

Towards Secure and Scalable Blockchain Systems: From Game-Theoretic Oracle Networks to AI-driven Smart Contract Repair

by

Behkish Nassirzadeh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

© Behkish Nassirzadeh 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Henry Kim
Associate Professor, Schulich School of Business,
York University

Supervisor(s): Vijay Ganesh
Professor, School of Computer Science, Georgia Institute of
Technology
Adjunct Professor, Dept. of Electrical and Computer Engi-
neering, University of Waterloo

Anwar Hasan
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal Member: Derek Rayside
Associate Professor, Dept. of Electrical and Computer Engi-
neering, University of Waterloo

Internal Member: Mark Crowley
Associate Professor, Dept. of Electrical and Computer
Engineering, University of Waterloo

Internal-External Member: Meng Xu
Assistant Professor, David R. Cheriton School of Computer
Science, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The adoption of blockchain technologies in security-critical and high-throughput domains remains limited by persistent challenges in scalability, reliability, and automated vulnerability mitigation. This thesis presents a cohesive body of work that addresses two fundamental limitations of modern blockchain systems: the difficulty of ensuring safe, efficient execution in smart contracts and the lack of robust mechanisms for secure data connectivity through decentralized oracle networks (DONs).

To address the first challenge, we introduce a suite of tools, *GasGauge*, *GasGuard*, and *GasGaugeAI*, that advance the detection, analysis, and automated repair of gas-related Denial-of-Service (DoS) vulnerabilities in Ethereum smart contracts. *GasGauge* leverages static-dynamic analysis to model safe loop bounds and identify Out-of-Gas (OOG) risks.

We examine how emerging AI methods, particularly large language models (LLMs) and program synthesis tools, provide a scalable path forward for developing self-healing blockchain systems. *GasGuard* builds on this foundation by integrating a fine-tuned LLM to insert guard conditions that prevent unsafe execution automatically. Finally, *GasGaugeAI* extends the pipeline with a novel multi-LLM framework that classifies gas-dependent vulnerabilities, generates Foundry-based test cases, synthesizes function-level repairs, and validates fixes iteratively. Across hundreds of real-world contracts, these systems demonstrate the potential of AI-guided repair to drastically reduce manual auditing efforts and prevent exploitable gas exhaustion patterns.

Beyond contract-level vulnerabilities, this thesis tackles the broader problem of trustworthy data connectivity in decentralized applications. We propose *CountChain*, a game-theoretic decentralized oracle network for secure aggregation in counting systems. Built on this foundation, *AdChain* applies DON principles to online advertising, mitigating discrepancy fraud through incentive-aligned protocols. Our experiments demonstrate that *CountChain* and *AdChain* provide both scalability and provable security against rational adversaries.

Together, the tools, systems, and theoretical insights presented in this thesis contribute to the vision of blockchain infrastructures that are both secure and scalable by design, bridging the gap between automated repair and game-theoretic connectivity.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors, Professors Vijay Ganesh and Anwar Hasan, for their unwavering support, insightful guidance, and tireless mentorship throughout my doctoral journey. Their mentorship has shaped not only this thesis but also the way I approach research, complexity, and impact.

Professor Ganesh's mentorship from the earliest stages of my graduate studies fostered my growth as a researcher through rigorous discussion, high standards of excellence, and a constant push toward deeper understanding. Professor Hasan's guidance brought fresh perspectives, strategic direction, and invaluable feedback that refined and strengthened this work. I am incredibly fortunate to have been supervised by two pioneers whose dedication and vision continually inspired me.

I am equally grateful to my examining committee members, Professors Derek Rayside, Mark Crowley, and Meng Xu, for their thoughtful engagement with my work and for the valuable feedback they provided throughout this process. Their questions and perspectives helped refine many aspects of this research.

Special thanks to Professor Henry Kim, my external examiner, for taking the time to review this work and provide constructive insights from a broader vantage point.

I would like to acknowledge the role of my comprehensive exam committee members, especially Professor Catherine Gebotys, for her guidance on the computer engineering background component.

I am especially grateful to my collaborators and colleagues whose ideas, feedback, and contributions enriched the work presented in this thesis. I would like to thank Dr. Albert Heinle, Dr. Stefanos Leonardos, Professor Karthik Pattabiraman, and Rui Xi for their insights, partnership, and intellectual generosity across various projects and publications, including the book chapter and research papers.

I also want to acknowledge the University of Waterloo's Department of Electrical and Computer Engineering for providing a dynamic environment encouraging curiosity and innovation. The countless hallway conversations, group meetings, and late-night brainstorming sessions played a vital role in shaping this work.

I am also grateful for the generous support I have received through fellowships and awards during my doctoral studies. In particular, I acknowledge the Ripple Graduate Fellowship in Blockchain Research, which supported my work in blockchain security, the Sharon & David Johnston Award for Excellence in Student Leadership, which recognized my contributions to the university community, and the Entrepreneurial PhD Fellowship, which provided invaluable encouragement and resources to pursue the entrepreneurial dimensions of my research.

Lastly, to my family and friends, thank you for your unconditional support, patience, and belief in me. This thesis would not have been possible without your grounding presence in my life.

Dedication

To those who never stopped believing in me, and to the one who gave me the courage to believe in myself.

Table of Contents

Examining Committee Membership	ii
Author's Declaration	iii
Abstract	iv
Acknowledgements	v
Dedication	vi
List of Figures	xii
List of Tables	xiv
List of Abbreviations	xv
List of Symbols	xvi
1 Introduction	1
1.1 Motivation and Context	1
1.2 Research Challenges	2
1.3 Common Design Themes and Unified Vision: Runtime and Data, the Two Pillars of Trust	2
1.4 Research Contributions	4
1.5 Thesis Organization	5
2 Preliminaries	6
2.1 Background	6
2.1.1 Blockchain Fundamentals	6

2.1.2	Overview and Taxonomy of System-Level Vulnerabilities	7
2.1.3	Smart Contracts and the Ethereum Virtual Machine	8
2.1.4	Ethereum Gas Model and DoS Vulnerabilities	9
2.1.5	Detection Techniques in Blockchain Security	10
2.1.6	AI for Blockchain: Applications and Advances	14
2.1.7	Large Language Models in Code Analysis and Repair	15
2.1.8	Blockchain Oracles and Decentralized Oracle Networks	15
2.1.9	Counting Systems in Blockchain Applications	16
2.1.10	Game Theory and Nash Equilibrium	17
2.2	Helper Tools	17
2.2.1	Slither: Static Analysis Framework	17
2.2.2	Foundry: Fuzzing and Test Automation	18
2.3	Related Work	18
2.3.1	Smart Contract Analysis and Gas-Related Vulnerability Detection	18
2.3.2	Automated Program Repair	19
2.3.3	Decentralized Oracle Networks	20
2.3.4	Blockchain-Based Advertising and Counting Systems	21
3	Automated Detection of Gas-Based Vulnerabilities in Smart Contracts	22
3.1	Motivation and Background	23
3.2	Hybrid Approaches for Detection and Repair of Security Vulnerabilities	23
3.3	Out-of-Gas Denial-of-Service Vulnerabilities	24
3.4	Steps to Identify and Mitigate OOG Vulnerabilities	26
3.5	GasGauge: A Security Analysis Tool for Smart Contract Out-of-Gas Vulnerabilities	28
3.5.1	Overview and Summary of Experimental Results	28
3.5.2	A Case Study: Analyzing the Airswap Protocol	28
3.5.3	Lessons Learned and Output Report	29
3.5.4	Integrating AI and Blockchain for Scalable, Automated Smart Contract Repair	30
3.6	Conclusion and Discussion	32

4	Towards Automatic Program Repair for Gas-Based Vulnerabilities in Smart Contracts	33
4.1	GasGuard: An LLM-based Automated Gas Vulnerability Detection and Mitigation System	33
4.1.1	GasGuard Architecture and Dataset Overview	34
4.1.2	The Output of GasGuard	36
4.2	Experimental Evaluation and Results of GasGuard	36
4.2.1	Experiment 1: Prompt Engineering Techniques On GPTs	37
4.2.2	Experiment 2: Effect of Dataset Size on Fine-Tuning Accuracy	39
4.2.3	Experiment 3: Effect of Explanatory Context on Fine-tuning	40
4.2.4	Experiment 4: Impact of Hyperparameter Optimization	42
4.2.5	Experiment 5: Combining Fine-tuning with Prompt Engineering	43
4.2.6	Summary of the Results	44
4.3	Optimization Theory for Model Performance of GasGuard	45
4.3.1	Managing Trade-offs with Adaptive Strategies	46
4.4	Initial Validation and Extensibility of GasGuard	46
4.5	Conclusion, Limitations and Next Steps	47
5	GasGaugeAI: A Multi-LLM APR Framework for State-Aware Gas Vulnerability Mitigation	49
5.1	GasGaugeAI Architecture	50
5.1.1	Phase I: Loop Detection and Risk Analysis via GasGauge	50
5.1.2	Phase II: Automated Repair for State-Controlled Loops	51
5.1.3	Modular Design and Synergy of AI with Program Analysis	53
5.1.4	Dataset Overview	55
5.1.5	Limitations of GasGaugeAI	56
5.2	Experimental Evaluation of GasGaugeAI	56
5.2.1	Preliminary and Experimental Setup	56
5.2.2	Experiment 1: Effectiveness of LLM-1 for Test Generation	57
5.2.3	Experiment 2: Impact of Prompt Engineering on LLM-2 Repair	58
5.2.4	Experiment 3: Impact of Prompt Combination Strategies on LLM-2 Repair	61
5.2.5	Experiment 4: Fine-Tuning GPT-4.1 for Loop Repair	62
5.2.6	Experiment 5: Multi-Round Feedback Using LLM-3	63

5.2.7	Summary of Findings	65
5.3	Summary, Discussion, and Future Work	65
5.4	Towards Trustworthy Decentralized Systems	66
6	Game-Theoretic Decentralized Oracle Network for Counting Systems	68
6.1	CountChain: A DON for Verifiable Counting	69
6.1.1	Motivation and Problem Scope	69
6.1.2	Design Principles of CountChain	70
6.1.3	Setup and Terminology	72
6.1.4	System Description	74
6.1.5	Reward and Penalty Mechanism	75
6.2	Incentive Design and Game-Theoretic Security of CountChain	76
6.2.1	Point System as a Utility Model	77
6.2.2	Incentive Compatibility and Nash Equilibrium Analysis	77
6.2.3	Sybil Attack Mitigation	79
6.2.4	Game-Theoretic Robustness Summary	79
6.3	Experimental Evaluation of CountChain	80
6.3.1	Experiment 1: Optimizing System Parameters	80
6.3.2	Experiment 2: Sybil Attack Resistance	82
6.4	Conclusion and Discussion	83
7	Towards Decentralized Header Bidding Solutions	84
7.1	Online Advertising and the Need for Verifiable Impressions	84
7.1.1	The Programmatic Stack and Header Bidding	85
7.1.2	Impression Discrepancy and Data Fragmentation	85
7.1.3	Blockchain’s Attempts to Save Ad Supply Chain	86
7.1.4	The Case for a Lightweight Oracle Layer	86
7.2	AdChain: A Decentralized Header Bidding Solution for Advertising Discrepancy Using CountChain	86
7.2.1	Overview and Objectives	87
7.2.2	Mapping CountChain Concepts to AdChain Context	88
7.2.3	System Architecture	90
7.2.4	Summary and Integration with Existing Infrastructure	90

7.3	Experimental Evaluation of AdChain	91
7.3.1	Impact of Honesty Rate and Verifier Count	91
7.3.2	Optimal Network Size	92
7.3.3	Sybil Attack Resilience	93
7.3.4	Comparative Experimental Analysis of CountChain and AdChain	94
7.4	Economic Model of AdChain	94
7.4.1	Assumptions	94
7.4.2	Revenue Estimation	95
7.4.3	Cost Estimation	95
7.4.4	Profitability Analysis	96
7.5	Discussion, Limitations, and Future Work	96
8	Conclusion and Future Work	98
8.1	Summary of the Insights	98
8.2	Limitations	99
8.3	Future Work	99
8.4	Closing Remarks	100
	References	101
	Glossary	113

