UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Doctoral Programme in
Information Engineering and Computer Science

Enhancing Blockchain System Security, Smart Contract Verification, and Blockchain Scalability: A Multi-Dimensional Approach

Atefeh Zareh Chahoki

Advisor
Prof. Marco Roveri
Università di Trento

September 2025

Abstract

Blockchain enables decentralized trust, transparent data integrity, and programmable smart contracts, but its adoption is limited by security threats and low transaction throughput. This thesis addresses these challenges through integrated contributions that strengthen user security and improve blockchain scalability.

The first challenge examined is the rise of cryptojacking malware, where attackers covertly exploit user devices to mine cryptocurrencies. Existing defenses struggle against evasive behaviors due to reliance on observable signatures or heuristics. To address this, the thesis introduces CryptojackingTrap, an evasion-resilient detection framework that leverages low-level memory traces and network activity to identify mining operations. This approach achieves robust detection even against sophisticated evasion strategies such as code obfuscation, encrypted command-and-control channels, and reduced hash-rate mining, improving detection accuracy by an order of magnitude.

The second challenge involves smart contract security, where immutability amplifies the impact of vulnerabilities. This thesis enhances VeriSolid, a formal methods tool for correct-by-design smart contract development. By addressing verification limitations and introducing new property templates, the improved VeriSolid significantly expands the range of contract behaviors that can be verified. Its coverage—the proportion of software requirement specifications that can be formally represented—was increased from 45.6% to 90.5% over a dataset of 555 specifications, enabling verification of a far wider set of property specifications.

The third focus addresses the scalability bottleneck in the Ethereum blockchain, where sequential transaction execution prevents validators from fully exploiting multi-core infrastructure. Following a systematic review of blockchain concurrency and a comprehensive taxonomy, this thesis proposes two key contributions: a static analysis framework for detecting transaction conflicts and Conthereum, a novel scheduler that safely enables intra-block parallelism. By combining conflict detection with a high-performance, conflict-aware scheduling algorithm, Conthereum avoids costly re-execution and achieves near-linear throughput gains on standard 8-core machines. Although scalability diverges from linear with higher core counts and more frequent conflicts, Conthereum still substantially outperforms sequential execution and existing concurrent solutions across diverse conditions.

Collectively, these contributions advance cryptojacking detection, formal verification of smart contracts, and scalable transaction execution, paving the way for more secure, efficient, and widely adopted blockchain systems.

Keywords

Blockchain, Smart Contract, Ethereum, Security, Scalability

Acknowledgements

I had the honour and privilege of being advised by Prof. Marco Roveri. I am grateful for his scientific guidance, unwavering support, and mentorship throughout my work. His expertise and precision in scientific thought have been foundational to my academic growth. I am also thankful for the autonomy he granted me to pursue my ideas, the opportunities he provided, and his support through every challenge I encountered. His guidance extended well beyond the academic scope and has been invaluable to my professional development, for which I am sincerely grateful.

I am grateful to Prof. Luca Viganò, Prof. Mirco Giacobbe, and Prof. Domenico Siracusa for honouring me by accepting to serve on my thesis committee. I thank them for taking the time to read my thesis and for their invaluable comments on my work.

I would like to sincerely thank the University of Trento and its staff for providing this remarkable opportunity and for fostering a productive and supportive research environment. I am also grateful for the university financial support that has made this research possible. This work was partially funded by the European Union under NextGenerationEU (PRIN 2022 Prot. n. 202297YF75), the NRRP MUR program FAIR – Future AI Research (PE00000013), and the MUR PRIN 2020 – RIPER: Resilient AI-Based Self-Programming and Strategic Reasoning (CUP E63C22000400001). I am sincerely grateful for all the facilities that have been provided by the European Union and Italian academic institutions. The rich cultural and artistic environment of Italy has had a positive, lifelong impact on my personal growth. *Grazie bella Italia.*

My sincere thanks to Prof. Hamid Reza Shahriari at Amirkabir University of Technology for his collaboration and insightful support in the research presented in Chapter 2. I am also thankful to Prof. Daniele Fontanelli at the University of Trento for his expert guidance in the mathematical foundations of Chapter 2. I would like to thank Lorenzo Masè for his invaluable endeavors in exploring an AI-based performance enhancement approach to the proposed CryptojackingTrap framework during his bachelor's thesis and internship project, which I had the pleasure of co-supervising, and whose results are summarized in Appendix B.

I am also indebted to Prof. Daniel Amyot, Prof. Luigi Logrippo, and Prof. John Mylopoulos from the Contract Specification Modelling (CSM) Lab, as well as all their group members at the University of Ottawa, for their invaluable collaboration and feedback on the research presented in Chapter 3. Their kind, generous, and insightful assistance has greatly enriched both my research and my skills throughout my doctoral journey. *Merci beaucoup.* I am grateful to Dr. Anastasia Mavridou for her genuine guidance in facilitating the open-source collaboration described in Chapter 3 and for

her generous support of researchers.

I would like to thank Prof. Maurice Herlihy from Brown University, whose insightful guidance contributed to the research in Chapters 4 and 6. I also thank Ms. Sonja Paulson and Ms. Katherine (Kate) Correia from Brown University for their kind logistical and administrative support during our collaboration.

I gratefully acknowledge Prof. Claudio J. Tessone and Dr. Mark C. Ballandies for the wonderful opportunity to conduct a research visit at the UZH Blockchain Center, which greatly enhanced my insights in this domain. I extend heartfelt thanks to them and to all members of the center for cultivating an inspiring and collaborative environment, as well as for their thoughtful reviews and discussions that significantly improved the quality of my research, specifically the work presented in Chapters 5 and 6.

I would also like to thank Samuel Welde, co-founder of Onocoy, and Matthias Egli, managing partner and co-founder of ChainSecurity, along with his professional team, for their warm welcome, practical insights, and valuable feedback. I also extend my gratitude to Trust Square and the Hedera Foundation for their meaningful contribution to the growth and support of the blockchain community in Zurich. Their contributions enriched both the practical and visionary dimensions of my research. *Besten Dank an alle.*

To my colleagues, office mates, and friends — thank you for the uplifting moments, your kind encouragement, and valuable insights, which made the daily work of this research far more enjoyable and meaningful.

To my parents, whose selfless love, sacrifices, and unwavering strength laid the foundation of my journey. To my siblings and their families, for their constant encouragement, kind support, and for shouldering responsibilities in my absence — enabling me to dedicate myself fully to this work. I am deeply grateful for their presence at every step of this path. To my motherland, ایران — I owe you my identity, my values, and the heritage that shaped my path. Above all, to Almighty God — for constant guidance, strength, clarity, and resilience that carried me through. All the blessings I have acknowledged above are reflections of Your grace. Thank You.

List of Publications

Journal

[1] Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. "Cryptojackingtrap: An evasion resilient nature-inspired algorithm to detect cryptojacking malware." *IEEE Transactions on Information Forensics and Security* 19 (2024): 7465–7477.

International Conferences

[2] Atefeh Zareh Chahoki, and Marco Roveri. "Static Analysis for Detecting Transaction Conflicts in Ethereum Smart Contracts." In *Proceedings of the 2025 8th International Conference on Blockchain Technology and Applications (ICBTA)*, Kanazawa, Japan, 2025.

[3] Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. "Conthereum: Concurrent Ethereum Optimized Transaction Scheduling for Multi-Core Execution." In *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, Zurich, Switzerland 2025.

International Workshops

[4] Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. "Unmasking the Unseen: CryptojackingTrap, the Most Resilient Evasion-Proof Cryptojacking Malware Detection Algorithm." In *2024 IEEE International Workshop on Information Forensics and Security (WIFS)*, pp. 1–4. IEEE, 2024.

[5] Atefeh Zareh Chahoki, Marco Roveri, Daniel Amyot, and John Mylopoulos. "Revisiting formal verification in VeriSolid: An analysis and enhancements." In *CEUR Workshop Proceedings*, vol. 3629, pp. 55–60. CEUR-WS, 2023.

Preprint

[6] Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. "SoK: Concurrency in Blockchain–A Systematic Literature Review and the Unveiling of a Misconception." *arXiv preprint arXiv:2506.01885* (2025).

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

AUs Atomic Units
BDLedger Big Data Ledger
BEMP Blockchain Energy Market Place
BFT Byzantine Fault-Tolerant
BG Block Graph
BL Block Log
BRL Block Read Log
BTC Bitcoin
C&C Command And Control
CIC Computationally Intensive Contracts
CO CryptojackingTrap Occurrence
Conthereum Concurrent Ethereum
CP Constraint Programming
CSP Communicating Sequence Processes
CTL Computation Tree Logic
DAG Directed Acyclic Graph
DApps Decentralized Applications
DDoS Distributed Denial of Service
DEX Decentralized Exchange
DPoS Delegated Proof of Stake
DVC Deterministic Concurrency Control
ET Energy Trading
ETH Ethereum
EVM Ethereum Virtual Machine
FDR Failure Divergence Refinement
FIFO First-In, First-Out
FJSS Flexible Job Shop Scheduling
GA Genetic Algorithm
GC Garbage Collection
HO Hash Occurrence
ILP Integer Linear Programming
JSS Job Shop Scheduling
LCCF Least Conflicting Count First
LCDF Least Conflicting Duration First
LIFO Last-In, First-Out
LTL Linear Temporal Logic
MCCF Most Conflicting Count First
MCDF Most Conflicting Duration First
MEV Maximal / Miner Extractable Value
MinVWC Minimum Vertex Weighted Coloring
MO Mining Occurrence
Mempool Memory Pool
P2P Peer to Peer
PBFT Practical Byzantine Fault Tolerance
PCE Power Consumption Efficiency
PPoV Parallel Proof of Vote
PoA Proof of Authority
PoS Proof of Stake
PoV Proof of Value / Proof of Vote
PoW Proof of Work
RL Read Log
RQ Research Question
RW Read-Write
SC Smart Contract
SCE Smart Contract Engineering
SLOC Source Lines Of Code
SMT Satisfiability Modulo Theories
SMuLC Sequential Multi-Layer Consensus
SO Split Occurrence
SoK Systemization of Knowledge
SPA Static Program Analysis
STM Software Transactional Memory
TE Time Efficiency

TOD Transaction Ordering Dependency
TPS Transactions Per Second
WW Write-Write
XMR Monero

Chapter 1

Introduction

Blockchain technology, since the pioneering Bitcoin in 2009 [8], has undeniably marked a paradigm shift in decentralized systems and financial infrastructures, evolving from a foundation for digital currencies to platforms supporting smart contracts and decentralized general applications, exemplified by Ethereum [9]. Blockchain technology has revolutionized trust, authority, and transparency in the digital age. By enabling participants to transact, exchange value, and execute agreements without centralized control, it embodies a vision of open participation where no single entity holds privileged power. This vision—rooted in consensus, immutability, and decentralization—has inspired applications far beyond its original cryptocurrency scope.

However, this innovation comes with inherent challenges. The rapid profitability and immense growth of this ecosystem—not only attracts legitimate investors but also presents an ideal new playground for cyberattackers, specifically through cryptojacking attacks, which exploit victims' machines and CPU cycles to mine cryptocurrencies on behalf of the attacker. While blockchain has evolved from cryptocurrency to general-purpose smart contracts, newer concerns have emerged. The very properties that make blockchain appealing —the immutability and transparency— are also exploited in many cyberattacks, leading to huge financial losses and demanding sophisticated preventive and mitigating solutions. The so-called *Blockchain Trilemma* captures a central truth: security, scalability, and decentralization are interdependent and often in tension. Improvements in one dimension can undermine another, making it difficult to achieve all three optimally. For blockchain to evolve into a robust, widely adopted infrastructure, advances must address these fundamental aspects without destabilizing the others. While blockchain-based solutions provide decentralized alternative infrastructures, this emerging technology requires continuous fundamental evolution to match—and

ultimately surpass—the throughput and scalability of well-established legacy systems. This thesis addresses these critical areas by presenting novel approaches to enhance user security in the presence of new blockchain threats, specifically focusing on cryptojacking malware detection, then broaden the security perspective from its general-purpose use of smart contract security through formal methods, and finally blockchain infrastructure scalability through optimized concurrency. In the following, these aspects are elaborated.

Cryptocurrencies, while offering unprecedented financial autonomy, have also become a lucrative target for malicious actors. The computationally intensive nature of cryptocurrency mining has led to the emergence of 'cryptojacking,' where cybercriminals surreptitiously exploit victims' computing resources to mine cryptocurrencies for their own gain. This illicit activity, often hidden within executable files or browser-based scripts, poses a significant threat to system integrity and user privacy. Traditional botnet detection techniques often fall short against the evolving evasion tactics employed by these attackers. Our work introduces CryptojackingTrap, a novel first-of-its-kind cryptojacking detection solution designed to withstand common malware defense mechanisms. By using a debugger and extensible cryptocurrency listeners, CryptojackingTrap identifies the execution of cryptocurrency hash functions—an indispensable behavior of all cryptojacking executors. This is achieved by correlating memory access traces of suspicious executables with publicly available cryptocurrency P2P network data. This assembly-level investigation, combined with a nature-inspired algorithm to triggering detection alarms, provides an accurate and evasion-proof method for cryptojacking detection. This contribution is crucial for safeguarding the users from blockchain ecosystems external security threats.

As Beyond external threats, the internal integrity of blockchain applications, particularly smart contracts, is paramount. Smart contracts, as self-executing agreements deployed on the blockchain, manage vast volumes of transactions and assets. Their immutable nature, while ensuring trust and transparency, means that any coding errors or vulnerabilities can lead to irreversible financial losses and complex mitigation efforts. Ensuring the correctness of smart contracts before deployment is therefore even more critical than in conventional software applications. This can be achieved through auditing the developed smart contract code using various approaches—most notably, formal methods for the highest level of assurance. Another approach is to use correct-by-design smart contracts, where formal methods are applied during the design phase to ensure security, and the implementation code is generated from the formally verified design. Our research contributes to this latter approach by enhancing VeriSolid, a

security mitigation tool that enables developers to create and formally verify smart contracts using predefined property templates. After ensuring design correctness, VeriSolid facilitates the automatic generation of equivalent Solidity code. Through our work, we have identified and resolved verification issues in existing VeriSolid templates and proposed seven new templates, significantly expanding its software requirement definition coverage. This advancement directly addresses the security concerns inherent in smart contract development, ensuring their reliable operation.

Finally, the widespread adoption of blockchain technology is often hampered by its inherent scalability limitations. Platforms like Ethereum, despite their innovative decentralized architecture, face substantial throughput constraints. One of the reasons is due to their sequential transaction processing model. While this sequential execution guarantees deterministic state transitions and consensus, it severely restricts the number of transactions that can be processed per second, creating a significant performance gap compared to traditional centralized systems. Although solutions like sharding offer inter-shard parallelism, intra-shard execution often remains sequential, preventing full utilization of multi-core processing capabilities. Existing speculative execution frameworks attempt to introduce concurrency by reordering or rolling back conflicting transactions at runtime, but their effectiveness diminishes under high-conflict workloads due to frequent re-executions. To overcome these limitations, we propose Conthereum, a deterministic and conflict-aware scheduling framework that enables safe, intra-block parallelism without compromising execution consistency. Conthereum is the first solution to leverage the public availability of smart contract code for runtime scheduling through static analysis, enabling a more agile alternative to existing intra-block concurrency. By incorporating a novel, lightweight, high-performance scheduler inspired by the Flexible Job Shop Scheduling Problem (FJSS) and extensive conflict data obtained through static analysis, Conthereum achieves substantial speedup with minimal overhead, even under realistic, conflict-heavy conditions. This approach shifts from conflict resolution at runtime to conflict avoidance, significantly improving throughput and transactions per second (TPS) and contributing to the overall scalability of blockchain infrastructure. Our empirical evaluations demonstrate near-linear throughput gains with increasing computational power on standard 8-core machines. Although scalability deviates from linear with higher core counts and increased transaction conflicts, Conthereum still significantly improves upon the current sequential execution model and outperforms existing concurrent solutions under a wide range of conditions.

1.1 Research Problem and Objectives

Blockchain technology has demonstrated significant potential for decentralized applications and secure digital transactions, yet its widespread adoption is hindered by persistent technical and security challenges. Key issues such as emerging malware threats, vulnerabilities in smart contracts, and limitations in system scalability continue to pose risks to both users and network performance. Addressing these challenges is essential to ensure robust, efficient, and trustworthy blockchain systems. The following sections outline the primary research problems and corresponding objectives that this thesis seeks to tackle.

- **Vulnerability to Emerging Blockchain Threats:** The increasing value transacted on blockchain platforms makes them targets for cryptojacking attacks. Existing defenses remain insufficient against evasive malware, exposing computational resources and network integrity. Dynamic network traffic analysis extracts botmaster-slave interactions but is bypassed when C&C communication is encrypted. Static malware code analysis is hindered by code obfuscation, and dynamic analysis of OS API calls for cryptographic functions fails if malware uses custom implementations. Static detection of cryptography constants is similarly evaded by obfuscation. CPU activity monitoring targets resource-intensive cryptojacking, but attackers evade detection by lowering mining rates and shifting resources to less CPU-demanding activities such as spam, DDoS, and phishing. Finally, static analysis for mining pool keywords and URLs is also circumvented by obfuscation. These limitations demonstrate that current detection methods struggle against sophisticated evasive cryptojacking malware, highlighting the need for more robust solutions.

- **Smart Contract Verification:** The immutability of smart contracts, while a core strength, also means that any vulnerabilities embedded within their code can lead to irreversible financial losses and systemic risks. Traditional software development and verification methods are often inadequate for the unique requirements of smart contracts, necessitating more rigorous, formal verification approaches and extensive properties support.

- **Limited Scalability:** The sequential execution model prevalent in many blockchain architectures severely limits transaction throughput, creating a bottleneck that prevents these systems from competing with centralized financial

infrastructures. While various solutions have been proposed, a truly effective and deterministic intra-block parallelism mechanism that avoids runtime conflicts remains missing.

This thesis aims to address these multifaceted problems by pursuing the following primary objectives:

1. **To develop an advanced cryptojacking detection system** that is resilient to evasion techniques and provides high accuracy with minimal false positives. This involves analyzing the fundamental behaviors of cryptojacking malware and correlating them with network-level data to identify illicit mining activities effectively.

2. **To improved the security of smart contracts** by improving the existing correct-by-design tools. The goal is to enable developers to verify smart contracts with extended properties to increase their software specifications, thereby mitigating the risks associated with immutable code.

3. **To design and implement a novel, conflict-aware scheduling framework for blockchain transactions** that enables efficient intra-block parallelism on multi-core architectures. This objective focuses on transforming the sequential execution model into a parallel one, ensuring deterministic and conflict-free execution to significantly improve transaction throughput and overall blockchain scalability.

By achieving these objectives, this research seeks to contribute significantly to the development of more reliable, secure, and scalable blockchain systems, thereby fostering greater trust and enabling broader adoption of decentralized technologies.

1.2 Contributions of the Thesis

The major contributions of the works presented in this thesis resulted in 3 accepted publications and 3 archived publications under review. Each work offers our proposed solutions to address the existing challenges in order to fill the research gaps in the fields of blockchain security, and scalability. Specifically, the contributions can be summarized as follows:

Contribution 1: Novel Cryptojacking Detection with CryptojackingTrap

- [1] Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. "Cryptojackingtrap: An evasion resilient nature-inspired algorithm to detect cryptojacking malware." *IEEE Transactions on Information Forensics and Security* 19 (2024): 7465–7477.

- [4] Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. "Unmasking the Unseen: CryptojackingTrap, the Most Resilient Evasion-Proof Cryptojacking Malware Detection Algorithm." In *2024 IEEE International Workshop on Information Forensics and Security (WIFS)*, pp. 1–4. IEEE, 2024.

We introduce CryptojackingTrap, an innovative cryptojacking detection solution that benefits assembly-level analysis and correlation with P2P network data to identify illicit cryptocurrency mining. This system demonstrates zero false negatives and an exceptionally low false positive rate (10^{-20}), providing a robust and evasion-proof mechanism for safeguarding computational resources against stealthy attacks. This contribution enhances the security of user systems interacting with blockchain environments.

Contribution 2: *Enhanced Formal Verification for Smart Contracts with VeriSolid*

- [5]Atefeh Zareh Chahoki, Marco Roveri, Daniel Amyot, and John Mylopoulos. "Revisiting formal verification in VeriSolid: An analysis and enhancements." In *CEUR Workshop Proceedings*, vol. 3629, pp. 55–60. CEUR-WS, 2023.

We significantly advance the capabilities of VeriSolid, a formal verification platform for smart contracts. Our work identifies and resolves existing verification issues and proposes seven new property templates, expanding VeriSolid's coverage from 45.6% to 90.5% across a dataset of 555 requirement specifications. This enables developers to create 'correct by design' smart contracts, reducing the risk of vulnerabilities and financial losses due. This directly contributes to the security of smart contract ecosystems.

Contribution 3: *A comprehensive systematic literature review to organize a taxonomy of concurrency in Blockchain identify challenges and open research problems related to concurrency in blockchain.*

- [6] Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. "SoK: Concurrency in Blockchain–A Systematic Literature Review and the Unveiling of a Misconception." *arXiv preprint arXiv:2506.01885* (2025).

We present the first systematic survey of concurrency in smart contracts, offering a comprehensive review of existing literature and organizing it into distinct dimensions.

This work establishes a detailed taxonomy of concurrency levels within blockchain systems, examines vulnerabilities and potential attacks related to concurrent operations, and critically analyzes the underlying assumptions in current research. Notably, we identify and correct a fundamental misconception regarding concurrency assumptions in a major research category, guiding future research toward more accurate foundations. This provides a foundational understanding for future advancements in blockchain scalability.

Contribution 4: *Static Analysis Method for Transaction Conflict Detection*

- [2] Atefeh Zareh Chahoki, and Marco Roveri. "Static Analysis for Detecting Transaction Conflicts in Ethereum Smart Contracts." In *Proceedings of the 2025 8th International Conference on Blockchain Technology and Applications (ICBTA)*, Kanazawa, Japan, 2025.

As a foundational component of Conthereum, we propose a novel static analysis method to detect potential transaction conflicts in Ethereum smart contracts. This method precisely identifies read-write, write-write, and function call conflicts by analyzing state variable access patterns in Solidity contracts. The implemented tool achieves high precision in identifying potential conflicts, supporting the design of proactive transaction scheduling strategies that reduce runtime failures and enhance validator throughput. This contribution is crucial for enabling the efficient and reliable parallel execution of smart contracts.

Contribution 5: *Conthereum: Optimized Concurrent Transaction Scheduling for Ethereum*

- [3] Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. "Conthereum: Concurrent Ethereum Optimized Transaction Scheduling for Multi-Core Execution." In *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, Zurich, Switzerland 2025.

We introduce Conthereum, a pioneering framework for intra-block parallel transaction execution in Ethereum. Conthereum features a novel, lightweight, and high-performance scheduler inspired by the FJSS. By utilizing static analysis for conflict detection and a custom greedy heuristic algorithm, Conthereum enables conflict-free execution in multi-core environments, transforming Ethereum's intra-block sequential model into a parallel one. Empirical evaluations demonstrate near-linear throughput gains and superior performance compared to existing concurrent solutions, even under high-conflict conditions. This represents a significant leap forward in blockchain scalability.

Collectively, these contributions provide a holistic approach to addressing the multifaceted challenges of security, and scalability in blockchain systems, paving the way for more robust, efficient, and widely adopted decentralized applications.

1.3 Outline of the Thesis

This thesis is structured into three main parts, each addressing a distinct theme within the broader scope of blockchain security, and scalability. Each part comprises several chapters, with each chapter focusing on a distinct contribution in that part domain, often corresponding to a paper. The first and last chapters exceptionally are not under any part categorizations and addressing the introduction and conclusion of the whole thesis respectively. The organization is as follows:

- **Chapter 1: Introduction** (This Chapter): Provides an overview of the thesis, including background, motivation, research problems, objectives, and a summary of contributions.

Part I: Blockchain System Security: Cryptojacking Malware Detection

- **Chapter 2: CryptojackingTrap: An Evasion-Proof Cryptojacking Detection System**: This chapter details the design and implementation of CryptojackingTrap, a novel solution for detecting cryptojacking malware using dynamic analysis and pattern recognition. It highlights its resilience against evasion techniques and its high accuracy in identifying illicit mining activities.

PartII: Formal Methods for Smart Contract Security

- **Chapter 3: Revisiting Formal Verification in VeriSolid: An Analysis and Enhancements**: This chapter presents the advancements made to VeriSolid, a tool that enables the formal verification and ‘correct by design‘ development of smart contracts. It elaborates on the expanded coverage and new property templates introduced to enhance its capabilities.

Part III: Scalability in Blockchain Infrastructure Using Concurrency

- **Chapter 4: SoK: Concurrency in Blockchain-A Systematic Literature Review and the Unveiling of a Misconception**: This chapter offers a Systematization of Knowledge of concurrency in smart contracts, providing a taxonomy of existing levels of concurrency and analyzing their vulnerabilities and

potential attacks. It also addresses misconceptions in current research and identifies future research directions.

- **Chapter 5: Static Analysis for Detecting Transaction Conflicts in Ethereum Smart Contracts**: This chapter introduces a novel static analysis method for identifying potential transaction conflicts in Ethereum smart contracts. It details how read-write, write-write, and function call conflicts are detected by analyzing state variable access patterns, laying the groundwork for efficient transaction scheduling.
- **Chapter 6: Conthereum: Concurrent Ethereum Optimized Transaction Scheduling for Multi-Core Execution**: This chapter presents Conthereum, a framework that leverages static analysis and a novel greedy heuristic algorithm to enable intra-block parallel transaction execution in Ethereum. It demonstrates how Conthereum achieves significant throughput gains and improved scalability by proactively avoiding conflicts.

And the thesis is concluded with the following chapter:

- **Chapter 7: Conclusion and Future Work**: This chapter summarizes the main findings and contributions of the thesis, discusses their practical implications, and outlines promising avenues for future research.
- **Appendices**: The appendices provide supplementary material, including detailed technical specifications and additional experimental data. More specificly, Appendix A provides additional details on the implementation of CryptojackingTrap, while Appendix B presents an extended exploratory study on enhancing CryptojackingTrap's performance using machine learning. Appendix C offers supplementary materials related to the implementation of VeriSolid.

This structured approach ensures a comprehensive exploration of the identified research problems and a clear presentation of the novel solutions proposed in this thesis.

Part I

Blockchain System Security: Cryptojacking Malware Detection

Chapter 2

CryptojackingTrap: An Evasion Resilient Nature-Inspired Algorithm to Detect Cryptojacking Malware

This chapter corresponds to the articles:

Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. "Cryptojackingtrap: An evasion resilient nature-inspired algorithm to detect cryptojacking malware." *IEEE Transactions on Information Forensics and Security* 19 (2024): 7465–7477.

Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. "Unmasking the Unseen: CryptojackingTrap, the Most Resilient Evasion-Proof Cryptojacking Malware Detection Algorithm." In *2024 IEEE International Workshop on Information Forensics and Security (WIFS)*, pp. 1–4. IEEE, 2024.

2.1 Introduction

Flourishing ecosystems of cryptocurrencies, captivating investors' attention, present an irresistible opportunity for cyber-criminals to monetize their resources. This includes leveraging the widespread bot infrastructure and exploiting victims' resources in mining [10]. The focus of this work is *cryptojacking*, a term used to describe the surreptitious utilization of a user's computing resources to mine cryptocurrencies on behalf of the cyber-criminal, also known as *Coinjacking*, *Cryptomining* or *drive-by mining*. Its descendent type that exclusively mines Bitcoin cryptocurrency was named *botcoin* [11]. The consequences of cryptojacking attacks would be financially significant for both individuals and businesses as it would result in system slow-down, the decreased lifespan of hardware, and increased electricity expenses. Botnets have a variety of behavioral features in higher-level classifications of propagation, rallying mechanism, command, and control (C&C) communication, purpose, evasion, and topology [12]. Cyber-security professionals enhance security products based on the botnet features in different life cycles to trap and protect the final user devices from exploitation. These features are extracted from significant patterns observed in the under-attack network or host. Although extensive literature exists for detecting botnets, the attack and counter-attack mechanisms associated with them are continuously evolving due to the emergence of new technologies.

In June 2011, the first Bitcoin mining malicious software was uncovered in the wild [13]; Afterward, countless malware families have adopted cryptocurrency mining. The first botcoin detection reports used general botnet feature detection methods. Although all these methods benefit from vast experience in botnet detection, they did not use the unique features of miners to increase their detection precision. In [14], authors introduced BotcoinTrap, which exploits the memory access trace partial predictability feature of botcoins. Unlike other approaches such as MineSwipper [15], BotcoinTrap can detect botcoins in executable files or running processes, and cannot be bypassed by changing general botnet features, including C&C protocols features. However, BotcoinTrap is limited to detecting Bitcoin mining malware and lacks precision metrics calculations for the proposed algorithm. In this paper, these limitations are addressed, and implementation is significantly extended with numerous evaluations to introduce a new approach called CryptojackingTrap, capable of detecting cryptojacking activity on any cryptocurrency network.

In this work, CryptojackingTrap is proposed, drawing inspiration from the Venus Flytrap (Dionaea muscipula) plant that has evolved a unique mechanism to capture in-

sects. Detection of prey movement is achieved through tiny trigger hairs on its leaves, which, when stimulated, initiate a rapid trap closure. Intriguingly, the Venus Flytrap has a 20-second time window for a second strike, which has been optimized to maximize capture rates while minimizing energy expenditure [16]. As computer scientists, inspiration is derived from this efficient mechanism to develop a novel approach for detecting cryptojacking malware. The proposed method analyzes to identify malicious behavior triggered by suspicious events within an optimized window, intending to reduce both false positive and false negative rates, akin to the Venus Flytrap's optimization for survival. This research goal is to find cryptojacking detection techniques that cannot be bypassed using known evasion approaches (2.2) and his paper makes the following contributions: (1) A novel approach and efficient algorithm named *CryptojackingTrap* is introduced for detecting cryptojacking malicious behavior in suspicious executables, including executable files and processes. This approach is resilient against malware evasion techniques including obfuscation, encryption, and hash rate reduction. Its design is open-to-extend, and it supports further cryptocurrencies without requiring changes to core modules, due to the incorporation of lightweight plugins. (2) The entire solution has been implemented, comprising 6.5K SLOC (source lines of code), and the source code is made available to the research and industrial communities under an open-source license. (3) A mathematical analysis has been conducted to calculate the false positive rate for CryptojackingTrap detection, yielding a value of 10^{-20} for a typical application. (4) Empirical evaluation results confirm the precision of this approach across five dataset categories, totaling over 15 GB of public data. (5) A marked field is proposed for other cryptocurrency block data structures. This marked data can be used in blockchains where significant predictable data is lacking, enabling the CryptojackingTrap solution to still function effectively. The marked field facilitates the CryptojackingTrap with seed data to track memory access and ensure successful detection. The remainder of this paper is organized as follows: The problem statement is presented in Section 2.2, Section 2.3 provides the most relevant background concepts used throughout the paper. Section 2.4 is dedicated to elaborating this novel approach. Section 2.6 evaluates this approach mathematically and experimentally. Comparing this research contribution with other researchers' related work is done in Section 2.8 and conclusions are discussed in Section 2.9.

2.2 Problem Statement

To determine the scale of the problem, researchers have analyzed CPU usage and campaign size. For instance, Zimba et al. [17] estimated that 32 percent of U.S. users are vulnerable to browser-based crypto mining. Hong et al. [18] found that approximately 10 million web users are affected by cryptojacking every month, resulting in a daily cost of 59,000 USD due to an additional 278K kWh of power consumption. Furthermore, [15] estimated that the profit of each cryptojacking campaign ranges from 14.36 USD to 31,060.80 USD per month on average, while [19] estimated an average profit of 340 USD per campaign per day (about 10,200 USD per month).

The malware detection problem becomes more complicated when attackers use techniques to evade detection tools. Botnet detection mechanisms and, consequently, evasion mechanisms target different life-cycle stages of botnets. Life-cycle stages are botnet conception, recruitment, interaction, marketing, attack execution, and attack success [20]. (1) A dynamic analysis of network traffic can be used to analyze the interactions between a botmaster and slaves by extracting features and behaviors on the network. However, the connection and interaction stages may be susceptible to bypassing if the C&C communication is directed through an encrypted channel. (2) Static analysis approaches over malware code are vulnerable to code obfuscation. Dynamic analysis of OS APIs for calling cryptography functions including hash is not effective if malware developers do not use OS APIs and develop their own customized code for calculations. (3) Static analysis of cryptography functions by focusing on finding cryptography function constants is another detection approach but it is not resilient to code obfuscation[21]. (4) Since cryptojacking is the most process-demanding attack, CPU activity analysis is also targeted as a detection solution. This security solution can still be evaded by decreasing the mining rate and dedicating that portion of botnet resource capacity to less computation-demanding tasks such as spam, click fraud, DDoS, identity theft, and information phishing. To maximize profits, botnet herders run resource-disjoint monetization activities simultaneously [22]. (5) Static code analysis to find constants related to mining pools keywords, and URLs is an alternative approach for detecting cryptojacking attacks. However, this approach is not resilient to code obfuscations.

2.3 Background

The glory of clear consensus over the world's most critical asset, money, without trusting any trusted party is rooted in blockchain and consensus protocols. Blockchain is a ledger of timestamped blocks of transactions chained by persisting the previous block hash in each block and the network is maintained by miners. They do a computation-intensive mathematical activity called mining which demands massive hash calculations on the block. In this section, detailed information on block structure is investigated in Bitcoin, Ethereum, and Monero. The targeted goal is to find predictable consecutive fields in the next valid block that is not broadcast on the network yet. These fields are clues to detect executables that deal with them very frequently in processing and memory retrievals. Amongst all cryptocurrencies' block parts, solely block header is potentially partially predictable which is elaborated in Table 2.1. Other parts due to their diversity based on miners' transactions selection are not the subject of this section elaboration.

2.4 CryptojackingTrap

This section elaborates CryptojackingTrap, an evasion resilient approach to detecting cryptojacking mining across various cryptocurrencies. Cryptojacking botnet matches general botnet life cycle stages as described in [20]. After examining different detection approaches, it was concluded that the life-cycle stage with the least diversity in evading techniques for cryptojacking detection is the last one, where the attacker must utilize the victim's machine to perform hash function calculations. In this stage, irrespective of variations in other life-cycle stages, all miners exhibit the same behavior of mining on the victim's machine, which is inevitable.

As expounded in Section 2.2, network, code, and CPU analysis are subject to potential evasion techniques from encryption, obfuscation, and mining rate reduction, respectively. Regardless of the characteristics of the code, network traffic, or hash calculation rate, attackers must still retrieve data and calculate the hash. This behavior is essential and therefore forms the basis for trapping cryptojacking attacks.

Dynamic analysis is utilized to trace the memory retrieval contents that malware uses as input parameters for hash function calculation. This information is practically impossible to conceal and can be partially predicted at any time. The input for the hash function consists of the data structure of the cryptocurrency block, and it is not entirely predictable what this data will be on the victim's machine during an attack.

Table 2.1: Bitcoin (top), Ethereum (middle), and Monero (bottom) block headers

	Field	Purpose	E.U.T. for the miners	Size (Byte)
Bitcoin	**Version**	indicates the set of block validation rules that must be followed	for soft-fork and update	4
	Previous Block Header Hash	chains valid blocks and ensures approved blocks integrity	almost once a ten minutes per newly broadcast mined block	32
	Merkle Root Hash	derives from the current block transactions hashes	by chaining any transactions	32
	Time	counts the current timestamp in second after 1970-01-01T00:00 UTC	every second	4
	Bits	indicates the current proof of work hardship	calculated every ten minutes	4
	Nonce	finds a random number in a brute-force search for PoW	random and unpredictable	4
Ethereum	**Parent Hash**	chains valid blocks and ensures approved blocks integrity	almost once a twelve seconds	32
	Ommers Hash	specifies ommers block	by changing ommers	32
	Beneficiary	declares beneficiary address, i.e mining fee recipient	by changing any beneficiaries	20
	State Root	specifies head node of the state trie	by changing an account state	32
	Transaction Root	keeps track of transactions in the current block	by changing any transactions	32
	Receipts Root	declares the hash of the root node of recipients in the block transactions	by changing any recipients	32
	Logs Bloom	logs searchable information about the block transactions	by changing any transactions	256
	Difficulty	indicates the number of leading zeros in the current block's hash	dynamically is calculated	32
	Number	counts the ancestor blocks for the current block	by increasing blockchain length	32
	Gas Limit	limits the total gas that may be used by the current block	average 15M but miner defines	32
	Gas Used	declares the total gas that was used by the current block	by changing transactions	32
	Timestamp	equals the block creation time and is the output of Unix time()	every second	32
	Extra Data	adds extra data relevant for this block	arbitrary by miners	32
	Mix Hash	specifies the current block hash and must meet the difficulty constraints	by changing the block	32
	Nonce	finds a random number in a brute-force search for PoS	random and unpredictable	8
Monero	**Major Version**	specifies the major block header version (always 1)	constant value (1)	1
	Minor Version	serves as a voting mechanism	updated based on miner vote	1
	Timestamp	indicates block creation time (UNIX timestamp)	each second	8
	Prev Id	specifies the identifier of the previous block	on new mined block broadcast	32
	Nonce	finds a random number in a brute-force search for PoW	random and unpredictable	4

This is because the block, including collected transactions, comprises flexible values that the cryptojacking miner determines. Nonetheless, the subsequent section explores which elements of a block can be predicted, and the formal evaluation presented in Section 2.6.1 demonstrates that this prediction is significant enough to serve as detection criteria.

2.4.1 Forecasting Attacks Data in Use

According to the E.U.T field on Table 2.1, it can be concluded that the "previous block hash" possesses outstanding features. First, this field is shared in all cryptocurrency block headers, as the most fundamental chaining feature of blockchains is implemented using this chaining field. Second, the value of this field is predictable for the miner's block under development at any given time, as it must be equal to the hash of the last accepted block on the network that is publicly available. Lastly, the size of this field (32 bytes) is sufficient for correlating suspicious executable memory traces and network data. While some fields, such as the "major version" in Monero, are constant and predictable, they are not candidates due to their small size (1 byte).

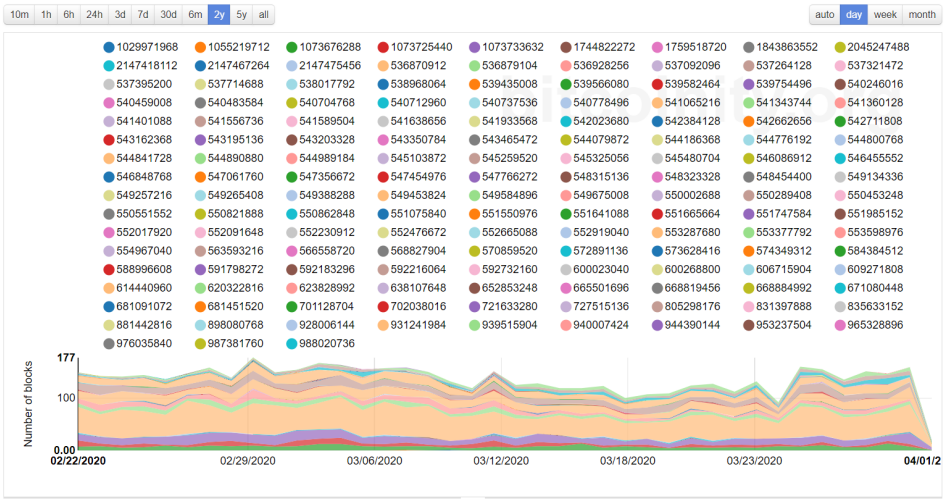


Figure 2.1: Diverse Bitcoin block versions actively used by network [7]

The "version" field in Bitcoin is followed immediately by the "previous block header," resulting in a predictable 32+4 byte data sequence. In 2018, BotcoinTrap detected predictable data by assuming the block version as predictable, as the average active block versions per day were only two. However, this field is no longer predictable. As shown in Figure 2.1, Bitcoin blocks were broadcast with 124 different versions in

one month (from 2/13/2020 to 3/13/2020). However, the diversity of versions in 2023 has increased significantly and is too vast to be represented in a pictorial diagram.

In consideration of the adoption of "previous block header hash" as the shared big predictable field for its algorithm, CryptojackingTrap is analyzed in Section 2.6.1 to possess sufficient precision in detection. It is crucial to note that the term 'hash' used throughout the rest of this paper to describe a predictable field can be substituted with any other predictable field and calculations are not affected.