List of Figures

2.1	Overview of Slither’s architecture and analysis pipeline [1].	17
3.1	A smart contract vulnerable to DoS with block gas limit [2]	25
3.2	Workflow of Identifying and Mitigating Out-of-Gas Vulnerabilities [3] . . .	26
3.3	GasGauge architecture consisting of detection, identification, and correction phases [4].	27
3.4	Vulnerable Code Snippet from one of Airtswap’s contracts	28
3.5	GasGauge output showing per-iteration gas usage and derived threshold formula [4].	30
4.1	GasGuard architecture: from loop detection to automated gas-safe patching [2].	34
4.2	Examples of each loop category [2]	35
4.3	Example of the input and output of GasGuard. GasGuard introduces a gas-aware runtime guard to prevent Out-of-Gas failures in an unbounded loop.	36
4.4	Summary of the results of Experiment 1. The format of each graph line is <i>LLM model.technique used.</i> [2]	38
4.5	Summary of the results of Experiment 2. Each line of the graph shows the results for a fine-tuning dataset size.[2]	39
4.6	Summary of the results of Experiment 3. Each line of the graph shows the results for a fine-tuning dataset size.[2]	40
4.7	The charts illustrate the impact of different hyperparameters. Top: model fine-tuned with 350 instances. Center: 50 instances. Bottom: The best results are from the center chart. E: Number of Epochs, B: Batch Size, L: Learning Rate [2].	41
4.8	Summary of the results of Experiment 5. The format of each graph line is <i>Datset size.technique used.</i> [2]	43
4.9	An Overview of the Extensibility of GasGuard [2]	46

5.1	GasGuard architecture: from loop detection to automated repair for state-dependent loops [3].	50
5.2	LLM-3 Feedback Impact on Repair Accuracy across 6 Configurations [3] .	64
6.1	General DON vs. Counting System DON Design [5].	69
6.2	An Overview of CountChain Mechanism [5].	73
6.3	Pseudocode for the Host and Verifier components in the CountChain protocol. The host handles proposition submissions and vote collection, while verifiers independently validate observed data against received propositions [5].	74
6.4	A verifier faces two sequential decisions: first, whether to incur a cost $-c$ to search for proof; and second, if successful, whether to act honestly and vote <i>True</i> , or disregard the proof and vote <i>False</i> [5].	77
6.5	Impact of honesty rate and verifier count on proposition accuracy [5]. . . .	80
6.6	Optimal node count for maximizing accuracy [5].	81
6.7	The success rate of Sybil attacks against CountChain. UNHR = Uncorrupted Node Honesty Rate [5]	82
7.1	Illustration of the header bidding process in online advertising. Multiple DSPs bid through SSPs using a Prebid.js integration on the publisher's website [6].	85
7.2	AdChain overview: adapting CountChain to verify ad impressions in a header bidding environment [6].	88
7.3	Pseudocode for the Host and Verifier components in the extended CountChain protocol. The Host manages verifier selection, proposition creation, and vote collection. Verifiers validate pixel-based input and vote based on hashed proofs and time-bounded constraints.	89
7.4	Impact of honesty rate and verifier count on proposition accuracy of AdChain[6].	91
7.5	Optimal node count for maximizing the accuracy of AdChain[6].	92
7.6	The success rate of Sybil attacks against AdChain. UNHR = Uncorrupted Node Honesty Rate [6]	93

List of Tables

2.1	Representative tools based on static analysis for smart contract vulnerability detection.	10
2.2	Symbolic execution tools for smart contract security.	11
2.3	Representative fuzzing tools for smart contract vulnerability detection. . .	12
2.4	AI-based approaches focused purely on learning-driven vulnerability detection.	13
3.1	Hybrid vulnerability detection and repair tools combining multiple techniques.	24
4.1	Summary of Benchmark of the Fine-Tuned Dataset [2].	35
4.2	Highlighted Results from the Experiments[2].	44
5.1	Summary of Common Repair Patterns Used in Fine-Tuning LLM-2	53
5.2	Summary of GasGaugeAI’s Design and Workflow.	54
5.3	RQ1: LLM-1 Test Generation Results [3]	58
5.4	RQ 2: LLM-2 Prompt Engineering Results[3]	59
5.5	Experiment 3: Prompt Combination Strategies for LLM-2 Repair (15 Combinations)[3]	60
5.6	RQ3: LLM-2 Prompt Combination Results	61
5.7	RQ4: LLM-2 Fine-Tuning Results [3]	62
5.8	RQ5: Repair Results After LLM-3 Feedback [3]	63
6.1	Summary of the terminology of CountChain [5].	72
6.2	Separated payoff matrix for CountChain. Each entry shows [Point, Stake]. Verifier payoff depends on their vote; submitter payoff depends only on the majority outcome.	76
7.1	The terminology used in the AdChain protocol [6].	87

List of Abbreviations

AI Artificial Intelligence [2](#), [6](#), [12](#), [14](#), [15](#), [30–32](#), [54](#), [55](#), [66](#), [100](#)

API Application Programming Interface [15](#)

APR Automated Program Repair [2](#), [15](#), [19](#), [20](#)

CoT Chain of Thought [52](#), [53](#)

CPM Cost Per Mille (cost per 1,000 ad impressions) [86](#)

dApp Decentralized Application [16](#), [99](#)

DeFi Decentralized Finance [3](#), [7–9](#), [20](#), [66](#), [100](#)

DON Decentralized Oracle Network [2](#), [3](#), [5](#), [6](#), [16](#), [17](#), [20](#), [21](#), [68–70](#), [83](#), [94](#), [98](#)

DoS Denial of Service [xii](#), [2](#), [3](#), [8–10](#), [16](#), [18–20](#), [22–25](#), [28–30](#), [32](#), [47](#), [49](#), [56](#), [65](#), [67](#), [75](#), [99](#)

EVM Ethereum Virtual Machine [2](#), [9](#), [18](#), [19](#), [24](#), [99](#)

IoT Internet of Things [15](#)

LLM Large Language Model [2–6](#), [12](#), [14](#), [15](#), [19](#), [20](#), [26](#), [27](#), [31](#), [33](#), [34](#), [36](#), [37](#), [40](#), [42](#), [46–53](#), [55–57](#), [59](#), [65](#), [98](#), [99](#)

NE Nash Equilibrium [77](#), [78](#)

OOG Out of Gas [4](#), [24](#), [26](#), [28–33](#), [35](#), [36](#), [47](#), [49](#), [50](#), [52](#), [53](#)

Oracle External Data Provider for Blockchain [2](#), [5](#), [8](#), [94](#), [96](#), [97](#)

List of Symbols

- $C(6, 2)$ The number of 2-element combinations from 6 prompting strategies; equals 15 [61](#)
- F Boolean value False [71](#)
- M A given model configuration including hyperparameters, dataset, and tuning strategies [45](#)
- N Total number of verifiers required to process global ad transactions at scale [95](#)
- N Total number of participating nodes in the system [71](#), [76](#), [79](#)
- T_a Start of the time interval for proposition validity [75](#)
- T_b End of the time interval for proposition validity [75](#)
- T Boolean value True [71](#)
- V Number of verifiers selected per advertisement, typically set to 15 [95](#)
- α Weight for model accuracy in the utility objective function [45](#), [46](#)
- β Weight for cost penalty in the utility objective function [45](#), [46](#)
- δ Weight assigned to human labor cost in the total cost function [45](#), [46](#)
- γ Weight assigned to computational cost in the total cost function [45](#)
- TC** Total cost function combining computation and labor costs for training and deploying a model [45](#)
- θ The vector of all tunable model parameters and design configurations [45](#)
- $b_i(p)$ Node i 's private belief about the truth value of proposition p [71](#)
- c Fixed computational cost incurred by a verifier when searching for a valid proof [77–79](#)
- i Index of a node (player) in the system, where $i \in \{1, \dots, N\}$ [71](#), [76](#)
- k Number of nodes controlled by the adversary [79](#)
- n Data size per ad transaction, approximately 1 KB [95](#)

n Number of verifiers randomly selected per proposition 76, 79

p_T Fraction of propositions that are actually True 78, 79

p_{v_i} Point total for player i 76

p_{v_j} Point total for player j 76

p Expected fraction of propositions that will be decided as True 79

p A proposition submitted to the system for verification 71

$q_i(p)$ Accuracy of node i 's belief about proposition p , i.e., the probability $b_i(p) = v(p)$ 71

$u(V_j)$ Heaviside function that returns 1 if $p_{v_j} > 0$, and 0 otherwise 76

$v(p)$ The binary ground truth value of proposition p , either $\{T, F\}$ 71

bandwidth The data transfer rate of a verifier's internet connection, affecting throughput capacity 95

BTC Bitcoin; the native cryptocurrency of the Bitcoin blockchain 96

Correct_Rate (CRate) The percentage of outputs classified as **Correct_Repair** out of total evaluated cases 60

Correct_Repair (CR) A successful fix that removes the vulnerability while preserving contract logic 59

CR1 Number of **Correct_Repair** cases after the first feedback round 63

CR2 Number of **Correct_Repair** cases after the second feedback round 63

IL1 Number of **Incorrect_Logic** cases after the first feedback round 64

IL2 Number of **Incorrect_Logic** cases after the second feedback round 64

Incorrect_Logic (IL) The repair introduces logic errors, breaks the contract, or deviates from intended behavior 60

KB Kilobyte; a unit of digital information equal to 1,024 bytes 95, 96

Mbps Megabits per second; a measure of network bandwidth 95, 96

ms Milliseconds, a unit of time equal to 10^{-3} seconds 81, 93, 94

No_Output_Generated (NOG) The model produces unusable output, such as empty, invalid, or unrecoverable Solidity code 60

NOG1 Number of **No_Output_Generated** cases after the first feedback round 64

NOG2 Number of `No_Output_Generated` cases after the second feedback round [64](#)

Not_Repaired_But_Correct_Logic (NRCL) The fix does not address the vulnerability but preserves correct contract behavior and good development practice [60](#)

NRCL1 Number of `Not_Repaired_But_Correct_Logic` cases after the first feedback round [64](#)

NRCL2 Number of `Not_Repaired_But_Correct_Logic` cases after the second feedback round [64](#)

TB Terabyte; a unit of digital storage equal to 2^{40} bytes or 1,024 GB [96](#)

Chapter 1

Introduction

Blockchain technology has emerged as a foundational infrastructure for decentralized systems, offering transparency, immutability, and automation without relying on centralized trust. Since its inception, it has enabled new paradigms in finance, supply chain, advertising, and governance. However, critical security and scalability challenges persist despite rapid proliferation and increasing adoption. These challenges are particularly acute in smart contract platforms like Ethereum, where vulnerabilities can be exploited to drain funds, halt protocols, or corrupt execution logic [7, 8]. Meanwhile, decentralized applications increasingly rely on external data sources, introducing a new class of trust and reliability risks that most existing blockchain consensus protocols struggle to address [9].

This thesis investigates two fundamental and interrelated limitations in modern blockchain systems. The first limitation is the lack of robust mechanisms to automatically detect and repair security vulnerabilities, especially state-dependent issues, in smart contracts. The second limitation is the demanding need for scalable, game-theoretic frameworks to ensure the correctness of external data ingestion, primarily in applications relying on counting systems. While these issues are often studied independently, this work argues that they form two sides of the same problem, building blockchain systems that are secure and scalable by design across both computation and connectivity layers.

1.1 Motivation and Context

Smart contracts, particularly those deployed on platforms like Ethereum, are immutable and public programs that autonomously manage digital assets and enforce business logic. However, their transparency also means any bug or vulnerability is immediately exploitable, and immutability causes the vulnerability to stay permanent in the system, unless using techniques like *CREATE2*¹. While some vulnerabilities like arithmetic operation flaws or access control errors are similar in traditional software, blockchain systems introduce unique attack surfaces, such as gas-based vulnerabilities, reentrancy, and oracle manipulation, that arise from their immutable, decentralized, state-dependent, and value-bearing nature.

¹<https://docs.openzeppelin.com/cli/2.8/deploying-with-create2>

Among the most pressing risks are gas-related [DoS](#) vulnerabilities, especially those caused by unbounded loops or excessive computation that exceed the [EVM](#)’s gas limits [10, 11, 8]. These vulnerabilities can render contracts unusable, freeze assets, or create opportunities for malicious exploitation. Detecting and mitigating such vulnerabilities currently requires extensive manual effort, specialized tooling, and deep familiarity with [EVM](#) internals. Even state-of-the-art analysis tools struggle with complex contract patterns or dynamic behaviors, making them unreliable for production-scale auditing [10, 12].

In parallel, decentralized applications increasingly depend on data from external sources, price feeds, counting systems, and voting outcomes, which are fetched through external nodes known as [Oracles](#). However, most blockchain platforms assume a closed-world model where all inputs are verifiable on-chain. Without a robust [DON](#), decentralized systems are susceptible to data manipulation, Sybil attacks, and incentive misalignment [13]. This tension between decentralization and real-world interaction is now one of the key bottlenecks for mainstream adoption.

The question this thesis seeks to answer is: ***How can we build blockchain systems that are not only secure in their execution but also trustworthy in their external connectivity, using automation, game theory, and [AI](#)?***

1.2 Research Challenges

This thesis focuses on two interconnected technical challenges:

1. **Gas-related vulnerabilities and lack of automated repair:** Traditional static analyzers can flag potential issues but often fail to suggest precise, verifiable, and deployable fixes. Also, loop-related gas risks require context-specific reasoning and symbolic constraints that are difficult to encode in fixed rule systems. Moreover, even after detection, repairing these issues requires a significant manual effort.
2. **Decentralized [Oracle](#) network ([DON](#)) design:** Ensuring data reliability across untrusted nodes without central authorities or external arbitration remains an open problem. Existing approaches do not apply to applications like counting systems. Also, they often compromise on scalability or rely on centralized fallback mechanisms, weakening the system’s integrity.