To tackle the challenge of multiple previous block hash in cryptocurrencies, particularly when dealing with orphan blocks, the authors proposed the incorporation of a dedicated "marked field" within the block data structures. This approach presents a streamlined and efficient solution that eliminates the necessity for cryptocurrency listeners and results in reduced computational overhead, as detection relies solely on the marked field. Importantly, this implementation seamlessly integrates with existing cryptocurrency implementations, requiring no significant change, and can be smoothly deployed for improved detection precision and performance. Another noteworthy suggestion is to incorporate this detection approach into hash function design, ensuring that the predictable field remains intact during hash method calculations and avoids the fragmentation that would otherwise necessitate fetching large segments of the predictable field from memory, thereby serving as a robust indicator for detecting miner malware.

2.4.2 The Architecture of CryptojackingTrap

In this section, the architecture of CryptojackingTrap is proposed in Fig. 2.2, comprising two primary blocks. The right block is labelled as "Core" which includes the main modules of CryptojackingTrap (monitor and detector), and the left as "Extensions" including listeners. In the following, different modules of the architecture are described:

Listener. The listeners are lightweight extensions, each one responsible for listening and gathering information from its specific blockchain P2P network. Each listener listens to its specific network block announcements and extracts valuable data, which is the previous block hash (Section 2.4.1), and stores it on the "Block Log (BL)" file(s). Block logs will be used for the rest of the detection process in the detector module. Since block logs have a predefined internal format that is also expected in the detector, developers can develop new listeners for the CryptojackingTrap solution without any need for chaining the core modules.

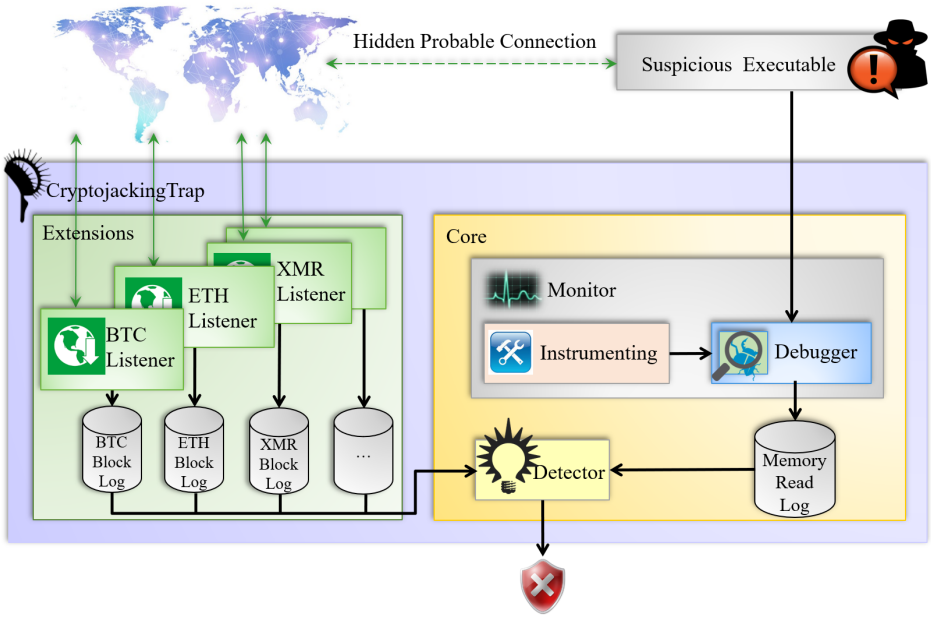


Figure 2.2: Asynchronous architecture of CryptojackingTrap.

Monitor. On the right side of Fig. 2.2, the debugger module is dedicated to tracing the suspicious executable's activities. The type of activity that CryptojackingTrap is interested in monitoring, which is a memory read access, is defined in the instrumenting part of the monitor module. Consequently, the output of this module is the "Read Log (RL)" file(s) capturing all memory read access traces of the suspicious executable.

Detector. The cryptojacking malware and CryptojackingTrap listeners are both connected to the same cryptocurrency networks and malware mines over the blocks that are also visible to the listeners. The detector module is responsible for taking BL and RL from listeners and monitor respectively and correlating and deciding if the application under test is malware. The detailed algorithm of the detector will be explained in the rest of this paper in Section 2.4.4.

The proposed CryptojackingTrap solution has two possible architectures: asynchronous and synchronous. The *asynchronous architecture*, which is currently adopted and has been explained, allows the listeners and monitor modules to run independently, without forcing them to execute simultaneously. This approach provides more flexibility and does not compromise the efficiency of the system. On the other hand, the *synchronous architecture* requires all modules to run at the same time, and the detector algorithm is merged inside the monitor module. This approach eliminates the need to store block logs and memory read logs since the monitor/detector module is aware of blocks and memory traces in real-time.

Therefore, although the synchronous architecture is a possible architecture from the design point of view, it is not practical with implementation considerations. The current asynchronous architecture forces the least interference in suspicious executable normal execution and provides more flexibility.

The CryptojackingTrap architecture is designed to be highly extensible and adaptable to evolving industry trends by facilitating the developers to incorporate new cryptocurrencies without affecting the integrity of core modules. Additionally, the architecture is modular and each module is capable of utilizing different platforms as long as they adhere to the internal data interface.

2.4.3 The Sequence of CryptojackingTrap Processes

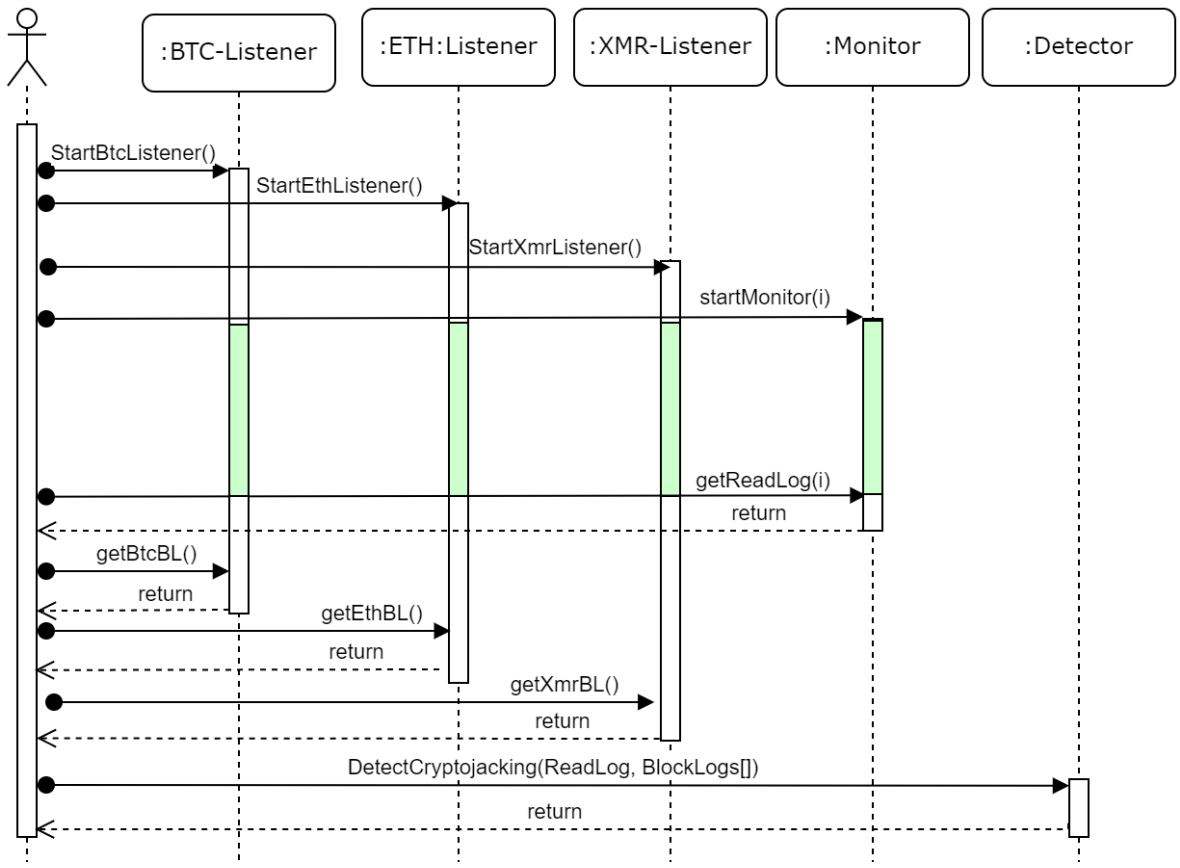


Figure 2.3: Sequence diagram of CryptojackingTrap.

Fig. 2.3 illustrates the time sequence of interactions between the client and CryptojackingTrap modules. The green section shared between listeners and monitor modules indicates that data for all these modules must pertain to the same time frame. Since

the history of cryptocurrency P2P network data is always available, the necessity of executing `startXListenr` before `startMonitor(i)` can be eliminated, and it suffices to cover debugging time in the listeners and execute `getXBL` after `getReadLog(i)` (X for currency names). Ultimately, presenting the entire log files to the detector using `DetectCryptojacking` provides a detection response, which its algorithm will be elaborated further in the subsequent section.

Table 2.2: Summary of Notation for CryptojackingTrap.

Level	Symbol	Meaning
SO	Ω_c	Predictable bytes count in each cryptocurrency c
	Ξ	Min size of each acceptable SO's substring
HO	Π	Min percentage of SO's hash coverage that specifies one HO
	Δ	Max window of all SOs to specify one HO
MO	Ψ	Min number of HOs to specify one MO
	Θ	Max window of all HOs to specify one MO
CO	Γ	Min number of MOs to specify the CryptojackingTrap alert
	Λ	Max window of all MOs to specify the CryptojackingTrap alert

2.4.4 The Detection Module Algorithm

Cryptojacking must calculate a hash function repeatedly as the main process and because this function input size is bigger than the processor's registers, a processor can not retrieve it once and keep it for further calculations. Consequently, the processor has to retrieve this value continuously during mining. This repeated data retrieval persisted in RL and significant repetition of BL value in RL with a consistent timestamp of each record implies mining on the suspicious executables.

The hash function input value including the previous block hash is not expected to be retrieved at once and intact but it is retrieved fragmented and also fragment's size depends on each hash algorithm. To tackle the uncertainty of splitting the hash value into unknown parts with undetermined size, four levels of abstraction were introduced to organize these data and provide a significant probability to specify the correct alert of cryptojacking detection. The notations that are used in the algorithms and calculations are listed on Table2.2 and organized based on the abstraction levels that will be described more in the remaining.

The levels of abstraction introduced by CryptojackingTrap are arranged from the lowest to the highest level, with each level building on the abstraction concept from the lower layer. As each level progresses, the criteria for true CryptojackingTrap detection becomes more stringent, resulting in higher detection precision. The examples provided in Fig. 2.4 visually illustrate these levels, aiming to help the reader develop an intuitive

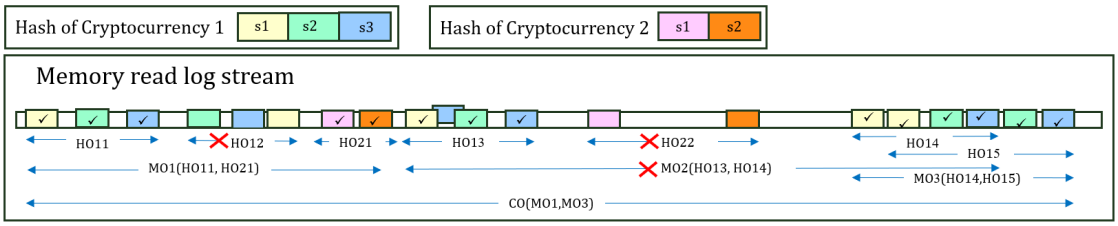


Figure 2.4: Illustrative examples demonstrating levels of CryptojackingTrap detection abstraction.

understanding of the algorithms discussed in this section.

Split Occurrence (SO). In the first level of detection abstraction, which is the lowest level, finding the substrings of the BL's hash values in the RL file is targeted. It is apparent from the statistics and probability basic calculations that the bigger the split substrings of hash value can appear in the RL file, the more probable it can be the indication that the executable file reads this specific hash value from memory. Consequently, in this level of abstraction, the detector searches for the substrings named splits with the following properties: Each split must be the largest feasible part of the hash that occurs in the RL file. The splits do not overlap with each other. The split sizes must be bigger than a threshold (Ξ), and they must appear exactly in the same order that they are in the hash value. In Fig. 2.4 example, the splits of the first cryptocurrency hash value are s1, s2, and s3, and the splits for the second cryptocurrency are s1 and s2.

After finding these substrings, this procedure constructs a `SplitOccurrence` object that includes detailed information about this substring. This detailed information includes the substring content and indexes to address it in both the hash string and RL file. To find the biggest possible splits a greedy divide and conquer procedure is declared in Algorithm 1, named `findSplitOccurrences`. It starts by calling `findLCS` procedure on line 2 which is a pure string processing procedure that finds the *longest common substring (LCS)* shared between `hash` and `readLog` and terminates the procedure by returning `null` if found substring is less than Ξ on line 18 and 19. `SplitOccurrence` object includes other integer attributes: `strStartIdx` and `rlStartIdxes` are the first indexes of the firstly found `splitStr` in the `hash` (line 20) and `readLog` (line 22) respectively. `strEndIdx` and `rlEndIdxes` are the last indexes of the firstly found `splitStr` in the `hash`(line 21) and `readLog` (line 23) respectively. To make it clearer, `strEndIdx` minus `strStartIdx` is equal to `rlEndIdxes[i]` minus `rlStartIdxes[i]` is equal to `splitStr` size.

`subStr(integer fromIdx, integer toIdx)` is used on line 6 and 10 to return the

Algorithm 1 Detector Algorithm: Find Split Occurrences

```
1: procedure FINDSPLITOCCURRENCES(hash: String, readLog: File):
   SplitOccurrence[]
2:   mSplit ← FINDLCS(hash, readLog)                          ▷ main split
3:   if  mSplit is null then
4:     return null
5:                                                  ▷ left side recursive call:
6:   lHash ← hash.subStr(0, mSplit.strStartIdx)
7:   lFile ← readLog.subFile(0, mSplit.rlStartIdxes[0])
8:   lSplits ← FINDSPLITOCCURRENCES(lHash, lFile)
9:                                                 ▷ right side recursive call:
10:  rHash ← hash.subStr( mSplit.strEndIdx, hash.size)
11:  rfile ← file.subFile(mSplit.rlEndIdxes[0], readLog.size)
12:  rSplits ← FINDSPLITOCCURRENCES(rHash, rfile)
13:  return new SplitOccurrence[] {lSplits,mSplit,rSplits}
14:
15: procedure FINDLCS(str: String, readLog: File): SplitOccurrence
16:  s ← new SplitOccurrence
17:  s.splitStr ← LCS (str, readLog)
18:  if  s.splitStr < Ξ then
19:    return null
20:  s.strStartIdx ← the left index of splitStr in str
21:  s.strEndIdx ← the right index of splitStr in str
22:  s.rlStartIdxes[] ← find all left indexes of splitStr in readLog
23:  s.rlEndIdxes[] ← find all right indexes of splitStr in readLog
24:  return s
```

substring of the callee string from the first index to the second index. `subFile(integer fromIdx, integer toIdx)` on line 7 and 11 similarly returns a new file containing the content of the callee file from the first index to the second index. consequently, `lHash` and `lFile` on line 6 and 7 are the left parts of `hash` and `readLog` respectively before the main split on them and `rHash` and `rFile` on line 10 and 11 are the right ones after the main split. `subFile` is a pseudocode to simplify its logic, but considering the performance concerns the same logic is implemented by passing indexes over one file. On line 13 a list including the left splits, main split, and right splits is returned as a result by preserving the order of split occurrences.

Hash Occurrence (HO). HO is the second level of detection abstraction indicates an attempt of reading a hash value from memory. Since an executable can not read a hash value from memory just by a single memory read access, HO encapsulates a list of splits that nondeterministically indicates one hash occurrence. HO guarantees a significant probability of correctness by enforcing criteria over splits selection specified

in Algorithm 2.

Algorithm 2 Detector Algorithm: Find Hash Occurrences

```
1: procedure FINDHASHOCCURRENCES(hash: String, readLog: File):
   HashOccurrence[]
2:   sos ← FINDSPLITOCCURRENCES(hash, readLog)
3:   hos ← new HashOccurrence[]
4:   maxHOCount ← min size for rlStartIdxes lists per split in sos
5:   for splitIdx = 0; splitIdx ⩽ maxHOCount; splitIdx++ do
6:     ho ← new HashOccurrence()
7:     ho.hash ← hash
8:     sections ← new Section[]
9:     for so: sos do
10:      section ← new Section
11:      section.splitStr ← so.splitStr
12:      section.rlStartIdx ← so.rlStartIdxes[splitIdx]
13:      section.rlEndIdx ← so.rlEndIdxes[splitIdx]
14:      sections.add(section)
15:    if sections[sections.size-1].rlEndIdx -
16:      sections[0].rlStartIdx ⩽ Δ then
17:      sum = ∑_{i=0}^{sections.size−1} sections[i].splitStr.size
18:      hcp ← sum / hash.size * 100
19:      if hcp ≥ Π then
20:        ho.sections ← sections
21:        ho.rlStartIdx ← sections[0].rlStartIdx
22:        ho.rlEndIdx ← sections[sections.size-1].rlEndIdx
23:        hos.add(ho)
24:  return hos
```

The algorithm utilizes a temporary data structure called `section` (instantiated on line 10) to organize splits that correspond to a single HO. While each `SplitOccurrence` generated on line 2 denotes a split string and all its occurrences in `readLog`, a `section` refers to a specific split occurrence. Line 8 defines a list of sections that pertains to all splits and their splitIdx$^{\text{th}}$ occurrences, which could potentially indicate an HO if it satisfies the following criteria.

The first criterion is that all splits within an HO must occur within a window of size Δ in `readLog`, as indicated on line 15 and 16. The second criterion is that concatenating the *splitStr* strings of the *sections* in the correct order must result in a string that is nearly identical to the input *hash* value. Specifically, the concatenated value can only differ by a few characters, which leads to the concept of *hash coverage percentage (HCP)*. The HCP is calculated as the sum of the sizes of all split strings in the HO's sections divided by the length of the hash and multiplied by 100. To ensure this criterion is

met, the HCP for each HO is checked against a threshold value of Π on line 19.

In Fig. 2.4 examples, HO_{ij} represents the j^{th} suggested hash occurrence for the i^{th} cryptocurrency, with accepted splits indicated by check marks. HO11 is valid since it includes all ordered splits within an accepted window, while HO12 is invalid due to unordered splits. HO21 demonstrates interleaved hash occurrences, HO13 shows a selection of non-overlapping splits, HO22 rejects too-distant splits, and HO14 accepts all splits within an accepted window even though there might be extra valid splits in that window.

Mining Occurrence (MO). The third level of abstraction is dedicated to structuring a mining activity based on found HOs. The pseudocode for finding MOs from HOs is explained in Algorithm 3. The algorithms benefit from object-oriented notions such as `DetectorSetting.lstnSetting.currencies` on line 3 to organize data based on supplier modules. Since the hash function must be repeatedly calculated in the cryptojacking algorithm, CryptojackingTrap is designed to be sensitive to the minimum count (Ψ) of detected HOs (line 30) in a limited window size of Θ (line 26). Initially findMiningOccurrences is called

In Fig. 2.4, MO1 is the initial valid mining occurrence considering $\Psi = 2$, encompassing HO11 and HO21 as valid hash occurrences. Notably, disparate cryptocurrency hash occurrences can collectively form an MO, curbing malware evasions across mining on varied platforms. MO2 is invalidated due to the distance between HO13 and HO14, exceeding Θ. By omitting HO13 and restarting the MO window from HO14, followed by HO15, it ended with the accepted MO3. This restarting of the window is done on line 37 in Algorithm 3.

CryptojackingTrap Occurrence (CO). This is the fourth and highest level of abstraction used to determine if a suspicious executable is a cryptojacking malware, as described in Algorithm 4. The `DetectorSetting` input object (line 1) contains all the necessary data for detector execution, including `lstnSetting` and `mntrSetting`, which represent the complete data received from the listener and monitor modules, respectively.

The `isCryptojacking` procedure output is a boolean value to indicate the detection result of cryptojacking activity in the input setting (line 1). This procedure retrieves all mining occurrences (line 2) and checks if there are at least Γ MOs (line 11) in a Λ-sized window (line 9). In Fig. 2.4, assuming $\Gamma = 2$ and counting MO1 and MO3 as the accepted mining occurrences, the algorithm ends to the final CryptojackingTrap alarm in the last layer of abstraction.

Although the algorithm elaboration in this section is self-sufficient, readers are facil-

Algorithm 3 Detector Algorithm: Find Mining Occurrences

```
1: procedure FINDMININGOCCURRENCES(s: DetectorSetting): MiningOccurrence[]
2:   hos ← new HashOccurrence[]
3:   for  String c ∈ s.lstnSetting.currencies do
4:     t_start ← min(s.mntrSetting.RL.getAllTimestamps())
5:     t_end ← max(s.mntrSetting.RL.getAllTimestamps())
6:     for i_t = 0; i_t ⩽ s.lstnSetting[c].BL.size-1; i_t++ do
7:       t ← s.lstnSetting[c].BL[i_t].timestamp
8:       if  t_start ⩽ t ⩽ t_end then
9:         t̂ ← s.lstnSetting[c].BL[i_t + 1].timestamp
10:        rl ← GETSUBRL(s.mntrSetting[c].RL, t, t̂)
11:        h ← s.lstnSetting[c].BL[i_t].blockHash
12:        hos.add(FINDHASHOCCURRENCES(h, rl))
13:  mos ← FINDMOSFROMHOS(hos)
14:  return mos
15:
16: procedure FINDMOSFROMHOS(hos: HashOccurrence[]): MiningOccurrence[]
17:  mos ← new MiningOccurrence[]
18:  for i = 0; i ⩽ hos.size-1; i++ do
19:    mo ← new MiningOccurrence()
20:    ho_start ← hos.get(i)
21:    selectedHOs ← new HashOccurrence[]
22:    selectedHOs.add(ho_start)
23:    mo.rlStartIdx ← ho_start.rlStartIdx
24:    for j = i+1; j ⩽ hos.size; j++ do
25:      ho_end ← hos.get(j)
26:      if ho_end.rlEndIdx-ho_start.rlStartIdx ⩽ Θ  then
27:                                          ▷ collect HOs in a Θ-sized window
28:        selectedHOs.add(ho_end)
29:        mo.rlEndIdx ← ho_end.rlEndIdx
30:        if selectedHOs.size ≥ Ψ then
31:          mo.setHOs(selectedHOs)
32:          mos.add(mo)
33:          i = j+1
34:          break                          ▷ continue with finding the next mo
35:        else continue                       ▷ add the next HO to selectedHOs
36:      else break                    ▷ could not collect Ψ hos in a Θ-sized
37:           window, start with the next HO as the first one
38:  return mos
39:
40: procedure GETSUBRL(srcRL: File, from: TimeStamp, to: TimeStamp): File
41:  return  a new file including selected lines from RL file that their
   timestamp are between from and to timestamps
```

Algorithm 4 Detector Algorithm: Is Cryptojacking

```
1: procedure isCryptojacking(setting: DetectorSetting): Boolean
   ▷ returns true if finds Γ MOs in a Λ-sized window
2:   mos ← findMiningOccurrences(setting)
3:   for i = 0; i ⩽ mos.size-1; i++ do
4:     mo_start ← mos.get(i)
5:     acceptedMOs ← new MiningOccurrence[]
6:     acceptedMOs.add(mo_start)
7:     for j = i+1; j ⩽ mos.size-1; j++ do
8:       mo_end ← mos.get(j)
9:       if mo_end.rlEndIdx-mo_start.rlStartIdx ≤ Λ  then
10:        acceptedMOs.add(mo_end)
11:        if acceptedMOs.size ≥ Γ then
12:          print "Positive Alarm!" + acceptedMOs
13:          return true
14:        else continue                    ▷ add the next MO to acceptedMOs
15:      else break                             ▷ restart accepted MO window
16:   print "Negative Alarm!"
17:   return false
```

itated to follow the implementation by using the same naming conventions simultaneously. Moreover, the algorithm serves as a seamless transition from the object-oriented declarative notation used in Java-based code to the concise abbreviations used in the formal evaluation section, which will be discussed in Section 2.6.1.

2.5 CryptojackingTrap Implementation

This section provides technical details on the implementation of the CryptojackingTrap modules. The listeners are implemented in Java and support Bitcoin, Ethereum, and Monero cryptocurrencies. Bitcoin implementation has a user interface to specify the P2P nodes count while others are console-based applications. Despite language and design differences, the listeners produce output files in a uniform format. The monitor module is implemented using the `Pin` debugger software [23], described in [24]. The current experimental evaluation benefits from `Pin` 3.22 (February 28, 2022), with instrumentation coded in C++ using Visual Studio 2022. The output DLL (Dynamic Link Library) file generated by the instrumentation is fed into the `Pin` debugger to log the memory read access traces of the suspicious application. Respecting the monitor module's predefined file format, it is possible to substitute the `Pin` debugger with any off-the-shelf debugger. The detector is a Java-based project equipped with an inter-

face, that facilitates users to specify the BL file(s) and cryptocurrencies of interest for scanning, as well as their corresponding RL file(s) obtained from the respective listeners. By running the detector, users can trace the debug logs until the final positive or negative alert is announced. The project's entire codebase and resulting datasets are available for open science on GitHub. [1]. The project encompasses nearly 6.5K SLOC, and has a distinct repository for any module for independent versioning. These modules are monitor, listener-bitcoin, listener-ethereum, listener-monero, detector, util, and dataset. Further details on the implementation and file formats can be found in the comprehensive project documentation.

2.6 Evaluation

CryptojackingTrap is evaluated through two approaches. The first approach involves a mathematical evaluation to calculate the false positive rate (Section 2.6.1), while the second approach focuses on calculating false positive and false negative rates using experimental evaluation approaches (Section 2.6.2).

2.6.1 Mathematical Evaluation

This section quantitatively evaluates the accuracy of the CryptojackingTrap algorithm in detecting cryptojacking activities. To carry out these actions, the data and calculations of all modules (listener, monitor, and detector) are formalized and used to calculate the false positive rate. For the sake of consistency in this evaluation, 'blocks' will be used to refer to cryptocurrency blocks, irrespective of their type (e.g., Bitcoin, Ethereum, Monero, etc.).

Listener. The listener module maintains block log (BL) files for all supported cryptocurrencies (C). The BL for a cryptocurrency $c \in C$ is denoted as BL_c and formulated in Eq. (2.1). BL_c is as a finite chronologically ordered sequence of pairs (t_i, h_i), where t_i denotes the creation timestamp of the i^{th} block and h_i represents the corresponding block hash value. The counter i iterates through the recorded block log hashes within the block log file(s) specific to cryptocurrency c, starting at 1 and ending at BC_c, which represents the total block count within the BL_c file(s).

$$BL_c = \{(t_i, h_i) \mid i \in \{1, 2, 3, \ldots, BC_c\}\} \tag{2.1}$$

[1] https://github.com/CryptojackingTrap/CryptojackingTrap

Monitor. The output read log (RL) of the monitor module is represented as a finite ordered sequence of (t_j,c_j) tuples in the following formula organized chronologically, with $j \in \{1, 2, 3, \ldots, RC\}$, where t_j represents the timestamp of the j^{th} memory read and c_j is the corresponding memory content. The value of RC ("read count") represents the number of memory reads recorded in the RL during the monitor module debugging time slot.

$$RL = \{(t_j, c_j) | j \in \{1, 2, 3, \ldots, RC\}\} \tag{2.2}$$

Detector. The detector module is responsible for processing the outputs of the listeners (Eq. 2.1) and monitor (Eq. 2.2) modules. The first step of the process formalization is to link memory read accesses to the current block on the cryptocurrency network at the precise time of the memory read. To achieve this objective, the Block Read Log (BRL_c), as defined in Eq. (2.3), is introduced. This is a combination of BL_c from Eq. (2.1) and RL from Eq. (2.2), jointed by the shared parameters of time t_i and t_j, respectively. Eq. (2.3) implies that the latest block hash of cryptocurrency c at the time of reading c_j by the suspicious executable was h_i. Since BRL_c tuples are defined according to each of the tuples in RL, the BRL_c list has the same size as RL file(s), and equals RC.

$$\begin{aligned} BRL_c = \ & \{(h_i, c_j) | \forall (t_j, c_j) \in RL, \exists (t_i, h_i) \in BL_c, \\ & t_i \leqslant t_j < t_{i+1})\} \end{aligned} \tag{2.3}$$

The second step of detector process formalization is breaking down c_j in BRL_c into its individual bytes and defining the ordered data list as described in Eq. (2.4) where D_c is data for cryptocurrency c and $||$ notation is used for concatenation operation. Each pair (h_i, b_i) of D_c means that the suspicious executable reads b_i byte from memory and at a time the current block hash in the cryptocurrency network c is h_i and this list is ordered chronologically based on memory read access bytes that are preserved from the origin RL sequence. D_c is the sole input for formalizing the detector abstraction levels in the rest of this section, which was introduced in Section 2.4.4.

$$\begin{aligned} D_c = \{(h_i, b_i) | & (h_i, c_i) \in BRL_c, \\ & c_i = b_1 || b_2 || b_3 || \cdots \in \{0, 1\}^8\} \end{aligned} \tag{2.4}$$

The purpose of this section is to formalize a method for calculating the false positive rates of CryptojackingTrap using statistical mathematics. However, due to the high complexity of the algorithm, it is not possible to include all of the implementa-

tion details in the mathematical calculations. In order to address this complexity, the formalization process is simplified by respecting a certain *simplification principle*. The principle is that these simplifications must designed in a way that makes the detection criteria weaker, thereby increasing the detection rate and guaranteeing an increase in the false positive rate. If the false positive rate obtained from the simplified formalization is satisfactory, it can be safely assumed that the rate for the actual algorithm and implementation, which has a lower false positive rate, will also be satisfactory. One of the adopted simplifications is to ignore the minimum size of splits (Ξ) in the first level of abstraction or in other words, it is assumed that $\Xi = 0$. Decreasing the minimum size of splits increases the possibility of acceptance of random and incorrect data that do not originate from the malware memory access, and consequently, it increases the false positive rate and is compatible with the described simplification principle. Other assumptions are $\Pi = 100$, which means full hash coverage percentage in any hash occurrence. Assuming $\Psi = 2$ and $\Gamma = 1$, Λ=0 simplifies the question of finding two hash occurrences to notify the detection. Since this assumption makes the detection criteria weaker and increases the false positive rate, it is consistent with the defined simplification principle.

The second level of abstraction is the hash occurrence, which is denoted as HO_c in Eq. (2.5). HO_c is a sequence of hash occurrences for cryptocurrency c where each hash occurrence is represented as a Ω_c-tuple of hash-byte pairs from D_c (Line 1 and first statement of line 2). K_{i+1} is not necessarily equal to $(K_i + 1)$ (Last statement of line 2), which means the hash-byte tuples in each hash occurrence can be interrupted by other memory reads, produced by other functionality on the suspicious executable except hash calculations. However, these interruptions must occur within a Δ-sized window in the RL, which is expressed by $k_{\Omega_c} - k_1 \leqslant \Delta$ in line 3. Note that, the indexes are originated from D_c and consequently are an indicator of its byte location in the RL file(s). It is evident that Δ is greater than Ω_c because there are Ω_c tuples in a Δ-sized window. All hash values in one hash occurrence tuple are identical and equal to the concatenation of all byte values in that tuple while preserving their order (Line 4).

$$
\begin{aligned}
HO_c = \{((h_{k_1}, b_{k_1}), (h_{k_2}, b_{k_2}), \ldots, (h_{k_{\Omega_c}}, b_{k_{\Omega_c}}))| \\
\forall i, 1 \leqslant i \leqslant \Omega, (h_{k_i}, b_{k_i}) \in D_c, k_i + 1 \leqslant k_{i+1}, \\
b_{k_i} \in \{0,1\}^8, k_{\Omega_c} - k_1 \leqslant \Delta \\
h_{k_i} = h_{k_j}, h_{k_i} = b_{k_1}||b_{k_2}||b_{k_3}||\ldots||b_{k_{\Omega_c}}\}
\end{aligned}
\tag{2.5}
$$

The next abstraction level, mining occurrence (MO), is formalized in Eq. (2.6).

$$\begin{aligned} MO = \{(ho_{k_1}, \ldots, ho_{k_\Psi}) | \\ \forall i, 1 \leqslant i \leqslant \Psi, \exists c, ho_{k_i} \in HO_c, k_\Psi - k_1 \leqslant \Theta\} \end{aligned} \tag{2.6}$$

and the occurrence of CryptojackingTrap (CO) using the formalization presented in Eq. (2.7).

$$CO = \{(co_{k_1}, \ldots, co_{k_\Gamma}) | co_{k_i} \in MO, k_\Gamma - k_1 \leqslant \Lambda\} \tag{2.7}$$

The ultimate CryptojackingTrap detection is then specified in (2.8) using these formalizations.

$$|CO| > 0 \Rightarrow CryptojackingTrap\ alarm \tag{2.8}$$

Using the formalized data and processes presented in this section, the false positive rate of the proposed algorithm can be calculated. This metric can be represented mathematically as the probability of obtaining the Eq. (2.8) event within a randomly generated RL file distribution. To calculate the false positive rate, a probability question arises that must first be discussed in a general format outside of the context of CryptojackingTrap, and then applied to the CryptojackingTrap context. Specifically, the contextualized question is as follows: What is the probability of the following event in a random sample space D_c? The event is defined as "finding Ω_c members of a predefined list in their original order in a Δ-sized sub-list of D_c, as well as finding these Ω_c members in another Δ-sized sub-list in D_c, where the distance between the first item of the first subset and the last item of the second subset is less than or equal to Θ."

To address the above-mentioned probability question, the *inclusion-exclusion principle* is employed, as expressed by the Eq. (2.9). This principle computes the number of surjective functions from set A, which consists of k elements, to set B, consisting of n elements. Eq. (2.9) also provides the number of distributing k distinct objects in n places, ensuring that all k objects are present and each of the n places receives at least one object and is non-empty.

$$T(k, n) = \sum_{i=0}^{n} \binom{n}{i} (-1)^i (n-i)^k \tag{2.9}$$

Thus, the probability of having Ω_c members in origin order in Δ series of sample space is calculated as below:

$$HOC_\Delta = \binom{\Omega_c}{1} \times T(\Delta - 1, \Omega_c - 1) \tag{2.10}$$

As a consequence, the count of Mining Occurrence is calculated as

$$MOC = (RC - \Theta) \times (HOC_{\Delta})^2 \times \binom{\Theta - 1}{1}. \tag{2.11}$$

Eq.(2.5) formalizes hash occurrences, while Eq.(2.10) calculates the count of hash occurrence events in an RL file containing RC records. As a result, the probability of hash occurrence events in a random sample space RL with a size of RC can be computed using the following equation:

$$FP = \frac{MOC}{2^{RC}}. \tag{2.12}$$

Since Eq. (2.12) formulates a probability of alerting CryptojackingTrap in a random file, it represents the false positive of the CryptojackingTrap. Referring to Section 2.4.1, the previous block hash is used as the predictable field in the current implementation over Bitcoin, Monero, and Ethereum, and consequently, its variable is initiated as $\Omega_c = 32$. Other variables are initiated based on the executions tuning over the implementation as $\Delta = 48$, $\Theta = 1{,}000{,}000$, and RC = 40,000,000 by scanning the suspicious executable file for 5 minutes. By substituting these values into Eq. (2.12) the false positive rate of CryptojackingTrapis calculated equal to 10^{-20}.

2.6.2 Experimental Evaluation

While the proposed solution undergoes evaluation using mathematical methods, assessing it with datasets offers a more comprehensive benchmark. However, obtaining authentic malware datasets adhering to policies presents significant limitations and challenges. The datasets organized before 2019, as detailed in [25], have become obsolete for benchmarking following the shutdown of Coinhive in March 2019. This investigation discovered that 99% of websites ceased cryptojacking activities, while 1% executed eight distinct mining scripts, revealing 632 potential cryptojacking sites upon script tracking. Given the dataset's lack of public accessibility, an additional investigation utilizing the same approach led to the identification of 130 websites. Nevertheless, none of these instances led to any negligible change in CPU usage, suggesting the inactive existence of these eight scripts on those websites. Given our emphasis on active malware, datasets featuring inactive scripts lack validity. Consequently, this study focused on diverse non-malware realistic datasets to strengthen the experimental evaluation of the method.

The Java code used to generate the files referenced in the evaluation, along with detailed testing results, is available in this project's dataset repository. The experiments

are conducted on a laptop running Microsoft Windows 11 Pro operating system, version 10.0.22621 Build 22621, with a 64-bit based PC system type. The laptop had an 11th Generation Intel(R) Core(TM) i5-1135G7 processor with four cores and eight logical processors, running at 2.40GHz. It was equipped with 16.0 GB of installed physical memory (RAM), with a total physical memory of 15.7 GB, and 2.36 GB of available physical memory. Additionally, it had 24.2 GB of total virtual memory with 3.25 GB available virtual memory, and a page file space of 8.50 GB. The evaluation time window targeted for any single test is on average one minute of monitoring that ends with a file of two million lines and around 80 MB. The precise values for each test are available in any test data file.

Table 2.3 provides more detailed information for the following experiments (1, 2, and 4) about the split occurrences that were found in each file that ultimately failed to meet the requirements for the accepted hash occurrence. Due to lack of space in Table 2.3, the column names are shortened and here the complete meaning of each column is specified. The experiment information is organized into two main sections (randomly generated data and benign non-miner applications). It follows with the degree of randomization or application name in the "Name" column. Then the detector module's average execution time is specified. Coverage is the HCP defined in 2.4.4. The next column is the average split counts that are found for each test. It is followed by the maximum split size and average split size. The next section of columns is dedicated to specifying the reasons for discarding the split occurrences as 1) The splits that are not found on the read log file, 2) The splits with a size less than acceptable threshold size (Ξ), and 3) The valid splits that are discarded in calculating the hash occurrence because they are too far from the preceding splits and exceeds the hash occurrence window threshold (Δ). These three columns represent percentages indicating the proportion of splits that failed due to any declared reasons. Since no hash occurrence was detected in any records within this table, the mining process was skipped in all of these instances.

1) Randomly Generated Dataset Evaluation.

To calculate the false positive rate of the algorithm, testing was conducted using a random dataset of memory read logs and random 32-byte hex hash values as the monitoring data. The randomly generated monitoring file is a 2,000,000 lines file, where each line has random memory read traces, including random size and content. The test is a 100-round test, and each round uses a new random hash and the same mentioned read log file. In each round no cryptojacking activity was detected, demonstrating the zero false positive of the detector module. This negative detection is a very strong detector that will be explained more in the following. As discussed in Section 2.4.4

Table 2.3: Random generated (upper) and Benign non-miner applications (lower) test results.

	Name	Time	Valid Split Occurrences Information				Discarded Split Occurrences Error Types		
			Coverage	Splits Count	Max Split Size	Avg Split Size	MISSED	TOO_SMALL	TOO_FAR
Random Generated	100% Random	35.55 s	27.91 %	17.53	5.11	3.63	0.91 %	34.54 %	37.47 %
	85% Random	36.34 s	60 %	13	18	6.0	0 %	30.77 %	23.07 %
	50% Random	49.57 s	68.5 %	8	18	12.5	0 %	37.5 %	18.75 %
	15% Random	47.92 s	75 %	8	18	12.0	25 %	12.5 %	12.5 %
Benign Non-Miners	Adobe Acrobat DC	37.03 s	21.60 %	32.33	4.15	3.00	8.76 %	62.37 %	16.13 %
	Microsoft Calculator	31.54 s	21.93	28.50	3.78	3.00	7.02 %	57.31 %	20.37 %
	Microsoft Word	30.87 s	25.17	30.17	4.00	3.00	8.29 %	57.46 %	17.68 %
	Nodepad++	24.76 s	19.76	27.83	3.30	3.00	11.38 %	53.29 %	20.26 %
	Photos	26.02 s	20.12	30.00	3.79	3.10	12.22 %	57.22 %	17.04 %

CryptojackingTrap detector employs a multi-level abstraction to iteratively search for patterns of split, hash, mining, and CryptojackingTrap occurrences in increasing order of restrictiveness and if any of them are found, detector imposes more restrictive criteria to search the next level and the detector comes with the positive answer if the last level is found. In this experiment, in addition to the negative response for detection, there were no occurrences of mining or even hash patterns found. This underscores the effectiveness of the CryptojackingTrap detector in accurately identifying the absence of cryptojacking activity. In Table 2.3, the first row labelled "100% Random" is the result of this dataset category.