To overcome these issues, this thesis proposes a novel integration of [APR](#) using [LLMs](#), static and dynamic analysis, and game-theoretic consensus mechanisms for decentralized [Oracles](#) systems.

1.3 Common Design Themes and Unified Vision: Runtime and Data, the Two Pillars of Trust

Although the GasGauge series and the oracle systems explored later in this thesis operate at different levels of the blockchain stack, they share a standard set of architectural and

philosophical principles. Both aim to make decentralized systems verifiable, efficient, and resilient in adversarial environments. This section highlights the overlapping design themes that underpin these otherwise distinct tools.

The security and correctness of a decentralized application depend on the smart contract’s logic, how it is executed, and what data it consumes. In this context, the integrity of execution and the integrity of input, two of the main pillars of trust, merge. Both GasGauge and CountChain enforce correctness not by relying on ideal behavior but by embedding constraints that limit the space of misbehavior. The GasGauge toolchain achieves this through static and dynamic analysis to compute safe iteration bounds for loops and uses [LLMs](#) to mitigate and repair the vulnerable code. In CountChain, constraint enforcement is implemented through voting rules, slashing penalties, and quorum thresholds, which prevent a small subset of nodes from distorting the results. In both cases, the protocol limits the power of any single actor and elevates verifiability as a first-class design goal.

The GasGauge tool series was designed to enhance runtime integrity by automatically detecting and repairing gas-based [DoS](#) vulnerabilities. These vulnerabilities are particularly dangerous because they can silently render smart contract functions unusable due to unbounded loop iterations that exceed Ethereum’s block gas limit [4]. Thus, GasGauge, GasGuard, and GasGaugeAI bring a form of runtime verifiability to smart contracts, turning gas usage from an unpredictable failure point into a measurable and enforceable constraint.

While runtime verification ensures that a contract executes safely, it does not guarantee that the data it uses is correct. Many decentralized applications, particularly in decentralized finance ([DeFi](#)), gaming, and advertising, depend on off-chain inputs that the blockchain cannot natively access, such as asset prices, event outcomes, or user actions. This gap is filled by decentralized oracle networks ([DONs](#)), which bridge on-chain logic and off-chain reality [9]. However, these oracles introduce their own vulnerabilities, including data manipulation, timestamp spoofing, and collusion among reporters [13, 14]. Without a robust accountability mechanism, the quality and truthfulness of data entering the blockchain remain difficult to guarantee.

The oracle systems introduced in this thesis, CountChain and AdChain, directly address the need for verifiable data input. CountChain focuses on decentralized counting in systems such as user engagement or voting, using techniques like lazy commitments and quorum-based voting to ensure that no single node can dominate the result. AdChain extends these principles to the domain of digital advertising, offering a way for publishers and advertisers to reconcile impression discrepancies transparently and without relying on centralized authorities. In both cases, the integrity of the reported data is preserved not through blind trust, but through decentralized agreement and rational game-theoretic incentives.

GasGaugeAI leverages [LLMs](#) to automatically synthesize gas-aware repairs, completing the loop from vulnerability detection to actionable patching. Similarly, AdChain’s automated aggregation of impression data and matching algorithms help translate raw ad telemetry into verifiable commitments without requiring manual reconciliation. These systems rely on assistive intelligence, not to replace human developers or auditors, but to

scale verifiability through automation.

Together, these two lines of work represent complementary pillars of trust. The GasGauge series safeguards how computations are performed, while CountChain and AdChain ensure the quality of the information they depend on. Addressing runtime and data-layer concerns creates a foundation on which provably trustworthy decentralized applications can be built.

These shared design themes point toward a unified vision for trustworthy decentralized systems, one in which correctness is not an optimistic assumption but a verifiable outcome. Whether the challenge is gas safety or data truthfulness, the solution lies in combining constrained autonomy, adversarial modeling, and automation to bring transparency and robustness to every layer of decentralized application logic.

This convergence is no coincidence. As decentralized applications become more complex, the need to reason formally about internal behavior and external inputs becomes essential. Without gas safety, applications may stall, and without data accountability, users may be misled.

Therefore, this thesis begins with the GasGauge toolchain, reflecting the foundational importance of runtime security in blockchain systems. Ensuring that smart contracts are secure is a prerequisite for building reliable connectivity and application layers. Without contract-level correctness, no amount of oracle design or external data verification can guarantee trust. Once security is established at the computation layer, the thesis progresses to connectivity and data-layer guarantees, culminating in practical applications. This ordering mirrors the layered architecture of blockchain systems themselves, from secure execution to decentralized data acquisition to vertical applications, emphasizing the importance of building trustworthy systems from the ground up.

1.4 Research Contributions

The contributions of this thesis are as follows:

- **GasGauge:** A hybrid static-dynamic analysis tool that identifies and models gas-intensive loop constructs in Ethereum smart contracts. This tool calculates safe bounds and outputs symbolic gas usage formulas to guide mitigation [4].
- **GasGuard:** An LLM-powered automated mitigation tool that builds on GasGauge by inserting ‘`require()`’ statements at vulnerable loop sites using prompt-engineered and fine-tuned GPT models. GasGuard achieves over 98% repair accuracy in lab settings [2].
- **GasGaugeAI:** A multi-LLM framework that automates both vulnerability detection and function-level repair using multiple methods, including static analysis, fuzzing, and LLMs. It introduces a pipeline for classifying state variables, generating test cases, and synthesizing secure logic to mitigate OOG vulnerabilities in complex loop-bound scenarios [3].

- **CountChain:** A [DON](#) for secure and scalable counting, leveraging lazy voting and game-theoretic incentives to align rational agents. CountChain is evaluated on large-scale experiments and provides fault tolerance without centralized trust [5].
- **AdChain:** A domain-specific implementation of CountChain applied to header bidding in online advertising. AdChain mitigates discrepancy fraud and introduces decentralized auction protocols for supply-side transparency [6].

Together, these contributions form a cohesive foundation for secure and scalable blockchain infrastructures grounded in automation, formalism, and incentive alignment.

1.5 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2** provides background on blockchain architecture, smart contracts, the Ethereum gas model, detection methods, decentralized [Oracles](#), and relevant work.
- **Chapter 3** outlines the motivation, architecture, and methodology for detecting gas-based denial-of-service vulnerabilities in Ethereum smart contracts. It introduces the GasGauge toolchain and explains a hybrid ten-step pipeline that integrates static analysis, fuzzing, symbolic execution, and [LLM](#)-guided repair.
- **Chapter 4** focuses on GasGuard, the first [LLM](#)-assisted tool in the GasGauge pipeline. It explains how GasGuard uses symbolic gas analysis and prompt-engineered [LLMs](#) to insert safe iteration guards, with real-world examples and evaluation results.
- **Chapter 5** presents GasGaugeAI, an advanced multi-agent [LLM](#) system for automated repair of gas vulnerabilities. It details the roles of three [LLMs](#), test generation, function synthesis, and feedback classification, and evaluates the prompt design, fine-tuning, and iterative feedback loops on 800 vulnerable loops.
- **Chapter 6** introduces CountChain, [DON](#), designed to ensure external data reliability using game-theoretic incentives and protocol-level security.
- **Chapter 7** explains AdChain, a real-world application of CountChain tailored to the online advertising ecosystem to address a billion-dollar problem.
- **Chapter 8** concludes with a summary of key findings, broader implications, and avenues for future research in secure, self-healing blockchain systems.

Chapter 2

Preliminaries

This chapter presents the foundational concepts and technical background required to understand this thesis’s contributions. It introduces key components of blockchain systems, such as smart contracts, gas economics, decentralized oracle networks (DONs), and consensus protocols, alongside the supporting methodologies drawn from program analysis, AI, and game theory. This chapter lays the groundwork for the proposed tools and systems discussed in later chapters by outlining both the technological context and existing research efforts.

2.1 Background

The technologies explored in this thesis intersect with secure computation, decentralized infrastructure, and automated reasoning. As such, this section introduces the essential elements that underpin the problem domains addressed in the remainder of this work. We review the fundamentals of blockchain architecture and smart contract execution models. We then discuss Ethereum’s gas model, oracle design, and economic mechanisms such as Nash equilibrium that ensure rationality in decentralized systems. We also introduce the emergence of LLMs as a new paradigm for reasoning about code at scale.

2.1.1 Blockchain Fundamentals

A blockchain is a distributed ledger technology designed to facilitate secure, verifiable transactions across a decentralized network of participants. Unlike most traditional databases, where a central authority maintains control over data consistency and trust, blockchains achieve consensus through cryptographic techniques and incentive-aligned protocols that eliminate the need for trusted intermediaries [8, 15].

A blockchain consists of a series of blocks containing a batch of transactions, metadata, and a cryptographic reference (i.e., hash) to the previous block. This structure forms a tamper-evident chain in which altering one block would require recomputing all subsequent

blocks, a computationally infeasible task without control over the majority of the network’s resources. Thus, once data is committed, it cannot be retroactively changed. This property, known as *immutability*, is foundational to blockchain’s appeal in adversarial settings [16].

Blockchain networks are typically categorized by their level of openness:

- **Public (permissionless) blockchains**, such as Bitcoin and Ethereum, are permissionless and allow anyone to participate as a validator, submit transactions, or run a full node [8].
- **Private (permissioned) blockchains** restrict participation to specific entities and are often used in enterprise environments where governance is centralized [17].
- **Consortium (hybrid) blockchains** strike a balance by allowing a group of pre-approved organizations to operate the network collaboratively [18].

Blockchain networks employ consensus protocols to reach an agreement on the next valid block. Early systems like Bitcoin utilize Proof of Work (PoW), which requires solving computational puzzles to produce blocks [15]. Ethereum initially adopted PoW but transitioned to Proof of Stake (PoS) in 2022 through the Merge [19], significantly reducing energy consumption and aligning validator rewards with financial commitment to the network. Other consensus mechanisms include Delegated Proof of Stake (DPoS) [20], Practical Byzantine Fault Tolerance (PBFT) [21], Proof of Authority (PoA) [22], and various hybrid models [18], each designed to balance throughput, security, and decentralization across different blockchain architectures.

In addition to consensus, blockchains rely on cryptographic primitives such as digital signatures and hash functions to guarantee transaction authenticity and data integrity. These mechanisms, when combined with economic incentives, allow blockchains to serve as a secure, trustless substrate for a wide range of applications, from decentralized finance (DeFi) to supply chain management and decentralized identity systems [23].

The rise of general-purpose blockchains, notably Ethereum, has expanded blockchain’s utility beyond asset transfers. These platforms support programmable logic in the form of *smart contracts*.

2.1.2 Overview and Taxonomy of System-Level Vulnerabilities

System-level vulnerabilities in blockchain refer to protocol-layer weaknesses arising from individual application logic and decentralized systems’ underlying architecture and assumptions. These vulnerabilities typically manifest in areas such as consensus mechanism, network assumptions, *mempool* dynamics, or cross-layer interactions, and they often transcend specific smart contract bugs or user errors. What makes them especially critical is their potential to impact all network users, undermine decentralization, or disrupt liveness and safety guarantees.

Unlike traditional software systems, where security is often confined to access control or memory safety, blockchain introduces new trust models rooted in economic incentives,

probabilistic finality, and distributed governance. These models make the system more robust in some ways but also more susceptible to novel attack vectors such as front-running, MEV (maximal extractable value), time manipulation, DoS with block gas limit, and validator collusion [16, 24].

System-level vulnerabilities can be exploited silently, profitably, and persistently, especially when the protocol’s assumptions do not hold in adversarial conditions. Examples include chain reorgs facilitated by selfish mining [25], validator bribery to reorder transactions [26], or misuse of Oracles update intervals to manipulate asset prices in DeFi platforms [14]. These issues are further exacerbated in multi-chain ecosystems where bridging protocols may introduce new failure modes due to inconsistent state and lack of atomicity.

A structured taxonomy helps distinguish between these vulnerabilities’ different origins and effects. While overlap exists across categories, we broadly classify system-level vulnerabilities into four types:

- **Consensus Layer Vulnerabilities:** These include chain reorganizations, time manipulation, selfish mining, and 51% attacks. They directly threaten safety, finality, and chain stability.
- **Smart Contract-Level Vulnerabilities:** These emerge from platform semantics such as gas metering, storage costs, and contract composability. Common examples include gas-based DoS, flash loan attacks, and unintended reentrancy. Although originating in smart contracts, their impact spans the system when exploited across composable DeFi protocols.
- **Oracle-Level Vulnerabilities:** These arise when external data inputs to the blockchain are delayed, manipulated, or selectively included. Front-running, data liveness failures, and low-frequency Oracles updates fall into this category [13, 9].
- **Cross-Layer and Interoperability Vulnerabilities:** These affect systems that span multiple chains or off-chain components, including token bridges, Layer 2 rollups, and relayer protocols. Vulnerabilities in this class often stem from inconsistent state assumptions, replay risks, or trust boundaries being violated [27, 28].