The accepted split occurrences only covered 27.91% of the hash value size, which falls significantly short of the minimum hash coverage requirement of 80%. Consequently, the generated split occurrences were not substantial enough to create a hash occurrence that met the minimum coverage requirement.

2) Benign Non-Miner Applications Evaluation.

CryptojackingTrap is evaluated in this section by testing on the benign non-miner highly used applications listed in Table 2.3. The reported numbers are the average of six different test results for each application. Likewise, randomly generated dataset, all the tests conducted in this category resulted in a strong negative alert which means a zero false positive rate.

3) Benign Miners Evaluation.

Due to the open-source nature of most crypto mining programs, attackers can modify the code base to create their variants. For instance, attack instances including CryptoSink, RubyMiner, JenkinsMiner and Graboid utilize the XMRig benign open source miner [26] [27]. As a result, the evaluation of CryptojackingTrap's true positive rate is conducted by executing it alongside the benign miners on Bitcoin, Ethereum, and Monero networks in the remaining of the current section.

Bitcoin used to be a profitable cryptocurrency for attackers in recent years that was the target of studies [14]. Bitcoin CPU mining is not beneficial anymore in recent years

and hence is not supported by many miner applications. For example, DiabloMiner [28] is not valid any longer, and BfgMiner [29] and CgMiner [30], which used to be the most famous Bitcoin miners, removed Bitcoin mining from their default configurations. This unprofitability is due to more electricity consumption per hash calculation than GPU, FPGA, or ASIC-based mining. In the scope of this study, Bitcoin mining is included in evaluations because Cryptojacking does not raise electricity consumption concerns for the attacker, as it utilizes victims' machines. Consequently, the current section analyzes Bitcoin CPU mining using CpuMiner [31], resulting in successful detection with a zero false-negative rate.

MinerGate [32] was selected as an Ethereum mining tool for the tests. This miner autonomously optimizes profit by selecting cryptocurrency types. While there is an option to disable certain currencies, the inability to exclude Monero makes it unsuitable for Ethereum testing. Additionally, WinEth [33], another Ethereum mining software, is incompatible with the evaluation system's video card. MinerGate [32], intended for Ethereum, was also utilized for Monero mining, and CryptojackingTrap successfully detected it with a zero false-negative rate.

4) Randomized Benign Miner Dataset Evaluation.

This dataset is created by randomizing an original RL file, created by monitoring MinerGate which mines Monero. Randomization evaluation is conducted by different "probabilities" that are (1, 5, 10, 15, ..., 85). For example, 20 percent randomization means all the lines of the files are replaced with a randomly generated line (random valid size and value) with 20% probability that ends up changing almost 20% lines of the origin file. Table 2.3 lists 3 records amongst 18 conducted experiments that resulted in negative alarms. The results show that CryptojackingTrap continues to give positive alarms till 5% randomization and stops detecting herein after and till 40% randomization detects HO but no MO that leads to a Negative alarm.

5) Expanded Benign Miner Dataset Evaluation.

The difference between this dataset and the preceding one is that this dataset does not create distortion in patterns and the target is to make them more sparse and spread random data between each two lines. This approach is a simulation of a miner that does not dedicate its all process to mining and decreases its hash rate for evasion purposes. The metric used in creating this dataset is how much "*times*" the data is targeted to expand. For example, using $times = 10$ expand the origin file ten times and consequently for every line of the origin file $times - 1$ random lines are added after each line. The evaluation of $times = 2, 3, 4, ..., 12$ concluded that CryptojackingTrap is resilient to detect until ten times of hash rate reduction.

2.7 Discussion

This section presents a summary of the evaluations conducted in the preceding subsections, focusing on the false positive and false negative rates of the detector algorithm. The analysis encompasses both mathematical calculations and experimental assessments, which are divided into five subsections. The mathematical subsection establishes an almost negligible false positive rate (10^{-20}). Subsequent assessments, including tests on randomly generated datasets and benign non-miner applications, consistently affirm the zero false positive rate, corroborated by mathematical results. Evaluations of benign miners also indicate a zero false negative rate among the tested miners. Moreover, the algorithm underwent rigorous evaluation in unconventional and challenging scenarios, specifically through assessments using randomized benign miner datasets and expanded benign miner datasets. The former demonstrates the robustness of the approach, displaying a resilient zero false negative rate even under conditions of up to 40% randomization of memory read content. The latter observations confirm that the algorithm maintains a zero false positive rate while accommodating up to a tenfold reduction in malware mining rates—an evasion technique used by *cryptj* malware that is not commonly addressed in many detection approaches, such as [34].

2.8 Related Work

This section provides an overview of the most pertinent and up-to-date research on cryptojacking malware detection and explains how CryptojackingTrap outperforms these methods in terms of resilience and precision.

In currently available cryptojacking detection approaches, certain assumptions are made that are not universally correct, resulting in a lack of resilience against malware evasion mechanisms and reduced precision. In the following the comprehensive classification of evasion techniques and the studies that are not resilient against them are listed. The first flawed assumption is that cryptojacking exhausts the victim's CPU resources [35]. However, studies have demonstrated that attackers use *hash rate reduction* by ten times to evade detection by the victim. For example, Gangwal and Conti introduced an innovative method utilizing smartphone magnetic sensors to profile processor magnetic field emissions across various mining algorithms, employing sophisticated machine learning techniques [34]. Their experimentation, conducted on two laptops, yielded notable results with an average precision exceeding 88% and an average F1

score of 87%. However, it is important to note a limitation of this approach: it may encounter challenges in accurately detecting hash rate reduction scenarios, and even when operating at full hash rate capacity, it does not ensure a guaranteed detection rate.

The second wrong assumption is that cryptojacking web pages consistently obtain exact keywords as malicious payload signatures [36]. Nonetheless, researchers have revealed *payload hiding mechanisms* of the attackers and presented counter examples [18]. The third CryptojackingTrap represents the first resilient solution against all these evasion mechanisms, consistently outperforming them in terms of resiliency and precision.

Konoth et al. used cryptocurrency mining features to propose MineSweeper for detecting cryptojacking [15]. They refer to the cryptojacking attacks as drive-by mining attacks. While MineSweeper uses features such as "Cryptomining Code," "CPU load as a side effect," and "mining pool communication," these features are all susceptible to bypassing by the malware. For example, the use of specific patterns in the code, such as "coinhive.min.js," is not mandatory, and the malware can establish its coin-mining service with unknown names. The CPU load can also be hidden by reducing the rate and allocating the remaining processing capacity to other malicious activities. Additionally, the mining pool communication can be concealed by using a proxy for communicating with miners and encrypting botnet communication. Therefore, MineSweeper's collected data is vulnerable to invalidation and evasion. All these diversity are discussed that can not affect the low-level activity of the miner that ends in valid detection by CryptojackingTrap.

The research in [14] proposed BotcoinTrap, which benefits from a new feature of botcoins, which is the memory access trace partial predictability feature. This method is resilient to obfuscation like the previous study [15], but in contrast with MineSwipper, BotcoinTrap can detect botcoins in terms of executable files or running processes. This approach can not be bypassed by changing general botnet features, including C&C protocol features. BotcoinTrap concerns solely malware that mines Bitcoin, which was the most used cryptocurrency at the time of research, 2018. Due to the cryptocurrency ecosystem dynamicity, this approach is not currently sufficient. CryptojackingTrap can detect cryptojacking activity on any cryptocurrency network. Another limitation of BotcoinTrap, which is covered in the current research, is the lack of precision metrics calculation for the proposed algorithm, neither formally nor with empirical proof.

The first opcode analysis attitude toward detecting crypto-miners was conducted by Carlin et al. [37]. This study analyzed opcode dynamically to detect mining activity over suspicious websites. Although opcode analysis is one of the well-known malware

detection methods, it suffers intrinsically from numerous intrinsic limitations. The first limitation is that polymorphic malware can change its code structure at runtime, which makes it difficult to detect using opcode analysis. The malware can generate different opcodes each time it executes, which makes it challenging to develop reliable signatures for detection. The second limitation is obfuscation to hide the malicious code's true purpose. This can include techniques like packing, encryption, and code obfuscation. These techniques can make the opcode analysis approach less effective since it can be challenging to identify malicious behavior from the obfuscated code. Although this approach is skippable by polymorphic malware and obfuscation, CryptojackingTrap relying solely on non-hideable characteristics of low-level memory contents can conquer all these evasions.

Compared to the above methods, this proposed method focuses on detecting the hash function execution over cryptocurrency data using low-level memory access tracing. This approach is not dependent on function signature, code features, or hard-coded data, making it resistant to obfuscation techniques. Moreover, it is immune to diversity in botnet network protocols such as pool URL, protocol type, and C&C encryption, making it an efficient detection tool. The mathematical evaluation shows that CryptojackingTrap achieves a false positive rate of 10^{-20}, which is a new record for the precise detection of cryptojacking malware.

In conclusion, several methods have been proposed to detect cryptojacking malware. While some of these methods have achieved high accuracy, they suffer from a high false positive rate or are dependent on specific features of the malware. This proposed method focuses on detecting the hash function execution over cryptocurrency data using low-level memory access tracing, making it resistant to obfuscation techniques and immune to diversity in botnet network protocols. The evaluation shows that the method achieves high accuracy and a low false positive rate, making it an efficient detection tool for cryptojacking malware.

2.9 Conclusions

In this paper, CryptojackingTrap, a novel and robust technique for detecting cryptocurrency mining activities in suspicious applications has been presented. This technique encompasses not only executable files but also operating system processes and websites. The approach focuses on the success phase of malware and tracing the low-level memory access of malware and predicts the data that miners must access. The predictable data

is obtained from the cryptocurrency network to detect mining activities. Importantly, this method does not rely on function signature, code features, or hardcoded data, rendering it resistant to obfuscation techniques. Furthermore, it demonstrates immunity to diversity in botnet network protocols such as pool URL, protocol type, and C&C encryption, making it an efficient detection tool.

The dynamic ecosystem of cryptocurrencies in terms of new cryptocurrency birth or changing cryptocurrencies inclusion in malware does not jeopardize the functionality of CryptojackingTrap. The tool currently supports the detection of profitable cryptocurrencies of Bitcoin, Ethereum, and Monero mining activities and can be easily extended to include other cryptocurrencies by adding a listener extension with no change in its core modules, making it scalable and flexible.

Evaluation results confirm that CryptojackingTrap achieves high accuracy and a low false positive rate in detecting stealthy crypto-miners. The mathematical evaluation of its false positive rate yields an exceptionally low value of 10^{-20}. Additionally, rigorous testing over randomly generated benign and malicious samples validates the effectiveness of this solution. This highlights the importance of cryptojacking detection in modern cyber-security, where attackers continuously develop new and sophisticated methods to evade detection.

Since the main advantage of this proposed method is the resilience against most evasion techniques, it inherently requires processing that is not recommended to be carried out as a life background process. This approach mainly is useful to find new families of malware, which can then be formulated as lightweight signatures in the final user's security products. Furthermore, it is suggested that CryptojackingTrap can be beneficial for cyber-security products aimed at identifying zero-day attacks.

This research work includes the implementation of 6.5K SLOC, released as open source, enabling researchers to extend upon both the approach and the benchmarks according to their ideas. For future work, it is proposed that branches of similar size on blockchains be included since CryptojackingTrap works with the largest blockchain branch.

In conclusion, CryptojackingTrap emerges as a creative and resilient detection technique that provides a reliable solution for detecting cryptocurrency mining activities. Its robustness, extendability, and flexibility make it an essential tool for contemporary cyber-security.

2.10 Exploratory Extension: Preliminary Investigation into AI-Enhanced Detection

The core of CryptojackingTrap is its detector module, which integrates information from all other modules and, through a sophisticated algorithm, identifies malicious behavior. Improving this module has the greatest potential impact on the overall system performance.

As part of the research direction on cryptojacking detection, an exploratory effort was undertaken to reduce processing overhead by using the potential of learning recurring patterns—particularly relevant given the limited diversity of cryptojacking under-beneath cryptocurrencies hash functions which results to limited behavior patterns in their memory trace log. This work was carried out as an undergraduate thesis that I co-supervised.The objective was to enhance the CryptojackingTrap detector by integrating a machine learning model capable of learning cryptojacking execution patterns while detecting, thereby reducing detection latency by avoiding full re-analysis of repeated behaviors.

This exploratory work produced a proposed learning model and a complete implementation. However, evaluation on miner datasets indicated that the current system is not yet able to reliably detect cryptojacking activity in practice. While AI-CryptojackingTrap did not improve detection performance in its current form, it opened a promising new line of investigation. This remains an open research problem requiring further development.

For completeness, the proposal, implementation, and results are summarized in Appendix B. Researchers interested in pursuing this direction are encouraged to review the full documentation and implementation available at `https://github.com/CryptojackingTrap/AI-CryptojackingTrap`.

Part II

Formal Methods for Smart Contract Security

Chapter 3

Revisiting Formal Verification in VeriSolid: An Analysis and Enhancements

This chapter corresponds to the article:

Atefeh Zareh Chahoki, Marco Roveri, Daniel Amyot, and John Mylopoulos. "Revisiting formal verification in VeriSolid: An analysis and enhancements." In *CEUR Workshop Proceedings*, vol. 3629, pp. 55–60. CEUR-WS, 2023.

3.1 Introduction and motivation

Smart contracts, evolving from cryptocurrencies pioneered by Bitcoin [8], offer decentralized, trustless execution of computable functions. They serve as a global computing platform, allowing parties to interact and transact without relying on a centralized authority or trusting each other implicitly. Ethereum [38, 39] introduced this transformative technology, thriving due to various blockchain imlementations, new languages, and business adoption [40]. However, blockchain's decentralization unveils vulnerabilities, exemplified by the DAO attack [41], underlining substantial financial risks due to the irreversibility inherent in smart contracts and emphasizing the urgent need for innovative security solutions for its specific vulnerabilities [42].

Solutions are varied [43] and include secure programming practices and developer patterns [44] to mitigate common vulnerabilities, automated vulnerability-detection tools [45, 46, 47] targeting specified properties, and formal verification approaches [48, 49], offering robust verification guarantees. The first category relies on programmer adherence, the second lacks contract-specific focus, and the third, although vital for ensuring correctness, poses automation challenges.

We have selected the Uniswap V2 Solidity code from the decentralized exchange (DEX) ecosystem as our real-world motivational example [50]. DEXs operate without intermediaries, facilitating peer-to-peer cryptocurrency transactions in a non-custodial manner. Our selection criteria included transaction count and percentage of total volume across all monitored DEXs on Etherscan.io. Uniswap V2 accounted for 82.51% of total transactions [51].

VeriSolid [52] employs a correct-by-design approach for smart contract development, emphasizing correctness during the design phase before code generation. However, our testing revealed verification errors and challenges in specifying desired properties, resulting in execution outcomes misaligned with defined properties. This paper addresses these issues, significantly improving VeriSolid's coverage from 45.6% to 90.5% across a dataset of 555 specifications, thereby enabling broader property verification.

We offer a literature review in 3.2, a concise overview of VeriSolid in Section 3.3, with an analysis of and improvements to VeriSolid's functionality in Section 3.4. Our contributions have been integrated into the VeriSolid codebase, now accessible to the community. Section 3.5 summarizes our findings and outlines potential future enhancements.

3.2 Related Work

Common Vulnerabilities and Design Patterns. Several studies have established taxonomies for common security vulnerabilities and design patterns for Ethereum smart contracts. Atzei et al. [42] identify twelve vulnerability types, while Luu et al. [46] delve into four of them, proposing mitigation techniques. VeriSolid, the correct-by-design solution analyzed and enhanced in this study, addresses two of these vulnerabilities: Reentrancy Vulnerability, exploited in the infamous DAO attack, and Transaction-Ordering Dependence. Bartoletti and Pompianu [53] identify nine common design patterns in Ethereum smart contracts, with authorization and time constraints being the most prevalent [53].

Verification and Automated Vulnerability Discovery. Researchers have devised various methods for identifying and mitigating vulnerabilities. Hirai performs formal verification [54] and defines the Ethereum Virtual Machine (EVM) instruction set in Lem [48]. Bhargavan et al. use F* to verify the safety and correctness of Ethereum contracts [45]. Luu et al. propose Ethereum semantics changes and offer the Oyente tool for vulnerability detection[46]. Fröwis and Böhme introduce an indicator for control flow immutability [55].

Smart Contract Verification Techniques. Researchers have explored multiple techniques for smart contract verification. Symbolic execution techniques have been employed [46], [47], [56],[57], [58] for detecting vulnerabilities by translating contract source code to bytecode. Tools like Securify [47], Ethainter [59] and Slither[60] fall into this category. Some tools, such as VERISMART [61], SMTCHECKER [62], Zeus [63], and Osiris [64], specialize in detecting specific vulnerabilities with high precision. Fuzzing techniques, such as ContractFuzzer [65] and sFUZZ [66], use automated generation of inputs for testing. Mutation testing [67], [68], [69] involves code mutation for analysis. There are numerous formal verification approaches, such as using Ethereum bytecode [70], EVM bytecode translation to F* [45], and formal modeling of the Bitcoin blockchain [71].

Formal Semantics and Visual Programming. Efforts have been made to provide a formal semantics for EVM bytecode and Solidity language. KEVM formalized EVM semantics into the K-framework [72], while Sereum is a runtime verification tool [73]. Microsoft offers Azure Blockchain Workbench with built-in verifier VERISOL [74]. SmartDEMAP [75] uses formal verification for contract transformation. Visual programming languages like Babbage [76], Bamboo [77], Obsidian [78], and Simplicity [79] automatically generate smart contract codes. Tokenscope [80] focuses on confor-

mance testing for tokens, while Smart contract engineering (SCE) [81] uses conformance testing for error detection. Wasm Smart contracts [82] employ conformance checking.

3.3 VeriSolid

VeriSolid is a correct-by-design tool that enables users to design contracts by modeling them as state machines. Using the Solidity language, users can specify details such as transition guards, statements, inputs, and contract definitions (e.g., internal variables). The contract models originate from a graphical editor (WebGME) where developers define smart contracts as transition systems, with states representing contract stages and transitions representing function executions. The user has access to predefined templates to specify desired properties on the contract, which are expressed in Computational Tree Logic (CTL) and associated with the model to capture safety and liveness requirements. Once verification is triggered, VeriSolid translates the designed model and specified properties into BIP [83, 84] and then into CTL specifications in nuXmv [85], which verifies whether the properties hold. After successful verification, the user can request Solidity code generation from the user interface, and the framework generates equivalent Solidity code from the verified model. This ensures that the deployed contract preserves the correctness and safety properties established during modeling. During code generation, the main logic of actions for each state and the guards for each transition must be provided by the user in Solidity via the interface. The framework then extracts this user-provided code and generates the corresponding control flow, creating the correct sequence of `if` statements and their bodies based on the transition sources, destinations, and guard conditions. In the case of negative verification results, nuXmv provides counterexamples at its level, which are then mapped back to the BIP level for developer inspection. In this paper, we assume the reader has basic knowledge of CTL [86].

3.4 VeriSolid Analysis and Extensions

While specifying our Uniswap model and verifying properties within the VeriSolid environment, a surprising incorrect verification verdict resulting from the second and third templates (Table 3.1) came to our attention. We explain this issue using a simpler model, i.e., one of the predefined smart contracts of the VeriSolid project called "*RockPaperScissors*" and depicted in Fig. 3.1.

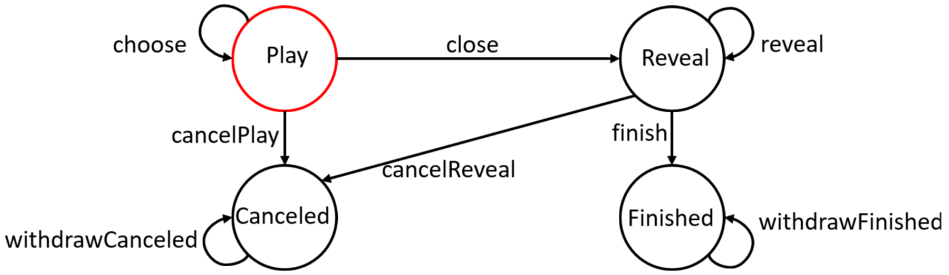


Figure 3.1: State machine of the *RockPaperScissors* VeriSolid predefined smart contract

The problematic scenario can be observed by providing "close#cancelPlay" as values to the second template definition (Table 3.1), which hence asserts that "the event (close) can happen only after the event (cancelPlay)". Surprisingly, the verification tool yields a result indicating that this property holds, despite our intuitive understanding suggesting otherwise. The same issue is also encountered with the third template, with parameters "close#cancelReveal#finish", which asserts that "if (close) happens, (cancelReveal) can happen only after (finish) happens". Once again, the verification reports that this property holds, despite the property trivially not holding.

Our investigation revealed that in the original VeriSolid tool, *weak until* ($A[xWy]$) properties were incorrectly encoded in nuXmv as *strong until* ($A[xUy]$) properties with an additional fairness constraint. Specifically, this encoding occurred for templates 2 and 3 (Table 3.1), which led to the generation of fairness constraints for the first (resp. third) parameter of the 2^{nd} (resp. 3^{rd}) template [87]. These wrong encodings are the root cause of the issues described previously.

Fairness constraints must be satisfied infinitely often within the system [88]. However, this encoding does not accurately capture the semantics of *weak until* properties. For instance, in the scenarios described earlier, it would require visiting infinitely often states such as "close" in the second template example and "finish" in the third template example, which is impossible due to the absence of a loopback mechanism. Since FAIRNESS(p) is used in nuXmv to restrict the verification to executions in which p holds infinitely often, this results in the fact that there are no paths satisfying this property; consequently any property becomes trivially true [86]. To address this issue, proper encoding of *weak until* using constructs supported by the verification engine is required. To this extent, we can leverage the equivalence $A[xWy] = \neg E[\neg yU(\neg x \wedge \neg y)]$ (see e.g., [89]), and rewrite each occurrence of *weak until* accordingly wherever needed.

We have also extended the supported verification templates of VeriSolid according to the most common CTL patterns [90]. Table 3.1 presents the VeriSolid property templates' logical descriptions and categorizations [90] between parentheses. The "CTL

Formula" column represents the encoding in CTL. "Sts." corresponds to the status:

Table 3.1: Safety and liveness property templates, including existing, fixed, implemented, and new ones

	No.	Property Template Description	CTL Formula	Sts.	%
Safety	1	**p** can never happen after **q** (Absence after)	$AG(q \rightarrow AG(\neg p))$	–	2.2
	2	**p** can happen only after **q** (Precedes globally)	$A[\neg pWq]$	fixed	4.5
	3	If **p** happens, **q** can happen only after **r** happens	$AG(p \rightarrow AXA[\neg qWr])$	fixed	–
	5	**p** can never happen (Absence globally)	$AG(\neg p)$	impl	7.4
	7	**p** can never happen before **q** (Absence before)	$A[(\neg p \vee AG(\neg q))Wq]$	impl	0.9
	8	**p** never happens between **q** and **r** (Absence between)	$AG(q \wedge \neg r \rightarrow A[(\neg p \vee AG(\neg r))Wr])$	new	3.2
	9	After **q** until **r**, **p** never happens (Absence Until)	$AG(q \wedge \neg r \rightarrow A[\neg pWr])$	new	1.6
	10	**p** always happens (Universality globally)	$AG(p)$	new	19.8
	11	After **q**, always the case that **p** happens (Universality After)	$AG(q \rightarrow AG(p))$	new	0.9
	12	Between **q** and **r**, always **p** happens (Universality between)	$AG(q \wedge \neg r \rightarrow A[(p \vee AG(\neg r))Wr])$	new	0.4
	13	**p** will eventually happen between **q** and **r** (Existence between)	$AG(q \wedge \neg r \rightarrow A[\neg rW(p \wedge \neg r)])$	new	1.4
Liveness	4	**p** will eventually happen after **q** (Response globally)	$AG(q \rightarrow AF(p))$	–	43.4
	6	**p** will eventually happen (Existence globally)	$AF(p)$	impl	2.2
	14	After **q**, **p** happens eventually (Existence after)	$A[\neg qW(q \wedge AF(p))]$	new	0.7
	15	After **q**, if **p** happens then in response **r** eventually happens (Response after)	$A[\neg qW(q \wedge AG(p \rightarrow AF(r)))]$	new	0.5
	16	If **p** happens then in response **q** eventually happens, followed by **r** eventually happens (Response chain globally)	$AG(p \rightarrow AF(q \wedge AX(AF(r))))$	new	1.4

"–" (unchanged), "fixed" (resolved in this paper), "impl" (introduced in a VeriSolid paper [52], implemented here), and "new" (introduced and implemented here). The last column reveals pattern coverage on a 555-example dataset [90]. The table encompasses all patterns with a score exceeding 1 in [90]. Our work significantly boosts VeriSolid's coverage, from 45.6% to 90.5%, enhancing its property verification capabilities. With these improvements, it is now possible to continue our analysis of the Uniswap smart contract.

3.5 Conclusion and Future Work

In this paper, we have improved the verification capabilities of VeriSolid, a correct-by-design development environment for smart contracts. Our investigations on this tool revealed incorrect encodings in nuXmv of two property templates. We corrected these issues and further contributed by implementing twelve new property templates. As a result, the verification templates now cover 90.5% of the 555 specification examples collected by Dwyer [90]. All these contributions are now available in the VeriSolid Git repository [91].

For future work, we are considering the following items. First, extend the current verification using advanced features in nuXmv by, e.g., encoding patterns in Linear Temporal Logic (LTL) [92] to then utilize advanced SAT-based model checking algorithms. Second, allow infinite state variables (reals, integers) beyond Boolean variables in guards and use SMT-based verification [93] available in nuXmv. Third, add checks to ensure the specified model is deadlock-free (each state has at least one future state). Fourth, bring back counterexamples in user level inspection. Finally, increase the granularity of the "lock" concept from FSolidM [94] (VeriSolid's predecessor) to eradicate reentrancy vulnerability in a "foolproof" manner, ensuring single transition execution at a time and preventing function nesting. Since this mechanism is overly restrictive, it is recommended to establish distinct locks for each method. Furthermore, we can offer users to define specific locks for individual transitions to ensure mutual exclusivity in concurrent transaction executions.

Part III

Scalability in Blockchain Infrastructure Using Concurrency

Chapter 4

SoK: Concurrency in Blockchain-A Systematic Literature Review and the Unveiling of a Misconception

This chapter corresponds to the article:

Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. "SoK: Concurrency in Blockchain–A Systematic Literature Review and the Unveiling of a Misconception." *arXiv preprint arXiv:2506.01885* (2025).

4.1 Introduction

Smart contracts are programs that can be stored and executed on a blockchain and that automate the execution of agreements according to predefined logic between untrusted parties [95]. Due to their tamper-proof nature and immutability of state transitions, they are ideal for enabling and facilitating trustless transactions in various domains. Deterministic execution is essential for smart contracts to ensure the predictable outcome of a transaction based solely on its inputs and the current state of the blockchain. This predictability is crucial for reliable decision-making and for avoiding unintended consequences. The concept known as the "Blockchain impossibility triangle" [96] or "Blockchain Trilemma" [97] suggests that a blockchain system cannot simultaneously achieve optimal scalability, decentralization, and security. Although this trilemma is not a formal proof within blockchain architecture, it effectively encapsulates the primary challenges blockchain systems face as observed in current implementations and industry practices. No blockchain system excels in all three areas. Traditionally, most blockchain systems have prioritized decentralization and security over scalability [98], which fundamentally limits their performance. Similarly to how different distributed systems must balance consistency and availability due to the CAP theorem, blockchain systems must also navigate trade-offs between scalability, decentralization, and security depending on the scenario. Scalability should not always be treated as the least important factor and sacrificed as a result.

One approach to addressing the scalability challenge is concurrency. This concept can be applied at various levels of the blockchain infrastructure, such as sharding, which enables transactions to be processed concurrently across different shards, or executing smart contracts within each shard and for each block by validators concurrently in a multi-core setting. Other strategies explored in this survey also leverage concurrency to enhance scalability.

However, concurrency introduces challenges related to synchronization, consistency, race conditions, deadlocks, and livelocks. These challenges can result in security vulnerabilities and unexpected behavior in smart contracts. Therefore, ensuring that these issues are addressed is critical for any concurrency solution, as discussed in this survey.

This survey aims to provide a comprehensive and systematic review of all types of concurrency issues in smart contracts. It assesses the validity of the assumptions and breaks them down into a taxonomy and a more detailed organization. In each area, the future open questions are elaborated on and presented. The study also aims to examine weaknesses, attacks, and countermeasures in the field, critically analyze assumptions,

and correct long-held misconceptions to align future research with real assumptions.

This paper presents several key contributions: (1) To the best of the authors' knowledge, it provides the first comprehensive survey of concurrency in smart contracts, offering a systematic review of a wide range of current literature to serve as an organized and insightful resource for researchers; (2) a detailed taxonomy is introduced, classifying various concurrency aspects into categories and organizing them based on their approaches and goals, whether implemented on existing blockchains or proposed for future developments; (3) open questions and emerging trends in all defined categories are highlighted, offering valuable guidance for future research; and (4) the flawed assumption of concurrent execution of transactions in Category Six of the presented taxonomy is critically examined and discussed, supported by sufficient references. All materials and code required to conduct this survey are publicly available on GitHub [1].

The remainder of this paper is organized as follows: Section 4.3 provides the main background concepts used throughout the paper. The methodology of the systematic review is presented in Section 4.4. Section 4.5 organizes the concurrency perspectives into three categories. In Section 4.6 a critical analysis reveals a misunderstanding in the third category.

4.2 Problem Statement And Motivation

Existing blockchain systems require that every node in the network maintain a complete replication of the transaction history and ledger states to ensure the total order of transactions, execution integrity, and data provenance. These data structures grow significantly over time, becoming prohibitively large. For instance, as of July 2024, the Bitcoin ledger is approximately 550GB [99], while the Ethereum ledger exceeds 18.5TB [100]. To address this, most blockchain nodes maintain a smaller and compact index known as *validation states*, which are sufficient to validate transactions. However, even these validation states can be substantial in size, such as Ethereum'cas validation states, which are around 1,120GB[100]. Additionally, blockchain nodes must locally replay all transactions based on these replicated states, incurring significant storage and computation costs that limit system scalability. This cumbersome stateful data also undermines system security and robustness by centralizing the network, as fewer nodes can handle such large data volumes.

To mitigate these issues, one approach is sharding, which partitions the blockchain

[1] https://github.com/Atefeh-Zareh/concurrency-in-smart-contracts

into multiple parallel chains, each managed by a subset of nodes. This reduces storage and computation duplications among different shards. However, sharding only alleviates the problem by a constant factor, as nodes within each shard still duplicate storage and computation. Moreover, sharding introduces new challenges, such as cross-shard transaction processing and vulnerability to attacks by slowly adaptive Byzantine adversaries. Shardening is studied in Section 4.5.2.

Another approach is the use of light nodes that store only block headers rather than full states. Although light nodes can follow consensus protocols, they cannot independently verify transaction validity, failing to address centralization due to state maintenance burdens.

The stateless blockchain concept has recently emerged as an off-chain scaling approach. This method moves ledger states and transaction executions off-chain to a subset of nodes, thereby reducing on-chain load. However, existing stateless blockchain systems are primarily designed for cryptocurrencies and face limitations when applied to general-purpose use cases involving smart contracts. Developing a general-purpose stateless blockchain that supports smart contracts presents several challenges. Transactions involving smart contracts can contain arbitrary logic, requiring novel proof techniques to ensure the integrity of off-chain executions. Smart contract transactions also introduce read-and-write sets of varying sizes, necessitating additional design for on-chain commitment updates. Furthermore, existing methods for cryptocurrency exchange offer limited support for parallel transaction executions. To enhance system throughput, new transaction processing methods are needed to validate and commit concurrent transactions from an asynchronous network, despite the stateless design. This category of off-chain solutions is studied in Section 4.5.6.

4.3 Background

This section briefly reviews the concepts related to concurrency and blockchain used in this paper.

4.3.1 Concurrency concepts

Concurrency refers to the ability of a computer program to execute multiple instructions or processes simultaneously. In distributed systems, concurrency can introduce challenges related to:

- **Consistency**: Maintaining data integrity across different replicas or copies in a distributed system.
- **Synchronization**: Ensuring that multiple processes access shared resources in a coordinated manner to avoid data inconsistencies.
- **Race Conditions**: Scenarios where the outcome of a program depends on the unpredictable timing of concurrent processes.
- **Nondeterminism**: Parallel systems commonly demonstrate this trait due to competition for communication resources. A system shows nondeterminism when two identical copies of it can act differently even with the same inputs upon which wins the race that results in an untestable trait of these systems. [101]
- **Deadlock**: is one of the most common issues in a parallel system, when no component can progress due to mutual waiting for communication, a concurrent system is deadlocked. A classic example is the 'five dining philosophers', where each philosopher requires both neighboring forks to eat, leading to a deadlock if all philosophers simultaneously pick up their left-hand fork and they deadlock and starve to death. One of the main reasons for deadlock is competition to acquire the resources [101].
- **Livelock**: The cause of livelock, is called *divergence* which means a program performs infinite unbroken sequences of internal actions and never interacts with its environment, such as infinite loops in programs. Although operationally and theoretically deadlock and livelock are different, they appear similar from the users' perspective because no response is received. However, livelock is worse since users may observe internal activities and mistakenly expect an eventual output [101].

4.3.2 Smart contracts

The *blockchain* is a foundational design pattern to facilitate pure peer-to-peer distributed computation. Initially implemented by Bitcoin in 2009 as a cryptocurrency [8], it has since evolved into a robust infrastructure capable of executing a wide range of generalized functionalities beyond cryptocurrency. Ethereum [9], in particular, pioneered this expansion in 2013 by introducing the *Ethereum Virtual Machine (EVM)*, a Turing-complete state machine that handles both deployment and execution of codified arbitrary business logic scripts named *smart contracts*. All blockchain networks

maintain securely shared data named *ledger* through a *consensus protocol* facilitated by volunteer operators named *miners* and *validators*. The transactions announced by users are initially stored in the *mempool* (short for "memory pool"), a temporary storage area where pending transactions wait to be included in the next block. Miners select transactions from the mempool that offer higher fees first, as this maximizes their rewards and persists them into the next block if the transaction is valid. This validity check includes the correctness of transaction format, having a sufficient gas fee, nonce verification, signature verification, balance check, double-spending prevention, compliance with consensus rules, and regarding smart contracts ensuring that the transaction can execute without error. Then miners extend the blockchain by one block in each *block interval*, which varies across different networks such as 10 minutes in Bitcoin or 12 seconds in Ethereum. Smart contracts introduce the concept of *gas*, irrespective of the consensus protocol they utilize. Gas represents the computational effort and cost required to execute operations within smart contracts to compensate for the resources used to execute smart contracts. It prevents infinite loops, restricts resource consumption, and maintains the network's efficiency and security. Smart contracts are written in various programming languages tailored to specific blockchain platforms. Solidity is the predominant language for Ethereum; Vyper aims to provide simplicity and security; Michelson is used on Tezos; and Move was developed for Diem (formerly known as Libra).

4.3.3 Proof of Work (PoW) & Bitcoin

Proof of Work (PoW) is a consensus protocol that is mainly introduced in Bitcoin. Participants, known as *miners*, gather newly broadcasted and not yet persisted transactions and add them in a *block* data structure if they are *valid*, then compete to solve a cryptographic puzzle, named *mining*, to add the block to the ledger. One part of the block is its *previous block hash*, which makes this data structure *tamper-proof*. The first miner to solve the puzzle earns the right to add the next block of transactions to the blockchain and win rewards. Rewards are in the form of the newly *minted* cryptocurrencies in that block and the *fee* and gas of the included transactions as the incentive. PoW is the original consensus mechanism used by Bitcoin [8] and several other cryptocurrencies, but because it is energy-intensive, it is being replaced by others such as Ethereum.

4.3.4 Proof of Stake (PoS) & Ethereum Mechanism

This section provides an overview of Proof of Stake (PoS), one of the most widely adopted consensus algorithms in blockchain infrastructures, including Ethereum. This section also briefly explores Ethereum's execution mechanism, highlighting its key components and processes.

In PoS, a node is selected as the proposer using methods like computational contests, predetermined sequences, or random algorithms, as seen in Ethereum. The proposer compiles a block by selecting a set of transactions and shares it with other nodes, known as attestors. Each attestor independently validates the transactions in the block, rejecting it if any discrepancies or invalid transactions are found.

Ethereum transitioned to PoS with its Ethereum 2.0 upgrade in 2022, aiming to enhance security, reduce energy consumption, and support new scaling solutions compared to the prior Proof of Work (PoW) architecture. In PoS, participants, referred to as *validators* or *nodes*, must deposit 32 ETH into a deposit contract, thereby *staking* it as collateral [102]. Validators are responsible for ensuring the validity of new blocks and occasionally for proposing and propagating them. Misbehavior, such as proposing multiple conflicting blocks or sending invalid attestations, results in penalties, including partial or total loss of staked ETH. To join as a validator, a user must pass through an activation queue, which limits the rate of new validators joining the network. This mechanism ensures system stability, efficient consensus, and protection against resource strain and Sybil attacks, where a single entity creates multiple identities to gain disproportionate influence over the network.

In Ethereum's PoS system, block timing follows a fixed schedule rather than being influenced by mining difficulty as in PoW. Time is divided into *slots* of 12 seconds and *epochs* comprising 32 slots, corresponding to 6.4 minutes. During each slot, a validator is pseudo-randomly selected using RANDAO to propose a new block and broadcast it to the network. Simultaneously, a randomly chosen committee of validators evaluates the proposed block's validity. These committees distribute the workload, ensuring that all active validators participate within each epoch without needing to act in every slot [102].

Transaction Process

The process of executing a transaction in Ethereum PoS involves several steps. A user creates and signs a transaction with their private key via a wallet or library, essentially making a request to a node through the `Ethereum JSON-RPC API`. The transaction is

submitted to the *execution client* part of that node, which verifies its validity, ensuring that the sender has sufficient ETH and has signed the transaction correctly. Valid transactions are added to the *local mempool* (a list of pending transactions) and broadcast over the execution layer gossip network. Advanced users may bypass public broadcasting and forward their transactions to specialized block builders, such as Flashbots Auction, to maximize profits through *Maximal Extractable Value (MEV)* [102].

The selected validator for the current slot is responsible for updating the global state. Nodes run three key software components: the *execution client*, the *consensus client*, and the *validator client*. The execution client collects transactions from the local mempool, executes them locally to generate state changes, and compiles them into an *execution payload*. This payload is passed to the consensus client, which incorporates it into a *beacon block*. The beacon block also contains metadata such as rewards, penalties, slashings, and attestations, enabling consensus on the blockchain's state and head of the chain [102].

Other nodes receive the new beacon block via the consensus layer gossip network and pass it to their execution clients for local validation of the proposed state changes. The validator client attests to the block's validity and confirms that it builds upon the chain with the greatest attestation weight, as determined by the fork choice rules. Once validated, the block is added to the local database of each node.

Transaction Finality

A transaction achieves *finality* when it is part of a block that cannot be reverted without burning a significant amount of staked ETH. Finality is managed through *checkpoint* blocks, which occur at the start of each epoch. Validators vote on checkpoint pairs, and if a pair receives votes from at least two-thirds of the total staked ETH, the target checkpoint becomes *justified*, and the source checkpoint is upgraded to *finalized*. Reverting a finalized block would require an attacker to control and sacrifice at least one-third of the total ETH staked across all participants in the network. To prevent an attacker from stalling finality, Ethereum employs an *inactivity leak*, which penalizes validators voting against the majority when finality fails for more than four epochs. This ensures the majority regains a two-thirds stake to finalize the chain.

Ethereum's Layered Architecture

Ethereum's architecture is organized into several layers, each responsible for specific functions that contribute to the overall operation and scalability of the network. The

execution layer handles transaction processing, smart contract execution, and state changes. This layer is responsible for bundling transactions into *execution blocks* and updating the global state. Above the execution layer is the **consensus layer**, also known as the *Beacon Chain*, which is responsible for maintaining network consensus, coordinating validators, and ensuring security. The *beacon block* is the key data structure in this layer, as it organizes validator attestations and records the consensus state of the network. Ethereum also integrates a **data availability layer**, which ensures the reliable retrieval of off-chain data, particularly for layer 2 solutions. Lastly, the network utilizes **layer 2 scaling solutions**, such as **Optimistic Rollups** and **zkRollups**, which offload transaction processing to achieve greater throughput while maintaining security through periodic anchors to the mainnet. This modular, layered architecture enhances Ethereum's scalability and security while enabling the development of decentralized applications and solutions.