Understanding these categories is essential for diagnosis and designing detection tools that target the root causes of systemic failure. However, this thesis primarily focuses on contract-level vulnerabilities, particularly those related to gas-based DoS risks, which are both subtle and under-addressed in current detection frameworks. Additionally, we explore Oracle-related vulnerabilities where applicable, especially in the context of decentralized data reporting for counting systems and online advertising. Consensus-layer and cross-layer vulnerabilities are acknowledged but lie outside the core scope of this work.

2.1.3 Smart Contracts and the Ethereum Virtual Machine

Smart contracts are self-executing programs deployed on a blockchain that autonomously enforce rules and trigger actions based on predefined conditions. Introduced with Ethereum

[8], smart contracts expanded the utility of blockchain beyond simple value transfer to programmable and composable applications in areas such as decentralized finance (DeFi), governance, and identity systems.

Unlike traditional software, smart contracts are immutable once deployed, meaning their logic cannot be modified without redeploying an entirely new contract, unless techniques like CREATE2 are used. This immutability ensures trust and auditability but also introduces a high level of fragility. Any bugs, vulnerabilities, or inefficiencies in the code are permanent and potentially exploitable [29]. Consequently, smart contract correctness is critical and requires rigorous verification before deployment.

Ethereum smart contracts are written in high-level languages like Solidity or Vyper and are compiled to bytecode for execution by the EVM, a stack-based virtual machine embedded in each Ethereum node. The EVM is Turing-complete, meaning it can compute anything given enough resources, but this also introduces complexity and undecidability challenges for formal analysis.

The EVM executes contracts deterministically and sequentially using an instruction set that manipulates the stack, memory, and persistent storage. Each instruction consumes a specific amount of *gas*, which acts as both a resource pricing mechanism and a guardrail against denial-of-service attacks caused by infinite loops or excessive computation.

Because of the composable nature of Ethereum, contracts can invoke each other recursively, leading to vulnerabilities such as reentrancy [30]. Numerous high-profile incidents, most notably the DAO exploit in 2016 [7], have demonstrated the real-world risks associated with insecure contract logic. These incidents have motivated the development of new programming patterns, static analysis tools, and formal verification frameworks to increase smart contracts' robustness before deployment. These vulnerabilities highlight the need for automated vulnerability detection and repair systems, such as those explored in this thesis.

2.1.4 Ethereum Gas Model and DoS Vulnerabilities

In Ethereum, *gas* is the fundamental unit of computational cost, designed to quantify and constrain the execution of smart contracts. Every operation in the EVM, from arithmetic instructions to data storage or contract calls, incurs a predefined gas cost.

Users specify a limit for each transaction, known as the transaction gas limit, and if execution exceeds this limit, the transaction is halted and reverted, but the gas spent up to that point is still consumed [8]. In addition to this limit, the miners and the network set a block gas limit. Each transaction must be included in at most one block by default. Hence, any limitation on a block automatically also applies to the transactions. Thus, the transaction gas limit cannot surpass the block gas limit.

The gas model serves two primary purposes. First, it prevents DoS attacks by making it economically infeasible to execute arbitrarily expensive computations or infinite loops. Second, it incentivizes efficient code by linking computation costs to transaction fees. This

Tool	Key Features	Analysis Level
Slither [1]	Rule-based detectors, inheritance graphs, function summaries, taint tracking, and plugin support.	Source code (Solidity AST)
SmartCheck [31]	Parses Solidity into XML AST and matches code against vulnerability patterns and coding style violations.	Source code
Securify 2.0 [32]	Uses control and data dependency analysis to verify compliance with security properties via Datalog queries.	Intermediate representation
Osiris [33]	Enhances static analysis with vulnerability-specific rules, especially for integer and overflow issues.	Bytecode

Table 2.1: Representative tools based on static analysis for smart contract vulnerability detection.

economic model, however, also introduces a new class of vulnerabilities unique to smart contracts that exploit the gas mechanism itself, known as gas-based vulnerabilities.

A typical example is unbounded loops that iterate over dynamic data structures like arrays or mappings. While such loops may be safe when data structures are small, they become dangerous as the on-chain state grows. In the worst case, a contract may enter a state where no function can be executed within the gas limit, effectively freezing the contract [11]. This vulnerability is classified as a gas-based DoS and has been identified in numerous real-world contracts.

Gas-related vulnerabilities can arise not only from control flow structures but also from indirect interactions between contracts. For instance, a contract that calls another contract without bounding its gas consumption could become vulnerable to cascading failures or intentional reentrancy that depletes gas before critical logic is executed.

2.1.5 Detection Techniques in Blockchain Security

Blockchain systems present unique security challenges, including immutability, gas constraints, composability, and adversarial execution environments. Traditional techniques such as static analysis and symbolic execution have been adapted to these constraints, while newer methods, like fuzzing, AI-assisted technique reasoning, and hybrid techniques, offer complementary strengths. Each technique varies in depth, automation, scalability, and suitability for specific classes of vulnerabilities. This section surveys the most prominent detection methods in the blockchain security landscape, explaining their core mechanisms and presenting representative tools through summary tables.

Tool	Key Features	Analysis Level
Oyente [29]	Early symbolic executor for Ethereum; detects reentrancy, timestamp dependency, and TOC-TOU bugs.	Bytecode
Mythril [34]	Combines symbolic execution and control flow analysis to detect a wide range of logic bugs.	Bytecode
Zeus [35]	Integrates symbolic reasoning with policy checking over Solidity; supports declarative security rules.	Source Code
Manticore [36]	Symbolic execution engine for smart contracts and binaries with fine-grained path exploration.	Bytecode

Table 2.2: Symbolic execution tools for smart contract security.

Static Analysis

Static analysis involves examining a smart contract’s source code or intermediate representation without executing it. It is among the most widely adopted approaches for early vulnerability detection due to its scalability, speed, and ability to scan contracts at development or compilation time. Static analyzers typically construct representations such as the [Abstract Syntax Tree \(AST\)](#), [Control Flow Graph \(CFG\)](#), or [Data Dependency Graph \(DDG\)](#) to model the program’s structure and identify patterns of known vulnerabilities.

The primary advantage of static analysis is its automation and breadth of coverage. However, static analysis is inherently limited by its lack of runtime context. It tends to over-approximate potential issues, often resulting in false positives. Additionally, static methods struggle to detect vulnerabilities that depend on evolving state, dynamic interactions, or gas usage behavior [12, 10].

Despite these limitations, static analysis remains a foundational layer in blockchain security. Its integration with more advanced methods (e.g., symbolic reasoning or fuzzing) enhances the precision and completeness of vulnerability detection pipelines. Table 2.1 summarizes leading static analysis tools and their characteristics.

Symbolic Execution

Symbolic execution explores all feasible paths in a smart contract by treating inputs as symbolic variables rather than concrete values. It builds constraints (path conditions) for each branch and uses [SMT Solver](#) solvers to determine feasibility [37]. This allows for detecting deep bugs such as reentrancy, overflows, and logic flaws that depend on specific input conditions or execution paths.

Symbolic execution provides strong semantic insight but suffers from scalability issues due to path explosion and complex constraints. It is also limited in modeling gas consumption or multi-contract interactions, often reducing its practical effectiveness in

Tool	Key Features	Fuzzing Type
Echidna [38]	Property-based fuzzer with Solidity assertion support and developer-defined invariants.	Source-level, Stateful
Foundry Fuzzer [39]	Fuzzing integrated into Solidity tests via forge ; supports input sequences and state persistence.	Source-level, Stateful
Contract Fuzzer [40]	Generates random ABI inputs to test for reentrancy, exceptions, and gas issues.	Bytecode-level, Stateless
Harvey [41]	Coverage-guided greybox fuzzer built on top of Mythril; targets deep path exploration.	Bytecode-level, Stateful
Medusa [42]	Parallel fuzzing engine using mutation strategies and execution trace guidance.	Bytecode-level, Stateful

Table 2.3: Representative fuzzing tools for smart contract vulnerability detection.

detecting gas-based or cross-call vulnerabilities. Table 2.2 summarizes the prominent symbolic execution-based tools in the smart contract domain.

Fuzzing

Fuzzing is a dynamic vulnerability detection technique that executes smart contracts repeatedly using randomly generated or mutated inputs. Unlike static analysis or symbolic execution methods, fuzzers observe real execution behavior, effectively uncovering runtime issues such as assertion violations, unhandled exceptions, and reentrancy under specific state transitions [43].

Fuzzing for Ethereum contracts is typically either stateless, resetting contract state on each run, or stateful, preserving execution context across input sequences. The latter is more potent for detecting multi-call or state-dependent vulnerabilities.

Fuzzing is scalable and easy to automate, but may miss deep bugs without adequate guidance. Its success depends on the quality of invariants, seed inputs, and execution feedback, making hybridization with symbolic execution increasingly common. Table 2.3 lists well-known smart contract fuzzers.

AI and LLM-Based Detection

Recent advances in AI, especially LLMs and reinforcement learning (RL), have opened new directions for smart contract vulnerability detection. These methods aim to learn patterns and semantics from data rather than relying on handcrafted rules or formal specifications. AI-based techniques are beneficial for identifying business logic bugs, contextual misuses, and natural language inconsistencies, challenges that traditional tools struggle to address [44, 47].

Tool / Model	Key Features	Technique
ContractWard [44]	Classifies contract functions as vulnerable/non-vulnerable using a graph neural network trained on control and data flow graphs.	Supervised ML (GNN)
SmartEval [45]	Uses GPT-style LLMs to evaluate smart contract functions and generate natural language explanations of potential risks.	LLM (GPT)
SmartInv [46]	Leverages LLMs to automatically synthesize invariants and assertions for runtime enforcement and fuzzing guidance.	LLM + Code Generation
GPT-4 (Few-Shot) [47]	Evaluates GPT-4’s ability to detect real-world bugs with minimal examples. Achieves 15% accuracy on MUB detection.	LLM (Few-shot)

Table 2.4: AI-based approaches focused purely on learning-driven vulnerability detection.

While promising, AI-based systems face challenges in explainability, dataset bias, and lack of provable guarantees. They are most effective when integrated with other static analysis or dynamic analysis forms as part of a hybrid technique pipeline.

Table 2.4 summarizes tools that rely purely on AI-based reasoning for vulnerability detection, without integrating other methods.

Other Noteworthy Detection Methods

Several other methods, such as runtime verification, formal verification, and transaction analysis, play important complementary roles. Although less commonly deployed in practical audits, these approaches are particularly effective for critical systems or post-deployment investigations.

Runtime verification (RV) involves monitoring the execution of a contract to check whether it adheres to predefined correctness properties at runtime [48]. **Formal verification** mathematically proves that a contract satisfies a specification across all possible executions [49, 50]. **Transaction analysis** shifts focus from the contract code to blockchain activity. It examines call trace patterns, token transfer, and opcode usage patterns to detect runtime anomalies or exploitation behaviors [51, 43].

Although these methods are not as widespread, they provide unique perspectives that are particularly valuable in high-assurance contexts or adversarial environments. Many hybrid systems integrate them selectively to enhance coverage, enforce correctness, or support post-deployment monitoring.

2.1.6 AI for Blockchain: Applications and Advances

The convergence of artificial intelligence (AI) and blockchain technologies marks a pivotal shift in how decentralized systems are designed, reasoned about, and secured. While blockchain offers a foundation for transparency, immutability, and distributed trust, it often lacks the flexibility and adaptability needed to respond dynamically to complex or evolving threats. Conversely, AI excels at learning patterns, automating tasks, and generalizing from limited information, but it lacks the verifiability, auditability, and incentive structures that blockchains provide. Their integration is complementary and transformative.

Recent advances in large language models (LLMs) such as GPT [52], Codex [53], and CodeT5 [54] have made it possible to perform sophisticated reasoning over smart contracts, automatically generate patches, and synthesize human-like explanations of program behavior. These capabilities have created a new class of AI-powered tools in the blockchain space. This advancement can detect vulnerabilities, propose and validate repairs, guide fuzzers, and generate invariants [47, 46, 45].

AI has emerged as a powerful force multiplier for blockchain systems, particularly in security, automation, and developer tooling. As smart contracts grow in complexity and the risks associated with incorrect execution increase, AI can augment traditional analysis techniques with pattern recognition, natural language reasoning, and learned code transformations. This shift has enabled a new generation of tools that are faster, more context-aware, and capable of learning from historical vulnerabilities without the need for handcrafted rules.

Security Auditing and Vulnerability Detection

As mentioned before, security auditing and vulnerability detection in smart contracts have been among the first areas to benefit from AI. Traditional tools based on static analysis or symbolic execution often struggle with code coverage, false positives, and scalability. In contrast, AI-based approaches can learn to recognize risky patterns or behaviors from labeled datasets. For instance, ContractWard [44] uses graph neural networks to classify smart contract functions as vulnerable or safe based on control and data flow structures. Similarly, MAIAN [55] integrates symbolic reasoning with machine learning heuristics to detect suicidal or greedy contracts at scale.

Fuzzing Guidance and Test Prioritization

Fuzzing is generally a strong technique to detect security and logic errors in a code. However, writing test cases for a fuzzer is manual and resource-consuming. Fuzzing guidance and test prioritization have been enhanced through reinforcement learning and LLMs. Tools like Confuzzius [56] adapt input mutation strategies based on learned feedback from prior runs, focusing on deeper or more semantically sensitive execution paths. Others use GPT-style models to suggest edge-case inputs or state transitions likely to reveal hidden bugs.

Automated Program Repair and Natural Language Processing

Perhaps the most impactful shift has been the integration of **LLMs** directly into smart contract reasoning workflows. GPT-based tools like SmartEval [45] and SolGPT [57] can explain contract behavior in natural language, identify vulnerabilities from code snippets, or generate Solidity code from high-level prompts.

APR is another promising domain. Early efforts like sGuard [58] and Elysium [59] applied static or bytecode-level rewriting to fix known bugs, but recent advances have focused on learned repair synthesis. Although **LLMs** might not be the best **AI** approach to detect vulnerabilities, they are strong models for writing code. Hence, tools like Contract-Tinker [60], TRT [61], and RepairCAT [62] use **LLMs** such as Codex or CodeT5 to suggest and refine contract patches. SmartInv [46] generates candidate invariants to guide fuzzing or runtime checks. These techniques allow vulnerability mitigation pipelines to operate semi-autonomously, reducing developer workload and shortening security response times.

2.1.7 Large Language Models in Code Analysis and Repair

LLMs, such as GPT-4 and Code LLaMA, have demonstrated remarkable capabilities in understanding, generating, and manipulating source code. Trained on vast corpora of natural and programming languages, **LLMs** can assist with tasks such as bug detection, code summarization, and automated repair [53, 63].

In the context of smart contracts and blockchain security, **LLMs** enable new forms of program reasoning that combine pattern recognition with contextual understanding. They can generate targeted patches for vulnerabilities, synthesize test cases, and even collaborate with static analyzers or fuzzers to support human auditors.

This thesis builds upon these advances by incorporating **LLMs** into the automated detection and repair pipeline for gas-related vulnerabilities in Ethereum smart contracts. It demonstrates high repair accuracy and low manual overhead.

2.1.8 Blockchain Oracles and Decentralized Oracle Networks

Smart contracts, by design, operate within a closed environment. Thus, they can access and manipulate only the data that resides on the blockchain. However, many real-world applications require interaction with off-chain data, such as asset prices, weather conditions, election results, or identity records. *Blockchain oracles* serve as the trusted bridge between on-chain contracts and off-chain information [9, 64].

Oracles can be broadly categorized into different types based on their design and trust assumptions. **Software oracles** fetch data from online sources such as **APIs**, while **Hardware oracles** interact with physical sensors and **IoT** devices. **Inbound oracles** bring external data onto the blockchain and **Outbound oracles** enable contracts to trigger off-chain actions. Finally, **Consensus-based oracles** aggregate inputs from multiple sources to increase reliability.

The simplest form of an oracle is a centralized data feed, a single node responsible for injecting off-chain data into the blockchain. While easy to implement, centralized oracles reintroduce a single point of failure and defeat the core promise of decentralization. An adversary controlling the oracle can corrupt contract behavior, steal funds, or cause denial-of-service, leading to systemic risk in decentralized applications (dApps) [65, 13].

To mitigate these risks, the concept of *Decentralized Oracle Networks* (DONs) has emerged. A DON consists of multiple independent nodes or data providers collectively validating and submitting data to the blockchain. By distributing trust, DONs enhance resilience against manipulation, downtime, or censorship.

One of the most widely used DONs implementations is **Chainlink** [9, 66]. Chainlink provides a flexible framework where multiple oracles retrieve and verify data independently before aggregating it on-chain through smart contracts. Other notable oracle systems include **Town Crier** [65], which uses trusted hardware enclaves (e.g., Intel SGX) to securely fetch and attest data, and **Band Protocol**, which leverages delegated proof-of-stake and off-chain computation for faster data feeds.

2.1.9 Counting Systems in Blockchain Applications

Accurate and tamper-resistant counting is a foundational requirement across many distributed systems, particularly where fairness, reputation, or monetary value depend on aggregated data. Examples include vote tallying, usage metering, impression counting in advertising, and user engagement analytics. In traditional systems, such counts are maintained by centralized servers, making them vulnerable to manipulation, misreporting, or unauthorized access [67].

In blockchain environments, counting poses additional challenges due to the need for decentralization, trustlessness, and gas efficiency. Naively implementing counters on-chain, for example, by iterating through a growing array to count items, quickly leads to scalability issues and potential DoS risks due to gas exhaustion. Furthermore, smart contracts cannot inherently observe or aggregate off-chain events (e.g., user clicks or impressions), requiring integration with oracles or other trusted middleware.

These limitations become particularly acute in blockchain applications related to online advertising. The advertising industry traditionally relies on centralized ad servers and measurement providers to count impressions, clicks, conversions, and viewability metrics. This centralized model has long been criticized for its opacity, lack of verifiability, and susceptibility to discrepancy fraud, where publishers and advertisers report different counts for the same campaign [68].

In the Web3 context, a decentralized approach to ad measurement and bidding can potentially resolve many of these issues. This thesis addresses these issues by introducing two blockchain-based systems: *CountChain*, which provides a decentralized and game-theoretic approach to counting across untrusted nodes, and *AdChain*, which applies this framework to online advertising to reduce fraud and automate verifiable bidding protocols.

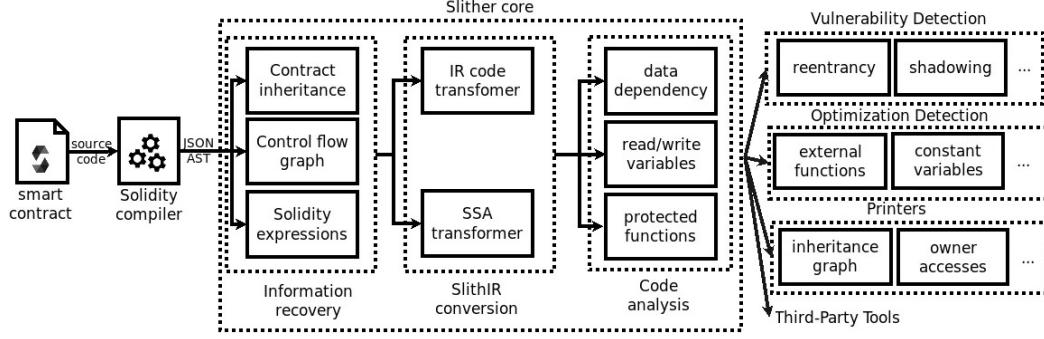


Figure 2.1: Overview of Slither’s architecture and analysis pipeline [1].

2.1.10 Game Theory and Nash Equilibrium

Game theory provides a mathematical framework for analyzing interactions among rational agents who make decisions based on incentives and expected outcomes. In blockchain systems, game-theoretic reasoning is essential for designing secure and effective protocols even when participants act strategically to maximize their own benefit [69].

A key concept is the *Nash equilibrium*, a stable state in which no participant can improve their outcome by unilaterally changing their strategy [70]. In decentralized settings, protocols designed around Nash equilibria ensure that honest behavior is the best response for each participant, given the behavior of others.

Applications of game theory in blockchain include consensus protocols, staking mechanisms, incentive alignment in [DONs](#), and fraud-resistant reporting in systems like those proposed in this thesis. By grounding protocol design in formal models of rationality, game-theoretic analysis enhances robustness against collusion, manipulation, and incentive misalignment.

2.2 Helper Tools

This thesis leverages two primary tools in the Ethereum security ecosystem, **Slither** and **Foundry**, to support static analysis, vulnerability detection, fuzz testing, and automated repair validation.

2.2.1 Slither: Static Analysis Framework

Slither is a widely used static analysis framework for Solidity smart contracts developed by Trail of Bits [1]. As shown in Figure 2.1, Slither analyzes source code by building an intermediate representation (IR) of the contract and extracting key semantic features such as function calls, inheritance relationships, and control-flow graphs.

Slither includes built-in detectors for common vulnerabilities (e.g., reentrancy, uninitialized storage, unbounded loops) and enables custom rule development. Its modular

design and command-line interface make Slither a foundational tool for integrating symbolic insights into automated repair workflows.

2.2.2 Foundry: Fuzzing and Test Automation

Foundry is a fast, modular Ethereum development toolkit built in Rust that includes tools like `forge` (for compiling, testing, and fuzzing contracts) and `cast` (for contract interaction). Foundry offers high-speed execution, first-class fuzzing, and native support for Solidity, making it well-suited for testing the runtime behavior of smart contracts [39].

Unlike earlier frameworks such as Truffle or Hardhat, Foundry compiles directly to [EVM](#) bytecode and avoids JavaScript-based bottlenecks, enabling fast iteration and tighter feedback loops between testing and repair.

2.3 Related Work

Secure and scalable blockchain systems depend on both trustworthy computation and reliable external data integration. Securing smart contracts has been a long-standing challenge in blockchain research. Prior work has introduced static analyzers, fuzzers, and oracle protocols, each tackling part of the problem. More recently, AI-driven tools and incentive-based designs have opened new possibilities for automation and verifiability.

Meanwhile, research on decentralized oracle systems highlights persistent challenges related to incentive alignment, verifiability, and real-time responsiveness. Many proposed architectures attempt to address these issues, but often rely on fixed data sources, manual reporting, or suffer from single points of failure.

2.3.1 Smart Contract Analysis and Gas-Related Vulnerability Detection

Many smart contract verification tools scan contracts for known vulnerabilities such as reentrancy, integer overflows, and access control misconfigurations. Some of the most well-known general-purpose tools include Manticore [36], MPro [71], Mythril [34], Securify (deprecated) [72], Securify 2.0 [32], Slither [1], SmartCheck [31], Verx [73], and Zeus [35]. These tools use a range of static analysis and symbolic execution techniques to flag known patterns of unsafe logic and provide audit support for Solidity contracts.

However, while effective at detecting logic-level bugs, most tools fail to detect gas-related vulnerabilities or suffer from high false negative rates when analyzing resource-exhaustion patterns such as unbounded loops or high-cost execution paths. For example, fuzzing-based tools like ContractFuzzer [40], Echidna [38], and Harvey [41] are effective at runtime path exploration but cannot reason about symbolic gas costs and [DoS](#) conditions.

A small number of tools explicitly target gas vulnerabilities. GASTAP [10] performs static gas-bound analysis using abstract interpretation. It infers size relations for public

functions, generates symbolic gas equations, and solves them to estimate worst-case costs. MadMax [12] statically detects gas-focused issues such as unbounded storage writes and loops. Despite these innovations, most existing gas-aware tools are limited to detection and offer little or no guidance on mitigation or repair.

To address these limitations, the GasGauge tool [4] combines static and dynamic techniques to model loop behavior, identify vulnerable patterns, and derive gas usage formulas. However, it stops short of suggesting direct code changes. GasGuard [2] builds on this by leveraging LLMs to insert ‘require’ guards based on symbolic gas thresholds. Finally, GasGaugeAI [3] introduces a feedback-driven multi-LLM system that classifies vulnerabilities, generates test cases using Foundry, and synthesizes secure logic to prevent gas-based DoS attacks. These tools are among the first to bridge symbolic gas reasoning with LLM-based automated repair workflows.

2.3.2 Automated Program Repair

APR has been a long-standing area of research in traditional software engineering and is now gaining traction in the domain of smart contracts. Early APR tools such as GenProg [74], Prophet [75], and SemFix [76] demonstrated the feasibility of automatic patch generation through techniques like genetic programming, probabilistic modeling, and symbolic execution. These systems perform effectively in general-purpose languages like C and Java, using unit test oracles to validate behavior. However, applying them directly to smart contracts is problematic due to domain-specific challenges such as immutability, gas constraints, storage cost sensitivity, and incomplete oracles for verification.

Recent APR frameworks have adopted learning-based approaches. Getafix [77], deployed at Facebook, mined recurring bug-fix patterns from historical commits to generate standard patches for Java code. ChatRepair [78] introduced an LLM-driven dialogue loop that iteratively refines patches using test feedback. Other notable approaches include CodeReviser [79] and RepairCAT [62], which employ encoder-decoder models like CodeT5 and StarCoder to learn buggy-to-fixed code mappings. While these techniques have improved general-purpose repair benchmarks, they do not address Ethereum-specific execution models, gas costs, or the composability requirements of smart contracts.