Ethereum Networks and Their Purposes

Ethereum operates across several networks, each tailored for specific purposes within its ecosystem. The **mainnet** is the primary public network where real transactions, decentralized applications (dApps), and smart contracts are executed, using ETH with actual monetary value. For development and testing, Ethereum offers **testnets** such as **Goerli** and **Sepolia**, which simulate mainnet conditions without risking real assets, with Goerli widely adopted for advanced testing and Sepolia preferred for lightweight development. **Private networks** are custom deployments used by organizations or developers for controlled experimentation and application testing. To address scalability, Ethereum integrates **layer 2 networks** like **Optimism** and **Arbitrum** (Optimistic Rollups) and **zkSync** and **StarkNet** (zkRollups), which reduce transaction costs and increase throughput by offloading computation from the mainnet while retaining security guarantees. Additionally, emerging **data availability solutions** like **EigenLayer** ensure that off-chain data for rollups can be reliably retrieved, further enhancing Ethereum's scalability and modularity.

Terminology and Roles in Ethereum PoS

In Ethereum's PoS architecture, the terms *validator* and *node* are used interchangeably. Each node performs multiple roles: proposing blocks when pseudo-randomly selected for a slot, attesting to other validators' blocks within an epoch as part of a committee, and validating transactions in each slot. These roles are not distinct among nodes but

represent different responsibilities within each node.

However, various terminologies in the literature might appear to suggest different roles. For instance, some references distinguish *miners* and *validators*, where miners propose new blocks and validators verify them [103]. Others, such as [104], use terms like *proposer* and *attestor* within a proposal-attestation paradigm. In this context, proposers select transactions and broadcast blocks, while attestors validate the blocks. In Ethereum, these distinctions are purely functional, as all nodes perform these tasks at different times.

In this survey, we adhere to the Ethereum convention, referring to participants as *validators*. For consistency and clarity, references to other terminology will include appropriate explanations, ensuring that readers are not misled by differing conventions.

4.3.5 Hyperledger Fabric

Nodes in Hyperledger Fabric

In Hyperledger Fabric, the nodes serve as communication entities within the blockchain network. Unlike many blockchains where nodes operate in a peer-to-peer manner, nodes in Hyperledger Fabric have distinct roles. There are three types of nodes:

- **Client**: The Client is operated by the end user and is responsible for submitting transaction proposals to the Endorser Peer and broadcasting the proposals and responses to the Orderer.
- **Peer**: Peers are primarily tasked with executing chaincode to read from and write to the ledger. All Peers act as committing peers (Committers), maintaining the ledger's state. Additionally, peers can serve as endorsing peers (endorsers) when an application makes a transaction endorsement request. This endorsement role is dynamic; otherwise, Peers function as regular committing peers.
- **Orderer**: The Orderer nodes collectively form the ordering service. In Hyperledger Fabric, which is a distributed system, each node maintains a copy of the ledger. The ordering service ensures the consistency of the ledger by ordering transactions into blocks.

4.3.6 Other Consensus Protocols

There are two categories of consensus protocols: Proof-based and Leader-based [98].

Byzantine fault tolerance (BFT) is a critical property in distributed system design. It refers to the system's ability to maintain its correct operation even when some components fail or behave maliciously. In blockchain networks, Consensus protocols are mechanisms used by distributed systems to achieve agreement on a single data value among multiple nodes or participants, even in the presence of faulty or malicious actors. This property is essential to maintain the integrity and security of the distributed ledger. Various consensus algorithms have been developed to achieve Byzantine fault tolerance in blockchain systems, each with its own trade-offs in terms of efficiency, scalability, and security. The following reviews some of the main types of consensus protocols, excluding PoW and PoS, which are discussed later [105]:

- **Delegated Proof of Stake (DPoS):** DPoS is a variation of PoS where token holders vote for a select few delegates who are responsible for validating transactions and adding them to the blockchain. EOS and TRON are examples of blockchain platforms that use DPoS.

- **Proof of Authority (PoA):** In PoA, consensus is achieved by a set of approved validators, typically known entities or organizations. Validators are responsible for creating new blocks and maintaining the blockchain. PoA is used in networks where a high degree of trust among participants is assumed, such as private or consortium blockchains.

- **Practical Byzantine Fault Tolerance (PBFT):** PBFT is a consensus algorithm designed to work in asynchronous networks with a certain number of faulty or malicious nodes. It ensures that all honest nodes reach a consensus on the order of transactions. PBFT is used in permissioned blockchains and distributed systems like Hyperledger Fabric.

- **Raft:** Raft is a consensus algorithm designed for fault-tolerant distributed systems. It elects a leader node responsible for managing the replication of the log among other nodes. Raft is often used in distributed databases and systems where simplicity and ease of understanding are prioritized.

- **Directed Acyclic Graph (DAG):** DAG-based consensus protocols, such as Tangle (used by IOTA) and Hashgraph, do not rely on blocks or miners. Instead, transactions are directly linked to each other in a graph-like structure, and consensus is achieved through a voting process based on transaction approval.

- **Proof of Vote (PoV):** Proof of Vote (POV) [106] represents a consensus algorithm designed specifically for consortium blockchains where PoW falls short. In this model, consensus is orchestrated by distributed nodes overseen by consortium partners, reaching decentralized decisions through voting-based arbitration. The primary concept involves assigning distinct security identities to network participants, enabling block submission and verification via voting among these entities, eliminating the need for third-party intermediaries or reliance on unregulated public awareness. In contrast to the fully decentralized PoW consensus, POV offers controllable security, convergence, and reliability, achieves transaction finality with a single block confirmation, and ensures swift transaction verification with minimal delay.

4.3.7 Performance in smart contracts

Table 4.1 shows different cryptocurrencies in the first column, the transaction speed as TPS in the second column, which stands for "transactions per second" and the last column shows the "average transaction confirmation time" of these cryptocurrencies [107]. The transaction verification process for cryptocurrencies is very slow and does not match the performance of traditional payment systems such as VISA, which handles an average of 10,547 TPS and can peak at 56,000 TPS and Paypal with a TPS of 1,700. TPS for Bitcoin is almost 7 and other TPSs for cryptocurrencies are listed in the second column on this paper. The calculation of the TPS for the cryptocurrencies in the last year is available in this paper git path.

Numerous approaches have been proposed to increase transaction speed and scalability by enhancing concurrency efficiency in blockchain technology. In Section 4.5, we will review these approaches, focusing on how concurrency is utilized at different levels of the blockchain to achieve improved performance.

Table 4.1: transaction speed of different cryptocurrencies

Cryptocurrency	TPS	Time
Bitcoin	$3-7$	60 min
Ethereum	$15-25$	6 min
Ripple	1500	4 sec
Bitcoin Cash	61	60 min
Stellar	1000	$2-5$ sec
Litecoin	56	30 min
Monero	4	30 min
IOTA	1500	2 min
Dash	48	$2-10$ min

4.4 Methodology

This section outlines the questions to be addressed, along with the source dataset and the query used to extract the relevant literature for this systematic review. Subsequently, it presents the exclusion criteria.

4.4.1 Research questions

This study aimed to address several crucial research inquiries:

(RQ1): What are the distinct concurrency aspects associated with smart contracts and their blockchain infrastructure?
- This question focuses on gaining insights into all types of concurrency aspects within the blockchain ecosystem and organizing them into categories.

(RQ2): What are the existing security threats and countermeasures available in the domain of blockchain concurrency, and how do they address the identified concerns?
- This investigation aims to identify the techniques currently utilized to address concurrency challenges in smart contracts in a detailed representation. It categorizes the literature based on whether the aspect already exists in the blockchain infrastructures or is proposed for further iterations. In addition, it examines whether the study's focus is on introducing vulnerabilities, attacks, or security solutions, and analyzes the approaches taken to address these issues.

(RQ3): What are the potential research gaps of current solutions that require attention?
- The purpose of this inquiry is to organize the open questions and trends in all these categories. These identified threats can serve as guidelines for future research in this field, directing attention to areas that require further investigation due to the absence of proposed solutions or the limitations of existing approaches.

4.4.2 Source databases and search query

This study used Scopus[2] as the primary source to identify relevant publications in this field. Scopus provides a vast repository of scientific literature with over 27,950 active titles, 90.6+ million records, and from more than 7,000 publishers. Scopus is continuously expanding its coverage of conference material primarily for the subject domains of publishers and societies of engineering and computer sciences [108]. In addition, Scopus offers a powerful search engine specifically designed for academic exploration. Although other databases such as Google Scholar, Web of Science, and IEEE Xplore exist, the extensive coverage and robust search capabilities of Scopus make it a sufficient choice for this particular research.

The search query utilized for all repositories can be found in Listing 4.1:

```
1 TITLE-ABS-KEY (
2   ( "smart contract" OR "Ethereum" OR "solidity" OR "Hyperledger" )
3   AND ( "concurren*" OR "race" OR "parallel" OR
4         "multithreaded" OR "synchroni*" OR "transaction ordering dependancy" OR "front
              running attack" OR shardening)
5   AND ( LANGUAGE( "English" ) )
6   AND NOT( DOCTYPE ( "cr" ) OR  DOCTYPE ( "ab" ) OR DOCTYPE ( "bk" ) OR DOCTYPE ( "ch" )
       OR DOCTYPE ( "cr" ) OR DOCTYPE ( "re" ) OR DOCTYPE ( "sh" ) )
```

Listing 4.1: Search query.

In line 1, TITLE-ABS-KEY indicates that the search query is applied to all the title, abstract, and keyword sections of the repositories. The primary objective of the search is to identify research works that specifically address concurrency solutions 4) in blockchain infrastructures (line 2). The query utilized the wildcard * in 'concurren*' and 'synchroni*' to encompass all variations within the word families of concurrency and synchronization. The query targeted all papers in the English language (line 5) and excluded document types of conference review (cr), abstract report (ar), book (bk),

[2] https://www.scopus.com/

book chapter (ch), conference review (cr), review (re) and short survey (sh), respectively, encoded on line 6. The described query execution in May 2024 returned 237 unique papers before the pruning process.

4.4.3 Inclusion and exclusion criteria

Table 4.2 lists the inclusion and exclusion criteria that are applied in this current study.

Table 4.2: Inclusion and Exclusion Criteria

Inclusion Criteria	**Exclusion Criteria**
• Research studying any concurrency aspects in smart contracts. • Studies that address at least one of the designated research inquiries. • Relevant studies identified through snowballing techniques from previously chosen literature.	• Theses, books, survey-based studies, or patents. • Non-peer-reviewed research. • Duplicate studies. • Studies lacking relevance to the designated research questions.

4.4.4 Screening Results

Figure 4.1 illustrates the percentage of relevant articles discovered during the screening process. Additionally, a sub-pie chart categorizes the non-relevant papers into different domains, highlighting the areas that were identified and filtered out from the main set of papers under consideration.

Figure 4.2 depicts the total number of citations for the relevant articles, categorized by their year of publication.

Figure 4.3 shows the number of relevant articles published each year.

4.5 Concurrency in smart contracts: a taxonomy

Three main categories of concurrency approaches for smart contracts have emerged that are represented in this section.

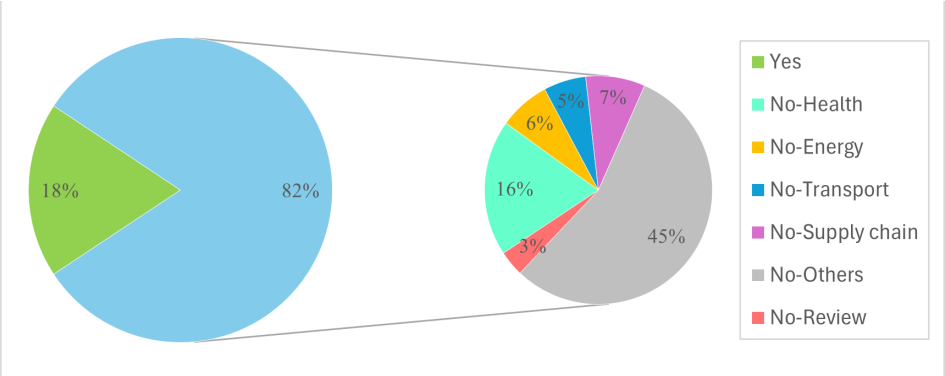


Figure 4.1: Screening results

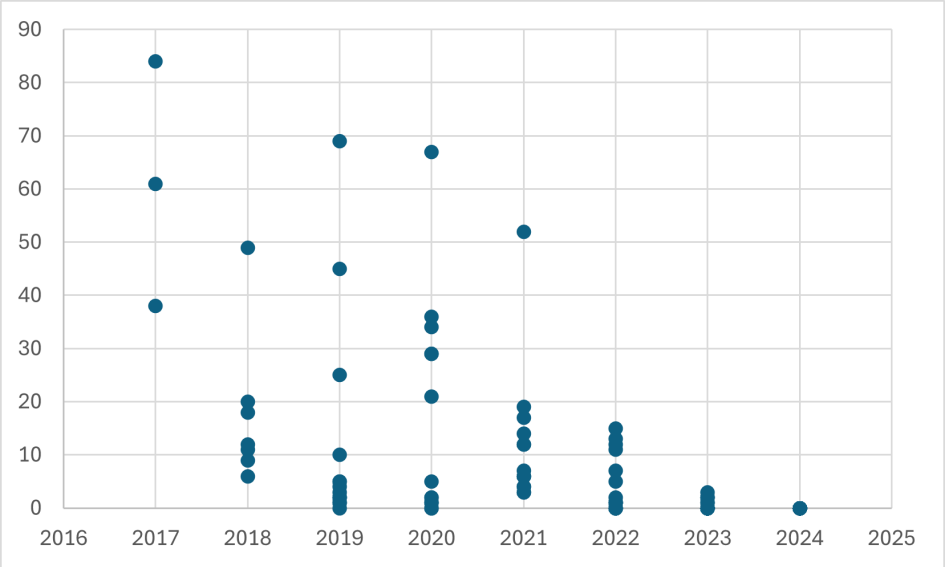


Figure 4.2: Paper citations per year

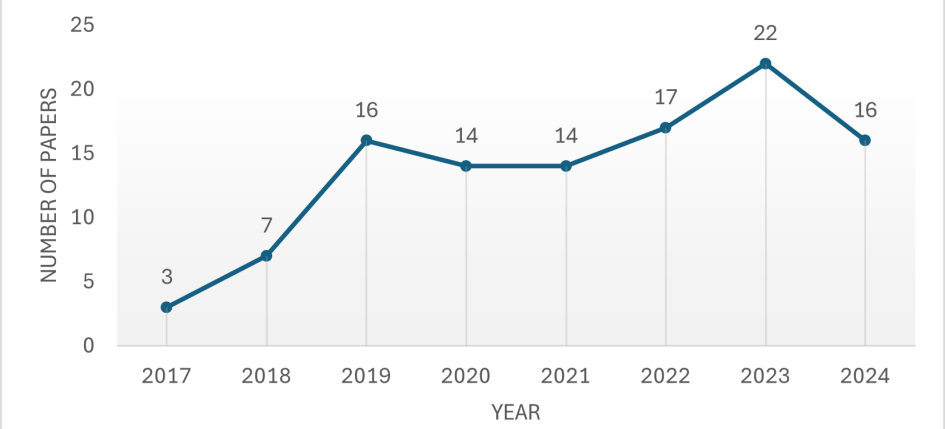


Figure 4.3: Paper number per year

4.5.1 Category 1: Concurrent Consensus

This category of solutions focuses on consensus protocols and believes that to scale blockchain systems, the fundamental structure of consensus needs modification. They propose novel consensus protocols to enhance concurrency in blockchain infrastructure aiming to improve performance.

One notable research in this category is done by Hazari and Mahmoud [109], who introduced *parallel PoW* based on parallel mining rather than solo mining. This approach eliminates simultaneous efforts of miners to solve a specific block by introducing a new role as a *manager*, along with processes for manager selection, work distribution, and a reward system. The introduction of a manager role does not compromise the decentralized nature of blockchain. This solution was implemented in a test environment mimicking Bitcoin's PoW features and tested across various scenarios with different difficulty levels and numbers of validators. The results indicate a 34% improvement in PoW scalability. This protocol aims to increase transaction processing speed, potentially reducing energy consumption. This research is extended by these authors in [107] which evaluated a cloud-based environment. The future plans for this research include evaluation against the 51% attack, evaluation on the Bitcoin testnet, refinement of the reward system, and evaluation of real-time network energy consumption, although it is expected to end reduced energy consumption due to speeding up transaction processing.

[110] introduces an approach to execute smart contracts in parallel and asynchronously. Unlike traditional methods, this approach minimizes coordination efforts and eliminates risks such as livelocks, even with small groups. This paradigm is implemented in two ways: first, enhancing Ethereum to support parallel and asynchronous execution seamlessly, without the need for hardforks. Second, introducing SaberLedger, a new public and permissionless blockchain, demonstrating its capabilities through a prototype implementation.

In [111], the authors introduce BDLedger (Big Data Ledger), a blockchain-like distributed ledger designed to achieve a nearly linear throughput scaling. BDLedger features a unique consensus mechanism called *Random Witness Consensus* and uses a DAG-based block structure. To support this scaling, the system makes two key tradeoffs: 1) it does not establish consensus on transaction order but focuses only on transaction content, which ensures that transactions do not conflict and can be processed in parallel; 2) it limits the nodes that store block content to a fixed number of random nodes, which helps control storage and bandwidth costs. The study demonstrates that BDLedger can scale throughput from approximately 20,000 TPS with 10 nodes to about

160,000 TPS with 100 nodes.

[112] focuses on improving PBFT, a consensus-based algorithm used in the Tendermint blockchain [113]. The paper introduces a lock-free algorithm for the blockchain, enhancing concurrency by allowing the proposal and voting phases to occur concurrently while keeping the commit phase sequential. This modification aims to mitigate the negative impacts of locks, thereby enabling greater parallelism within the blockchain system.

[114] focuses on concurrency in the consensus algorithm level to avoid a throughput decrease by node numbers. They have proposed a more sufficient consensus algorithm based on PoV (read more in 4.3.6) named *Parallel Proof of Vote (PPoV)*. In this protocol, concurrency is integrated into the consensus cycle, enabling numerous nodes to generate blocks simultaneously. Its research findings and experimental data indicate that PPoV outperforms traditional BFT consensus by a factor of 2-5, particularly when the number of nodes ranges between 4 and 100.

4.5.2 Category 2: Sharding

Sharding, originally a database technique for splitting large datasets into smaller parts across multiple storage systems, improves efficiency by reducing server load. In blockchain, sharding was first introduced in 2016 [115] as a method to enhance scalability and transaction throughput. The protocol *ELASTICO*, proposed for permissionless blockchains, scales transaction rates almost linearly with available computational power. It efficiently uses network messaging and tolerates Byzantine adversaries up to one-fourth of the computational power. ELASTICO achieves secure partitioning of the mining network into smaller committees, each handling a distinct set of transactions or *shards*. It was the first secure sharding protocol designed to operate in the presence of Byzantine adversaries.

Since then, sharding in blockchain has become one of the most advanced and specialized areas in the context of concurrent executions on blockchain networks. Sharding, in particular, has been extensively examined in several key surveys, and readers are encouraged to consult these sources for in-depth analysis. This survey aims to address critical gaps in the concurrency domain, which remains underexplored. For a more detailed overview of recent developments in the specific area of sharding, see [116], [117], [118], [119], [120], [121], [122].

4.5.3 Category 3: Intra-block Concurrent Processing

This approach acknowledges the sequential execution of transactions within a block by miners or validators in modern cryptocurrency systems like Ethereum and aims to address the limitations of serial execution of both inter and intra-Smart Contracts, which result in inefficiencies in system throughput and resource utilization.

Speculative

To overcome this challenge, [103] presented a novel approach to enable the parallel execution of smart contracts by miners and validators. This approach utilizes established concepts from Herlihy et al.'s distributed computing approaches *software transactional memory (STM)* [123] and *transactional boosting* [124]. The speculative approach has two strategies. In the first strategy, "execute then order," the leader speculatively executes smart contracts (SCs) in any arbitrary order, generates a corresponding schedule, and then sends this schedule to the validators, who replay the transactions and vote on the execution outcome. In the second strategy, "order then execute," the leader pre-determines the execution order of the SCs. Validators then speculatively execute the SCs in parallel, but they ensure that the final outcome reflects the predetermined order.

The first one is "order then execute," in which the leader chooses an order in which the SCs should appear to execute. The validators speculatively execute the SCs but ensure that the result seems to be executed in that order. The second strategy is "execute then order," in which the leader executes the SCs speculatively in an arbitrary order, generates a schedule, and sends that schedule to the validators who replay and vote on that execution. These strategies are discussed in more detail in the following.

Order then execute. Miners execute smart contracts *speculatively* in parallel, allowing non-conflicting contracts to proceed concurrently and generating a serializable concurrent *schedule* for a block's transactions. Miners encode this schedule as a deterministic *fork-join program*, facilitating parallel re-execution by validators. Despite its efficacy in increasing transaction throughput, this approach is based on the sequential nature of transaction execution within a block and does not introduce any uncertainty in transaction executions. Experimental results demonstrate significant speedups for both miners and validators, with a 1.33x speedup for miners and a 1.69x speedup for validators achieved with just three concurrent threads. Additionally, the paper presents a prototype implementation and discusses the potential for future advancements in multithreading support and programming language design to further improve

throughput and concurrency control in smart contract execution. Furthermore, the paper highlights the applicability of the proposed mechanisms in permissioned blockchain systems and outlines future research directions, including support for multithreading in the Ethereum virtual machine and advancements in programming language support for smart contracts to maximize throughput while avoiding concurrency pitfalls [103].

This approach has been experimentally evaluated in [125] to represent the potential benefits of speculative techniques for the execution of Ethereum smart contracts in parallel, using historical data to estimate their effectiveness. Transaction traces from sampled blocks on the Ethereum blockchain are replayed over time using a simple speculative execution engine, with miners attempting to execute all transactions in parallel and rolling back those causing data conflicts. Validators follow the same schedule as miners. The findings reveal that even simple speculative strategies yield notable speed-ups, with estimated speed-ups starting at approximately 8-fold in 2016 and declining to about 2-fold by the end of 2017, as transaction traffic increased. Moreover, the study identifies that a small set of contracts is responsible for a significant portion of data conflicts resulting from speculative concurrent execution, named *storage hot-spots*. Several observations emerge from the study, including the importance of distinguishing between reads and writes to reduce conflict rates, limited additional benefits from more aggressive speculative strategies, and the potential for accurate static conflict analysis to yield modest benefits. Furthermore, increasing the number of cores in the simulated virtual machine improved speed-ups, but with little improvement beyond 64 cores. The study suggests directions for future research, including incentivizing contracts that produce fewer data conflicts and extending the virtual machine to support common commutative operations. In conclusion, while simple speculative strategies can produce significant speed-ups, these diminish as transaction rates and conflict rates increase. The study underscores the need to focus on reducing conflict rates to further increase parallelism in Ethereum-style smart contract execution, potentially through improved recognition of commuting operations at the semantic level and investigating the effects of intrinsic data types on contention.

Execute then order. [126] addresses the challenge of integrating black-box concurrent data structures into existing STM frameworks, focusing on optimistic transactional usage. It introduces conflict abstractions to define data-structure conflicts in terms of commutativity separately from object implementation, enabling efficient cooperation with generic STM runtimes. Additionally, shadow speculation allows individual transactions to make private speculative updates to highly concurrent black-box objects, facilitating opaque speculative updates that can be later dropped or atomically applied.

These concepts are realized in a new transactional system called ScalaProust, built on ScalaSTM, supporting objects of arbitrary abstract type and integrating with underlying STM conflict-detection mechanisms. The paper's contributions include conflict abstractions, shadow speculation, ScalaProust system implementation, and experimental validation demonstrating competitive scalability with existing specialized approaches. However, the described mechanisms currently do not support mixtures between transactional objects and STM-managed read/write operations. Future directions include integrating pessimistic and optimistic treatment of black-box ADTs with standard STM memory operations, improving performance, and utilizing conflict abstractions for STM support of "pure writes" and automatic verification techniques for synthesis.

Deterministic

[104] presents a two-stage approach for efficiently executing Blockchain transactions in parallel by adopting deterministic concurrency control. The paper addresses the inefficiencies of generating dependency graphs, which are often time-consuming and require re-executing transactions multiple times. Instead of building a large dependency graph, the authors partition transactions into *batches* to avoid inter-batch conflicts, thus reducing the complexity of computing partial orders. The proposed *deterministic concurrency control (DVC)* method enables high parallelism by solving an equivalent MinVWC problem (minimum vertex weighted coloring) [127], which helps in finding an approximate optimal partial order without the need for re-execution of transactions. The authors demonstrate the effectiveness of their approach through experimental results, which show a significant reduction in the costs of computing the partial order and achieving a high degree of parallelism compared to existing solutions. This work offers a promising solution to improve the efficiency of concurrent Blockchain transaction execution, especially in the context of modern hardware utilization.

Partitioning

This approach proposes a scalable smart contract execution scheme for blockchain-based systems, aiming to overcome the limitations of serial processing and improve system throughput. Using the decentralized nature of blockchain technology, the scheme enables the parallel execution of multiple smart contracts, thus enhancing the system's capability to handle a large volume of transactions. The study carried out in [128] approach employs a divide-and-conquer strategy, using a fair contract partition algorithm based on *integer linear programming (ILP)* to partition smart contracts into subsets.

Table 4.3: Speed Up Comparisons.

Reference	Latest Year Test	Availability	Evaluation Execution Environment	Cores	Speed Up Rate	
					Proposer	Attestor
Conthereum [3], 2025	2025	Public[1]	Intel Core i5-1135G7 (2.40GHz), eight logical processors, 16.0 GB RAM, Windows 11 Pro	3	2.93	2.48
				4	3.73	2.77
				5	4.31	3.23
				32 (0%conflict)	25.92	25.92
				32 (15%conflict)	9.08	5.58
[103], 2020	N/A	Not available	4-core Intel Xeon W3550 (3.07 GHz), 12 GB RAM, Ubuntu 16	3 + 1 for GC	1.33	1.69
[125], 2019	2017	Not available	N/A	16	1.13	N/A
				64	2.26	N/A
Block-STM [129], 2023	2023	public[2]	AmazonWeb Services c5a.16xlarge instance, (AMD EPYC CPU and 128GB memory) Ubuntu 18.04	32 (low-contention)	20	-
				32 (high-contention)	9	-
ScalaProust [126], 2019	-	Public[3]	Amazon EC2 m4.10xlarge instance, which has 40 vCPUs and 160 GB of RAM	-	-	-
[104], 2023	N/A	Private	4-core processor of 2.2 MHz, 4 GB main memory	4	3	N/A
				20 to 24	5	N/A

[1] `https://github.com/Conthereum`,
[2] `https://github.com/danielxiangzl/Block-STM`,
[3] `https://github.com/ScalaProust/ScalaProust`

Each subset is then randomly assigned to a group of users, named sub-committee. Participants within each sub-committee exclusively handle a portion of smart contracts, enabling the simultaneous execution of various smart contracts. The contributions of this work include the development of a scalable execution scheme, theoretical analyses demonstrating its correctness and security, and simulations showcasing its practicality. The study evaluates the efficiency of the proposed scheme using data from existing smart contract systems, demonstrating its scalability and improved efficiency compared to sequential processing. However, a potential limitation lies in the efficiency of the ILP solver, which may pose challenges for larger sets of smart contracts. Future research directions include exploring more efficient and scalable partitioning techniques without relying solely on ILP solvers. This paper focuses on PoW and is grounded in data collected from Ethereum. However, it is important to acknowledge that Ethereum transitioned from PoW to PoS during "the merge" in September 2022. Therefore, this study conducted in 2017 requires re-evaluation or revision to align with the changes resulting from Ethereum's transition to PoS.

Intra-block Performance Comparison

Table 4.3 presents a comparative analysis of the peak performance of various concurrency approaches at the intra-block level in blockchain, sorted in descending order based on the year of presentation. The `Reference` column specifies the cited work along with the year of publication. `Latest Year Test` denotes the most recent year in which blockchain transaction characteristics were evaluated within the respective study. `Evaluation Execution Environment` describes the hardware and software configura-

tions of the experimental environment, including processor specifications, memory capacity, and operating system, as these factors can significantly impact performance results.

The `Cores` column represents the number of processor cores explicitly allocated for transaction scheduling and execution. GC in this column refers to the cores that are allocated for garbage collection (GC) or other system processes. The `Speed Up Rate` column reports the speedup achieved in each study, further categorized into two distinct metrics: `Proposer` and `Attestor`, denoted as an ordered pair (p, a) for comparative analysis. Since attestors must maintain transaction order within a block, their speedup values are inherently lower than those of proposers for any given row in the table. If a value, including attestor performance, is not reported in a study, it is denoted as `N/A` (Not Available). For studies that present results across multiple core configurations, this table includes only representative values that allow for a comprehensive comparison.

The performance trends indicate that Conthereum [3] demonstrates the highest recorded speedup, achieving (2.92, 2.27) with 3 cores, surpassing the results of [103], which reported (1.33, 1.69) for the same core count. Notably, Conthereum's proposer speedup of 2.92 with just 3 cores exceeds the 16-core and 64-core speedups of [125], which reported 1.13 and 2.26, respectively, without attestor evaluation. Additionally, [104] did not examine attestor performance but tested proposers ranging from 4 to 24 cores, reporting a speedup of 3 for 4 cores and 5 for 24 cores. These values are outperformed by Conthereum, which achieves a proposer speedup of 3.73 for 4 cores and 5.81 for only 9 cores. This comparative analysis highlights the efficiency of Conthereum in achieving superior speedup with fewer computational resources.

4.5.4 Category 4: Directed Acyclic Graph (DAG) approaches

This category of solutions redefines the structural foundation of blockchain systems by replacing the traditional linear sequential chain of blocks with a *Directed Acyclic Graph (DAG)* [130]. The promise of DAG-based blockchain systems is to enable fast confirmation (complete transactions within million seconds) and high scalability (attach transactions in parallel) without significantly compromising security.

In DAG-based systems, transactions are represented as nodes, with directed edges indicating dependencies or confirmations between them. This approach eliminates the sequential bottlenecks inherent in conventional blockchains, such as Bitcoin's PoW and Ethereum's PoS, enabling parallel transaction processing and significantly enhancing scalability and concurrency. DAG allows users to validate prior transactions to con-

firm their own, reducing energy consumption and transaction costs. By improving throughput and reducing latency, DAG-based architectures are particularly well-suited for high-frequency or microtransaction environments.

Despite these advantages, DAG-based systems face significant challenges, such as ensuring network consistency, preventing double-spending attacks in low-activity networks, and maintaining resilience against attacks that exploit transaction dependencies. These limitations, combined with DAG's early stage of development and the absence of proven security guarantees, explain why some well-established blockchains do not use it. In particular, security is often sacrificed for speed, and the blockchain trilemma cannot be fully resolved by DAG alone.

Several well-known DAG-based systems include IOTA, which uses the Tangle where each transaction confirms two prior transactions for enhanced scalability; Nano, which employs a block-lattice structure for fast, fee-less transactions by allowing each account to maintain its own blockchain; and Hashgraph, which offers a DAG-based consensus protocol focused on fairness, security, and efficiency.

In conclusion, while DAG-based systems have become a well-established area of research in blockchain concurrency, this survey focuses on the broader landscape of concurrency in blockchain systems. Extensive literature on DAGs is covered in several key surveys, and readers are directed to those sources for in-depth analysis. This survey aims to fill critical gaps in the concurrency domain, which remains underexplored. For specialized surveys on specific concurrency aspects, we provide references, ensuring a comprehensive overview while guiding readers to relevant detailed studies. For further exploration of DAG-specific advancements, please refer to [131], [132], [133], [134], and[135].

4.5.5 Category 5: Semi-Centralized Solutions

Semi-centralized solutions in blockchain technology aim to balance the benefits of decentralization with the efficiency of centralized control. These approaches often seek to enhance scalability, transaction throughput, and governance by introducing certain centralized elements while maintaining the core principles of blockchain.

An illustrative example of a semi-centralized approach is the RCANE architecture [136]. RCANE proposes a network of parallel blockchains managed under a semi-centralized framework. This design allows for concurrent processing of transactions across multiple chains, effectively distributing the load and enhancing overall system performance. The semi-centralized governance model facilitates coordinated decision-

making and efficient management, addressing scalability challenges inherent in fully decentralized systems.

4.5.6 Category 6: Off-chain solutions

This research category discussed an intrinsic restriction of the blockchain in executing complex smart contracts. This limitation is made by considering gas to execute each instruction of a smart contract (Section 4.3.2) and a limited threshold of gas supported for any block. For instance, sorting 256 integers using the selection sort algorithm demands 17 million gas, exceeding the current block limit of 8 million. The quick sort also reaches this gas limit after processing 2,000 elements. Consequently, more sophisticated applications, such as decentralized blockchain oracles, become infeasible to execute under these constraints [137]. These smart contracts are name *complex* [137] or Computationally Intensive Contracts (CIC) [138]. There are some solutions to increase these per-block execution limits for complex smart contracts, and one of them is using *off-chain* mechanism. An off-chain execution model, allows contract issuers to delegate contract execution to a selected group of service providers, thus operating independently from the blockchain's consensus layer. This section of the survey encompasses the concurrent solutions in this category of architecture.

ACE (Asynchronous and Concurrent Execution of Complex Smart Contracts) [137] facilitates the execution of more complex smart contracts on permissionless blockchains using off-chain and the main benefit of ACE compared to earlier solutions is its ability to let one contract securely invoke another contract, even when they are executed by different sets of service providers. The system's key innovations include a flexible trust model and a robust concurrency control protocol, making it a significant advancement over previous solutions. The evaluation results show that ACE can facilitate the development of smart contracts that are significantly more complex than those on standard Ethereum.

In this protocol, concurrency is integrated into the consensus cycle, enabling numerous nodes to generate blocks simultaneously. PoV, in contrast, is an incentive mechanism designed to reward those who create *value*. The value is defined for any smart contract based on the smart contract token transaction volume. POV allocates more tokens to the owners of smart contracts with higher transaction volumes. POV incentivizes value creators and manages the supply of coins through an adjustable incentive coefficient algorithm. To enhance POV performance on permissioned blockchains, the authors proposed and developed *Hypernet*, a rapid off-chain transaction system that

allows transactions to be processed independently of the blockchain. It consists mainly of the client and server components. The Hypernet client sends transactions through a centralized server that does not require trust. The centralized server's role in Hypernet is merely to forward tamper-resistant transaction messages. Hypernet introduces the concepts of multi-signature address and contract address to facilitate secure and fast off-chain transactions. This solution is evaluated by generating random parameters to trigger contracts and recording the actual time taken to issue rewards. Their results indicate that the system incurs a loss of approximately 2% when POV is implemented, and Hypernet achieves transaction speeds four times faster than traditional permissioned blockchain systems.

[139] introduces SlimChain, a novel blockchain system designed to enhance transaction scalability through off-chain storage and parallel processing. Emphasizing a stateless approach, SlimChain retains only short commitments of ledger states on-chain, while off-chain nodes handle transaction executions and data storage. New schemes are proposed for off-chain smart contract execution, on-chain transaction validation, and state commitment. Additionally, optimizations are included to minimize network transmissions, and a new sharding technique is introduced to further boost system scalability. Extensive experiments validate SlimChain's performance, demonstrating a reduction in on-chain storage requirements by 97% to 99% and an improvement in peak throughput by 1.4 to 15.6 times compared to existing systems.

4.5.7 Category 7: Cross-chain solutions

Cross-chain solutions constitute a distinct category in smart contract concurrency due to their role in enabling atomic transactions and state synchronization across isolated blockchain networks. Unlike single-chain concurrency models, these approaches must address heterogeneous consensus protocols, varying finality times, and trust-minimized bridging mechanisms-all while preserving atomicity and preventing double-spending. Foundational work by [140] established the theoretical framework for cross-chain swaps, while later surveys ([141, 142, 143, 144, 145]) systematized the landscape. Key concurrency challenges include: i) race conditions during cross-chain contract execution due to delayed finality, ii) verification of atomicity in multi-chain transactions, and iii) composability risks when smart contracts span multiple VMs.

4.5.8 Category 8: Transaction Ordering Dependency(TOD) and Front-Running Attacks

This category focuses on race conditions resulting from different miners' decisions regarding transaction orders within persisted blocks. As discussed in Section 4.3.2, each block of the blockchain contains verified transactions. *Transaction Ordering Dependency (TOD)* refers to the uncertainty concerning the state of a contract from the user's perspective within the blockchain system at the time of contract invocation. When multiple transactions invoking the same contract occur almost simultaneously and are included in one block, the miner arbitrarily determines their order. This can lead to uncertainty for users regarding the state of the contract at the time of their transaction's execution, depending on the miner's decision. It is important to note that not all smart contracts are sensitive to transaction orders. Consequently, contracts affected by transaction ordering are termed TOD contracts. TOD contracts are susceptible to both benign and malicious invocations, the former potentially resulting in unexpected outcomes and the latter being exploited for unfair profit or theft [46]. Malicious exploitation of TOD-vulnerable smart contracts is termed *front-running attacks*, which originate on the Chicago Board Options Exchange (CBoE) in 1988 [146]. Its definition by the Securities Exchange Commission (SEC) in 1977 is as follows: "The practice of effecting an option transaction based upon nonpublic information regarding an impending block transaction in the underlying stock, to obtain a profit when the options market adjusts to the price at which the block trades [147]". In a blockchain system, front-running occurs when malicious entities exploit the visibility of pending transactions in the mempool to execute transactions for their own benefit at the expense of the original transaction owner. In this attack adversaries can influence the outcome of this transaction race to prioritize their transaction order in the block through participation in mining, offering higher `gasPrice` to incentivize miners, or collusion with other miners.

As a benign scenario example, consider a smart contract that publishes puzzles and rewards participants for correct solutions. Suppose the contract owner can alter the reward amount by a contract invocation. In such cases, when a participant submits their solution transaction, there is no certainty they will receive the current prize amount if their solution is correct. This uncertainty arises because the owner may simultaneously submit a transaction to change the prize amount, and depending on the miner's decision, the participant may receive either the old or updated reward. Another example is a decentralized exchange or marketplace allowing users to buy and sell tokens, with

fluctuating price functions callable by token owners. If a buy order and a price change transaction are submitted nearly simultaneously and included in the same block, the buyer cannot be certain whether they will purchase the token at the original or updated price, depending on the transaction order.

In malicious scenarios, the owner of a puzzle smart contract could monitor the network and, upon observing a solution being submitted, quickly send a transaction to change the reward amount to zero within the block interval (explained in 4.3.2), increasing the likelihood of their transaction being included before the solution transaction in the subsequent block. Consequently, the adversary can exploit the system to acquire puzzle solutions at minimal cost.

Detecting TOD involves recognizing valid transactions that influence a contract's global or state variables, akin to identifying read-after-write dependencies in race detection, which poses challenges for developers. The study conducted in [148] identifies different types of TODs, including a previously undocumented variant. To address TOD detection, the paper proposes TODler, a static analyzer based on information flow analysis. Evaluation of 108 Ethereum smart contracts demonstrates that TODler surpasses existing methods in terms of both speed and accuracy, while also identifying the newly discovered TOD pattern.

In [149] the authors introduce TransRacer, an innovative tool designed to detect transaction races in Ethereum smart contracts. Transaction races, which can result from the concurrent execution of transactions within the same block, have been largely overlooked despite their potential to cause inconsistent states and other unexpected outcomes. TransRacer employs symbolic execution to analyze function dependencies, enabling it to identify and generate witness transactions that reveal hidden races in specific contract states. This method significantly narrows the search space compared to random fuzzing techniques, enhancing detection accuracy and efficiency. Experimental results on 50 real-world smart contracts demonstrated TransRacer's capability to detect 426 races within approximately 256 minutes, including 149 significant race bugs. Further empirical analysis on 6,943 smart contracts underscores the prevalence and potential harm of transaction races in practice, highlighting the critical need for tools like TransRacer to maintain smart contract integrity.

[150] presents an open-source simulation tool designed to educate users on the vulnerabilities and mitigations associated with front-running attacks in Ethereum. This work introduces a comprehensive simulation environment that allows users to perform and understand various types of front-running attacks, such as displacement attacks, sandwich attacks, and priority gas auctions. The tool also demonstrates how to mitigate

these attacks using the MEV-geth protocol, enhancing transaction privacy. By providing a hands-on educational platform, the simulation software aims to bridge the gap in educational resources regarding blockchain security, particularly in teaching smart contract vulnerabilities and mitigation strategies. This tool has been integrated into the curriculum at the University of Bristol, underscoring its utility in training the next generation of blockchain developers. The simulation's experimental framework highlights the practical challenges and potential solutions in managing front-running attacks, contributing valuable insights into the security dynamics of blockchain transactions.