APR faces unique challenges in the smart contract domain: contract immutability, gas metering, and deployment constraints. Several tools have emerged to address these issues. Template-based systems like SCRepair [80], ContractFix [81], and TIPS [82] apply predefined patches based on static analysis warnings from detectors, such as Slither and Securify. SmartFix [83] and HCC [84] explore constraint-solving and meta-learning approaches but require detailed symbolic specifications and are typically limited to arithmetic and reentrancy vulnerabilities.

Elysium [59] and Aroc [85] introduce low-level repair strategies. Elysium performs context-aware bytecode patching without requiring source code access, enabling fixes post-deployment. Aroc uses automatic recompilation and off-chain rollback mechanisms but depends on legacy EVM features. While these approaches expand coverage, they often lack scalability or suffer from low precision under complex logic.

Several [APR](#) tools have explicitly focused on Solidity. [sGuard](#) [58] rewrites contract logic using predefined patterns but struggles with complex dependency chains. [SmartShield](#) [86] and [DeFinery](#) [87] use symbolic execution and heuristics for vulnerability resolution, but often alter business logic or degrade performance. [sGuard+](#) [88] builds upon [sGuard](#) using refined rule learning and partial machine learning models to enhance coverage, though its lack of theoretical soundness limits robustness.

Recent work has explored [LLMs](#)-based repair for smart contracts. [TRT](#) [61], [ContractTinker](#) [60], and [SolGPT](#) [57] use GPT-based models and vulnerability descriptions to generate secure code suggestions. These tools demonstrate the potential of natural language-conditioned repair for Solidity but remain limited by the need for static analysis scaffolding, human validation, and a lack of support for state-aware or gas-sensitive logic.

The tools introduced in this thesis, [GasGuard](#) and [GasGaugeAI](#), address many limitations by combining symbolic gas modeling with [LLMs](#)-based patch generation and test-driven validation. They are among the first systems to specialize [LLMs](#) for gas-based [DoS](#) vulnerabilities and demonstrate fully automated repair pipelines grounded in Ethereum-specific semantics.

2.3.3 Decentralized Oracle Networks

[Chainlink](#) [9] is the most prominent [DON](#) architecture to date, providing cross-chain services for verifiable data feeds, random number generation, external API integration, and automated execution. Other platforms adopt alternative designs. [Neo Oracle Service](#) [89] facilitates external data access through a committee of designated oracle nodes, maintaining a semi-decentralized governance model. [TON](#) [90], a decentralized messaging application, introduces a messaging oracle layer where off-chain messages are transmitted and verified by lightweight agents before being fed on-chain. These models trade off scalability, trust assumptions, and implementation complexity.

Prediction markets have also inspired [DON](#)-like systems. [Augur](#) [91] and [Gnosis](#) [92] use staking and consensus games to crowdsource answers to binary or categorical propositions. These platforms reward truthfulness under game-theoretic equilibrium assumptions but are limited to specific use cases and do not provide general-purpose data feeds.

More theoretical [DONs](#) have emerged as well. [Astraea](#) [93] is a decentralized truth-determination system that treats oracle data submission as a voting game, deciding the veracity of claims through economic incentives. Similarly, a Smart Contract System for Decentralized Borda Count Voting [94] introduces a ranked-choice e-voting system based on the Borda count, offering self-tallying and privacy-preserving computation suitable for decentralized governance.

In the context of financial applications, [Distributed Blockchain Price Oracle \(DBPO\)](#) [95] provides exchange rate feeds for digital assets using a decentralized system of aggregators. It is used to safeguard collateral valuations and settlement procedures in [DeFi](#) ecosystems. Likewise, [DAG-Based Decentralized Oracle Model](#) [96] uses a directed acyclic graph structure to validate centralized data inputs (e.g., Bitcoin price) through decentralized consensus mechanisms.

Despite their diversity, most [DONs](#) either rely on predefined sources, do not support real-time data delivery, or require significant manual reporting. In contrast, the CountChain protocol introduced in this thesis enables decentralized oracles to initiate data submissions based on local observations autonomously. It uses distributed voting and lazy commitment to support real-time, auditable counts, paving the way for vertical applications like AdChain in online advertising.

2.3.4 Blockchain-Based Advertising and Counting Systems

Several platforms have explored blockchain-based advertising systems. AdHash [\[97\]](#) proposes a custom blockchain that eliminates intermediaries and enables advertisers and publishers to transact directly, with complete control over campaign data. Alkimi Exchange [\[98\]](#) is a decentralized ad exchange built to replace legacy real-time bidding (RTB) infrastructure with more transparent, user-friendly auction protocols. Adwatch [\[99\]](#) offers advertisers customizable rule-based campaign contracts and tracking via smart contracts, enabling full transparency of engagement metrics.

While these systems make valuable strides, they often require a complete overhaul of the current digital advertising infrastructure. Many aim to remove intermediaries entirely, some offering real value in fraud prevention, auction management, and content curation. Moreover, platforms like AdHash and Adwatch store campaign metrics directly on-chain, raising privacy concerns when sensitive impression or user behavior data is shared publicly.

AdChain [\[6\]](#) provides a reusable counting layer across verticals, from advertising to resource tracking, ensuring high precision with minimal overhead and no reliance on external trust.

Chapter 3

Automated Detection of Gas-Based Vulnerabilities in Smart Contracts

As blockchain technology evolves and adoption expands across finance, governance, and infrastructure, its foundational layers have become critical targets for adversaries. Unlike most traditional software, blockchains are public, immutable, and incentive-driven systems where vulnerabilities at the protocol level can have catastrophic consequences [16, 24]. These system-level vulnerabilities, rooted in the architecture, consensus mechanism, and network assumptions of blockchain platforms, pose persistent threats to decentralization, security, and liveness.

Many of these vulnerabilities overlap with those found in traditional distributed systems, such as Sybil attacks and network partitions [100, 27]. However, blockchain introduces novel risks due to its open validator model, mempool mechanics, gas metering, and probabilistic consensus [14]. For instance, blockchain’s gas-based issues, MEV, and finality reversion are unique and stem directly from economic and timing assumptions embedded in block production [26, 101].

A prominent example of such a gas-related vulnerability is DoS via the block gas limit, where unbounded loops or excessive on-chain computation can permanently lock functions, degrade network responsiveness, or stall contract execution. This chapter introduces GasGauge [4, 102], a static and dynamic analysis tool designed to detect such DoS vulnerabilities. This work was published at *the MARBLE 2022 conference*.

The immutable nature of blockchain protocols heightens the need for robust detection methods [29]. This chapter categorizes key system-level vulnerabilities and reviews the detection methods used to identify and analyze them. We emphasize the evolution of tooling from traditional static analysis and symbolic execution to modern hybrid techniques and AI-assisted techniques. By exploring the intersection of vulnerabilities and detection strategies, this chapter and the next chapters lay the groundwork for understanding how blockchain security tooling has evolved and how emerging techniques can address long-standing gaps in visibility, automation, and resilience.

3.1 Motivation and Background

Gas-related [Denial-of-Service \(DoS\)](#) vulnerabilities pose a persistent and often underestimated risk in Ethereum smart contracts. Unlike conventional vulnerabilities such as reentrancy or arithmetic overflows, gas-based flaws do not necessarily result in immediate fund loss. Yet, they can permanently disable critical contract functionality or render [Governance](#) and upgrade mechanisms unusable. In high-value deployments, these silent failures have the potential to halt protocols, censor users, or lock funds indefinitely [29].

The root cause of these vulnerabilities lies in Ethereum’s gas model, which charges a per-instruction execution fee to prevent infinite loops and denial-of-service attacks. However, this metering mechanism creates a new attack surface where carefully crafted transactions can trigger costly loops or nested operations that exceed the block [Gas Limit](#), causing transactions to revert. If a smart contract’s core logic, such as token migration, batch payments, or governance proposal processing, depends on user input or unbounded [State Bloat](#), it becomes vulnerable to gas exhaustion. Attackers can exploit this by bloating the state or crafting inputs that cause legitimate operations to fail repeatedly [4, 10, 12].

Although several tools such as MadMax [12], Gastap [10], and Securify [32] attempt to analyze gas usage or detect unbounded loops, they suffer from high false-positive rates, limited repair suggestions, or lack of semantic context about loop behavior. Moreover, most existing tools do not provide actionable fixes for vulnerable contracts, leaving developers with limited recourse beyond manual refactoring or complete rewrites. Therefore, we developed GasGuard and GasGaugeAI to address this.

To address this gap, we developed **GasGauge** [102] as a static analysis and fuzzing framework that could detect and simulate gas-exhausting inputs across smart contracts. The tool decomposed contract code into [Control Flow Graph \(CFG\)](#) summaries, identified loops, and predicted worst-case input behaviors using symbolic fuzzing. Although the initial prototype was effective at identifying vulnerable patterns, it did not yet offer automated mitigation or repair.

3.2 Hybrid Approaches for Detection and Repair of Security Vulnerabilities

As discussed in Chapter 2, the most common methods to detect and repair vulnerabilities in smart contracts are static analysis, symbolic execution, fuzzing, and AI-based approaches. However, each of these methods has some limitations.

Hybrid approaches combine multiple vulnerability detection techniques to compensate for the limitations of individual methods [55, 56]. These methods are particularly valuable for analyzing complex smart contracts with dynamic states, composability, or resource-bound behaviors.

Common patterns in hybrid detection include static pre-analysis followed by guided fuzzing, symbolic path exploration refined with reinforcement learning, and runtime ver-

Tool / Framework	Key Features	Combined Techniques
MAIAN [55]	Symbolically explores contract paths and uses learned heuristics to detect suicidal, greedy, and prodigal contracts.	Symbolic + ML
Confuzzius [56]	Integrates symbolic execution and reinforcement learning to guide fuzzing toward attack sequences.	Symbolic + Fuzzing + RL
MPro [71]	Uses static analysis and symbolic checks to validate user-defined security properties.	Static + Symbolic
GasGauge [4]	Analyzes loop structures using static summarization and fuzzing to derive safe iteration bounds.	Static + Fuzzing + Runtime Verification
GasGuard [2]	Leverages symbolic gas formulas and LLMs to automatically insert ‘require’ guards for gas safety.	Static + Symbolic + LLM
GasGaugeAI [3]	Multi-LLM pipeline that combines test-driven feedback (Foundry) with learned repair logic and symbolic gas modeling.	Symbolic + Fuzzing + LLM

Table 3.1: Hybrid vulnerability detection and repair tools combining multiple techniques.

ification inserted through AI-generated invariants. While hybrid systems are more computationally intensive, they are often the most effective at uncovering subtle or context-dependent vulnerabilities in Ethereum and other smart contract platforms. Table 3.1 lists some of the main tools that use the hybrid approach in smart contract security.

The final three tools in Table 3.1, GasGauge, GasGuard, and GasGaugeAI, are central contributions of this thesis. They represent a progressive evolution of hybrid vulnerability detection and repair for gas-based DoS vulnerabilities. These systems combine static analysis, fuzzing, runtime verification, symbolic reasoning, and large language models to detect and fix looping constructs that may exceed Ethereum’s gas limits.

This chapter and the next two chapters provide a deep exploration of these tools, detailing their architecture, implementation, and the novel techniques used to synthesize secure repairs. Together, they represent a step toward fully automated and explainable mitigation systems for critical vulnerabilities in smart contracts. The remainder of the chapter details the motivation, system design, and empirical findings behind each iteration.

3.3 Out-of-Gas Denial-of-Service Vulnerabilities

Ethereum smart contracts are susceptible to vulnerabilities where execution fails due to insufficient gas, resulting in an Out-of-Gas (OOG) exception. When a transaction exceeds its gas allocation, either due to a user-defined limit or the Block Gas Limit, the EVM halts execution, reverts State Changes, and still charges the sender for the consumed

```

1 contract SmallBank {
2     address[1000] fixedUsers = [address1, address2, ..., address1000];
3
4     function addInterestWithFixedArray(uint interest) public {
5         for(uint i = 0; i < fixedUsers.length; i++) {
6             fixedUsers[i].call.value(interest)();
7         }
8     }
9
10    function addInterestWithInputVariable(address[] memory users, uint interest) public {
11        for(uint i = 0; i < users.length; i++) {
12            users[i].call.value(interest)();
13        }
14    }
15
16    address[] public users;
17    function addInterestWithStateVariable(uint interest) public {
18        for(uint i = 0; i < users.length; i++) {
19            users[i].call.value(interest)();
20        }
21    }
22
23    function addUser(address newUser) public {
24        users.push(newUser);
25    }
26 }

```

Figure 3.1: A smart contract vulnerable to DoS with block gas limit [2]

gas [103]. While this model safeguards network stability, it introduces a new class of DoS vulnerabilities when smart contracts are not Gas-Aware.

A typical scenario is the *DoS with Block Gas Limit*, where unbounded or poorly bounded loops exceed the maximum gas that can be spent in a single block [4]. This is especially problematic in loops controlled by user input or state variables, where iteration counts may unpredictably grow. Such loops, once exceeding the gas threshold, render critical contract functions permanently unusable.

To secure smart contracts from this failure mode, developers must identify gas-sensitive loops and bind their execution using safe constraints. These constraints are often based on loop complexity, prior gas consumption, and the dynamic nature of loop bounds, particularly those influenced by state variables.

The contract in Figure 3.1 shows three loop variants:

- A **constant-bound loop** over a fixed number of elements,
- An **input-bound loop** over user-supplied data,
- A **state-bound loop** where the size of the state variable increases over time.

Each case requires different strategies for detection and remediation.

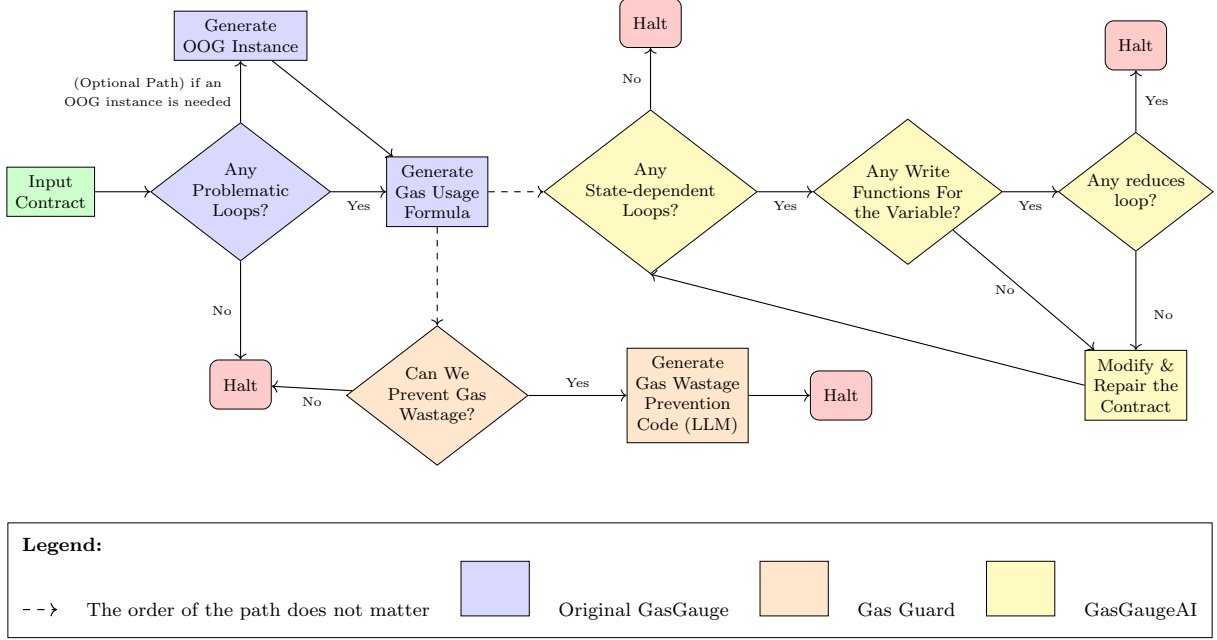


Figure 3.2: Workflow of Identifying and Mitigating Out-of-Gas Vulnerabilities [3]

3.4 Steps to Identify and Mitigate OOG Vulnerabilities

To mitigate gas-related vulnerabilities, especially those resulting from unbounded loops, we propose a 10-step pipeline that blends static analysis, runtime evaluation, fuzzing, and LLM-based repair strategies. These 10 steps are as follows [3]:

1. **Loop Detection:** Perform static analysis to locate all loops within the contract.
2. **Loop Risk Assessment:** Analyze each loop's structure and an assessment of how easy it is to trigger an OOG instance.
3. **OOG Trigger Discovery:** Use a fuzzer to simulate realistic usage scenarios and find an OOG instance.
4. **Iteration Bound Discovery:** Perform runtime verification to calculate the maximum number of safe loop iterations.
5. **Gas Usage Formula Derivation:** Derive a gas-based formula using `gasleft()` that models the safe upper bound for the loop.
6. **Variable Classification:** Identify and categorize all variables influencing the loop bounds into constant, input, or state categories. This classification determines the repair strategy.

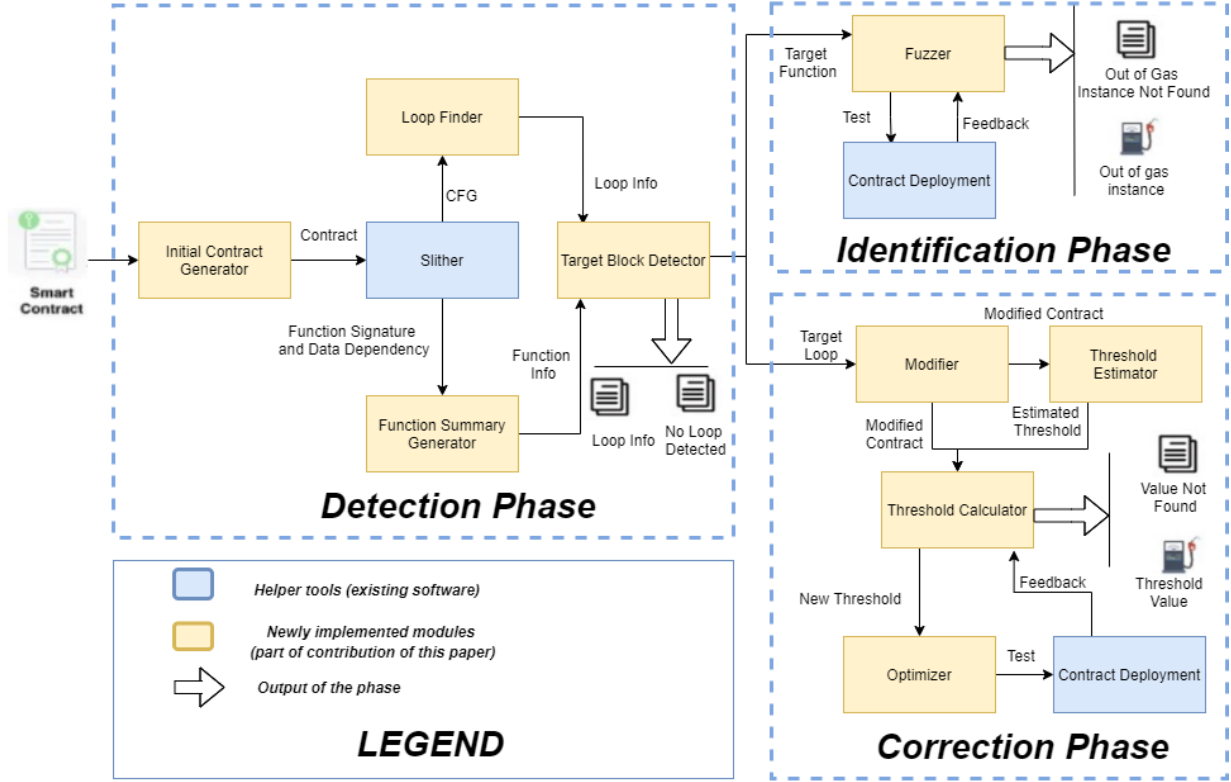


Figure 3.3: GasGauge architecture consisting of detection, identification, and correction phases [4].

7. **Preventive Guard Insertion:** Insert a `require()` statement before the loop using the derived formula to prevent execution when the gas limit is exceeded.
8. **State Variable Analysis:** If a state variable controls the loop, statically identify whether any existing functions in the contract write to this variable in a way that reduces its value.
9. **Functionality Validation via Testing:** Generate Foundry test cases to determine whether any identified functions correctly reduce the loop-bound state variable in realistic usage patterns.
10. **Missing Logic Synthesis:** If no such behavior exists, automatically synthesize the missing decrement functionality using an LLM and revalidate it through further fuzzing and testing to ensure the loop is now bounded and safe.

Figure 3.2 presents an overview of this process, with each phase annotated by the GasGauge tool that performs it. As shown, Steps 1–5 are handled by the original **GasGauge** tool [4]. **GasGuard** [2] performs steps 6–7 by inserting gas-aware guards using LLMs. **GasGaugeAI** [3] completes the pipeline through steps 8–10, applying multi-LLM reasoning to identify state-bound loop controls and generate secure repairs through testing and synthesis.

```

1 function distribute(uint256 amount) public {
2     for (uint i = 0; i < recipients.length; i++) {
3         recipients[i].transfer(amount);
4     }
5 }

```

Figure 3.4: Vulnerable Code Snippet from one of Airswap’s contracts

Together, these tools represent a complete framework that goes beyond vulnerability detection to offer secure, automated patch generation tailored to gas-constrained decentralized environments.

3.5 GasGauge: A Security Analysis Tool for Smart Contract Out-of-Gas Vulnerabilities

GasGauge was initially developed to address a subtle yet impactful class of **DoS** vulnerabilities due to **OOG** failures in unbounded loops. While most security tools focused on known patterns like reentrancy or arithmetic overflows, GasGauge targeted a less explored but increasingly common issue where loop iterations grow with state or input and eventually cause the contract to fail due to **Block Gas Limit** [4].

3.5.1 Overview and Summary of Experimental Results [102]

As shown in Figure 3.3, the tool operates in three core phases: (1) a static analysis phase that summarizes loop structures using a **Abstract Syntax Tree (AST)** and **Control Flow Graph (CFG)**, (2) a white-box fuzzing phase that generates transactions to trigger gas exhaustion, and (3) a correction phase that computes a gas-safe iteration threshold based on runtime simulations. When a vulnerability is confirmed, GasGauge suggests inserting a **require()** guard to prevent **OOG** execution.

The tool was evaluated on a benchmark of 1,000 real-world contracts, achieving 0% false positives and 0% false negatives in the detection phase. GasGauge was also competitive in runtime performance and consistently outperformed seven state-of-the-art tools, including GasTap, MadMax, MPro, Mythril, SmartCheck, Securify 2.0, and Slither, in its ability to detect gas **DoS** vulnerabilities without over-reporting [4]. These results established GasGauge as one of the first practical tools to detect and simulate gas-related failures in Ethereum smart contracts accurately.

3.5.2 A Case Study: Analyzing the Airswap Protocol

To showcase the practical impact and real-world utility of GasGauge, we conducted a focused case study on the *Airswap* smart contract¹, a well-known decentralized token

¹<https://etherscan.io/address/0x8d48f9b61affaaa2e5a7402e87d27eaa0cc27b6f>

trading protocol that had previously undergone formal manual auditing by Quantstamp², one of the industry’s leading smart contract security firms.

The code snippet for one of the vulnerable functions in this contract is provided in Figure 3.4. As shown, the function `distribute()` iterates over the `recipients` array, which grows over time via user registrations. Without enforcing an upper bound, this function becomes increasingly gas-intensive. Eventually, it fails for all users, effectively freezing contract funds, a vulnerability that can emerge even without malicious actors.

According to Quantstamp engineers, it took them around 4 hours of comprehensive manual review to detect the unbounded loop vulnerability in one of the contracts and to provide a gas analysis report. Meanwhile, GasGauge automatically uncovered this critical gas-related vulnerability in the `distribute` function. This loop iterated over a dynamically growing array of *recipients*, introducing an OOG risk as the array expanded over time. When triggered, this vulnerability could cause transaction failures once the loop consumed more gas than the *block gas limit*, allowing a subtle and difficult-to-detect issue during static code review.

GasGauge autonomously analyzed the `distribute` function, identified the unbounded loop, and performed per-iteration gas estimation, determining that each iteration consumed approximately 5,000 gas units. Based on this, it automatically derived a threshold-based formula to safely cap loop iterations at runtime according to the available gas, thereby mitigating the risk of OOG failures.

What makes this result particularly compelling is the efficiency gained. While manual auditing of similar gas-related loop behaviors typically requires 4–5 hours of expert analysis, GasGauge completed the full detection, analysis, and reporting process in less than 15 minutes, with minimal human supervision required. This represents a 16× reduction in auditing time, allowing auditors and developers to focus their efforts on more complex or business-critical logic.

This case study highlights the value of automated gas analysis tools in practice. Even in professionally audited, production-grade contracts, subtle gas-related vulnerabilities can persist unnoticed. GasGauge fills this gap by delivering targeted, automated, and highly actionable analysis for *gas safety*, transforming a traditionally manual and labor-intensive process into a fast, reliable, and repeatable workflow.

3.5.3 Lessons Learned and Output Report

Through its deployment on real-world contracts such as Airswap and others, several key lessons emerged from the development and application of GasGauge:

- **Loop-Based Vulnerabilities Are Underreported:** Many audited smart contracts still contain gas-based DoS vulnerabilities due to loops that grow with user input or persistent state. These vulnerabilities often escape detection by traditional tools focused on reentrancy or overflow issues.