4.5.9 Category 9: Available concurrent transaction execution vulnerabilities (misconception)

In 2017, [151] introduced a concurrent perspective on smart contracts. It explained the similarities between the behaviors of smart contracts within cryptocurrency frameworks and the classic challenges encountered in shared-memory concurrency scenarios. By explaining two instances sourced from the Ethereum blockchain, the authors explained vulnerabilities to bugs that mirror those commonly encountered in traditional concurrent programs. In elaborating on these examples, the paper underscores the intrinsic relationship between observable contract behaviors and well-established concepts in concurrency, including atomicity, interference, synchronization, and resource ownership.

This study analogized smart contract utilization in blockchains to threads that engage with concurrent objects in shared memory (*contracts-as-concurrent-objects*). Notably, the paper critiques the prevalent non-modular design of locking contracts, advocating instead for the implementation of synchronization primitives as standalone libraries. This proposed shift aligns with established software engineering best practices, where such modularization fosters better reasoning about contract behaviors. However, the separation of contract logic from locking mechanisms presents its own set of challenges, necessitating a holistic approach to contract verification that considers interactions with rigorously specified external contracts.

Moreover, the discourse extends to address concerns regarding liveness properties in contract implementations involving locks and exclusive access. By contemplating scenarios where certain accounts may indefinitely retain access, impeding progress and fairness within the system, the paper raises pertinent questions about ensuring eventual progress and incentivizing parties to release locks. This discussion underscores the importance of fairness assumptions akin to those found in concurrency theory and

presents avenues for leveraging existing proof methods for reasoning about progress and termination in multi-contract executions.

Concluding with reflections on the broader implications of the proposed analogy, the paper advocates the use of insights from concurrency research to enhance understanding, debugging, and verifying complex contract behaviors within distributed ledger environments. Although acknowledging the limitations and unique aspects of contract programming, such as gas-bounded executions and fund management, the authors invite speculation on potential challenges and insights inspired by the observed parallels. These speculations range from exploring scenarios reminiscent of the ABA problem in non-garbage-collected languages to considering the influence of mining protocols on scheduling priorities and defining consistency notions for composite contracts with multi-transactional operations. Through this comprehensive examination, the paper underscores the interdisciplinary nature of blockchain research and its potential to benefit from insights gleaned from established fields such as concurrency theory.

[152] discusses the vulnerability of smart contracts, with a particular focus on concurrency related issues. The study highlights the critical challenges posed by the concurrent nature of smart contracts, which share similarities with concurrent programs with shared memory. Using formal verification methods, specifically the *Communicating Sequence Processes (CSP)* theory [153] and the *Failure Divergence Refinement (FDR)* model checking tool, the research aims to address these challenges. The paper illustrates a race condition in a smart contract example borrowed from Solidity language documents, named The Safe Remote Purchase smart contract [154], alongside an attack scenario resulting from specific concurrent executions of this smart contract. The authors utilize CSP theory to formally model this smart contract and the considered attack, subsequently using the FDR model checker to demonstrate the possibility of this attack occurring. The findings underscore the potential advantages of using CSP and FDR tools for vulnerability detection within smart contracts, particularly in the context of concurrency.

However, our analysis reveals an error in the original assumption made in this paper regarding the race condition and the designed attack in smart contracts, specifically within Ethereum which is the subject of this study. The truth is that inherently, Ethereum's infrastructure does not support concurrency, contrary to the assumption made in this study. The concurrency assumption in [152] is based on the study in [151], which has led to ongoing debate despite its flaws. Consequently, it underscores the importance of rectifying such misconceptions within this branch of the body of literature in this field; the next section discusses this in more detail.

4.6 Addressing the Flawed Assumption (Category 9)

This section delves into an erroneous assumption identified in the investigation, categorized as Category 9, as elaborated in Section 4.5.9. The assumption is the presence of concurrent executions within smart contracts, implying the possibility of race conditions in the execution of multiple transactions inside one smart contract in the Ethereum infrastructure. However, this section demonstrates why this assumption is invalid. The first subsection presents evidence from the Ethereum references to refute the notion of concurrent execution. References from Ethereum's architecture clearly outline the sequential and non-concurrent execution of transactions within each block that consequently prevent any concurrent executions inside one smart contract. The subsequent subsection explores an alternative perspective on the potential for concurrent execution through the mechanism of sharding.

4.6.1 Evidence from blockchain design

Ethereum. Buterin et al. explicitly stated the sequential execution of transactions on Ethereum white paper as "Note that the state is not encoded in the block in any way; it is purely an abstraction to be remembered by the validating node and can only be computed (securely) for any block starting from the genesis state and sequentially applying every transaction in every block" [9].

4.6.2 Sharding and potential for transactions concurrency issue

Sharding is a scalability technique that partitions the blockchain state and transaction processing across multiple shards (databases). This allows for parallel processing of transactions, potentially enabling true concurrency in smart contracts.

Sharding architecture

In a sharded architecture, transactions are routed to their designated shard based on a sharding key (e.g., user address). This enables concurrent processing of transactions within different shards. However, challenges remain:

Table 4.4: Comparison

Year	Ref	**Miner Approach**	**Locks**	**Require Block Graph**	**Validator Approach**	**Blockchain Type**
2020	Dickerson et al. [103]	Pessimistic ScalaSTM	Yes	Yes	Fork-join	Permissionless
2018	Zhang and Zhang [155]	-	-	Read, Write Set	MVTO Approach	Permissionless
2019	Anjana et al. [156]	Optimistic RWSTM	No	Yes	Decentralized	Permissionless
2019	Amiri et al. [157]	Static Analysis	-	Yes	-	Permissioned
2019	Saraph and Herlihy [125]	Bin-based Approach	Yes	No	Bin-based	Permissionless
2019	Anjana et al. [158]					
2021	Anjana et al. [159]	Optimistic ObjectSTM	No	Yes	Decentralized	Permissionless
2021	Anjana [160]					
2022	Baheti et al. [161]					
2023	Piduguralla et al. [162]					
2024	Anjana et al. [163]	Bin+Optimistic RWSTM	No	No (if no dependencies)/Yes	Decentralized	Permissionless

- **Cross-Shard Transactions:** Transactions that interact with data in multiple shards require careful coordination to maintain consistency.
- **State Synchronization:** Sharding introduces the need for efficient mechanisms to keep all shard states consistent with the overall blockchain state.
- **Security Considerations:** Careful design is necessary to ensure the security of the sharded system and prevent attacks that exploit shard boundaries.

Potential benefits of sharding for concurrency

Despite the challenges, sharding offers potential benefits for achieving true concurrency in smart contracts:

- **Increased Throughput:** Parallel processing can significantly increase the number of transactions processed per second.
- **Improved Scalability:** Sharding can accommodate a growing number of users and transactions.
- **Enhanced Flexibility:** Developers can design smart contracts that take advantage of shard-specific functionalities.

4.7 Conclusion and future work

In this paper, we have critically examined the assumptions surrounding concurrency in smart contracts, identifying misconceptions in the discourse on concurrent transaction execution. Through a comprehensive analysis of existing literature and concepts such as sharding, we have challenged prevailing paradigms and highlighted the need for a nuanced understanding of concurrency in blockchain systems. Moving forward, we

advocate for continued research and development efforts to address the inherent trade-offs between scalability, security, and decentralization in blockchain networks.

Chapter 5

Static Analysis for Detecting Transaction Conflicts in Ethereum Smart Contracts

This chapter corresponds to the article:

Atefeh Zareh Chahoki, and Marco Roveri. "Static Analysis for Detecting Transaction Conflicts in Ethereum Smart Contracts." In *Proceedings of the 2025 8th International Conference on Blockchain Technology and Applications (ICBTA)*, Kanazawa, Japan, 2025.

5.1 Introduction

Static program analysis (SPA) is a cornerstone of software engineering, providing rigorous techniques to reason about program behavior without execution. Techniques such as abstract interpretation, data-flow analysis, and control-flow analysis detect subtle errors, ensure correctness, and verify security properties in complex systems [164]. Beyond correctness and security, SPA supports scalability by providing compile-time insights to optimize runtime execution strategies.

The emergence of decentralized applications (DApps) and smart contracts on the Ethereum Virtual Machine (EVM) introduces a critical new domain for SPA. Smart contracts manage significant financial value, and their immutability amplifies the consequences of programming errors. While prior work has focused on security vulnerabilities such as reentrancy and integer overflow, transaction conflicts remain a less-explored challenge affecting both scalability and security.

Ethereum smart contracts, typically written in Solidity, compile into bytecode executed by the EVM, which enforces sequential transaction execution to ensure determinism and prevent inconsistent state updates. This sequential execution limits parallelization and underutilizes multi-core architectures, creating a scalability bottleneck. Runtime solutions like optimistic concurrency control and speculative execution [103] degrade as conflict rates rise, whereas Conthereum [3] uses pre-execution conflict information to generate conflict-free schedules. As Layer 2 rollups and sharding evolve, static conflict analysis becomes increasingly important for batch reordering and parallel execution.

From a security perspective, immutability makes correctness critical. Vulnerabilities arise from low-level bugs (e.g., arithmetic overflows), control flow misuse (e.g., reentrancy), or transaction-level interactions such as Transaction Ordering Dependence (TOD). Existing fuzzy, dynamic, and static analysis approaches detect many vulnerabilities. Tools like Oyente [46], Mythril [57], Slither [60], and Securify [47] are effective for known patterns but generally cannot identify higher-order transaction conflicts crucial for TOD detection.

In this work, we leverage static analysis to predict potential transaction conflicts, addressing both scalability and security. By analyzing state variable access patterns, our method identifies read-write, write-write, and function call conflicts, enabling schedulers to plan parallel execution with minimal rollbacks. Developers can also detect potential TOD-related conflicts to mitigate vulnerabilities before deployment, supporting safer and more scalable smart contract execution.

This paper makes the following contributions. First, we present a novel static analysis framework for identifying potential transaction conflicts without execution traces, enabling early detection of conflict-prone function pairs for optimized scheduling, risk-aware testing, and enhanced auditability. Second, we introduce a conflict taxonomy covering direct and indirect conflicts for inspecting overlapping state accesses. Third, we implement a tool that parses Solidity contracts, analyzes state variable patterns, detects conflicts with severity ratings, and generates comprehensive reports. Finally, we evaluate our framework on real-world Ethereum contracts, demonstrating high precision, with read-write conflicts most prevalent, followed by write-write and function call conflicts.

This paper is organized as follows. Section 5.2 provides background on smart contract execution and analysis. Section 5.3 reviews related work. Section 5.4 details our static analysis approach. Section 5.5 describes the tool implementation. Evaluation results are presented in Section 5.6, and Section 5.7 concludes with future directions.

5.2 Background

This section briefly reviews the EVM memory model [165] with focus on the memory locations whose semantics and persistence determine whether two distinct transactions may produce interfering state updates and then follows with minimal technical background of Solidity which is necessary for the rest of this document. These memory types are summarized in Table 5.1. Since only `storage` persists across transactions, our analysis focuses on state variables mapped to storage slots and on code paths that may result in reads or writes to those slots directly or indirectly. Memory, calldata, and stack locations are local to a single execution and do not create inter-transaction conflicts; events (logs) are recorded externally and are not read by contracts. This focused scope enables both precision (by avoiding spurious dependencies) and efficiency (by limiting the analysis domain).

In Solidity, among function visibilities, only `public` and `external` functions can be invoked through transactions and are therefore the focus of this study, however for extracting call chains and identifying indirect conflicts all of them are included. Among function state mutabilities, only `pure` functions, which neither read nor write data, and `view` functions, which read but do not modify storage, are relevant. Others, such as `non-reentrant` and `non-payable`, are excluded. For variable visibility, all types (`private`, `public`, and `internal`) are stored on the blockchain and persist across

Table 5.1: EVM memory kinds and relevance to inter-transaction conflict detection.

Memory kind	Description	Conflict relevance
`storage`	Persistent contract state (slot-indexed, persisted across transactions).	High: primary source of inter-transaction conflicts.
`memory`	Transient memory per call (reset after a call finishes).	Low: local to a single call/-transaction.
`calldata`	Read-only view of external arguments.	Low: per-call and immutable within a call.
`stack`	EVM operand stack (transient).	None: ephemeral for execution.
`logs (events)`	Emitted data stored off-chain for indexing.	None: cannot be read by contracts to form conflicts.
`bytecode`	Immutable contract code.	None: no effect for data conflicts, affects control flow only.

transactions, even private ones, and are considered in our algorithm.

5.3 Related Works

Research on smart contract analysis has focused primarily on detecting security vulnerabilities rather than transaction conflicts. In this section, we review existing work on smart contract analysis and highlight the gaps that our approach addresses.

Smart Contract Vulnerability Detection. Several tools exist to identify security vulnerabilities in smart contracts. Oyente [46] pioneered using symbolic execution to detect issues like reentrancy, timestamp dependence, and mishandled exceptions. Mythril [57] enhances this with control flow analysis and taint tracking. Securify [47] employs formal verification to ensure security compliance. These tools prioritize known vulnerability patterns over transaction conflicts.

Concurrency in Smart Contracts. Kolluri et al. [166] show how transaction-ordering dependencies can be exploited in decentralized exchanges. Chakravarty et al. [167] suggest the UTXO model to address concurrency issues in place of Ethereum's account-based model. These studies focus more on theory or specific applications, lacking a general method for detecting transaction conflicts. [6] provides a comprehensive study of concurrency across different layers of blockchain, and this preventive static analysis is beneficial for many of these layers.

Static Analysis of Smart Contracts. Static analysis is commonly used for smart contract verification. Slither [60] detects vulnerabilities and offers contract structure insights, MadMax [168] identifies gas-related issues, and SmartCheck [169] utilizes pattern matching on XML code. These tools are informative but do not address transaction conflicts. There are numerous studies focused on conflict detection using the static analysis of Ethereum bytecode [170], [171], but the analysis of Solidity source code still requires further investigation.

5.4 Conflict Detection Using Static Analysis.

Although the Turing-complete nature of smart contract languages makes it impossible to predict conflicts precisely in advance, this algorithm adopts a pessimistic approach. It may report many false positives (potential conflicts that may not occur in any real execution flows) but ensures no false negatives (if it detects no conflict, there is certainty that no conflict will occur). Using static analysis of storage memory, this algorithm effectively enables these preventive and reliable conflict predictions. Our approach for detecting transaction conflicts in Ethereum smart contracts written in Solidity consists of three main steps, described in this section: (1) parsing the contract code, (2) analyzing state variable access patterns, and (3) detecting potential conflicts.

5.4.1 Step 1: Contract Parsing

The first step involves parsing the Solidity smart contract code to extract its structural components, including state variables, functions, events, memory types (storage, memory, and calldata), and visibility modifiers (public, private, internal, and external). We use Slither [60] to leverage its existing analysis capabilities. Slither internally relies on `solc`, the official Solidity compiler, to generate the abstract syntax tree (AST) representing the syntactic structure of the program [172]. Hence, the parsing in our approach is ultimately performed by `solc` [173].

The information extracted from the parser for our study is as follows: (1) Contract name and source code, (2) State variables: name, type, visibility, and memory type, (3) Functions: name, parameters, return values, visibility, and state mutability, (4) Modifiers: name and the list of state variables they access, (5) Spacial modifiers: like `nonReentrant` that apart from state variable access, influence function behavior, and (6) Special functions: such as constructors, fallback functions, and receive functions

In the context of detecting potential conflicts between smart contract functions

based on their read and write access to persistent blockchain storage, *event* declarations are excluded from consideration. This is because events in Ethereum-like blockchains are merely log entries that are stored separately from the contract's state variables and are inaccessible from within the contract after emission; hence, they cannot induce read-write or write-write conflicts in contract storage.

Similarly, *constructor*s can also be disregarded in conflict analysis. A constructor is executed only once during contract deployment and cannot be invoked thereafter; moreover, no other function can execute concurrently during this phase, as the contract is not yet deployed for external calls. Consequently, neither events nor constructors contribute to runtime state conflicts between concurrently executing contract functions.

In contrast, *modifiers*, despite their primary role in handling cross-cutting concerns, often access and evaluate contract state variables. For example, the commonly used `onlyOwner` modifier checks whether `msg.sender` matches the value of the `owner` state variable. If any function modifies `owner`, this operation can conflict with the execution of other functions guarded by the same modifier. Therefore, modifiers are treated equivalently to inlined function calls when analyzing potential state conflicts. This modifier conflict is also bring into consideration in function call chains as well.

Any state variable referenced within a *require* statement used to validate execution conditions—for example, `require(balances[msg.sender] >= amount, "Insufficient balance")`—is classified as being read either being in a function or a modifier. Consequently, the enclosing function may conflict with any other function that modifies the same variable inside the require block.

5.4.2 Step 2: State Variable Access Analysis

This stage extract a precise characterization of state variable accesses (reads and writes) per function and an effective propagation of these effects through intra-contract call graphs.

Given a compiled Solidity project resulted from the previous step, we construct a Slither model and query each function's direct state variable reads/writes. Concretely, the analysis consumes Slither's per-function (1) Direct attributes `state_variables_read` and `state_variables_written` when sufficient, and (2) IR-driven inspection of `Assignment`, `Index`, and `Member` operations to robustly classify accesses, including compound assignments (e.g., "`+=`") that semantically imply both a read and a write of the left-hand state variable.

For each function, we identify: (1) Read Operations: State variables that the

function reads, (2) Write Operations: State variables that the function modifies, and (3) Function Calls: Other functions that the function calls

After extracting the *Control Flow Graph (CFG)*, it is used to perform transitive state effect analysis. State effects are propagated across intra-contract call graphs to capture transitive reads and writes: if function f calls g, f inherits g's state effects. We compute the transitive closure over the internal call adjacency and union callees' read/write sets into callers' effective sets. This models the net state footprint of externally invocable entry points (public/external functions), which is crucial for accurate conflict detection.

This analysis achieves precision by using Slither's data-dependency utility (*is_dependent*) for indirect value-flow reasoning and its intermediate representation (IR) for accurate handling of compound assignments and member/indexed updates. While our approach provides an initial static analysis for conflict detection on Solidity code, precision is aimed be further enhanced in future work possibly by incorporating full alias analysis, points-to analysis.

5.4.3 Step 3: Conflict Detection

The third step is to detect potential conflicts between functions. We consider three types of conflicts. Direct access conflicts that arise when transactions directly access the same state variable, including: (1) Read-Write (RW): One transaction reads a variable while another writes to it, (2) Write-Write (WW): Both transactions write to the same variable, and (3) Transitive Access Conflicts: Conflicts caused by indirect state access through nested or chained function calls, where one transaction invokes a function that modifies or reads state variables accessed by another transaction.

For each pair of functions that can be called as transactions (public or external functions that are not pure), we check for these conflicts based on their state variable access patterns. We also calculate a conflict percentage for each contract, which is the ratio of *function pairs* with conflicts to the total number of possible function pairs. This metric provides an overall measure of the contract's susceptibility to transaction conflicts.

Cross-contract interactions in Solidity primarily arise through composition, libraries, and proxy patterns, all of which can create inter-contract conflicts by enabling multiple contracts to access or modify shared storage. Inheritance, by contrast, does not produce such conflicts since the derived and base contracts share a single unified storage layout.

5.5 Implementation

The workflow of the proposed approach is illustrated in Figure 6.1. It consists of several modules, each responsible for a specific stage of the analysis. The process begins with pre-processing, where smart contracts are checked for supported versions and, if necessary, converted accordingly. In this step, hidden functions (e.g., getters) are added and normalization is applied. Next, contracts are parsed using the Slither library, complemented by additional coding and adjustments. Relevant variables, functions, and modifiers are then extracted, and a call graph is constructed between functions and contracts. The resulting data are forwarded to the detector module, which is followed by statistical analysis and visualization through the report generator. The tool is released as open source at `https://github.com/Conthereum/conflict-detector-static-analysis`.

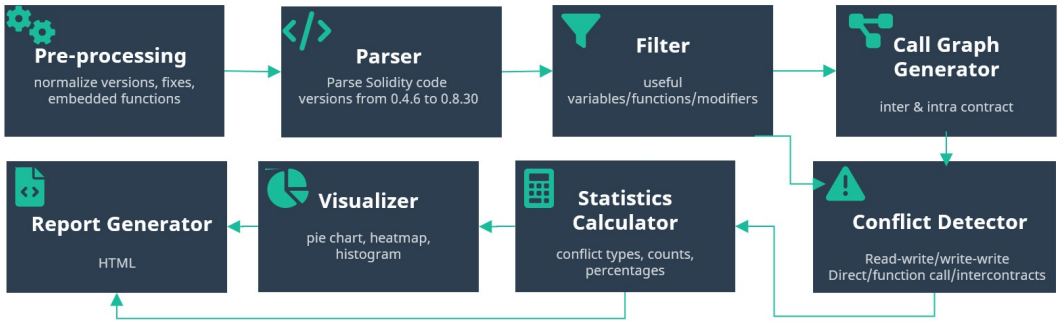


Figure 5.1: Workflow Diagram of the Proposed Solution.

The pseudocode in 5 summarizes the main logic of the batch static analyzer for Solidity smart contracts. Each Solidity contract is preprocessed; in this process, some hidden functions such as getters are added to the contract, and for a few older, unsupported contract versions, the syntax is converted to a supported one. Then, its public or external functions, all modifiers, and non-constant state variables are extracted. Using Slither's intermediate representation and call graph, effective read and write sets are propagated along function call chains, allowing conflicts to be detected not only from direct variable access but also through transitive calls, modifiers, and intercontract calls. Conflicts can occur in two ways: read–write and write–write. These can happen directly, through function chains, or via intercontract calls, resulting in six more granular types of conflicts, which we categorize. It is also worth noting that the original implementation included robustness fixes to account for limitations in Slither's default analysis—e.g., correcting cases where compound assignments (such as `+=` or `-=`) might be incorrectly

Algorithm 5 Conflict Detection in Smart Contracts

Require: Set of Solidity contracts C
for all contract $c \in C$ **do**
  preprocess (c) ▷ normalize version, minimal fixes
  extract state variables V_c and functions F_c
  filter V_c to non-constant
  filter F_c to public/external (non-constructors)
  build call graph and effective read/write sets:
  for all $f \in F_c$ **do**
    $R_f \leftarrow$ vars read (directly or via call chain or inter-contract)
    $W_f \leftarrow$ vars written (directly or via call chain or inter-contract)
  for all unordered pairs (f_i, f_j) in F_c **do**
    if $R_{f_i} \cap W_{f_j} \neq \emptyset$ or $R_{f_j} \cap W_{f_i} \neq \emptyset$ **then**
      record **Read-Write Conflict** between f_i, f_j
    if $W_{f_i} \cap W_{f_j} \neq \emptyset$ **then**
      record **Write-Write Conflict** between f_i, f_j
compute statistics: conflict types, counts, and percentages

classified as pure reads rather than simultaneous read–write operations. These technical adjustments ensure accurate conflict detection but are abstracted away in the high-level pseudocode.

5.5.1 Data Model

The tool employs a structured data model to represent smart contracts and their analysis results. Each *Contract* includes its name, source code, state variables, functions, and events, while *StateVariable* and *Function* capture the properties of variables and functions, including types, visibility, mutability, and parameters. *Parameter* and *Event* describe function inputs and contract events, respectively. *ContractAnalysisData* records variable access patterns and function pair conflicts, and *Conflict* stores conflict type, severity, and description. Finally, *AnalysisResult* consolidates the contract, its analysis data, conflicts, and associated metrics, providing a complete representation of the analysis outcome.

5.5.2 Report Generation and Visualization

The tool produces comprehensive reports and visualizations to support the analysis of smart contract conflicts. Reports are generated in HTML format, summarizing key contract information, including functions, state variables, and detected conflicts, while

CSV files provide structured data for further analysis or integration with other tools. Visualizations are automatically generated using Python libraries such as Matplotlib and Seaborn, offering intuitive representations of the conflict landscape through pie charts, histograms, heatmaps, and scatter plots. These visualizations highlight patterns such as the distribution of conflict types, conflict counts per contract, relationships between contract characteristics and conflicts, and analysis time, enabling developers to quickly identify and address potential issues. Together, the reports and visualizations provide a clear, accessible overview of contract conflicts and facilitate informed decision-making during development.

5.6 Evaluation

This section reports the experimental results obtained from the implementations presented in the previous section (6.4). Experiments were conducted on a Windows 11 Pro (64-bit) laptop with an Intel Core i5-1135G7 CPU (4 cores, 8 threads, 2.40GHz) and 16GB RAM.

We collected a dataset of 100 Ethereum smart contracts from the Ethereum mainnet. The contracts were selected to represent a diverse range of applications, including tokens, decentralized exchanges, lending protocols, and governance contracts. The contracts vary in size and complexity.

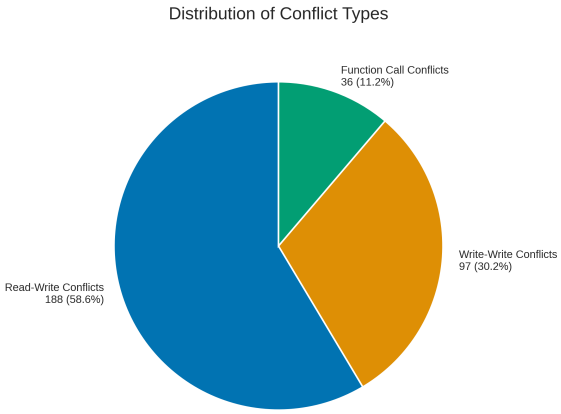
We ran our tool on each contract in the dataset and collected the analysis results, including the number and types of conflicts, the conflict percentage, and the analysis time. We also made a comprehensive collection of automated test cases that evaluate the correctness of the detections precisely to assess the precision of our approach.

5.6.1 Results

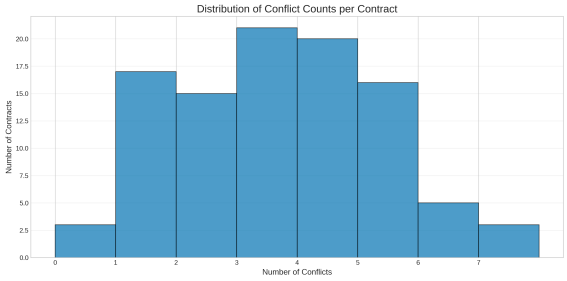
Conflict Distribution.

Our analysis identified a total of 423 conflicts across the 100 contracts. Figure 5.2a shows the distribution of conflicts by type.

As shown in the figure, read-write conflicts are the most common (58.6%), followed by write-write conflicts (30.2%) and function call conflicts (11.2%). This distribution reflects the typical access patterns in smart contracts, where functions often read and write to shared state variables.



(a) Distribution of Conflict Types



(b) Distribution of Conflict Counts per Contract

Figure 5.2: Conflict distributions in contracts.

Conflict Prevalence.

We found that 78% of the contracts in our dataset have at least one potential transaction conflict. The average number of conflicts per contract is 4.23, with a maximum of 27 conflicts in a single contract. Figure 5.2b illustrates the distribution of the number of conflicts per contract, which, as expected, follows a normal (Gaussian) distribution.

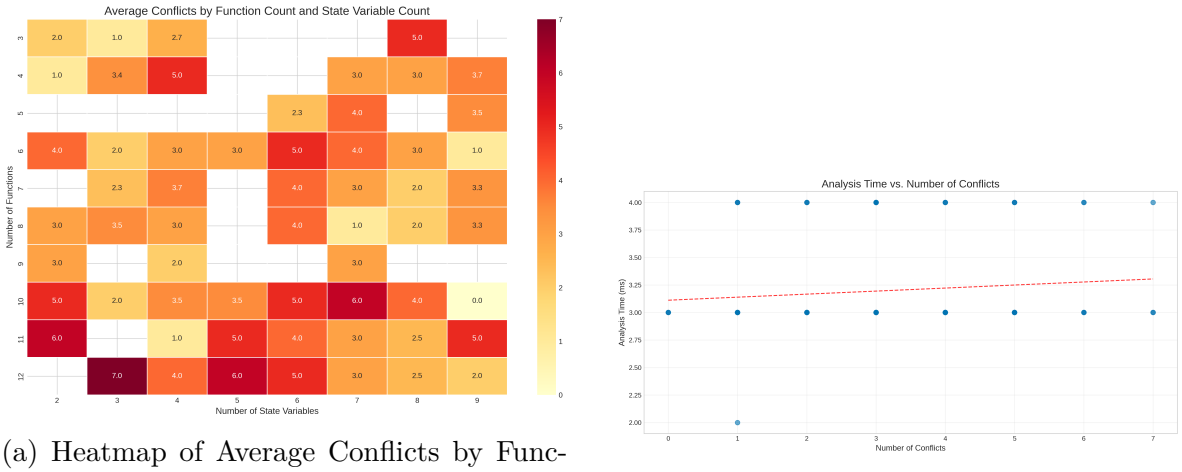
The conflict percentage, which is the ratio of function pairs with conflicts to the total number of possible function pairs, ranges from 0% to 87.5%, with an average of 32.7%. This metric provides an overall measure of the contract's susceptibility to transaction conflicts.

Conflict Patterns.

We observed several patterns in the detected conflicts:

- **State Variable Hotspots**: Some state variables, such as balances and ownership information, are accessed by multiple functions and are frequent sources of conflicts.
- **Function Hotspots**: Some functions, such as transfer and withdraw functions, are involved in multiple conflicts due to their interaction with critical state variables.
- **Conflict Clusters**: Conflicts often occur in clusters, where a group of functions interact with a common set of state variables.

Figure 5.3a shows a heatmap of the average number of conflicts by function count and state variable count.



(a) Heatmap of Average Conflicts by Function Count and State Variable Count

(b) Analysis Time vs. Number of Conflicts

Figure 5.3: Comparison of conflict patterns and analysis time.

As expected, contracts with more functions and state variables tend to have more conflicts. However, the relationship is not strictly linear, as the specific access patterns and function interactions also play a significant role.

Analysis Performance

The analysis time ranges from 52 ms to 1,873 ms per contract, with an average of 312 ms. Figure 5.3b shows the relationship between the number of conflicts and the analysis time.

The analysis time is primarily influenced by the size and complexity of the contract, particularly the number of functions and state variables. The number of detected conflicts also has a moderate impact on the analysis time, as more conflicts require more processing for report generation.

5.6.2 Precision and Recall

To assess the precision of our approach, we manually reviewed a random sample of 50 detected conflicts. We found that 46 of them (92%) are true positives, where the functions could indeed conflict if called concurrently. The false positives (8%) were primarily due to limitations in our static analysis, such as imprecise identification of state variable access patterns or failure to account for control flow dependencies.

Evaluating recall (the percentage of actual conflicts that are detected) is challenging without ground truth data. However, we compared our results with a subset of contracts that have known transaction ordering dependencies, and our tool successfully identified all of them.

5.6.3 Case Studies

We conducted detailed case studies on three contracts with high conflict percentages to understand the nature of the conflicts and their potential impact.

Token Contract.

A standard ERC20 token contract had 12 conflicts, primarily involving the transfer, transferFrom, and approve functions. These functions modify the balances and allowances state variables, leading to potential read-write and write-write conflicts. For example, if two transfers from the same account are executed concurrently, the final balance may be incorrect if the second transfer reads the balance before the first transfer updates it.

Decentralized Exchange.

A decentralized exchange contract had 27 conflicts, the highest in our dataset. The conflicts involved functions for placing orders, executing trades, and withdrawing funds. These functions interact with complex state variables such as order books and user balances, leading to numerous potential conflicts. For example, if an order is placed and executed concurrently, the execution may use outdated order information.

Governance Contract.

A governance contract for a decentralized autonomous organization (DAO) had 18 conflicts. The conflicts involved functions for proposing, voting on, and executing governance actions. These functions modify shared state variables such as proposal status and vote counts, leading to potential conflicts. For example, if two users vote on a proposal concurrently, one vote may be lost if both transactions read the same vote count before updating it.

5.6.4 Discussion

Our evaluation demonstrates that transaction conflicts are prevalent in Ethereum smart contracts, with 78% of contracts in our dataset having at least one potential conflict. The high precision of our approach (92%) indicates that it effectively identifies genuine conflicts that could lead to issues if not properly addressed.

The distribution of conflict types provides insights into the nature of transaction conflicts in smart contracts. Read-write conflicts are the most common, reflecting the typical access patterns where functions read shared state before making decisions. Write-write conflicts, while less common, are potentially more severe as they can lead to state corruption and lost updates. Function call conflicts are the least common but can be complex to analyze and address due to the interaction between multiple functions.

The conflict patterns we observed highlight the importance of careful state variable and function design in smart contracts. Developers should be aware of potential hotspots and clusters of conflicts and implement appropriate synchronization mechanisms or redesign their contracts to avoid issues.

Our tool's performance is suitable for integration into development workflows, with an average analysis time of 312 ms per contract. This enables developers to receive immediate feedback on potential conflicts as they write or modify their contracts.

5.7 Conclusion

This paper presented a static analysis approach for detecting transaction conflicts in Ethereum smart contracts before runtime. By analyzing contract code, our method identifies read-write, write-write, and function call conflicts between transaction pairs, and we implemented it in a tool that parses Solidity contracts, detects potential conflicts, and provides severity ratings. Evaluation on 100 real-world contracts showed that 78% contain at least one potential conflict, with 58.6% read-write, 30.2% write-write, and 11.2% function call conflicts, achieving a precision of 92%.

Our contributions include a novel static analysis technique, a classification of transaction conflicts with severity ratings, a supporting tool with report and visualization capabilities, and an empirical evaluation providing insights into conflict prevalence. This approach enables developers to detect potential conflicts early, facilitating safer contract design and reducing risks from transaction ordering dependencies.

Future work includes enhancing the precision of static analysis and conducting larger-scale empirical studies, including comparisons with bytecode-level static analysis,

to gain more comprehensive insights into the differences between analyses at various language levels.

Chapter 6

Conthereum: Concurrent Ethereum Optimized Transaction Scheduling for Multi-Core Execution

This chapter corresponds to the article:

Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. "Conthereum: Concurrent Ethereum Optimized Transaction Scheduling for Multi-Core Execution." In *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, Zurich, Switzerland 2025.

6.1 Introduction

A fundamental performance limitation for most blockchain platforms such as Ethereum lies in the limited transaction execution throughput, which is crucial to supporting high-volume decentralized applications. The *ledger* acts as a distributed state machine, where validators execute the transactions of each block sequentially to guarantee deterministic state transitions and maintain consensus over this shared global state. This conservative sequential execution model ensures correctness but severely restricts throughput, particularly when compared to traditional systems. While traditional financial platforms like Visa and PayPal process thousands of transactions per second (TPS), blockchain networks such as Bitcoin [8] and Ethereum [95] achieve significantly lower TPS [107] (Bitcoin processes only 3-7 TPS, and Ethereum approximately 15-25 TPS [107]). In contrast, Visa averages 9,840 TPS with a peak of 65,000 TPS [174], while PayPal averages 8,340 TPS [175]. This performance gap underscores the need for faster blockchain transaction processing to compete with centralized systems.

While recent advances such as sharding [115] improve scalability by enabling inter-shard parallelism, intra-shard execution remains strictly sequential. This prevents validators from using the full computational potential of multi-core systems during block execution. Parallel execution of transactions within a block is theoretically feasible when those transactions access disjoint states; however, the practical realization is complicated by dynamic and unpredictable access patterns, which often lead to memory access conflicts. These conflicts often arise from popular contracts such as auctions and arbitrage opportunities [176]. Speculative execution frameworks [103] have added concurrency by reordering or rolling back conflicting transactions at runtime. While these techniques provide significant speedups, their effectiveness diminishes under high-conflict workloads due to frequent re-executions [125].

To overcome these limitations, we propose *Conthereum* (short for *Concurrent Ethereum*), a deterministic and conflict-aware scheduling framework that enables safe, intra-block parallelism without compromising execution consistency. By incorporating static analysis to conflict detection, and a novel scheduling algorithm, our approach allows validators to achieve substantial speedup with minimal overhead, even under realistic, conflict-heavy conditions by enabling conflict avoidance rather than resolution at runtime.

This paper makes the following key contributions: 1. A novel scheduler for multi--core transaction execution that avoids conflicts and ensures consistency through safe, deterministic parallelism. 2. An open-source implementation based on a variant of the

Flexible Job Shop Scheduling (FJSS) problem, solved using a greedy heuristic that reduces execution time and achieves efficient suboptimal scheduling, outperforming existing benchmarks. 3. Experimental results demonstrate near-linear throughput scaling on 8-core machines, maintaining performance significantly above that of serial execution and earlier intra-block parallelism solutions, even with increased cores and transaction conflicts. 4. Robust performance under high-conflict transaction sets, where speculative approaches typically degrade and may perform worse than sequential execution. 5. An extensible multi-objective optimization framework that supports tuning cross-cutting concerns such as energy consumption, enabling validators to balance resource usage according to their priorities.

The rest of the paper is structured as follows: Section 6.2 presents background concepts.Section 6.3 introduces our scheduling algorithm. Implementation details are provided in Section 6.4, followed by experimental evaluation in Section 6.5. Section 6.6 discusses related work, and Section 6.7 concludes the paper and outlines future research directions.

6.2 Background

This section covers essential background on Solidity [154] and the Job Shop Scheduling (JSS) problem.

6.2.1 Smart Contracts

Blockchain systems have evolved from cryptocurrency platforms into decentralized computation via smart contracts. Ethereum [9] enabled this with the *Ethereum Virtual Machine (EVM)*, a Turing-complete state machine for deploying and executing smart contracts. Below, we summarize key

Terminology of Roles and Responsibilities. With the 2022 transition from Proof of Work (PoW) to Proof of Stake (PoS), *miners* were replaced by *validators*, responsible for *proposing* and *attesting* blocks. Proposers, selected pseudo-randomly based on staked ETH, execute transactions and broadcast their *proposed block*; attestors independently verify them. Some works distinguish miners and validators [103], while others adopt the proposer/attestor terminology [104]. We follow Ethereum's current convention, referring to participants as *validators* and their tasks as *proposing* and *attesting*.

Transaction Types. As illustrated in Fig. 6.1 Solidity transactions fall into two categories: those persisted on-chain and those pending in the mempool. Each category includes two types: (1,3) contract creation and (2,4) contract invocation-where 1 and 2 are on-chain, 3 and 4 are in-mempool. Throughout this document, "smart contract" refers to type 1. We use Txn_i to denote transactions of type i, and $Txn_{i,j} = Txn_i \cup Txn_j$ to represent combined sets; e.g., $Txn_{3,4}$ includes all mempool transactions.

Function Visibility. Solidity defines four function visibilities: *public* (callable by any contract), *private* (only within the current contract), *internal* (within the current and derived contracts), and *external* (only by external contracts). We denote public functions of contract C as $C.Func.Public$, and similarly for other visibilities.

6.2.2 Job Shop Scheduling Problems

Job Shop Scheduling (JSS) is a classic combinatorial optimization problem [177] with many variants [178]. A set of *jobs*, each consisting of ordered *tasks* or *operations*, is assigned to *machines* to minimize the *makespan*-the span from the start of the first task to the completion of the last task. *Flexible Job Shop Scheduling (FJSS)* extends JSS by allowing each task to be executed on one of several machines, improving adaptability. Given its *NP-hard* nature, finding an *optimal* solution is computationally impractical for large instances, especially as the number of jobs and machines increases. Therefore, *heuristic* and *metaheuristic* methods are employed to find *suboptimal* or *feasible* solutions within a limited *wall time*, beyond which results are *unknown*. In this document, *total execution time* refers to the sum of the wall time (for scheduling) and the makespan (for executing all jobs in parallel based on the obtained schedule).

6.3 Conthereum Solution

This section presents *Conthereum*, a scheduling-based algorithm that enables intra-block transaction concurrency in Ethereum. As illustrated in Fig. 6.1 and detailed in the following subsections, the Conthereum architecture consists of: cost analysis estimating transaction durations from gas usage (6.3.1); conflict analysis for safe parallelization (6.3.2); and a scheduling algorithm that formalizes transaction placement as a constrained optimization problem (6.3.3). Conthereum improves core utilization and reduces block execution time, thereby increasing throughput. The algorithm also supports cross-cutting concerns such as energy-aware scheduling, enabling validators to balance execution speed and power consumption based on hardware constraints and

cost preferences.

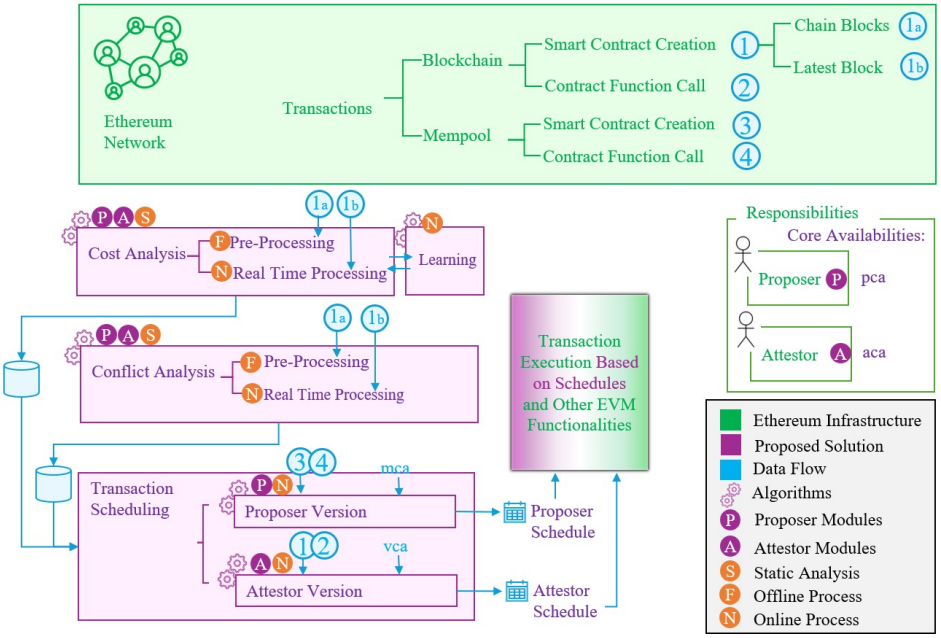


Figure 6.1: Workflow Diagram of the Proposed Solution.

6.3.1 Cost Analysis

Gas is the unit of computational cost in Ethereum, representing the resources required to execute operations. Users pay gas fees to incentivize validators to process their transactions. Building on prior findings [125], which demonstrate a robust linear correlation between gas consumption and execution time for individual transactions, we use this relationship to calculate the gas expenditure, enabling us to estimate the expected execution time for each transaction.

In the *EVM*, gas consumption is not statically embedded in a transaction's structure but dynamically determined at runtime, depending on the execution path (control flow, loops, and external calls) of the smart contract, thus making pre-execution estimation inherently uncertain.

Conthereum uses a hybrid gas-to-time model, starting with *static analysis*, where using Slither [60] estimates gas usage from control flow and operations. Though fast and data-free, it can miss execution nuances. Therefore *dynamic feedback* refines this by using execution times from previous scheduler runs to adjust predictions, letting the system learn the actual transaction costs. Each transaction t receives an estimated execution time $\widehat{T}(t) \propto \widehat{G}(t)$, based on $\widehat{G}(t)$, the static gas estimate. The proportion-

ality constant is tweaked with observed data, enhancing accuracy while maintaining scalability.

6.3.2 Conflict Analysis

Conthereum is the first solution to use the public availability of smart contract code to perform runtime scheduling through static analysis, enabling a more agile alternative to speculative intra-block concurrency approaches. It statically analyzes Solidity code to construct a repository of potential conflicts between pairs of public and external functions callable by Ethereum transactions. The complete methodology for this conflict detection is detailed in [2]. The architecture supports *interchangeability*, allowing other static conflict analysis techniques, such as [179], to be substituted without impacting the overall system design.

[2] identifies read-write, write-write, and function call conflicts between transaction pairs by analyzing state variable access patterns in Solidity contracts. The study employs Slither [60] to parse a wide range of Solidity versions and detect conflicts. Evaluation on a dataset of Ethereum smart contracts shows zero false negatives and a low false positive rate in identifying potential conflicts.

In Conthereum, this conflict detection is performed in two phases. First, a bulk analysis is run on all existing and deployed contracts on the blockchain. Second, the conflict dataset is incrementally updated by analyzing new contract-related transactions as they appear in incoming blocks. This design enables nodes to independently maintain consistent conflict data without the need for explicit data exchange. These two phases correspond to steps 1 and 3 in Figure 6.1.

6.3.3 Scheduler Description and Formalization

In the core of Conthereum there is a novel scheduler to optimize transaction execution in Ethereum, addressing two critical challenges: throughput limitations and cost efficiency. In the existing Ethereum framework, transactions within a block are processed sequentially to maintain state consistency and avoid conflicts. While this approach ensures correctness, it limits the number of transactions that can be executed concurrently, ultimately reducing overall performance. Additionally, validators, responsible for blocks proposing and attestation, face operational costs tied to processing transactions, making efficiency an important consideration.

Conthereum overcomes these challenges by parallelizing transaction execution on

Table 6.1: Nomenclature.

Constants	
$Prcs$	List of processes P_i, ordered by i, with cardinality n
t_i	Expected execution time of process P_i (in milliseconds)
op_i	Number of operations for process P_i
Cps	Set of conflicting process pairs, where $(P_i, P_j) \in Cps$ indicates that P_i and P_j cannot execute concurrently
Ca	Set of cores (indexed by C_k), with cardinality m
att	true if attestor executes the scheduler, false otherwise
c_k^{op}	Cost per operation of core $C_k \in C$
c_k^{idle}	Cost per idle time of core C_k (per unit time)
α_{time}	Weight of execution time in the optimization objective
α_{cost}	Weight of operational cost in the optimization objective
Decision Variables:	
$x_{i,k}$	$x_{i,k} = 1$ if P_i is assigned to C_k; $x_{i,k} = 0$ otherwise Each P_i is assigned to exactly one core, so $\sum_{C_k \in Ca} x_{i,k} = 1 \quad \forall P_i \in Prcs$.
s_i	Start time of process P_i ($s_i \geq 0$)
Derived Variables:	
f_i	Finish time f_i of process P_i is calculated as the sum of start time s_i and execution time t_i for each P_i in $Prcs$
$Prcs(C_k)$	Set of processes executing on core C_k, where $Prcs(C_k) = \{P_i \in Prcs \mid x_{i,k}$ is true$\}$
$Idle_k$	Total idle time of core C_k
E_k	Total energy consumption of core C_k

multi-core systems, ensuring correctness while reducing costs. Inspired by FJSS constraints, it dynamically distributes transactions with varying execution times to different cores, allowing concurrent processing of non-conflicting transactions.

Conthereum provides a deterministic, conflict-free mechanism for processing smart contracts on Ethereum, ensuring efficient use of computational resources. The workflow of this solution involves: i) Using non-conflicting transactions and scheduling them for parallel execution. ii) Ensuring sequential execution for dependent or conflicting transactions to maintain state correctness. iii) Optimizing resource allocation based on computational complexity and power consumption models.

Hereafter, we use the term "process" to refer to the execution units derived from public and external function calls within Ethereum transactions described in Section 6.2. Given that transactions can create new smart contracts or invoke existing functions, defining a process as a specific function execution enables a more granular optimization of the transaction execution while maintaining blockchain state correctness.

The Conthereum transaction scheduling problem can be formulated as a constraint programming problem as follows (see Table 6.1 for the complete nomenclature): Given:

- A set of processes $Prcs = \{P_1, \ldots, P_n\}$, where each process P_i, derived from $Txn_{3,4}$, is characterized by its execution time t_i (milliseconds) and operation count op_i (number of operations involved in the process).
- A set of conflicting process pairs $Cps = \{(P_i, P_j) \mid P_i \text{ and } P_j \text{ cannot execute concurrently }\}$.
- A set of cores $Ca = \{C_1, \ldots, C_m\}$: each C_k is defined by its cost per operation c_k^{op} and cost per idle time c_k^{idle}.
- A Boolean variable att, true if the scheduler is executed by an attestor, or false if by a proposer.
- A weight $\alpha_{time} \in [0, 1]$, which represents the relative importance of minimizing execution time in the overall optimization objective. $\alpha_{cost} = 1 - \alpha_{time}$ represents the importance of minimizing cost.

To determine:

- A boolean variable $x_{i,k}$ true if process P_i is executed on core C_k, and false otherwise.
- The starting time s_i (in milliseconds) of process P_i.
- A boolean variable $o_{i,j}$ which is true if process P_i is scheduled before process P_j, and false otherwise, used to enforce ordering constraints under validation mode.

The main goal to meet is:

- Time Efficiency: The makespan, representing the total time span from the start of the earliest process to the end of the latest in all cores, is defined as TE.

$$TE = \max_{C_k \in Ca} \sum_{P_i \in Prcs(C_k)} t_i \tag{6.1}$$

- Power Consumption Efficiency: The total power consumption, factoring in both processing and idle times can be declared as PCE.

$$PCE = \sum_{C_k \in Ca} \left(\sum_{P_i \in Prcs(C_k)} op_i \cdot c_k^{op} + \text{Idle}_k \cdot c_k^{idle} \right) \tag{6.2}$$

- Minimize the total time across cores and operational costs respectively using weighted coefficients.

$$\min \left(\alpha_{time} \cdot TE + \alpha_{cost} \cdot PCE \right) \tag{6.3}$$

Subject to:

- Core Availability Constraint: Each core can only execute one process at any given time, ensuring that no processes are assigned concurrently to a single core. Let $Ca = \{C_1, \ldots, C_m\}$ represent the set of available cores; thus, for any core $C_k \in Ca$, only one process $P_i \in Prcs$ may be assigned without conflicts.

  C1: No Overlap on a Core

$$s_i \geq f_j \vee s_j \geq f_i \quad \forall P_i, P_j \in Prcs, P_i \neq P_j, \\ \text{where } x_{i,k} \wedge x_{j,k} \text{ is true} \tag{6.4}$$

- Conflicting Processes Constraint: For any pair of conflicting processes $(P_i, P_j) \in Cps$, these processes must execute sequentially in any order on any core to avoid conflict. Non-conflicting processes can be scheduled concurrently on different cores.

  C2: No Overlap on Conflicting Processes

$$s_i \geq f_j \vee s_j \geq f_i \quad \forall (P_i, P_j) \in Cps \tag{6.5}$$

  For any $(P_i, P_j) \in Cps$, this constraint enforces that process P_j cannot start execution until process P_i has completed or viseversa, thus ensuring that conflicting processes are executed sequentially possibly on different cores. Here, f_i represents the finish time of process P_i.

  C3: Order Preservation Under Attestor Mode

$$att \Rightarrow (s_i + t_i \leq s_j), \quad \forall (P_i, P_j) \in Cps, \quad i < j \tag{6.6}$$

  This constraint ensures that if the scheduler is executed by an attestor, conflicting processes must follow their predefined order while non-conflicting processes can be reordered freely.

Moreover, the following assumptions are made:

- An external module is utilized to identify conflicting process pairs.
- The execution times and operation counts of the processes are accurately estimated and known before scheduling.
- The core availability data reflect the real-time processing capacities of each core within the multi-core environment and are provided before scheduling.
- Processes are assumed to be indivisible: they cannot be split across multiple cores during execution.
- All cores are assumed to be homogeneous, and each core has the same processing

capabilities.

- The power consumption characteristics of the computing resources remain constant during the scheduling period.
- A core can only work on one process at a time.
- Once started, a process must run to completion.

6.4 Implementation

Since Conthereum's scheduling problem is a variant of FJSS (6.2.2), we began implementing the scheduler using state-of-the-art optimization techniques for this problem. Facing the performance gap in the available solutions (discussed in Section 6.5), we then introduced a greedy algorithm to cope with time constraints demanded by the application. The required implementation for all the approaches is available as open source on github.com/conthereum/conthereum.

Although an optimal solution exists and can be computed given sufficient time, the NP-hard nature of the problem often implies that the required computation exceeds the time needed for sequential execution, making it impractical. Therefore, the primary objective of the Conthereum scheduler is to minimize the *total execution time*, defined as the sum of the wall time (time spent computing a suboptimal schedule) and the makespan (time required to execute all processes in parallel based on the found schedule). Importantly, this total execution time must be less than of serial execution time (refereed as *Conthereum's performance requirement*), with greater reductions yielding higher speedup.

Genetic Algorithm (GA). We first used GA with Tabu Search to solve the optimization problem of Sec. 6.3.3, and we utilized OptaPlanner[1] framework. Following a standard approach (see OptaPlanner documentation), we modeled *hard* (conflict) and *soft* (load balancing) constraints. GA, while non-optimal, produced high-quality schedules, but fell short of Conthereum's performance requirements.

Constraint Programming (CP). We then encoded the optimization problem of Sec. 6.3.3 in OR-Tools[2] using the provided `CpModel`. While CP can yield optimal schedules, its wall time even for suboptimal solutions exceeds the serial execution time which makes it impractical. To speed up the search and accelerate convergence, we added an initializer that efficiently computes a feasible solution (Sec. 6.4.1) that narrows the search space and accelerates the calculation. As listed in table 6.2 CP required

[1]OptaPlanner - `https://www.optaplanner.org/`

[2]Google OR-tools - `https://developers.google.com/optimization`

wall time is significantly longer than the serial execution and its results are useful for benchmarking but not as the final solution for Conthereum scheduler.

Table 6.2: Performance Metrics (3 Cores): OR-Tools vs. Greedy Scheduler

Grp	Procs	Conf. %	OR-Tools		Greedy	
			Wtime (s)	Mkspn (ms)	Wtime (ms)	Mkspn (ms)
1	50	15	7.11	125.00	0.03	126.33
2	50	25	7.11	175.00	0.05	125.67
3	50	35	7.11	124.67	0.05	132.00
4	50	45	7.11	131.33	0.08	128.33
5	100	15	82.19	266.67	0.05	254.00
6	100	25	85.54	318.67	0.06	258.67
7	100	35	80.54	268.33	0.09	257.00
8	100	45	83.83	257.00	0.15	268.00
9	150	15	131.19	523.00	0.09	380.67
10	150	25	131.25	505.33	0.11	384.33
11	150	35	131.19	515.00	0.15	384.00
12	150	45	151.43	564.00	0.19	384.00
13	200	15	141.71	1130.67	0.13	504.67
14	200	25	142.04	1235.67	0.18	507.00
15	200	35	142.13	1449.00	0.22	508.33
16	200	45	130.38	1502.33	0.30	515.67

Existing Implementations. We also analyzed existing implementations [180]. Although this code base reflect a more relaxed version of our problem and lacked conflicting processes constraint, the execution performance failed to meet Conthereum's performance requirements.

6.4.1 Proposed Outperforming Greedy Iterative Algorithm

Building on insights from CP, GA, and existing scheduling implementations, we introduce the Conthereum Scheduling Algorithm, a *conflict-aware iterative greedy approach* with *constraint-driven resource allocation* specifically designed for Ethereum transaction scheduling. This method prioritizes transactions based on conflict properties (count and duration) and assigns them iteratively to the earliest available cores while preserving constraints. It first accommodates feasible transactions with no idle time and then assigns the previously skipped transactions by considering the minimal idle time.

The pseudocode of the proposed greedy approach is reported in Algorithm 6. The key elements are as follows:

- **Strategy**: Defined in line 1, this structure specifies the heuristic configurations for the algorithm.
  - `sortType` determines the priority order of processes, which can be: First In First Out (FIFO), Most Conflicting Count First (MCCF), Most Conflicting

Algorithm 6 Conthereum Scheduler

```
1: Structure: Strategy
2:    sortType ∈ {FIFO, MCCF, MCDF, LCCF, LCDF}
3:    assignType ∈ {LOOSE, STRICT}
4:    looseReviewRound ∈ ℕ
5: Input: facts, strategy
6: horizon ← 0
7: computingPlan ← new ComputingPlan(fact)
8: facts.sortProcesses(strategy.sortType, facts.isAttestor)
9: if strategy.assignType = STRICT then
10:    for each process in facts.processes do
11:       horizon ← horizon + process.executionTime
12:       computingPlan.assignStrictly(process, facts.isAttestor)
13: else if strategy.assignType = LOOSE then
14:    for round ← 0 to strategy.looseReviewRound do
15:       unassignedProcesses ← 0
16:       for each process in facts.processes do
17:          if round = 0 then
18:             horizon ← horizon + process.executionTime
19:          if process.core = null then
20:             couldAssign ← computingPlan.assignLoosely(process, facts.isAttestor)
21:             if not couldAssign then
22:                unassignedProcesses ← unassignedProcesses + 1
23:       if unassignedProcesses = 0 then
24:          break
25:    for each process in facts.processes do
26:       if process.core = null then
27:          computingPlan.assignStrictly(process, facts.isAttestor)
28: output.processes ← facts.processes
29: output.horizon ← horizon
30: output.scheduleMakespan ← computingPlan.getScheduleMakespan()
31: output.wallTimeInMs ← current function execution time
32: return output
```

Duration First (MCDF), Least Conflicting Count First (LCCF), or Least Conflicting Duration First (LCDF). In MCDF and LCDF, processes are ranked based on the cumulative duration of conflicts with other processes, in descending and ascending order, respectively. In MCCF and LCCF, priority is determined by the number of conflicting transactions instead of conflict duration.

  - `assignType` can be either `LOOSE` or `STRICT`, determining the assignment strategy as described in the following sections.

- **Initialization**: The algorithm begins by taking `facts` as input, is a data structure which includes processes, available cores, and the specified `strategy`. The variable `horizon` is initialized with 0, and accumulates the time corresponding to the makespan of serial execution of all transactions. Later, it is used to calculate the speedup factor.

- **Sorting Phase**: In line 8, transactions are sorted according to the specified `sortType`, which determines the order in which processes will be assigned in subsequent steps. The sorting also depends on whether the scheduler is executed by the initial proposer or the later attestors of the proposed block, as indicated by `facts.isAttestor`. The embedded algorithm in `sortProcesses` works as follows: if `facts.isAttestor` is false, a regular sort is applied to the processes based on `sortType`. If `facts.isAttestor` is true, all conflicting transactions that can affect each other by changing their order are collected at the beginning of the process list while preserving their initial order. The remaining non-conflicting transactions are then added in their original order. Although these non-conflicting transactions can theoretically be executed in different orders, since our `sortType`s are either `FIFO` or conflict-based, applying any of these sorts does not alter the list, and we can skip sorting them.

- **Strict Assignment Strategy**: If the strategy is set to `STRICT`, the algorithm directly applies the `assignStrictly` method (line 12). This method greedily assigns transactions to the least occupied core. In case of a conflict, minimal idle time is added to the assigned core to resolve the conflict before scheduling the process. The assigned core and conflict-free start time are updated in the process object accordingly. The `assignLoosely` method accepts `isAttestor` as an input parameter and is designed to respect the transaction order if `isAttestor` is set to `1`. Specifically, it ensures that if the input transaction has any conflicting

processes, all its preceding conflicting transactions-based on their original order- are already assigned.

- **Loose Assignment Strategy**: If the strategy is set to `LOOSE`, the process follows a two-step approach: an initial loose assignment followed by a strict assignment for any remaining processes. The loose assignment phase iterates up to `looseReviewRound` times (line 14), attempting to assign transactions using the `assignLoosely` method (line 20). This method attempts to schedule a transaction on the least occupied core only if no conflicts arise, returning `true` upon a successful assignment. This method is also has `isAttestor` input variable and its functionality is the same as explained for `assignStrictly` method. Regardless of the value of `isAttestor`, the method checks for conflicts. If a conflict is detected, the transaction remains unassigned. Importantly, `assignLoosely` does not introduce idle time to resolve conflicts. If the method is unable to assign the transaction, it returns `false`; otherwise, it returns `true`.

  Since unassigned transactions are revisited in each `looseReviewRound`, they may get assigned in later iterations as the state of core assignments evolves. This phase maximizes the number of assignments without introducing idle time. If all transactions are assigned before reaching the maximum number of rounds, the algorithm terminates early (line 24).

  In the second sub-step of the `LOOSE` strategy, any remaining unassigned transactions are handled using the `assignStrictly` method (line 25), ensuring that all transactions are ultimately scheduled.

- **Output**: The algorithm returns an output object containing: 1) `processes`: where each process is assigned an execution core and a start time; 2) `horizon`: The estimated makespan in a serial execution model; 3) `scheduleMakespan`: the maximum occupied time across all cores, including both process execution and idle periods; 4) `wallTimeInMs`: the actual execution time taken by the scheduling function:

This approach aims to balance execution efficiency and computational feasibility, offering a scalable solution for Ethereum's concurrent execution model.

6.4.2 Complexity

This section analyzes the runtime complexity of the scheduler. Conflict detection and cost analysis are performed offline and thus excluded from complexity calculations. These modules respectively provide t_i (execution time of process P_i) and conflict sets Cps, which are inputs to the scheduling algorithm.

Each block contains n processes. To determine conflicts, we generate all $O(n^2)$ process pairs and perform constant-time lookups in the conflict repository. Sorting the processes requires $O(n \log n)$ time.

The most effective assignment strategy, (`LOOSE`), comprises two distinct phases: the loose and the strict assignment phases. In the worst-case scenario, the loose phase results in the allocation of only a single, highly conflicting, and large process, leading to all `looseReviewRound` iterations failing to allocate any additional process, incurring a cost of $O(n \times \texttt{looseReviewRound})$. During the strict phase, the remaining processes are allocated to m cores, with a cost of $O(n \times m)$. Consequently, the overall complexity of the scheduler is $O(n^2)$.

6.4.3 Conflict-Aware Upper Bound Calculation

To establish the most realistic upper bound for the schedule makespan in the presence of transaction conflicts, we model the scheduling problem using conflict graphs and analyze the scheduling implications under a given conflict rate.

Model and Assumptions

We consider n processes to be scheduled on m cores, each with an average execution time $\bar{t}$. The conflict rate $cr \in [0, 1]$ denotes the probability that any two processes conflict. We represent the conflict structure as a graph $G = (V, E)$, where $|V| = n$ and edges $(v_i, v_j) \in E$ indicate conflicting process pairs that cannot run concurrently. The conflict graph is modeled as an Erdős-Rényi random graph $G(n, cr)$.

Chromatic Number and Scheduling Layers

The chromatic number $\chi(G)$ approximates the number of scheduling layers needed. For the Erdős-Rényi random graph $G(n, cr)$, it is asymptotically approximated as [181]:

$$\chi(G) \approx \frac{n}{2 \log_{1/(1-cr)} n}. \tag{6.7}$$

Each color class represents a set of mutually non-conflicting processes executable in parallel. Thus, the number of sequential layers required is at least $\chi(G)$.

Upper Bound on Runtime

Assuming a uniform average process time $\bar{t}$, the upper bound runtime under conflict-aware parallel execution is:

$$UB_{cr} = \bar{t} \cdot \left\lceil \frac{\chi(G)}{m} \right\rceil = \bar{t} \cdot \left\lceil \frac{n}{2m \log_{1/(1-cr)} n} \right\rceil \tag{6.8}$$

This captures the best-case runtime possible under a given conflict structure and core limit.

Boundary Analysis

No Conflicts ($cr = 0$) The conflict graph has no edges, and all processes are independent. Thus:

$$UB_0 = \bar{t} \cdot \left\lceil \frac{n}{m} \right\rceil \tag{6.9}$$

Full Conflicts ($cr = 1$) The graph is complete; all processes conflict. Hence $\chi(G) = n$ and only one process can run at a time:

$$UB_1 = \bar{t} \cdot n \tag{6.10}$$

Interpretation

As cr increases, the chromatic number $\chi(G)$ increases, requiring more sequential scheduling layers and reducing parallelism. Although idealized, this bound offers a tight benchmark for evaluating practical schedulers under conflict constraints.

6.5 Experimental Evaluation

This section reports the experimental results obtained from the implementations presented in the previous section (6.4).

Environment. Two environments are used to benchmark scheduling quality. The first is a high-performance server (Intel i9-10940X, 28 cores, 258 GB RAM), and the

second is a standard laptop (Intel i5-1135G7, 8 cores, 16 GB RAM). Except for the data reported for OR-Tools in Table 6.2, which is collected on the high-performance server to facilitate optimal solution generation, all other experiments are conducted on the standard laptop to emulate typical validator conditions.

Benchmark Data. The dataset consists of transaction sets ranging from 50 to 200, based on features extracted from Ethereum transactions on the network [182]. For each transaction count, multiple datasets with different conflict percentages are used to reflect the proportion of conflicting transactions, which can impact overall scheduling efficiency. The conflict rates vary from 15% to 45%, divided into four groups (15, 25, 35, and 45), derived from the empirical evaluation of Ethereum transactions conducted in [125]. Transactions are scheduled for execution on 1 to 32 cores, following the configurations used in prior work [103, 104, 125, 126, 129], to enable performance comparison.

Table 6.3: Performance Metrics - Proposers and Attestors Speedups

Data			**2 cores**		**4 cores**		**8 cores**		**16 cores**		**32 cores**	
Group No.	**Process Count**	**Conflict %**	**proposer**	**attestor**	**proposer**	**attestor**	**proposer**	**attestor**	**proposer**	**attestor**	**proposer**	**attestor**
1	50	15	1.99	1.88	3.89	3.14	7.08	4.73	7.52	6.38	10.33	8.08
2	50	25	1.98	1.77	3.84	2.70	5.50	3.77	6.13	4.89	8.13	6.17
3	50	35	1.98	1.65	3.65	2.34	3.70	3.14	5.17	4.12	7.01	5.09
4	50	45	1.97	1.53	3.35	2.07	3.25	2.72	4.67	3.53	6.41	4.39
5	100	15	1.99	1.88	3.89	3.14	7.08	4.73	7.52	6.38	10.33	8.08
6	100	25	1.98	1.77	3.84	2.70	5.50	3.77	6.13	4.89	8.13	6.17
7	100	35	1.98	1.65	3.65	2.34	3.70	3.14	5.17	4.12	7.01	5.09
8	100	45	1.97	1.53	3.35	2.07	3.25	2.72	4.67	3.53	6.41	4.39
9	150	15	1.99	1.88	3.89	3.14	7.08	4.73	7.52	6.38	10.33	8.08
10	150	25	1.98	1.77	3.84	2.70	5.50	3.77	6.13	4.89	8.13	6.17
11	150	35	1.98	1.65	3.65	2.34	3.70	3.14	5.17	4.12	7.01	5.09
12	150	45	1.97	1.53	3.35	2.07	3.25	2.72	4.67	3.53	6.41	4.39
13	200	15	1.99	1.88	3.89	3.14	7.08	4.73	7.52	6.38	10.33	8.08
14	200	25	1.98	1.77	3.84	2.70	5.50	3.77	6.13	4.89	8.13	6.17
15	200	35	1.98	1.65	3.65	2.34	3.70	3.14	5.17	4.12	7.01	5.09
16	200	45	1.97	1.53	3.35	2.07	3.25	2.72	4.67	3.53	6.41	4.39
AVG:	**125**	**30**	**1.98**	**1.71**	**3.68**	**2.56**	**4.88**	**3.59**	**5.87**	**4.73**	**7.97**	**5.93**

Benchmark Results. Table 6.3 presents the speedup results of the proposed algorithm in both proposer and attestor modes, for selected core counts for brevity. Each row corresponds to the average result of a distinct `group` of three scheduling

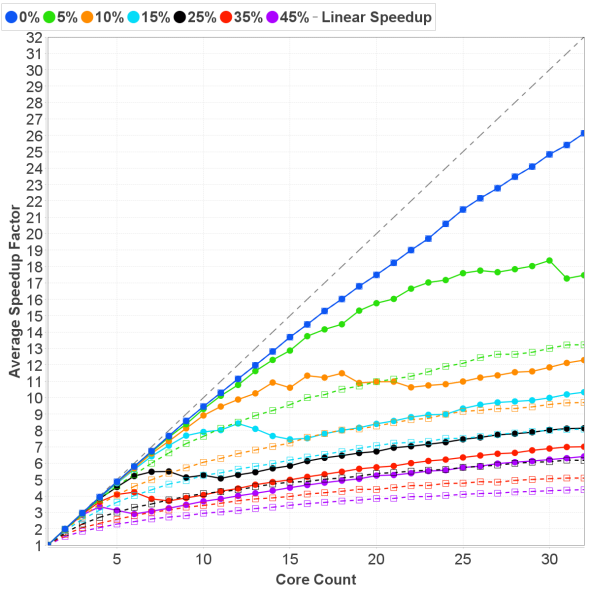
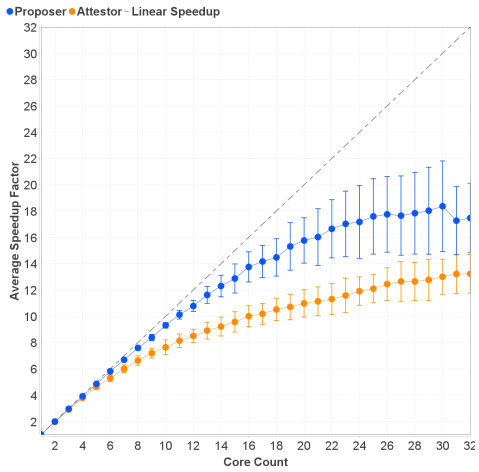


Figure 6.2: Speedup Factor vs. Core Count - comprehensive

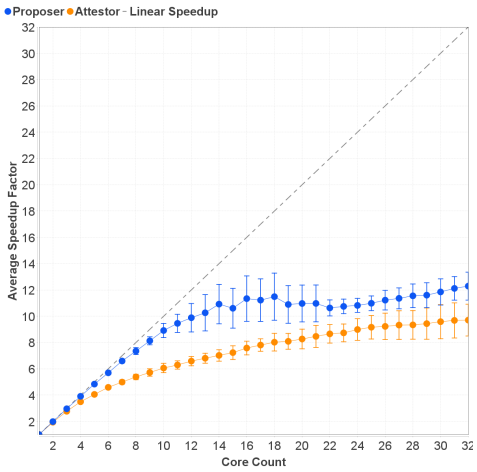
`instances`, characterized by the same parameters and a unique random seed (values 1, 2, and 3). This ensures dataset reproducibility, reliability, and reduces randomness bias. For all samples, the time weight is set to 100%, prioritizing time minimization while disregarding energy efficiency, enabling direct comparison with related works in this domain.

Table 6.2 compares the wall time and makespan between the OR-Tools CP implementation (on the server environment) and the greedy iterative heuristic implementation (on the standard system). As observed, the greedy implementation significantly reduces wall time-from 7 to 130 seconds down to 0.03 to 0.3 milliseconds. While makespan is also improved, the most substantial gains are seen in wall time. These results confirm that our greedy algorithm achieves near-optimal makespans within milliseconds on standard hardware, demonstrating its real-world applicability.

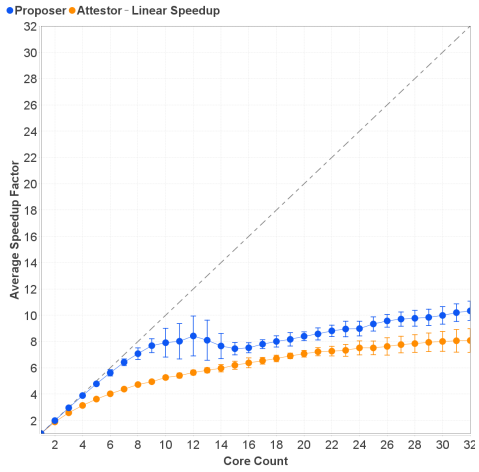
Figure 6.2 illustrates comprehensive results of all experiments, showing how average speedup evolves with increasing core counts from 1 to 32. Each color represents a distinct conflict rate among datasets, with solid and dashed lines indicating proposer and attestor modes, respectively. As expected, for all conflict levels, proposers consistently achieve higher speedups than attestors because of their flexibility of reordering the transactions-except in the 0% conflict case, where performance is identical. Overall speedup improves with more cores, but degrades with increasing conflict rates. This degradation becomes more pronounced at higher conflict levels, where a majority of processes face contention, leaving many cores underutilized. The well-known



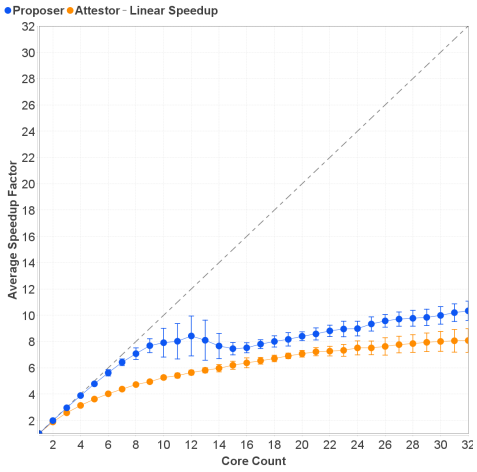
(a) 5% Conflict



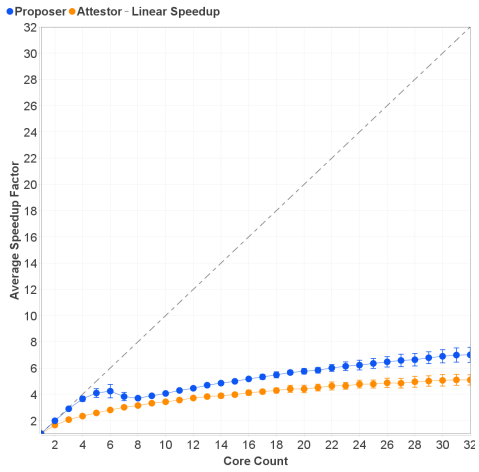
(b) 10% Conflict



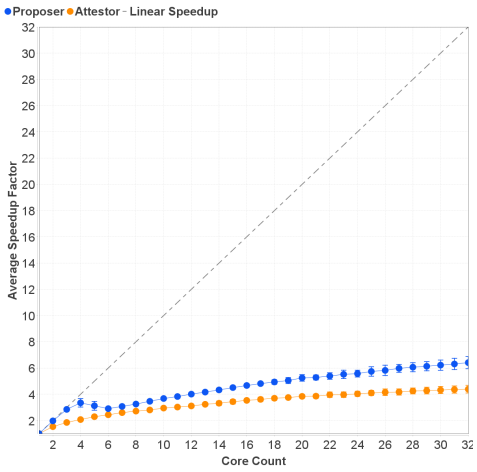
(c) 15% Conflict



(d) 25% Conflict



(e) 35% Conflict



(f) 45% Conflict

Figure 6.3: Speedup vs Core Count

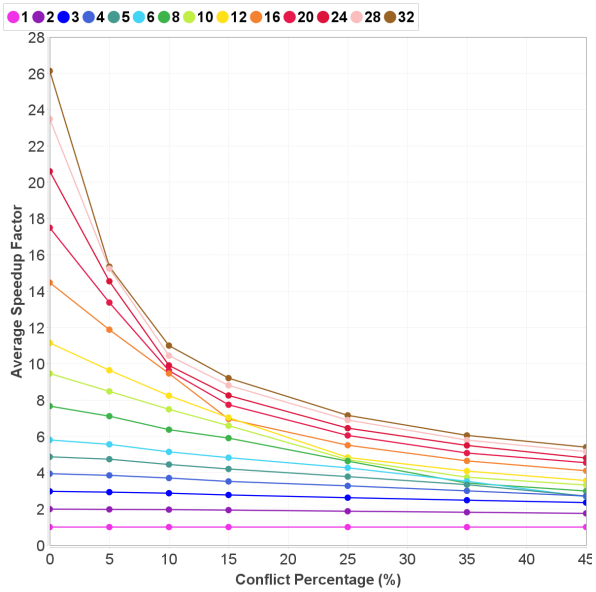


Figure 6.4: Conflict vs. Speedup.

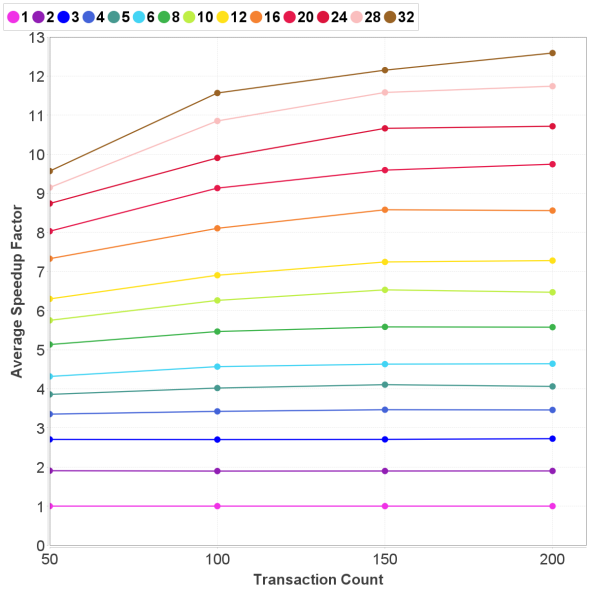


Figure 6.5: Transaction Count vs. Speedup.

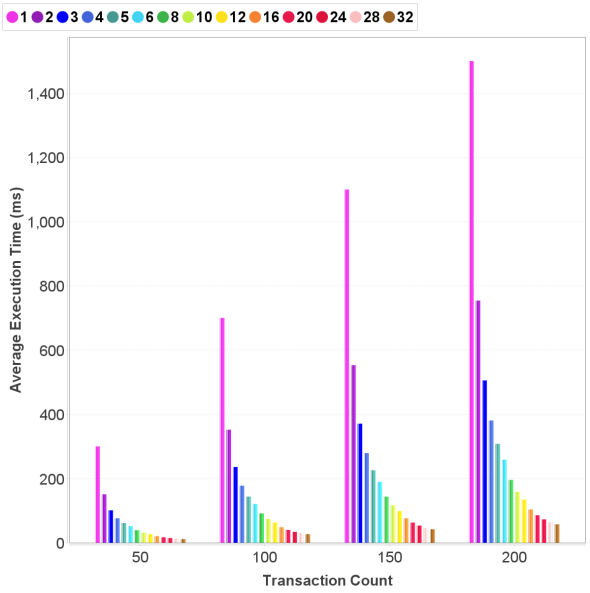


Figure 6.6: Parallel vs. Serial Execution Time.

non-monotonic speedup phenomenon is also observed, where speedup declines with additional cores due to load imbalance [183]. Another key observation from this diagram is that the trend remains approximately linear up to 8-core parallelism which is more distinguishable in individual diagrams (Figure 6.3a, 6.3b, 6.3c, 6.3d, 6.3e, and 6.3f).

Figure 6.4 illustrates how average speedup declines with increasing conflict rate. This decline is most significant in high-core parallelism settings, whereas lower-core configurations maintain nearly linear speedup regardless of conflict level. Figure 6.5 shows a modest improvement in speedup as the transaction count increases, particularly in parallelism levels above 10 cores, though the gains are not drastic. Figure 6.6 shows average execution time across core counts (1–32) and transaction group sizes, highlighting Conthereum's effectiveness in accelerating transaction processing. Further benchmark results including the solutions that excluded from the main solution is available in Conthereum git documentations.

6.6 Related Work

This section reviews recent literature on smart contracts concurrency, identifying concurrency layers in blockchain. It also assesses the compatibility of Conthereum with existing approaches or its need for exclusive adoption, followed by a performance comparison in the later case.

A central challenge in smart contract concurrency lies in intra-block parallelism,

aiming to improve transaction throughput beyond sequential execution. Speculative concurrency, as proposed in [103] parallelize execution based on optimistic conflict assumptions and enhance the parallelism significantly specially at low conflict rates, it suffers from rollback overhead and performance degradation under rising conflict rates, as shown in empirical studies [125]. Enhancements such as conflict abstractions and shadow speculation [126] have improved speculative methods by better managing concurrent updates.

Block-STM [129], as the current state-of-the-art speculative approach, uses *dependency estimation* instead of explicitly computing transaction dependencies and executing them via a fork-join schedule. It does not precompute dependencies; rather, for each transaction, it treats the write-set of an aborted incarnation (i.e., each execution attempt of a transaction) as an estimate for the write-set of its next incarnation, helps reduce the transaction abort rate.

Conthereum takes a pessimistic approach, pre-analyzing contracts to avoid conflicts entirely-unlike speculative methods that resolve them at runtime. This enables faster, rollback-free execution, positioning Conthereum for future scalability as conflicts grow more frequent [125]. The comparative analysis indicates that Conthereum demonstrates the highest recorded speedup, achieving (2.93, 2.48) with 3 cores, surpassing the results of [103], which reported (1.33, 1.69) for the same core count. Notably, Conthereum's proposer speedup of 2.93 with just 3 cores exceeds the 16-core and 64-core speedups of [125], which reported 1.13 and 2.26, respectively, without attestor evaluation. Additionally, [104] did not examine attestor performance but tested proposers ranging from 4 to 24 cores, reporting a speedup of 3 for 4 cores and 5 for 24 cores. These values are outperformed by Conthereum, which achieves a proposer speedup of 3.73 for 4 cores and 5.81 for only 9 cores. Block-STM [129] has demonstrated performance improvements of up to 20× in low-contention workloads and up to 9× in high-contention scenarios compared to sequential execution. However, the exact block size, conflict rate, and execution modes (e.g., proposer or attestor) are not clearly specified in their study, making precise comparisons challenging. Conthereum's performance using 32 cores showing speedups of 25.92 under no-conflict conditions and 9.08 under 15% conflict-both higher than Block-STM's reported 20× and 9×, respectively.

Apart from inter-block concurrency, several other solutions have been proposed to enhance blockchain throughput by introducing concurrency across different domains. These include *Sharding* [117], *Off-chain concurrency solutions* [139, 184], and *Concurrent Consensus mechanisms* [107, 109, 110]. While all three approaches significantly enhance network throughput, they can perform along with the intra-block concurrency

without introducing conflicts.

6.7 Conclusion and Future Work

This paper introduced Conthereum, an optimized scheduling framework for concurrent transaction execution in Ethereum. It proposes a novel greedy iterative heuristic that ensures state consistency by avoiding transaction conflicts while maximizing multi-core utilization. Experimental results demonstrate near-linear throughput speedup up to 8 cores, with continued gains beyond, outperforming existing intra-block concurrency methods. Conthereum is also extensible to support cross-cutting concerns such as energy-aware scheduling. While developed for Ethereum, the framework is applicable to permissioned blockchains such as Hyperledger Fabric.

Future work includes refining the greedy heuristic for improved efficiency on higher-core systems and validating energy-aware scheduling extensions. We also plan to explore EVM integration and assess OS-level enhancements for optimized multi-core scheduling. While the algorithm outperformed all recent solutions using benchmarks incorporating features from prior works to ensure fair comparison, a dedicated empirical study would be beneficial. Although Conthereum is theoretically adaptable to other blockchain platforms, platform-specific experiments are needed to confirm performance benefits. Finally, our scheduler outperforms existing FJSS implementations, highlighting its potential for general-purpose JSS applications.

Chapter 7

Conclusion and Future Work

7.1 Main Conclusions

This doctoral thesis has explored and addresses numerous critical aspects of security, and scalability which has emerged with new blockchain systems. Through a series of contributions, we have addressed an emerged blockchain ecosystem Cyber attack named cryptojacking to the formal verification of smart contracts and the optimization of transaction scheduling for enhanced Ethereum throughput.

7.1.1 Summary of Contributions

Our contributions are structured around three main themes:

- **Security in Blockchain Systems by Cryptojacking malware detection (CryptojackingTrap initiative):** We introduced novel methods for detecting cryptojacking malware, named CryptojackingTrap system. This involved using dynamic analysis and introducing and recognition of a novel pattern to identify and mitigate malicious activities that exploit users infrastructure for unauthorized cryptocurrency mining. Our approach provides a significant step forward in safeguarding within the blockchain ecosystem.
- **Smart Contract Security using formal Methods (VeriSolid enhancements):** We delved into the critical area of smart contract security, emphasizing the application of formal methods. Our work on analyzing and enhancing VeriSolid, demonstrated how rigorous mathematical techniques can be employed to formally verify the correctness and security properties of smart contracts. This is crucial for preventing vulnerabilities that could lead to significant financial

losses or system failures, offering a higher degree of assurance compared to traditional testing methods.

- **Scalability in Blockchain Infrastructure Using Transaction Scheduling (Conthereum initiatives):** Addressing the inherent scalability limitations of blockchain, particularly Ethereum, we proposed and developed Conthereum. This solution focuses on optimizing intra-block transaction scheduling through static analysis and advanced optimization techniques. By enabling efficient parallel execution of transactions, Conthereum significantly enhances throughput and reduces latency, paving the way for more performant and widely adoptable decentralized applications.

Each of these contributions, while distinct, collectively advances the state-of-the-art in blockchain technology by tackling fundamental challenges in its security, and scalability. We have shown that a multi-faceted approach, combining empirical analysis, formal verification, and algorithmic optimization, is essential for building resilient and high-performance systems in Blockchain domain.

7.1.2 Synthesis of Findings

This thesis has approached the broader challenges of blockchain systems which they are foundational tensions in the blockchain trilemma (security, decentralization, scalability). The thesis has three interrelated dimensions: protection against external threats, assurance of internal correctness, and enhancement of execution efficiency.

In Part I, the proposed Cryptojacking Trap addresses the increasingly evasive nature of cryptojacking malware, contributing to the protection of computational integrity in decentralized infrastructures. In Part II, the VeriSolid Enhancement extends formal verification tools to support correct-by-design development of smart contracts, addressing the critical need for secure and verifiable code in immutable systems. Finally, Part III tackles the scalability limitations of current blockchain architectures. Through a comprehensive study of concurrency, a conflict-detection framework, and the Conthereum proposal for concurrent transaction execution, this part outlines a practical path toward more efficient blockchain performance. While each part addresses a different technical concern, collectively they form a cohesive contribution toward the sustainable evolution of blockchain technology.

7.1.3 Practical Implications

The practical implications of this thesis are significant for various stakeholders in the blockchain space:

- **For Cybersecurity service providers:** The innovative *CryptojackingTrap* approach offers an evasion-resilient method for detecting cryptojacking malware. It can significantly support cybersecurity service providers by enhancing their detection capabilities, particularly in identifying zero-day attacks. Moreover, it enables the refinement of signature and blacklist databases, facilitating more lightweight and efficient detection mechanisms on the end-user side.

- **For Developers:** The methodologies and tools presented, such as VeriSolid provide developers with powerful instruments to write more secure and efficient smart contracts. By integrating formal verification developers can reduce the likelihood of bugs and vulnerabilities, leading to more reliable decentralized applications.

- **For Blockchain Operators/Validators:** Conthereum offers a tangible solution for improving the throughput of blockchain networks. Validators can leverage our scheduling algorithms to process more transactions per second, leading to reduced operational costs and enhanced network capacity. This is particularly relevant for high-volume applications and the broader adoption of blockchain technology.

- **For Users:** Ultimately, the advancements in reliability, security, and scalability translate into a more trustworthy and responsive blockchain experience for end-users. Reduced risks of malware, more secure smart contract interactions, and faster transaction confirmations contribute to a more positive and confident engagement with decentralized platforms.

- **For Researchers:** The thesis identifies several promising avenues for future research, particularly in the intersection of dynamic analysis, formal methods, static analysis, and performance optimization. It provides a strong foundation for further exploration into advanced scheduling algorithms, more comprehensive formal verification techniques, and novel approaches to cybersecurity in decentralized environments.

7.1.4 Future Directions

Building upon the work presented in this thesis, several exciting directions for future research emerge. The items below are numbered using a two-level scheme $i.j$, where i indicates the chapter and j denotes the jth future work within that chapter:

2.1 **AI/ML for Enhanced Threat Detection:** Further exploration into the application of advanced AI and Machine Learning techniques for cryptojacking detection and other blockchain-specific threats. This could involve developing predictive models that identify anomalous behavior patterns indicative of malicious activity with greater accuracy and speed.

3.1 **Integration of Advanced nuXmv Features through LTL Encoding:** A promising direction is to extend the current verification capabilities by encoding property patterns directly in Linear Temporal Logic (LTL). This would allow VeriSolid to leverage advanced SAT-based and Bounded Model Checking (BMC) algorithms available in nuXmv, enabling deeper temporal reasoning and improved detection of behavioral inconsistencies in smart contracts.

3.2 **Support for Infinite-State Verification via SMT Solvers:** The current verification process is limited to Boolean variables. Future work could introduce support for infinite-state variables such as integers and real numbers within guards and state transitions. This enhancement, enabled by integrating SMT-based verification in nuXmv, would significantly increase the expressiveness and applicability of VeriSolid to complex smart contracts with arithmetic and quantitative logic.

3.3 **Automated Deadlock-Freedom Checking:** Extending VeriSolid to automatically verify deadlock-freedom properties is a critical enhancement. By ensuring that every reachable state has at least one successor, the tool can guarantee system liveness and prevent scenarios of execution halting, thereby strengthening the overall correctness assurance of the generated smart contracts.

3.4 **Counterexample Generation and User-Level Visualization:** Reintroducing counterexample generation and providing intuitive visualization at the user level would enhance the usability and educational value of VeriSolid. This feature would help developers better understand and debug specification violations by tracing the precise execution path that leads to a failed property.

3.5 **Granular Lock Mechanism for Reentrancy Prevention:** The existing global locking mechanism inherited from FSolidM effectively prevents reentrancy but is overly restrictive, as it enforces exclusive execution for the entire contract. Future improvements could focus on implementing finer-grained locks, allowing distinct locks per transition or method. Additionally, integrating user-configurable lock definitions within the VeriSolid interface would enable customized concurrency control, ensuring both safety and efficiency in multi-transaction smart contract environments.

5.1 **Large-Scale Empirical Evaluation of Conflict Patterns:** Future work includes conducting larger-scale empirical studies using this solution to gain deeper insights into conflict patterns, statistical distributions, and temporal trends of potential conflicts among smart contracts and their transactions. This analysis would help in understanding how conflicts evolve over time and across different contract categories, guiding further refinement of detection and mitigation strategies.

6.1 **Heuristic Refinement for High-Core Systems:** As multi-core processors continue to scale, refining the greedy iterative heuristic to maintain efficiency and load balance across a larger number of cores is a key direction. This could involve integrating adaptive partitioning strategies or hybrid heuristics that combine greedy and metaheuristic approaches to sustain scalability and reduce scheduling overhead.

6.2 **Energy-Aware Scheduling in Conthereum:** While we touched upon energy efficiency in the formalization of Conthereum, a dedicated experimental evaluation of its impact on power consumption and the development of more sophisticated energy-aware scheduling policies would be a valuable extension.

6.3 **EVM Integration and Evaluation:** Extending Conthereum with direct EVM (Ethereum Virtual Machine) integration would allow the framework to interact seamlessly with Ethereum's runtime environment. This would enable end-to-end evaluation on real-world smart contract workloads, providing deeper insights into its performance, gas efficiency, and transaction validation consistency.

6.4 **Operating System-Level Scheduling Enhancements:** Exploring OS-level scheduling optimizations can complement Conthereum's algorithmic scheduling. Integrating thread affinity, CPU pinning, or kernel-assisted task prioritization

mechanisms could improve cache locality and minimize context-switching overhead, leading to further performance gains in concurrent transaction execution.

6.5 **Dedicated Empirical Study for Comparative Evaluation:** Although benchmark-based evaluations provided strong evidence of Conthereum's superiority, conducting a large-scale empirical study using heterogeneous datasets and network configurations would yield more comprehensive insights. This would also help in identifying workload-dependent behaviors and fine-tuning the scheduling model for broader deployment scenarios.

6.6 **Generalization of Conthereum for other Blockchain:** The insights gained from adapting Conthereum for Ethereum blockchain transaction scheduling could be generalized to other blockchain throughput enhancement, opening new avenues for optimization.

6.7 **General-Purpose Job Shop Scheduling Applications:** Given that Conthereum's scheduling mechanism outperforms conventional Flexible Job Shop Scheduling (FJSS) approaches, future research could focus on adapting it for non-blockchain domains. Examples include manufacturing, cloud task orchestration, or data center workload scheduling, where concurrency, resource efficiency, and state consistency are equally critical.

6.8 **Dynamic Adaptation in Conthereum:** While Conthereum utilizes static analysis for conflict detection, future work could explore dynamic adaptation mechanisms. This would involve real-time monitoring of network conditions and transaction patterns to dynamically adjust scheduling strategies, further optimizing performance under varying loads and conflict rates.

Bibliography

[1] Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. Cryptojackingtrap: An evasion resilient nature-inspired algorithm to detect cryptojacking malware. *IEEE Transactions on Information Forensics and Security*, 19:7465–7477, 2024. doi: 10.1109/TIFS.2024.3353072.

[2] Atefeh Zareh Chahoki and Marco Roveri. Static analysis for detecting transaction conflicts in ethereum smart contracts. In *Proceedings of the 2025 8th International Conference on Blockchain Technology and Applications*, ICBTA '25, New York, NY, USA, 2025. Association for Computing Machinery.

[3] Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. Conthereum: Concurrent ethereum optimized transaction scheduling for multi-core execution. In *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2025.

[4] Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. Unmasking the unseen: Cryptojackingtrap, the most resilient evasion-proof cryptojacking malware detection algorithm. In *2024 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–4. IEEE, 2024.

[5] Atefeh Zareh Chahoki, Marco Roveri, Daniel Amyot, and John Mylopoulos. Revisiting formal verification in verisolid: An analysis and enhancements. In *CEUR Workshop Proceedings*, volume 3629, pages 55–60. CEUR-WS, 2023.

[6] Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. Sok: Concurrency in blockchain–a systematic literature review and the unveiling of a misconception. *arXiv e-prints*, pages arXiv–2506, 2025. URL `https://arxiv.org/pdf/2506.01885`.

[7] Bitcoinity Authors. Blockchain blocks version - bitcoinity.org, 2023. URL `https://data.bitcoinity.org/bitcoin/block_version`.

[8] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

[9] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013. URL `https://github.com/ethereum/wiki/wiki/White-Paper`.

[10] Houssain Kettani and Polly Wainwright. On the top threats to cyber systems. In *2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT)*, pages 175–179, 2019. doi: 10.1109/INFOCT.2019.8711324. URL `https://doi.org/10.1109/INFOCT.2019.8711324`.

[11] Danny Yuxing Huang, Hitesh Dharmdasani, Sarah Meiklejohn, Vacha Dave, Chris Grier, Damon McCoy, Stefan Savage, Nicholas Weaver, Alex C. Snoeren, and Kirill Levchenko. Botcoin: Monetizing stolen cycles. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, pages 1–16, 2014. URL `https://www.ndss-symposium.org/ndss2014/botcoin-monetizing-stolen-cycles`.

[12] Sheharbano Khattak, Naurin Rasheed Ramay, Kamran Riaz Khan, Affan A. Syed, and Syed Ali Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE Communications Surveys & Tutorials*, 16(2):898–924, 2014. doi: 10.1109/SURV.2013.091213.00134. URL `https://doi.org/10.1109/SURV.2013.091213.00134`.

[13] Ryan Naraine. Researchers find malware rigged with bitcoin miner, 2011. URL `https://www.zdnet.com/article/researchers-find-malware-rigged-with-bitcoin-miner/`.

[14] Atefeh Zareh and Hamid Reza Shahriari. Botcointrap: Detection of bitcoin miner botnet using host based approach. In *2018 15th International ISC (Iranian Society of Cryptology) Conference on Information Security and Cryptology (ISCISC)*, pages 1–6, 2018. doi: 10.1109/ISCISC.2018.8546867. URL `https://doi.org/10.1109/ISCISC.2018.8546867`.

[15] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1714–1730, New York, NY, USA, 2018. Association for

Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243858. URL https://doi.org/10.1145/3243734.3243858.

[16] Dieter Hodick and Andreas Sievers. On the mechanism of trap closure of venus flytrap (dionaea muscipula ellis). *Planta*, 179:32–42, 1989. doi: 10.1007/BF00395768. URL https://doi.org/10.1007/BF00395768. Publisher: Springer.

[17] Aaron Zimba, Zhaoshun Wang, and Mwenge Mulenga. Cryptojacking injection: A paradigm shift to cryptocurrency-based web-centric internet attacks. *Journal of Organizational Computing and Electronic Commerce*, 29(1):40–59, 2019. doi: 10.1080/10919392.2019.1552747. URL https://doi.org/10.1080/10919392.2019.1552747.

[18] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1701–1713, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243840. URL https://doi.org/10.1145/3243734.3243840.

[19] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. Thieves in the browser: Web-based cryptojacking in the wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450371643. doi: 10.1145/3339252.3339261. URL https://doi.org/10.1145/3339252.3339261.

[20] Rafael A. Rodríguez-Gómez, Gabriel Maciá-Fernández, and Pedro García-Teodoro. Survey and taxonomy of botnet research through life-cycle. *ACM Comput. Surv.*, 45(4), aug 2013. ISSN 0360-0300. doi: 10.1145/2501654.2501659. URL https://doi.org/10.1145/2501654.2501659.

[21] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, 2007. doi: 10.1109/ACSAC.2007.21. URL https://doi.org/10.1109/ACSAC.2007.21.

[22] Rashid Tahir, Muhammad Huzaifa, Anupam Das, Mohammad Ahmad, Carl Gunter, Fareed Zaffar, Matthew Caesar, and Nikita Borisov. Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 287–310. Springer International Publishing, 2017. ISBN 978-3-319-66332-6. doi: 10.1007/978-3-319-66332-6_13. URL https://doi.org/10.1007/978-3-319-66332-6_13.

[23] Intel Corporation. Pin - a dynamic binary instrumentation tool, 2023. URL https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065034. URL https://doi.org/10.1145/1065010.1065034.

[25] Said Varlioglu, Bilal Gonen, Murat Ozer, and Mehmet Bastug. Is cryptojacking dead after coinhive shutdown? In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, pages 385–389, 2020. doi: 10.1109/ICICT50521.2020.00068. URL https://doi.org/10.1109/ICICT50521.2020.00068.

[26] Keshani Jayasinghe and Guhanathan Poravi. A survey of attack instances of cryptojacking targeting cloud infrastructure. In *Proceedings of the 2020 2nd Asia Pacific Information Technology Conference*, APIT '20, page 100–107, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376853. doi: 10.1145/3379310.3379323. URL https://doi.org/10.1145/3379310.3379323.

[27] xmrig. XMRig, 2017. URL https://github.com/xmrig/xmrig. original-date: 2017-04-15T05:57:53Z.

[28] Patrick McFarland. DiabloMiner: OpenCL miner for bitcoin, 2018. URL https://github.com/Diablo-D3/DiabloMiner. original-date: 2010-11-06T14:56:14Z.

[29] Luke Dashjr. luke-jr/bfgminer, 2023. URL https://github.com/luke-jr/bfgminer.

[30] Con Kolivas. CGMiner 3.7.2 download (windows 10) AMD, doge [2023], 2023. URL `https://cgminer.info/`.

[31] CPU Miner Contributors. pooler/cpuminer, 2011. URL `https://github.com/pooler/cpuminer`. original-date: 2011-12-04T19:26:44Z.

[32] The MinerGate Authors. Smart multicurrency mining pool & 1-click GUI miner, 2023. URL `https://minergate.com/`.

[33] The Wineth Authors. Easy graphical ethereum mining software, 2023. URL `https://wineth.net/`.

[34] Ankit Gangwal and Mauro Conti. Cryptomining cannot change its spots: Detecting covert cryptomining using magnetic side-channel. *IEEE Transactions on Information Forensics and Security*, 15:1630–1639, 2020. doi: 10.1109/TIFS.2019.2945171.

[35] Dan Goodin. Cryptojacking craze that drains your CPU now done by 2,500 sites, 2017. URL `https://arstechnica.com/information-technology/2017/11/drive-by-cryptomining-that-drains-cpus-picks-up-steam-with-aid-of-2500-sites/`.

[36] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. A first look at browser-based cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 58–66, 2018. doi: 10.1109/EuroSPW.2018.00014. URL `https://doi.org/10.1109/EuroSPW.2018.00014`.

[37] Domhnall Carlin, Philip O'Kane, Sakir Sezer, and Jonah Burgess. Detecting cryptomining using dynamic analysis. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–6, 2018. doi: 10.1109/PST.2018.8514167. URL `https://doi.org/10.1109/PST.2018.8514167`.

[38] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[39] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[40] Samya Dhaiouir and Saïd Assar. A systematic literature review of blockchain-enabled smart contracts: platforms, languages, consensus, applications and choice

criteria. In *Research Challenges in Information Science: 14th International Conference, RCIS 2020*, pages 249–266. Springer, 2020. doi: 10.1007/978-3-030-50316-1_15.

[41] Quinn DuPont. Experiments in algorithmic governance: A history and ethnography of "the DAO," a failed decentralized autonomous organization. In *Bitcoin and beyond*, pages 157–177. Routledge, 2017.

[42] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (SoK). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, Lecture Notes in Computer Science, pages 164–186. Springer, 2017. ISBN 978-3-662-54455-6. doi: 10.1007/978-3-662-54455-6_8.

[43] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys*, 53(3):1–43, 2021-05-31. ISSN 0360-0300, 1557-7341. doi: 10.1145/3391195.

[44] The Solidity Authors. Security considerations – Solidity 0.8.14 documentation, 2022. URL `https://docs.soliditylang.org/en/develop/security-considerations.html`.

[45] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, and Nikhil Swamy. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016.

[46] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, CCS '16, pages 254–269, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978309. URL `https://doi.org/10.1145/2976749.2978309`.

[47] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[48] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[49] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*, pages 243–269. Springer, 2018.

[50] etherscan.io. Uniswap (UNI) token tracker | Etherscan, 2023. URL `https://etherscan.io/token/0x1f9840a85d5af5bf1d1762f925bdaddc4201f984`.

[51] etherscan.io. DEX activity | Etherscan, 2023. URL `http://etherscan.io/stat/dextracker`.

[52] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-design smart contracts for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 446–465. Springer, 2019.

[53] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 494–509. Springer International Publishing, 2017. ISBN 978-3-319-70278-0. doi: 10.1007/978-3-319-70278-0_31.

[54] Yoichi Hirai. Formal verification of deed contract in ethereum name service. *November-2016.[Online]. Available: https://yoichihirai. com/deed. pdf*, 2016.

[55] Michael Fröwis and Rainer Böhme. In code we trust? measuring the control flow immutability of all smart contracts deployed on ethereum. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2017 International Workshops, DPM 2017 and CBT 2017, Oslo, Norway, September 14-15, 2017, Proceedings*, pages 357–372. Springer, 2017.

[56] Manticore Contributors. Manticore, 2023. URL `https://github.com/trailofbits/manticore`.

[57] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 9(54):4–17, 2018.

[58] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis*, pages 513–520, Cham, 2018. Springer International Publishing. ISBN 978-3-030-01090-4.

[59] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 454–469, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385990. URL `https://doi-org.ezp.biblio.unitn.it/10.1145/3385412.3385990`.

[60] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, 2019. doi: 10.1109/WETSEB.2019.00008.

[61] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1678–1694, 2020. doi: 10.1109/SP40000.2020.00032.

[62] Leonardo Alt and Christian Reitwiessner. Smt-based verification of solidity smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 376–388, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03427-6.

[63] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Proceedings of the 2018 Network and Distributed Systems Security Symposium (NDSS)*, pages 1–12, 2018.

[64] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 664–676. Association for Computing Machinery, 2018. ISBN 978-1-4503-6569-7. doi: 10.1145/3274694.3274737. URL https://dl.acm.org/doi/10.1145/3274694.3274737.

[65] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 259–269, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238177. URL https://doi-org.ezp.biblio.unitn.it/10.1145/3238147.3238177.

[66] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: an efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 778–788, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380334. URL https://doi-org.ezp.biblio.unitn.it/10.1145/3377811.3380334.

[67] Haoran Wu, Xingya Wang, Jiehui Xu, Weiqin Zou, Lingming Zhang, and Zhenyu Chen. Mutation testing for ethereum smart contract. *arXiv preprint arXiv:1908.03707*, 2019.

[68] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.*, 29(4), September 2020. ISSN 1049-331X. doi: 10.1145/3402450. URL https://doi-org.ezp.biblio.unitn.it/10.1145/3402450.

[69] Joran J. Honig, Maarten H. Everts, and Marieke Huisman. Practical mutation testing for smart contracts. In Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 289–303, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31500-9.

[70] Yoichi Hirai. Formal verification of deed contract in ethereum name service. *November-2016.[Online]. Available: https://yoichihirai. com/deed. pdf*, 2016.

[71] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of bitcoin transactions. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 541–560, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg. ISBN 978-3-662-58387-6.

[72] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, and Andrei Stefanescu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[73] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934*, 2018.

[74] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. Formal specification and verification of smart contracts for azure blockchain. *arXiv preprint arXiv:1812.08829*, 2018.

[75] Markus Knecht and Burkhard Stiller. Smartdemap: A smart contract deployment and management platform. In *Security of Networks and Services in an All-Connected World*, pages 159–164, Cham, 2017. Springer International Publishing. ISBN 978-3-319-60774-0.

[76] Christian Reitwiessner. Babbage a mechanical smart contract language. *Accessed: Sep*, 21, 2019.

[77] Yoichi Hirai. Bamboo: an embryonic smart contract language. *Accessed: Sep*, 25, 2019.

[78] Michael Coblenz. Obsidian: a safer blockchain programming language. In *2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C)*, pages 97–99. IEEE, 2017.

[79] Russell O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120, 2017.

[80] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of*

the 26th ACM Conference on Computer and Communications Security (CCS 2019), CCS '19, page 1503–1520, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3345664. URL https://doi-org.ezp.biblio.unitn.it/10.1145/3319535.3345664.

[81] Kai Hu, Jian Zhu, Yi Ding, Xiaomin Bai, and Jiehua Huang. Smart contract engineering. *Electronics*, 9(12):2042, 2020. Publisher: Multidisciplinary Digital Publishing Institute.

[82] Rikard Hjort. Formally verifying webassembly with kwasm: Towards an automated prover for wasm smart contracts. Master's thesis, Chalmers University of Technology & University of Gothenburg, 2020. URL https://odr.chalmers.se/server/api/core/bitstreams/a06be182-a12e-46ce-94d3-cff7a5dc42ba/content.

[83] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE software*, 28(3):41–48, 2011. Publisher: IEEE.

[84] Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. Formal verification of infinite-state BIP models. In *International Symposium on Automated Technology for Verification and Analysis*, pages 326–343. Springer, 2015.

[85] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Computer Aided Verification: 26th International Conference, CAV 2014*, pages 334–342. Springer, 2014. doi: 10.1007/978-3-319-08867-9_22.

[86] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

[87] Anastasia Mavridou. VerifyContract.js, 2023. URL https://github.com/anmavrid/smart-contracts/blob/master/src/plugins/VerifyContract/VerifyContract.js.

[88] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore, and Marco Roveri. NuSMV 2.6 tutorial, 2015. URL `https://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf`.

[89] Matthew Dwyer. Property pattern mappings for CTL, 2023. URL `https://matthewbdwyer.github.io/psp/patterns/ctl.html`.

[90] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, 1999. doi: 10.1145/302405.302672.

[91] Anastasia Mavridou. VerifyContract.js, 2023. URL `https://github.com/anmavrid/smart-contracts`.

[92] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. doi: 10.1109/SFCS.1977.32. URL `https://doi.org/10.1109/SFCS.1977.32`.

[93] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-825.

[94] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*, pages 523–540. Springer, 2018.

[95] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[96] Long Chen, Lin William Cong, and Yizhou Xiao. *A Brief Introduction to Blockchain Economics*, chapter Chapter 1, pages 1–40. worldscientific, 2021. doi: 10.1142/9789811220470_0001. URL `https://www.worldscientific.com/doi/abs/10.1142/9789811220470_0001`.

[97] Mauro Conti, Ankit Gangwal, and Michele Todero. Blockchain trilemma solver algorand has dilemma over undecidable messages. In *Proceedings of the 14th*

International Conference on Availability, Reliability and Security, ARES '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450371643. doi: 10.1145/3339252.3339255. URL https://doi.org/10.1145/3339252.3339255.

[98] Kaiwen Zhang and Hans-Arno Jacobsen. Towards dependable, scalable, and pervasive distributed ledgers with blockchains. In *ICDCS*, pages 1337–1346, 2018.

[99] Blockchain.com. Blockchain.com | charts - blockchain size (mb), 2024. URL https://www.blockchain.com/explorer/charts/[id].

[100] etherscan.io. Ethereum full node sync (archive) chart | etherscan, 2024. URL https://etherscan.io/chartsync/chainarchive.

[101] Anthony Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, NJ, USA, 1998. ISBN 978-0130935804.

[102] ethereum.org. Proof-of-stake (pos), 2024. URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[103] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Distributed Computing*, 33(3-4):209–225, 2020. doi: 10.1007/s00446-019-00357-z. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85068876264&doi=10.1007%2fs00446-019-00357-z&partnerID=40&md5=dd7311fa1468b935494b5c6719d86e3e. All Open Access, Green Open Access.

[104] Huahui Xia, Jinchuan Chen, Nabo Ma, Jia Huang, and Xiaoyong Du. Efficient execution of blockchain transactions through deterministic concurrency control. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13943 LNCS:509–518, 2023. doi: 10.1007/978-3-031-30637-2_33. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85161654879&doi=10.1007%2f978-3-031-30637-2_33&partnerID=40&md5=7d0d01cd2a6bb0d750eaa00ae34bc744.

[105] Bahareh Lashkari and Petr Musilek. A comprehensive review of blockchain consensus mechanisms. *IEEE Access*, 9:43620–43652, 2021. doi: 10.1109/ACCESS.2021.3065880.

[106] Kejiao Li, Hui Li, Hanxu Hou, Kedan Li, and Yongle Chen. Proof of vote: A high-performance consensus protocol based on vote mechanism & consortium blockchain. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 466–473, 2017. doi: 10.1109/HPCC-SmartCity-DSS.2017.61.

[107] Shihab Shahriar Hazari and Qusay H. Mahmoud. Improving transaction speed and scalability of blockchain systems via parallel proof of work. *Future Internet*, 12(8), 2020. doi: 10.3390/FI12080125. URL `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85089548009&doi=10.3390%2fFI12080125&partnerID=40&md5=ae0d6542c52eb6252771b3998d65a3dc`.

[108] Scopus team. Scopus content | elsevier, 2024. URL `https://www.elsevier.com/products/scopus/content`.

[109] Shihab Shahriar Hazari and Qusay H. Mahmoud. A parallel proof of work to improve transaction speed and scalability in blockchain systems. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference, CCWC 2019*, pages 916–921, 2019. doi: 10.1109/CCWC.2019.8666535. URL `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85063863999&doi=10.1109%2fCCWC.2019.8666535&partnerID=40&md5=48e48f90293f465ef556ece7331b6ae9`.

[110] Jian Liu, Peilun Li, Raymond Cheng, N. Asokan, and Dawn Song. Parallel and asynchronous smart contract execution. *IEEE Transactions on Parallel and Distributed Systems*, 33(5):1097–1108, 2022. doi: 10.1109/TPDS.2021.3095234. URL `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85112672240&doi=10.1109%2fTPDS.2021.3095234&partnerID=40&md5=9b639907ed892ed22587c0ec40947605`.

[111] Gang Huang, Kaidong Wu, Chaoran Luo, Su Zhang, Huaqian Cai, Xiang Jing, and Yun Ma. Bdledger: A scalable distributed ledger for large-scale data recording. *Communications in Computer and Information Science*, 1490 CCIS: 87–100, 2021. doi: 10.1007/978-981-16-7993-3_7. URL `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85121764911&doi=10.1007%2f978-981-16-7993-3_7&partnerID=40&md5=5d89292da80717632830c135dcd1fe9c`.

[112] Basem Assiri and Wazir Zada Khan. Fair and trustworthy: Lock-free enhanced tendermint blockchain algorithm. *Telkomnika (Telecommunication Computing Electronics and Control)*, 18(4):2224 – 2234, 2020. doi: 10.12928/TELKOMNIKA.V18I4.15701. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85087566813&doi=10.12928%2fTELKOMNIKA.V18I4.15701&partnerID=40&md5=a3f2ded577d7ecb8b53fd7a5b85f2738.

[113] Jae Kwon. tendermint/tendermint, 2024. URL https://github.com/tendermint/tendermint. original-date: 2014-05-14T23:21:35Z.

[114] Yongjie Bai, Yang Zhi, Hui Li, Han Wang, Ping Lu, and Chengtao Ma. On parallel mechanism of consortium blockchain: Take pov as an example. In *ACM International Conference Proceeding Series*, pages 147 – 154, 2021. doi: 10.1145/3460537.3460560. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85115674506&doi=10.1145%2f3460537.3460560&partnerID=40&md5=f01c2263d7c6e4f9d9a2a06584ab07cb.

[115] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978389. URL https://doi-org.ezp.biblio.unitn.it/10.1145/2976749.2978389.

[116] Guangsheng Yu, Xu Wang, Kan Yu, Wei Ni, J. Andrew Zhang, and Ren Ping Liu. Survey: Sharding in blockchains. *IEEE Access*, 8:14155–14181, 2020. doi: 10.1109/ACCESS.2020.2965147.

[117] Xinmeng Liu, Haomeng Xie, Zheng Yan, and Xueqin Liang. A survey on blockchain sharding. *ISA Transactions*, 141:30–43, 2023. doi: 10.1016/j.isatra.2023.06.029. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85164573433&doi=10.1016%2fj.isatra.2023.06.029&partnerID=40&md5=1f360f22b8f3ad34707b24e44bde5a6c.

[118] Faiza Hashim, Khaled Shuaib, and Nazar Zaki. Sharding for scalable blockchain networks. *SN Computer Science*, 4(1):2, 2022.

[119] Firas Hammoodi Neamah Al-Mutar, Ahmed Ali Talib Al-Khazaali, and Baqar Assam Hataf. Scalability of blockchain: Review of cross-sharding with high com-

munication overhead. In *BIO Web of Conferences*, volume 97, page 00075. EDP Sciences, 2024.

[120] Md Mohaimin Al Barat, Shaoyu Li, Changlai Du, Y. Thomas Hou, and Wenjing Lou. Sok: Public blockchain sharding. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 766–783, 2024. doi: 10.1109/ICBC59979.2024.10634422.

[121] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, AFT '19, pages 41–61, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367325. doi: 10.1145/3318041.3355457. URL `https://doi-org.ezp.biblio.unitn.it/10.1145/3318041.3355457`.

[122] Brandon Liew Yi Quan, Nur Haliza Abdul Wahab, Arafat Al-Dhaqm, Ahmad Alshammari, Ali Aqarni, Shukor Abd Razak, and Koh Tieng Wei. Recent advances in sharding techniques for scalable blockchain networks: A review. *IEEE Access*, pages 1–1, 2024. doi: 10.1109/ACCESS.2024.3523256.

[123] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137087. doi: 10.1145/872035.872048. URL `https://doi.org/10.1145/872035.872048`.

[124] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595937957. doi: 10.1145/1345206.1345237. URL `https://doi.org/10.1145/1345206.1345237`.

[125] Vikram Saraph and Maurice Herlihy. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376*, 2019.

[126] Thomas Dickerson, Eric Koskinen, Paul Gazzillo, and Maurice Herlihy. Conflict abstractions and shadow speculation for optimistic transactional objects. In Anthony Widjaja Lin, editor, *Programming Languages and Systems*, pages 313–331, Cham, 2019. Springer International Publishing.

[127] Yiyuan Wang, Shaowei Cai, Shiwei Pan, Ximing Li, and Monghao Yin. Reduction and local search for weighted graph coloring problem. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(03):2433–2441, Apr. 2020. doi: 10.1609/aaai.v34i03.5624. URL `https://ojs.aaai.org/index.php/AAAI/article/view/5624`.

[128] Zhimin Gao, Lei Xu, Lin Chen, Nolan Shah, Yang Lu, and Weidong Shi. Scalable blockchain based smart contract execution. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 352–359, 2017. doi: 10.1109/ICPADS.2017.00054.

[129] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 232–244, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700156. doi: 10.1145/3572848.3577524. URL `https://doi.org/10.1145/3572848.3577524`.

[130] K. Thulasiraman and M. N. S. Swamy. *Graphs: Theory and Algorithms*. John Wiley & Sons, 2011. ISBN 978-1-118-03025-7. Google-Books-ID: rFH7eQffQNkC.

[131] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. Sok: Dag-based blockchain systems. *ACM Comput. Surv.*, 55(12), March 2023. ISSN 0360-0300. doi: 10.1145/3576899. URL `https://doi-org.ezp.biblio.unitn.it/10.1145/3576899`.

[132] Huan Yu Wu, Xin Yang, Chentao Yue, Hye-Young Paik, and Salil S. Kanhere. Chain or dag? underlying data structures, architectures, topologies and consensus in distributed ledger technology: A review, taxonomy and research issues. *Journal of Systems Architecture*, 131:102720, 2022. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2022.102720. URL `https://www.sciencedirect.com/science/article/pii/S1383762122002077`.

[133] Qian Wei, Bingzhe Li, Wanli Chang, Zhiping Jia, Zhaoyan Shen, and Zili Shao. A survey of blockchain data management systems. *ACM Trans. Embed. Comput. Syst.*, 21(3), May 2022. ISSN 1539-9087. doi: 10.1145/3502741. URL `https://doi-org.ezp.biblio.unitn.it/10.1145/3502741`.

[134] Xiang Fu, Huaimin Wang, and Peichang Shi. A survey of blockchain consensus algorithms: mechanism, design and applications. *Science China Information Sciences*, 64:1–15, 2021.

[135] Xiaofeng Lu, Cheng Jiang, and Pan Wang. A survey on consensus algorithms of blockchain based on dag. In *Proceedings of the 2024 6th Blockchain and Internet of Things Conference*, BIOTC '24, pages 50–58, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717000. doi: 10.1145/3688225.3688232. URL https://doi-org.ezp.biblio.unitn.it/10.1145/3688225.3688232.

[136] Nguyen Van Toan, Ung Park, and Geunwoong Ryu. Rcane: Semi-centralized network of parallel blockchain and apos. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, volume 2018-December, pages 695–700, 2018. doi: 10.1109/PADSW.2018.8644573. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85063318541&doi=10.1109%2fPADSW.2018.8644573&partnerID=40&md5=34fd356f01861d9ec6366861441a54f3.

[137] Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. Ace: Asynchronous and concurrent execution of complex smart contracts. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 587–600, 2020. doi: 10.1145/3372297.3417243. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85096161986&doi=10.1145%2f3372297.3417243&partnerID=40&md5=86515ecf98b7c301c895e45e27353ba4.

[138] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. Yoda: Enabling computationally intensive contracts on blockchains with byzantine and selfish nodes, 2018. URL https://arxiv.org/abs/1811.03265.

[139] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. Slimchain: Scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment*, 14(11):2314–2326, 2021. doi: 10.14778/3476249.3476283. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85119678964&doi=10.14778%2f3476249.3476283&partnerID=40&md5=0083c4152ed2912bdf744a0f96bcbb98.

[140] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, page 245–

254, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357951. doi: 10.1145/3212734.3212736. URL https://doi-org.ezp.biblio.unitn.it/10.1145/3212734.3212736.

[141] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A survey on blockchain interoperability: Past, present, and future trends. *ACM Comput. Surv.*, 54(8), October 2021. ISSN 0360-0300. doi: 10.1145/3471140. URL https://doi-org.ezp.biblio.unitn.it/10.1145/3471140.

[142] Panpan Han, Zheng Yan, Wenxiu Ding, Shufan Fei, and Zhiguo Wan. A survey on cross-chain technologies. *Distrib. Ledger Technol.*, 2(2), June 2023. doi: 10.1145/3573896. URL https://doi-org.ezp.biblio.unitn.it/10.1145/3573896.

[143] Hanyu Mao, Tiezheng Nie, Hao Sun, Derong Shen, and Ge Yu. A survey on cross-chain technology: Challenges, development, and prospect. *IEEE Access*, 11:45527–45546, 2023. doi: 10.1109/ACCESS.2022.3228535.

[144] Wei Ou, Shiying Huang, Jingjing Zheng, Qionglu Zhang, Guang Zeng, and Wenbao Han. An overview on cross-chain: Mechanism, platforms, challenges and advances. *Computer Networks*, 218:109378, 2022. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2022.109378. URL https://www.sciencedirect.com/science/article/pii/S1389128622004121.

[145] Peter Robinson. Survey of crosschain communications protocols. *Computer Networks*, 200:108488, 2021. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2021.108488. URL https://www.sciencedirect.com/science/article/pii/S1389128621004321.

[146] Jerry W Markham. Front-running-insider trading under the commodity exchange act. *Cath. UL Rev.*, 38:69, 1988.

[147] États-Unis. Securities and Exchange Commission. Special Study of the Options Markets. *Report of the Special Study of the Options Markets to the Securities and Exchange Commission*. US Government Printing Office, 1979.

[148] Sundas Munir and Christoph Reichenbach. Todler: A transaction ordering dependency analyzer - for ethereum smart contracts. In *Proceedings - 2023 IEEE/ACM 6th International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB 2023*, pages 9–16, 2023. doi:

10.1109/WETSEB59161.2023.00007. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85169085075&doi=10.1109%2fWETSEB59161.2023.00007&partnerID=40&md5=cc67ae8bf141b5a2f8c8a706660b3b67.

[149] Chenyang Ma, Wei Song, and Jeff Huang. Transracer: Function dependence-guided transaction race detection for smart contracts. In *ESEC/FSE 2023 - Proceedings of the 31st ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 947 – 959, 2023. doi: 10.1145/3611643.3616281. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85180548834&doi=10.1145%2f3611643.3616281&partnerID=40&md5=250a0fdf5719e743a09df50aa0cb43e6.

[150] Zachary Stucke, Theodoros Constantinides, and John Cartlidge. Simulation of front-running attacks and privacy mitigations in ethereum blockchain. In *European Modeling and Simulation Symposium, EMSS*, 2022. doi: 10.46354/i3m.2022.emss.041. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85142909699&doi=10.46354%2fi3m.2022.emss.041&partnerID=40&md5=e02a26112e1e8e8677bb705ab0165988.

[151] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 478–493, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70278-0.

[152] Meixun Qu, Xin Huang, Xu Chen, Yi Wang, Xiaofeng Ma, and Dawei Liu. Formal verification of smart contracts from the perspective of concurrency. In *International Conference on Smart Blockchain*, pages 32–43. Springer, 2018.

[153] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, aug 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL https://doi.org/10.1145/359576.359585.

[154] Solidity Contributors. Solidity by example — solidity 0.8.26 documentation, 2024. URL https://docs.soliditylang.org/en/latest/solidity-by-example.html#safe-remote-purchase.

[155] An Zhang and Kunlong Zhang. Enabling concurrency on smart contracts using multiversion ordering. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10988 LNCS:425–439, 2018. doi: 10.1007/978-3-319-96893-3_32. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85051129282&doi=10.1007%2f978-3-319-96893-3_32&partnerID=40&md5=ad208a90b3e81dd28f09f5e7c643bd57.

[156] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts. In *Proceedings - 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019*, pages 83–92, 2019. doi: 10.1109/EMPDP.2019.8671637. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85063875625&doi=10.1109%2fEMPDP.2019.8671637&partnerID=40&md5=d739ee512c8d47e1f9ab15356e94d63d.

[157] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *Proceedings - International Conference on Distributed Computing Systems*, volume 2019-July, pages 1337–1347, 2019. doi: 10.1109/ICDCS.2019.00134. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85074865320&doi=10.1109%2fICDCS.2019.00134&partnerID=40&md5=da89308a8d5454664d2fe873b8e8009d.

[158] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. Entitling concurrency to smart contracts using optimistic transactional memory. In *ACM International Conference Proceeding Series*, page 508, 2019. doi: 10.1145/3288599.3299723. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85060934158&doi=10.1145%2f3288599.3299723&partnerID=40&md5=7993743a21a60f85c7159688d1533d58.

[159] Parwat Singh Anjana, Hagit Attiya, Sweta Kumari, Sathya Peri, and Archit Somani. Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12129 LNCS:77–93, 2021. doi: 10.1007/978-3-030-67087-0_6. URL https://www.scopus.com/inward/

record.uri?eid=2-s2.0-85101539609&doi=10.1007%2f978-3-030-67087-0_6&partnerID=40&md5=445f0bc290b7bfef3c7c0c869e9d3b0d.

[160] Parwat Singh Anjana. Efficient parallel execution of block transactions in blockchain. In *Middleware 2021 Doctoral Symposium - Proceedings of the 22nd International Middleware Conference: Doctoral Symposium*, pages 8–11, 2021. doi: 10.1145/3491087.3493676. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85121106502&doi=10.1145%2f3491087.3493676&partnerID=40&md5=58bbd83f3256405ca871f41dbda52aa8.

[161] Shrey Baheti, Parwat Singh Anjana, Sathya Peri, and Yogesh Simmhan. Dipetrans: A framework for distributed parallel execution of transactions of blocks in blockchains. *Concurrency and Computation: Practice and Experience*, 34(10), 2022. doi: 10.1002/cpe.6804. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85122252293&doi=10.1002%2fcpe.6804&partnerID=40&md5=7c971a4acf78188851ee429342ff56e5.

[162] Manaswini Piduguralla, Saheli Chakraborty, Parwat Singh Anjana, and Sathya Peri. Dag-based efficient parallel scheduler for blockchains: Hyperledger sawtooth as a case study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 14100 LNCS:184–198, 2023. doi: 10.1007/978-3-031-39698-4_13. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85171582845&doi=10.1007%2f978-3-031-39698-4_13&partnerID=40&md5=ffc244a7fab78042dfe584fc0f9d18d4.

[163] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. Optsmart: a space efficient optimistic concurrent execution of smart contracts. *Distributed and Parallel Databases*, 42(2):245–297, 2024. doi: 10.1007/s10619-022-07412-y. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85129749817&doi=10.1007%2fs10619-022-07412-y&partnerID=40&md5=fe84cbbb3450d096d6079ad0914c7296.

[164] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2025.

[165] Vitalik Buterin et al. Solidity latest document. https://docs.soliditylang.org/en/latest/, 2025. Solidity project documentation.

[166] Ananya Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 363–373. ACM, 2019.

[167] Manu M Chakravarty, James Chapman, Kenneth MacKenzie, Omar Melkonian, Michael P Jones, and Philip Wadler. The extended utxo model. In *Int. Conf. on Financial Cryptography and Data Security*, pages 525–539. Springer, 2020.

[168] Nicolas Grech, Michael Kong, Aleksandar Jurisevic, Logan Brent, Benjamin Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

[169] Sergey Tikhomirov, Evgenia Voskresenskaya, Igor Ivanitskiy, Rustam Takhaviev, Egor Marchenko, and Yury Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16. ACM, 2018.

[170] Michele Pasqua, Andrea Benini, Filippo Contro, Marco Crosara, Mila Dalla Preda, and Mariano Ceccato. Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode. *Journal of Systems and Software*, 200:111653, 2023. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2023.111653. URL `https://www.sciencedirect.com/science/article/pii/S0164121223000481`.

[171] Parwat Singh Anjana and Srivatsan Ravi. Empirical analysis of transaction conflicts in ethereum and solana for parallel execution. *arXiv preprint arXiv:2505.05358*, 2025.

[172] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006. ISBN 978-0321486813. Also known as the Dragon Book.

[173] Ethereum Foundation. Solidity compiler (solc). `https://github.com/ethereum/solidity`, 2025. Accessed: 2025-10-20.

[174] Visa Inc. Visa fact sheet: Transaction processing. Technical report, Visa Inc., 2024. URL `https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/visa-net-fact-sheet.pdf`.

[175] PayPal Holdings, Inc. Paypal holdings, inc. annual report 20241231. Technical report, PayPal Holdings, Inc., 2024. URL `https://www.sec.gov/Archives/edgar/data/1633917/000163391725000019/pypl-20241231.htm`.

[176] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

[177] Selmer Martin Johnson. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly*, 1(1):61–68, 1954.

[178] Hegen Xiong, Shuangyuan Shi, Danni Ren, and Jinjin Hu. A survey of job shop scheduling problem: The types and models. *Comp. & Op. Res.*, 142:105731, 2022. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2022.105731. URL `https://www.sciencedirect.com/science/article/pii/S0305054822000338`.

[179] Parwat Singh Anjana and Srivatsan Ravi. Empirical analysis of transaction conflicts in ethereum and solana for parallel execution. *arXiv preprint arXiv:2505.05358*, 2025.

[180] Robbert Reijnen, Kjell van Straaten, Zaharah Bukhsh, and Yingqian Zhang. Job shop scheduling benchmark: Environments and instances for learning and non-learning methods. *arXiv preprint arXiv:2308.12794*, 2023.

[181] Béla Bollobás and Béla Bollobás. *Random graphs*. Springer, 1998.

[182] etherscan.io. Ethereum (ETH) blockchain explorer, 2025. URL `https://etherscan.io/`.

[183] Dror G Feitelson. Job scheduling in multiprogrammed parallel systems. Technical report, IBM T.J. Watson Research Center, 1997.

[184] Weiqi Dai, Deshan Xiao, Hai Jin, and Xia Xie. A concurrent optimization consensus system based on blockchain. In *2019 26th International Conference on Telecommunications, ICT 2019*, pages 244–248, 2019. doi: 10.1109/ICT.2019.8798836. URL `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85071474109&doi=10.1109%2fICT.2019.8798836&partnerID=40&md5=17d1b6beb72b55921f9e732fb5e2ea65`.

[185] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[186] Atefeh Zareh Chahoki. Cryptojackingtrap dataset. https://ieee-dataport.org/documents/cryptojackingtrap, 2023.

Appendix A

CryptojackingTrap

A.1 Implementation

Figure A.1 represents the technical view of the Cryptojacking current implementation and the programming language or libraries that has been used. This architecture is designed to be modular, extensible, and adaptable to evolving industry needs, ensuring that new cryptocurrencies can be incorporated without affecting the core system's integrity.

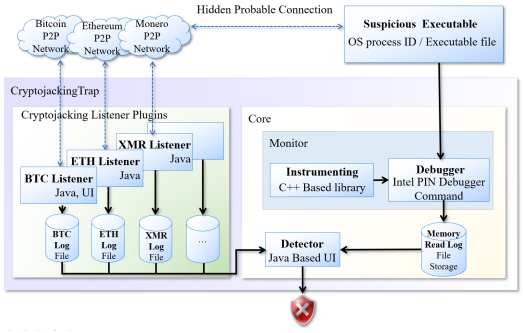


Figure A.1: Technical View of the CryptojackingTrap Architecture.

Figure A.2 illustrates the listener user interface for Bitcoin.

Listing A.1 shows the commands to monitor an executable file or process id, respectively.

Listing A.1: PIN Commands for CryptojackingTrap Monitor

```
$ pin.exe -t "${PinPath}\source\tools\cryptojackingtrap-monitor\x64\Debug\
    CryptojackingtrapMonitor.dll" -support_jit_api -o fileName.out -- "App-
    path\application.exe"
```

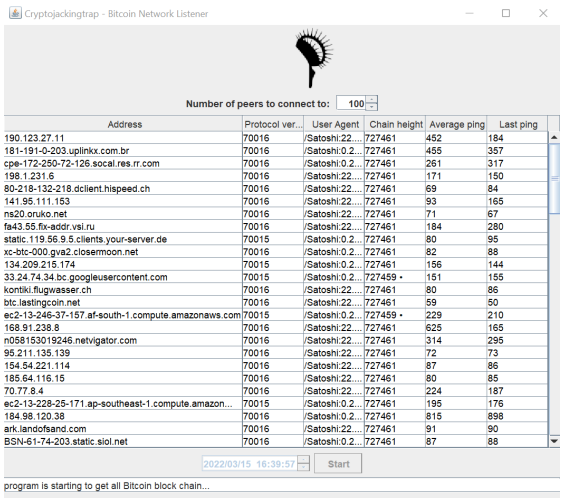


Figure A.2: CryptojackingTrap Bitcoin Listener User Interface.

```
$ pin.exe -pid 36999 -t "${PinPath}\source\tools\cryptojackingtrap-monitor\
    x64\Debug\CryptojackingtrapMonitor.dll" -support_jit_api -o fileName.out
```

The detector which is implemented as a Java-based project with a user interface, shown in Figure A.3, that enables users to specify the BL files, select the cryptocurrencies of interest, and input the corresponding RL files for analysis.

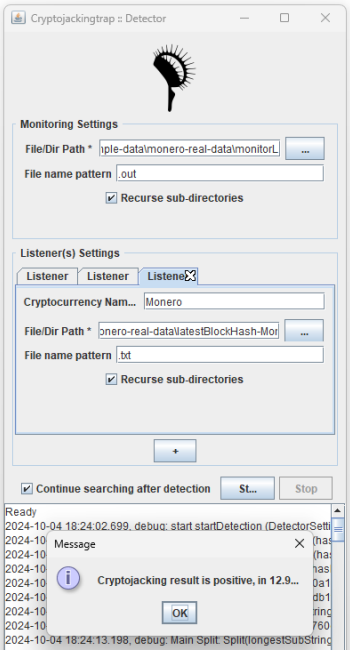


Figure A.3: CryptojackingTrap Detector User Interface.

Appendix B

AI-CryptojackingTrap: Exploratory Extension

B.1 Introduction

This appendix presents an exploratory investigation into extending the *CryptojackingTrap* framework with a machine learning aided detector aimed at improving performance while preserving the solution inherent evasion-resilient advantage. At the heart of CryptojackingTrap lies its detector module, which consolidates data from all other components and applies an advanced detection algorithm to identify malicious activity. Enhancements to this module offer the highest potential for improving the overall system's effectiveness.

As part of the research on cryptojacking detection, an exploratory study was conducted to reduce processing overhead by leveraging the potential to learn recurring patterns—a particularly relevant approach given the limited behavioral diversity of cryptojacking, stemming from the uniformity of cryptocurrency hash functions and their corresponding memory trace logs. This work, carried out as an undergraduate thesis I co-supervised, aimed to enhance the CryptojackingTrap detector by incorporating a machine learning model capable of learning cryptojacking execution patterns during detection, thereby reducing latency by avoiding repeated full analyses of previously observed behaviors.

The study begins with an examination of the dataset derived from *CryptojackingTrap*, focusing on memory-read log files and their suitability for detecting cryptojacking activity. To enable supervised learning, a new dataset was constructed, incorporating custom-made encoding algorithms and file-modification techniques implemented via

regular expressions. Several encoding strategies were tested, ultimately converging on a single approach that demonstrated relatively better detection performance. Furthermore, a dedicated preprocessing pipeline was designed to resolve format inconsistencies within the memory-read logs, followed by a custom normalization procedure optimized for machine learning workflows.

The trained model, developed through multiple training and validation cycles, highlights both the promise and the challenges of applying ML to this problem domain. Despite these efforts, the model did not sufficiently learn the discriminative patterns necessary to reliably identify cryptojacking. While these results fall short of a production-ready solution, they reinforce the complexity of recognizing cryptojacking patterns within heterogeneous log data and point toward the necessity for more advanced modeling techniques. Nevertheless, this work illustrates that machine learning has the potential to augment the *CryptojackingTrap* architecture, offering a path to greater detection efficiency without compromising its evasion resilience. By documenting these intermediate findings and the detailed methodology, this appendix aims to provide a foundation for future research that seeks to utilize AI to enhance the robustness and performance of cryptojacking detection systems, while aligning with the novel design of CryptojackingTrap to utilize its inherent evasion resilience.

B.2 AI-CryptojackingTrap Solution

For modeling the CryptojackingTrap detection this study adopted Neural Network to solve this problem. Generally in machine learning, a *task* defines the problem the model aims to solve, typically classification, regression, or clustering. In this work, the task is binary classification: determining whether a given memory-read log corresponds to *mining* or *non-mining* activity. While the original CryptojackingTrap framework already achieves reliable classification using its detector, the objective here is to train a machine learning model to learn from its outputs and potentially improve detection speed.

The envisioned model must infer patterns from *memory-read log content* and associated *hash values*. If recurring patterns occur within a fixed time window strongly suggest cryptomining activity. The hypothesis is that a trained model could replicate or exceed the performance of the existing detector with lower computational cost. This aligns with *eager learning* strategies, in which the predictive model is constructed during training and subsequently applied to unseen data without re-computation. This con-

trasts with *lazy learning* methods (e.g., k-Nearest Neighbors), which postpone model construction until a query is made, performing prediction by directly comparing the query with stored training instances.

B.2.1 Dataset Transformation Algorithm

The initial dataset consists of memory-read log files generated by the Cryptojacking-Trap monitoring module, each containing timestamps and corresponding memory-read contents (hexadecimal-encoded). These logs are paired with their associated hash values and labeled as either *mining* or *benign*. For the model to learn effectively, these heterogeneous raw files must be transformed into a uniform numerical format suitable for neural network input.

The primary challenge is that log files vary in size, making feature alignment non-trivial. Early attempts considered splitting logs into equal segments, but this risked losing context and introducing sparsity. To address this, a *sliding window* approach was adopted with size of N, enabling systematic extraction of overlapping, fixed-length sequences from each log file. This method preserves temporal locality, increases the number of usable samples from each file, and strengthens pattern recognition by exposing the model to repeated overlapping subsequences.

B.2.2 Preprocessing Algorithm

Preprocessing for addresses formatting errors, better encoding, and scales features was performed in three stages: **(1) Data cleaning:** Removal of debugger artifacts and structural inconsistencies, including concatenated lines and misplaced newline characters, to ensure a uniform pattern of "date–timestamp–memory content". Only memory content is retained for model input. **(2) Encoding:** Neural networks require numerical input. Both the target hash values and memory-read contents are converted to a consistent byte-based numerical representation. The resulting feature vector for each sample comprises: 32 features for the hash, N features for memory content and a binary label (`1` for mining, `0` otherwise). **(3) Normalization:** Feature scaling is applied using `sklearn.preprocessing` to place all features on a comparable range, mitigating training instabilities and improving convergence.

B.3 Implementation

The proof-of-concept implementation was developed in Python and utilized several key libraries to support the data pipeline and model training workflow. For data modification and transformation, including reading structured and semi-structured logs and preparing them for training, the `pandas` library was employed. `Scikit-learn` was used for splitting the dataset into training and test sets using the `train_test_split` function, as well as for feature scaling and other essential preprocessing steps. The machine learning model was implemented using the `TensorFlow` deep learning framework in conjunction with its high-level API, `Keras`. The model was constructed using the `Sequential()` class to define a linear stack of fully connected `Dense()` layers. The network was then compiled using the `compile()` method to specify appropriate loss functions, optimizers, and evaluation metrics. Training was conducted over multiple epochs with the `fit()` method, followed by performance evaluation on held-out test data via `evaluate()` to assess generalization. Finally, the trained model was persisted using `save()` and could be restored later through `load_model()`. This end-to-end setup facilitated experimentation with different architectures and parameters in a modular and reproducible manner.

Training and Testing. The supervised learning pipeline begins by loading the standardized dataset into memory using `pandas`. Labels are separated from feature columns and the dataset is split into training and testing sets using `train_test_split` from `scikit-learn`. The neural network model is then defined using the `Sequential` API in Keras with three fully connected layers: an input layer with 1033 neurons (matching the number of features), a hidden layer of 128 neurons, and a single-neuron output layer representing the binary classification output. The model is compiled with the `adam` optimizer [185], selected for its efficiency and adaptive learning capabilities, and `BinaryCrossentropy` as the loss function. Accuracy is used as the primary evaluation metric. Training is performed over 20 epochs and saved to disk.

For testing, two previously unseen memory-read logs are used—one from a mining process and one from a non-mining process. The trained model is reloaded and applied to these new inputs. Feature columns are extracted and formatted to match the training input schema. The model produces predictions indicating whether the input traces exhibit mining-like behavior.

The full documentation and implementation available open source at `https://github.com/CryptojackingTrap/AI-CryptojackingTrap`.

B.4 Experimental Evaluation

Deployment Environment. The development and training processes were conducted on a high-performance erver specifications tailored for deep learning workloads. The system operated on Ubuntu 22.04.4 LTS and featured an NVIDIA RTX A5000 GPU with 8192 CUDA cores and 24 GB of dedicated memory. It was connected via a PCIe Gen3 interface at x16 link width and a 384-bit memory interface.

Evaluation Dataset. This study employed the publicly available Cryptojacking-Trap dataset [186] which consists of memory-read log files paired with corresponding hash values. The dataset used in this study experimental evaluation was constructed from 10 log files (3 mining, 7 benign) and various mining algorithm was used for the mining files. With a window size of $N = 1000$ bytes and variable step sizes (500 for mining, 100 for benign), the preprocessing pipeline generated a 6.19 GB CSV file containing 1,845,767 samples, each with 1033 features.

Evaluation Result. Initial training yielded a low accuracy of 35%, traced to inconsistencies in feature encoding—specifically, the mismatch between large integer-encoded memory-read contents and small-scale hash values. This prevented the model from learning effective correlations. To address this, a coherent byte-level encoding strategy was adopted, ensuring more uniform feature scales. The revised dataset led to moderate improvements: the model reached 54% training accuracy and 69% loss. However, loss stagnation across epochs indicated limited learning progress. Evaluation on a held-out portion of the dataset confirmed this, with the same accuracy and loss values, suggesting that the model cannot generalize beyond familiar patterns in structurally similar data. Further testing on two new memory-read logs—one from a benign miner and one from a non-miner—revealed shortcomings. The model achieved 100% accuracy (55% loss) on the non-mining data but failed entirely on the mining data (0% accuracy, 87% loss), indicating it had defaulted to predicting non-mining behavior for all inputs.

B.5 Conclusion and Future Work

This exploratory thesis presented the first integration of machine learning into CryptojackingTrap's resilient detection architecture, aiming to reduce computational overhead while preserving detection capabilities. Although the proposed AI-CryptojackingTrap model did not yet achieve reliable cryptojacking detection—particularly on unseen min-

ing data—the study established a complete pipeline for dataset generation, model training, and evaluation.

The findings revealed key limitations in feature encoding and generalization, but more importantly highlighted the inherent challenges of modeling memory-read behavior. Despite modest performance, this work opens a new research direction in applying learning-based methods to memory-read log analysis for Cryptojacking detection.

Future work could explore more advanced architectures—such as deep or recurrent neural networks—or hybrid systems that combine rule-based logic with learning models. Techniques like dimensionality reduction, alternative encoding strategies, or case-specific training may improve effectiveness and scalability. This exploratory study lays the groundwork for a promising and underexplored area in cryptojacking defense.

Appendix C

VeriSolid

C.1 RockPaperScissors Predefined Example

Listing C.1: Generated nuXmv code for RockPaperScissors example having property type two

```
MODULE BAUC(active_interaction)

  VAR
    NuPachoose_guard    :   boolean;
    NuPa1   :   boolean;
    NuPacancelReveal_guard    :   boolean;
    NuPa3   :   boolean;
    NuPawithdrawCanceled_guard    :   boolean;
    NuPawithdrawCanceled_revert   :   boolean;
    NuPawithdrawCanceled_no_revert    :   boolean;
    NuPa7   :   boolean;
    NuPa8   :   boolean;
    NuPaclose_guard   :   boolean;
    NuPa10    :   boolean;
    NuPacancelPlay_guard    :   boolean;
    NuPa12    :   boolean;
    NuPareveal_guard    :   boolean;
    NuPa14    :   boolean;
    NuPa15    :   boolean;
    NuPafinish_guard    :   boolean;
    NuPa17    :   boolean;
    NuPawithdrawFinished_guard    :   boolean;
    NuPawithdrawFinished_revert   :   boolean;
    NuPawithdrawFinished_no_revert    :   boolean;
    NuPa21    :   boolean;
    NuPa22    :   boolean;
    Nuplace   :   {NuSReveal, NuSInitialState, NuSPlay, NuSFinished, NuSCanceled,
        NuSFinalState, NuSState, NuSchoose, NuScancelReveal, NuSwithdrawCanceled,
        NuSwithdrawCanceled_no_revert, NuSs11_1, NuSclose, NuScancelPlay, NuSreveal,
        NuSs15_1, NuSfinish, NuSwithdrawFinished, NuSwithdrawFinished_no_revert,
```

```
        NuSs19_1};
28
29
30  INIT
31    ( (Nuplace) = (NuSPlay) );
32
33  INVAR
34    ( (NuPawithdrawCanceled_revert) <-> (( (Nuplace) = (NuSwithdrawCanceled) )) ) &
35    ( (NuPa7) <-> (( (Nuplace) = (NuSwithdrawCanceled_no_revert) )) ) &
36    ( (NuPachoose_guard) <-> (( (Nuplace) = (NuSPlay) )) ) &
37    ( (NuPa8) <-> (( (Nuplace) = (NuSs11_1) )) ) &
38    ( (NuPacancelReveal_guard) <-> (( (Nuplace) = (NuSReveal) )) ) &
39    ( (NuPawithdrawFinished_revert) <-> (( (Nuplace) = (NuSwithdrawFinished) )) ) &
40    ( (NuPa3) <-> (( (Nuplace) = (NuScancelReveal) )) ) &
41    ( (NuPa1) <-> (( (Nuplace) = (NuSchoose) )) ) &
42    ( (NuPaclose_guard) <-> (( (Nuplace) = (NuSPlay) )) ) &
43    ( (NuPawithdrawFinished_no_revert) <-> (( (Nuplace) = (NuSwithdrawFinished) )) ) &
44    ( (NuPawithdrawFinished_guard) <-> (( (Nuplace) = (NuSFinished) )) ) &
45    ( (NuPafinish_guard) <-> (( (Nuplace) = (NuSReveal) )) ) &
46    ( (NuPawithdrawCanceled_guard) <-> (( (Nuplace) = (NuSCanceled) )) ) &
47    ( (NuPa12) <-> (( (Nuplace) = (NuScancelPlay) )) ) &
48    ( (NuPa10) <-> (( (Nuplace) = (NuSclose) )) ) &
49    ( (NuPacancelPlay_guard) <-> (( (Nuplace) = (NuSPlay) )) ) &
50    ( (NuPareveal_guard) <-> (( (Nuplace) = (NuSReveal) )) ) &
51    ( (NuPa21) <-> (( (Nuplace) = (NuSwithdrawFinished_no_revert) )) ) &
52    ( (NuPa22) <-> (( (Nuplace) = (NuSs19_1) )) ) &
53    ( (NuPa17) <-> (( (Nuplace) = (NuSfinish) )) ) &
54    ( (NuPa14) <-> (( (Nuplace) = (NuSreveal) )) ) &
55    ( (NuPa15) <-> (( (Nuplace) = (NuSs15_1) )) ) &
56    ( (NuPawithdrawCanceled_no_revert) <-> (( (Nuplace) = (NuSwithdrawCanceled) )) );
57
58  TRANS
59    ( (( (( (Nuplace) = (NuSPlay) )) & (( (next(Nuplace)) = (NuSchoose) )) )) & (( (
          active_interaction) = (NuI1) )) ) |
60    ( (( (( (Nuplace) = (NuSchoose) )) & (( (next(Nuplace)) = (NuSPlay) )) )) & (( (
          active_interaction) = (NuI2) )) ) |
61    ( (( (( (Nuplace) = (NuSReveal) )) & (( (next(Nuplace)) = (NuScancelReveal) )) )) &
          (( (active_interaction) = (NuI3) )) ) |
62    ( (( (( (Nuplace) = (NuScancelReveal) )) & (( (next(Nuplace)) = (NuSCanceled) )) ))
          & (( (active_interaction) = (NuI4) )) ) |
63    ( (( (( (Nuplace) = (NuSCanceled) )) & (( (next(Nuplace)) = (NuSwithdrawCanceled) ))
           )) & (( (active_interaction) = (NuI5) )) ) |
64    ( (( (( (Nuplace) = (NuSwithdrawCanceled) )) & (( (next(Nuplace)) = (NuSCanceled) ))
           )) & (( (active_interaction) = (NuI6) )) ) |
65    ( (( (( (Nuplace) = (NuSwithdrawCanceled) )) & (( (next(Nuplace)) = (
          NuSwithdrawCanceled_no_revert) )) )) & (( (active_interaction) = (NuI7) )) ) |
66    ( (( (( (Nuplace) = (NuSwithdrawCanceled_no_revert) )) & (( (next(Nuplace)) = (
          NuSs11_1) )) )) & (( (active_interaction) = (NuI8) )) ) |
67    ( (( (( (Nuplace) = (NuSs11_1) )) & (( (next(Nuplace)) = (NuSCanceled) )) )) & (( (
          active_interaction) = (NuI9) )) ) |
68    ( (( (( (Nuplace) = (NuSPlay) )) & (( (next(Nuplace)) = (NuSclose) )) )) & (( (
          active_interaction) = (NuI10) )) ) |
69    ( (( (( (Nuplace) = (NuSclose) )) & (( (next(Nuplace)) = (NuSReveal) )) )) & (( (
          active_interaction) = (NuI11) )) ) |
```

```
70      ( (( (( (Nuplace) = (NuSPlay) )) & (( (next(Nuplace)) = (NuScancelPlay) )) )) & (( (
            active_interaction) = (NuI12) )) ) |
71      ( (( (( (Nuplace) = (NuScancelPlay) )) & (( (next(Nuplace)) = (NuSCanceled) )) )) &
            (( (active_interaction) = (NuI13) )) ) |
72      ( (( (( (Nuplace) = (NuSReveal) )) & (( (next(Nuplace)) = (NuSreveal) )) )) & (( (
            active_interaction) = (NuI14) )) ) |
73      ( (( (( (Nuplace) = (NuSreveal) )) & (( (next(Nuplace)) = (NuSs15_1) )) )) & (( (
            active_interaction) = (NuI15) )) ) |
74      ( (( (( (Nuplace) = (NuSs15_1) )) & (( (next(Nuplace)) = (NuSReveal) )) )) & (( (
            active_interaction) = (NuI16) )) ) |
75      ( (( (( (Nuplace) = (NuSReveal) )) & (( (next(Nuplace)) = (NuSfinish) )) )) & (( (
            active_interaction) = (NuI17) )) ) |
76      ( (( (( (Nuplace) = (NuSfinish) )) & (( (next(Nuplace)) = (NuSFinished) )) )) & (( (
            active_interaction) = (NuI18) )) ) |
77      ( (( (( (Nuplace) = (NuSFinished) )) & (( (next(Nuplace)) = (NuSwithdrawFinished) ))
             )) & (( (active_interaction) = (NuI19) )) ) |
78      ( (( (( (Nuplace) = (NuSwithdrawFinished) )) & (( (next(Nuplace)) = (NuSFinished) ))
             )) & (( (active_interaction) = (NuI20) )) ) |
79      ( (( (( (Nuplace) = (NuSwithdrawFinished) )) & (( (next(Nuplace)) = (
            NuSwithdrawFinished_no_revert) )) )) & (( (active_interaction) = (NuI21) )) ) |
80      ( (( (( (Nuplace) = (NuSwithdrawFinished_no_revert) )) & (( (next(Nuplace)) = (
            NuSs19_1) )) )) & (( (active_interaction) = (NuI22) )) ) |
81      ( (( (( (Nuplace) = (NuSs19_1) )) & (( (next(Nuplace)) = (NuSFinished) )) )) & (( (
            active_interaction) = (NuI23) )) ) |
82      ( (( (active_interaction) != (NuI23) )) & (( (( (active_interaction) != (NuI22) )) &
             (( (( (active_interaction) != (NuI21) )) & (( (( (active_interaction) != (NuI20
            ) )) & (( (( (active_interaction) != (NuI19) )) & (( (( (active_interaction) !=
            (NuI18) )) & (( (( (active_interaction) != (NuI17) )) & (( (( (
            active_interaction) != (NuI16) )) & (( (( (active_interaction) != (NuI15) )) &
            (( (( (active_interaction) != (NuI14) )) & (( (( (active_interaction) != (NuI13)
             )) & (( (( (active_interaction) != (NuI12) )) & (( (( (active_interaction) != (
            NuI11) )) & (( (( (active_interaction) != (NuI10) )) & (( (( (active_interaction
            ) != (NuI9) )) & (( (( (active_interaction) != (NuI8) )) & (( (( (
            active_interaction) != (NuI7) )) & (( (( (active_interaction) != (NuI6) )) & ((
            (( (active_interaction) != (NuI5) )) & (( (( (active_interaction) != (NuI4) )) &
             (( (( (active_interaction) != (NuI3) )) & (( (( (active_interaction) != (NuI2)
            )) & (( (( (active_interaction) != (NuI1) )) & (( (next(Nuplace)) = (Nuplace) ))
             )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) );
83
84
85  MODULE main
86
87    VAR
88      NuInteraction   :   {NuI1, NuI2, NuI3, NuI4, NuI5, NuI6, NuI7, NuI8, NuI9, NuI10,
            NuI11, NuI12, NuI13, NuI14, NuI15, NuI16, NuI17, NuI18, NuI19, NuI20, NuI21,
            NuI22, NuI23};
89      bauc  : BAUC(NuInteraction);
90
91    DEFINE
92      BAUC_a1   :=    bauc.NuPa1;
93      BAUC_a17    :=    bauc.NuPa17;
94      BAUC_achoose_guard    :=    bauc.NuPachoose_guard;
95      BAUC_awithdrawFinished_revert   :=    bauc.NuPawithdrawFinished_revert;
96      BAUC_a21    :=    bauc.NuPa21;
```

```
97      BAUC_a22    :=    bauc.NuPa22;
98      BAUC_afinish_guard    :=    bauc.NuPafinish_guard;
99      BAUC_acancelPlay_guard    :=    bauc.NuPacancelPlay_guard;
100     BAUC_a7    :=    bauc.NuPa7;
101     BAUC_a10    :=    bauc.NuPa10;
102     BAUC_awithdrawCanceled_guard    :=    bauc.NuPawithdrawCanceled_guard;
103     BAUC_awithdrawFinished_guard    :=    bauc.NuPawithdrawFinished_guard;
104     BAUC_awithdrawCanceled_revert    :=    bauc.NuPawithdrawCanceled_revert;
105     BAUC_awithdrawCanceled_no_revert    :=    bauc.NuPawithdrawCanceled_no_revert;
106     BAUC_awithdrawFinished_no_revert    :=    bauc.NuPawithdrawFinished_no_revert;
107     BAUC_acancelReveal_guard    :=    bauc.NuPacancelReveal_guard;
108     BAUC_a12    :=    bauc.NuPa12;
109     BAUC_a15    :=    bauc.NuPa15;
110     BAUC_a14    :=    bauc.NuPa14;
111     BAUC_a3    :=    bauc.NuPa3;
112     BAUC_a8    :=    bauc.NuPa8;
113     BAUC_areveal_guard    :=    bauc.NuPareveal_guard;
114     BAUC_aclose_guard    :=    bauc.NuPaclose_guard;
115
116
117   INVAR
118     ( (( (NuInteraction) = (NuI2) )) -> (BAUC_a1) ) &
119     ( (( (NuInteraction) = (NuI18) )) -> (BAUC_a17) ) &
120     ( (( (NuInteraction) = (NuI1) )) -> (BAUC_achoose_guard) ) &
121     ( (( (NuInteraction) = (NuI20) )) -> (BAUC_awithdrawFinished_revert) ) &
122     ( (( (NuInteraction) = (NuI22) )) -> (BAUC_a21) ) &
123     ( (( (NuInteraction) = (NuI23) )) -> (BAUC_a22) ) &
124     ( (( (NuInteraction) = (NuI17) )) -> (BAUC_afinish_guard) ) &
125     ( (( (NuInteraction) = (NuI12) )) -> (BAUC_acancelPlay_guard) ) &
126     ( (( (NuInteraction) = (NuI8) )) -> (BAUC_a7) ) &
127     ( (( (NuInteraction) = (NuI11) )) -> (BAUC_a10) ) &
128     ( (( (NuInteraction) = (NuI5) )) -> (BAUC_awithdrawCanceled_guard) ) &
129     ( (( (NuInteraction) = (NuI19) )) -> (BAUC_awithdrawFinished_guard) ) &
130     ( (( (NuInteraction) = (NuI6) )) -> (BAUC_awithdrawCanceled_revert) ) &
131     ( (( (NuInteraction) = (NuI7) )) -> (BAUC_awithdrawCanceled_no_revert) ) &
132     ( (( (NuInteraction) = (NuI21) )) -> (BAUC_awithdrawFinished_no_revert) ) &
133     ( (( (NuInteraction) = (NuI3) )) -> (BAUC_acancelReveal_guard) ) &
134     ( (( (NuInteraction) = (NuI13) )) -> (BAUC_a12) ) &
135     ( (( (NuInteraction) = (NuI16) )) -> (BAUC_a15) ) &
136     ( (( (NuInteraction) = (NuI15) )) -> (BAUC_a14) ) &
137     ( (( (NuInteraction) = (NuI4) )) -> (BAUC_a3) ) &
138     ( (( (NuInteraction) = (NuI9) )) -> (BAUC_a8) ) &
139     ( (( (NuInteraction) = (NuI14) )) -> (BAUC_areveal_guard) ) &
140     ( (( (NuInteraction) = (NuI10) )) -> (BAUC_aclose_guard) );
141
142
143
144 -- A [ !((NuInteraction) = (NuI10)) U ((NuInteraction) = (NuI12))]
145 CTLSPEC A [ !((NuInteraction) = (NuI10)) U ((NuInteraction) = (NuI12))]
146
147 FAIRNESS ( (NuInteraction) = (NuI10));
```

C.2 NuXmv Fairness Code Generation

VerifyContract.js

src > plugins > VerifyContract > VerifyContract.js > define() callback > VerifyContract > verifyContract

```
234            actionNamesToTransitionNames = VerifyContract.prototype.actionNamesToTransitions(model['transitions'], actionNamesToTransitionNames);
235            if (currentConfig['templateTwo'] != '' || currentConfig['templateThree']!=''){
236              fairnessProperties = 'FAIRNESS ( ';
237            }
238            if (currentConfig['templateOne']!=''){
239              propertiesSMV += VerifyContract.prototype.generateFirstTemplateProperties(currentConfig['templateOne'], model, bipTransitionsToSMVNames, actionNamesToTransiti
240              propertiesTxt += VerifyContract.prototype.generateFirstTemplatePropertiesTxt(currentConfig['templateOne'], model, bipTransitionsToSMVNames, actionNamesToTrans
241            }
242            if(currentConfig['templateTwo']!=''){
243              propertiesSMV += VerifyContract.prototype.generateSecondTemplateProperties(currentConfig['templateTwo'], model, bipTransitionsToSMVNames, actionNamesToTransit
244              fairnessProperties += VerifyContract.prototype.generateSecondTemplateFairnessProperties(currentConfig['templateTwo'], model, bipTransitionsToSMVNames, actionN
245              propertiesTxt += VerifyContract.prototype.generateSecondTemplatePropertiesTxt(currentConfig['templateTwo'], model, bipTransitionsToSMVNames, actionNamesToTrar
246            }
247            if(currentConfig['templateThree']!=''){
248              propertiesSMV += VerifyContract.prototype.generateThirdTemplateProperties(currentConfig['templateThree'], model, bipTransitionsToSMVNames, actionNamesToTransi
249              fairnessProperties += VerifyContract.prototype.generateThirdTemplateFairnessProperties(currentConfig['templateThree'], model, bipTransitionsToSMVNames, actior
250              propertiesTxt += VerifyContract.prototype.generateThirdTemplatePropertiesTxt(currentConfig['templateThree'], model, bipTransitionsToSMVNames, actionNamesToTra
251            }
252            if(currentConfig['templateFour']!=''){
253              propertiesSMV += VerifyContract.prototype.generateFourthTemplateProperties(currentConfig['templateFour'], model, bipTransitionsToSMVNames, actionNamesToTransi
254              propertiesTxt += VerifyContract.prototype.generateFourthTemplatePropertiesTxt(currentConfig['templateFour'], model, bipTransitionsToSMVNames, actionNamesToTra
255            }
256            if (currentConfig['templateTwo'] != '' || currentConfig['templateThree']!=''){
257              fairnessProperties = fairnessProperties.slice(0,-1);
258              fairnessProperties += ');';
259            }
260            fs.appendFileSync(path + '/' + model.name+ '.smv', propertiesSMV);
261            fs.appendFileSync(path + '/' + model.name+ '.smv', fairnessProperties);
262            fs.writeFileSync(path + '/' + model.name+ 'Prop.txt', propertiesTxt);
```

Figure C.1: NuXmv fairness code generation, main section

VerifyContract.js

src > plugins > VerifyContract > VerifyContract.js > define() callback > VerifyContract > generateThirdTemplateProperties > properties

```
411
412  /* Get the properties specified by the user in Template Two */
413  VerifyContract.prototype.generateSecondTemplateFairnessProperties = function (templateTwo, model, bipTransitionsToSMVNames, actionNamesToTransitionNames){
414    var self = this,
415        properties = [],
416        property, clause,
417        fairnessProperties = '';
418
419    properties =VerifyContract.prototype.parseProperties.call(self, model, templateTwo);
420    for (property of properties){
421      for (clause of property[0]){
422        fairnessProperties += bipTransitionsToSMVNames[actionNamesToTransitionNames[clause]] + "|"
423      }
424    }
425    return fairnessProperties;
426  };
```

Figure C.2: NuXmv fairness code generation, template two

VerifyContract.js

src > plugins > VerifyContract > VerifyContract.js > define() callback > VerifyContract > generateFourthTemplateProperties

```
504  VerifyContract.prototype.generateThirdTemplateFairnessProperties = function (templateThree, model, bipTransitionsToSMVNames, actionNamesToTransitionNames){
505    var self = this,
506        properties = [],
507        property, clause,
508        fairnessProperties = '';
509
510    properties =VerifyContract.prototype.parseProperties.call(self, model, templateThree);
511    for (property of properties){
512      for (clause of property[2]){
513        fairnessProperties += bipTransitionsToSMVNames[actionNamesToTransitionNames[clause]] + "|"
514      }
515    }
516    return fairnessProperties;
517  };
```

Figure C.3: NuXmv fairness code generation, template three