²<https://quantstamp.com/>

```

1 Report for 0xd848f9b61affaaa2e5a7402e87d27eaa0cc27b6f.sol:
2 Found 1 loop in 1 function with the following information:
3 Function Name: VAPE.multiTransfer(address[],uint256[]), Visibility: public, Number of Loops: 1
4 -----
5 Results for VAPE.multiTransfer(address[],uint256[]):
6 inputs affecting the function loop bound(s) are: input 1 (address[] memory receivers);
7
8 Results of the Threshold Finder:
9 for loop 1 in VAPE.multiTransfer(address[],uint256[]):
10 The type is Normal
11 For gas limit = 6721975, the calculated threshold = 899
12 gas consumption for the first iteration = 47420; average gas consumption for other iterations = 5932
13 Threshold formula (gasleft() is the global function gasleft() placed before entering the loop ) =
14 (gasleft() - 47420) / (5932)
15 -----
16 Run time = 422.0240927560001 seconds

```

Figure 3.5: GasGauge output showing per-iteration gas usage and derived threshold formula [4].

- **Tool-Aided Audits Are Essential:** Manual auditing alone is insufficient for identifying gas-related risks. GasGauge’s ability to systematically analyze loop structures and simulate realistic gas usage scenarios proved vital for identifying latent issues.
- **Runtime Thresholds Enable Practical Repairs:** Estimating per-iteration gas consumption and calculating gas-aware iteration thresholds allows developers to insert runtime guards using the `gasleft()` function. This enables bounded execution without sacrificing core logic.

To support developer adoption, GasGauge provides an output report summarizing the vulnerability analysis for each loop. Figure 3.5 shows a sample output generated by the tool. The report includes the function name, loop location, variables affecting the loop, gas consumption for the first and subsequent iterations, and the calculated threshold formula. This formula ensures that the loop will halt before triggering an `OOG` error, even if the number of elements increases over time. By placing this check before the loop, developers can maintain functional correctness while safeguarding against `DoS` conditions.

The output format was designed to be interpretable by both smart contract developers and downstream automated repair tools, such as GasGuard and GasGaugeAI, as discussed in the following chapters.

This work highlights how runtime accountability and data accountability reflect a shared architectural vision. This vision aims to make smart contracts not only self-executing but also self-verifying in both logic and input. Through this convergence, we move closer to building decentralized systems that are not merely permissionless but provably trustworthy.

3.5.4 Integrating AI and Blockchain for Scalable, Automated Smart Contract Repair

The growing intersection of `AI` and blockchain is integrated into the practical tools and methodologies developed across multiple chapters. In particular, the GasGuard and Gas-

GaugeAI frameworks exemplify how [LLMs](#) can be incorporated into blockchain security pipelines to achieve automated, explainable, and scalable smart contract repair.

GasGuard is one of the first tools in this work to use [LLMs](#) for guided vulnerability mitigation. Built on top of the original GasGauge engine, GasGuard takes vulnerability reports, specifically loops susceptible to [OOG](#) denial-of-service, and leverages a prompt-engineered GPT-based model to generate proper guards to avoid gas wastage. These guards are tailored to enforce safe iteration bounds derived from symbolic gas analysis. Rather than relying solely on predefined templates, the [LLM](#) dynamically synthesizes code that respects loop context, state dependencies, and Ethereum’s gas limits [2].

GasGaugeAI extends this concept into a multi-[LLM](#) architecture with a feedback-driven repair loop. The first [LLM](#) (LLM-1) focuses on test-case generation. It uses loop metadata and symbolic constraints to generate Foundry-based unit tests that verify whether a state-dependent loop can be safely bounded or whether an existing control function can reduce gas usage. The second [LLM](#) (LLM-2), built on GPT-4.0 (32k) and trained via few-shot learning, is tasked with synthesizing secure and context-aware patches that may introduce guards, decrement logic, batching systems, or mapping-based refactors [3]. The third [LLM](#) (LLM-3), built on GPT-4.1 and trained on a curated taxonomy of known repair patterns, classifies candidate patches and provides structured feedback to LLM-2 when refinements are required.

This architecture transforms [LLMs](#) from static code generators into interactive agents that reason over symbolic conditions, test outputs, and loop states. Patches are classified and validated through a feedback loop involving LLM-2, LLM-3, and LLM-1 until the repair is proven safe through empirical testing.

A key feature of both tools is that [AI](#) does not replace static or formal methods; it augments them. The [LLMs](#) used in GasGuard and GasGaugeAI rely on grounding context extracted from symbolic loop summaries, static state variable classification, and Foundry test outputs. This hybrid design ensures that the [AI](#)-driven suggestions are not generated in a vacuum but are instead tied to concrete behavioral insights and verified runtime feedback.

Ultimately, this thesis positions [LLMs](#) not as magic boxes, but as intelligent assistants embedded in an adversarial-aware repair loop. They translate abstract vulnerability reports into actionable patches, generate human-readable invariants, and reason across symbolic and behavioral layers. These capabilities point toward a new model of smart contract security, where detection, reasoning, and repair co-exist in an autonomous but auditable workflow.

The lessons from these tools can also be generalized. Similar [LLM](#)-based assistants could be extended to detect and repair other classes of vulnerabilities, such as access control flaws, reentrancy, or unhandled exceptions. As language models continue to improve, so too will their ability to reason about decentralized systems in automated and accountable ways.

Despite this progress, challenges remain. [LLMs](#) are prone to hallucinations and may generate code that appears correct but fails under edge cases [53]. Datasets used for training

can be noisy or biased, and current models provide no formal correctness guarantees. As a result, AI-driven tools are best used in tandem with traditional techniques, augmented by symbolic solvers, test feedback, and static reasoning, to ensure soundness and verifiability.

3.6 Conclusion and Discussion

This chapter introduced GasGauge, a tool for detecting OOG vulnerabilities caused by unbounded loops in Ethereum smart contracts. By combining static analysis, fuzzing, and runtime gas profiling, GasGauge identifies gas-intensive patterns and suggests safe iteration limits. Its effectiveness was demonstrated through large-scale testing and a real-world case study on the Airswap protocol.

This work laid the foundation for the automated analysis of gas-based DoS vulnerabilities. However, GasGauge focuses primarily on detection and estimation, leaving the repair process to the developer.

The next chapter builds on this with GasGuard, which introduces automated mitigation using large language models. This evolution continues in GasGaugeAI, which closes the loop by synthesizing, testing, and verifying secure patches through AI-driven repair.

Chapter 4

Towards Automatic Program Repair for Gas-Based Vulnerabilities in Smart Contracts

Gas-based vulnerabilities remain a persistent and subtle threat to the reliability and availability of Ethereum smart contracts. While logic bugs and security flaws such as reentrancy or access control issues often draw immediate attention, gas-related risks, particularly those leading to Out-of-Gas (OOG) failures, are equally capable of disrupting contract execution, especially under dynamic runtime conditions [104, 105, 106]. These vulnerabilities are notoriously difficult to catch through traditional static analysis or manual review alone, as they often depend on the evolving size of arrays, mappings, or other state variables that influence the control flow [2].

GasGuard [2] was designed to address this gap through an automated, scalable, and intelligent analysis pipeline. By leveraging the outputs of the *GasGauge* toolchain and combining them with a fine-tuned Large Language Model (LLM), GasGuard performs targeted repairs on vulnerable loops, introducing safeguards and runtime checks to mitigate gas risks. This chapter presents the design, architecture, and evaluation of GasGuard, demonstrating how it transforms hours of manual vulnerability triage and mitigation into a streamlined, automated process, thereby accelerating secure smart contract development and reducing reliance on expert auditors. This work is published at *the MARBLE 2025 conference*.

4.1 GasGuard: An LLM-based Automated Gas Vulnerability Detection and Mitigation System

GasGuard builds upon the original GasGauge by introducing a LLM-powered system to patch gas-based vulnerabilities in Ethereum smart contracts automatically. While GasGauge identifies and reports risky loops, it leaves the mitigation to developers. GasGuard

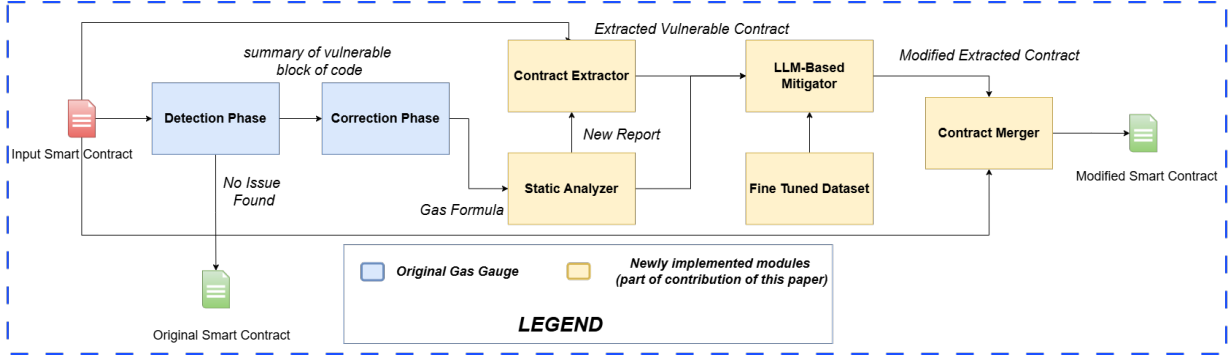


Figure 4.1: GasGuard architecture: from loop detection to automated gas-safe patching [2].

closes this gap by inserting [Gas-Aware](#) safeguards directly into vulnerable functions, leveraging prompt-engineered [LLMs](#) to suggest secure, context-sensitive fixes.

4.1.1 GasGuard Architecture and Dataset Overview

While GasGauge focuses on identifying risky loops and estimating safe execution thresholds, GasGuard uses this information to generate patched smart contracts automatically. The system is composed of several modular components, illustrated in Figure 4.1, each designed to tackle a specific phase of vulnerability remediation.

Static Analyzer: The first step in GasGuard builds on the analysis provided by GasGauge. This is the report generated during the correction phase of the original GasGauge tool. Instead of relying on raw textual output, the static analyzer translates the vulnerability report into a structured [JSON](#) format. This includes detailed metadata such as loop line numbers, loop types (e.g., fixed-bound, state-dependent), and a gas estimation formula. By doing so, it bridges the gap between detection and automated repair, enabling the [LLM](#) to focus solely on generating effective patches without interpreting unstructured data. It also detects and includes the start and end lines of each loop, which will then help the [LLM](#) navigate faster.

Contract Extractor: To minimize input token length and improve [LLM](#) performance, the contract extractor isolates only the relevant contract from a [Flattened Contract](#) Solidity file. Flattened files often include numerous contracts, libraries, and dependencies that are irrelevant to the mitigation task. This component utilizes the metadata provided by the static analyzer to parse and extract the vulnerable contract, ensuring syntax validity while excluding unrelated code. This preprocessing significantly improves both the accuracy and efficiency of the [LLM](#).

LLM-Based Mitigator: At the core of GasGuard lies the [LLM](#)-based mitigator. It takes the structured vulnerability data and the extracted contract as input and generates a repaired version of the code. The [LLM](#) is fine-tuned to insert [Gas-Aware](#) guards, typically in the form of [require\(\)](#) statements, at strategic points in the loop, enforcing safe execution

Dataset Name	Number of Contracts	Total Detected Loops	Case 1 Loops	Case 1.1 Loops	Case 2 Loops	Case 2.1 Loops	Case 3 Loops	Case 4 Loops
Training & Validation	246	500	392	25	9	12	35	27
Testing	230	393	315	26	4	8	27	13
Total	476	893	707	51	13	20	62	40

Table 4.1: Summary of Benchmark of the Fine-Tuned Dataset [2].

```

1 // Loop Case 1: simple loops
2 for (uint256 i = 0; i < lockBoxStructs.length; ++i) {...}
3 // Loop Case 1.1: simple loops with constant iterations
4 for (uint256 i = 0; i < 5; ++i) {...}
5 // Loop Case 2: nested loops
6 for (uint8 i = 0; i < arrNumbers.length - 1; i++) {
7     for (uint j = 0; j < arrNumbers.length - i - 1; j++) {...}}
8 // Loop Case 2.1: nested loops with constant iterations
9 for (uint8 i = 0; i < 4; i++) {
10     for (uint8 j = i+1; j < 5; j++) {...}}
11 // Loop Case 3: more complicated loops
12 while (true) {
13     if (poolUsers[_userID].a < _a) {return _userID;}
14     _userID++;}
15 // Loop Case 4: complex loops
16 for (n /= 2; n != 0; n /= 2) {
17     x = rmul(x, x);
18     if (n % 2 != 0) {z = rmul(z, x);}}

```

Figure 4.2: Examples of each loop category [2]

bounds. This process maintains the original logic of the contract while eliminating the risk of OOG failures. At the time of the tool’s design, GPT-3.5 was the most suitable fine-tunable model for Solidity tasks. After extensive experimentation with prompt formats, system messages, and few-shot examples, GasGuard achieved over 98% accuracy in generating correct repairs.

Contract Merger: Once a contract is repaired, the merger component reinserts the patched version into the original **Flattened Contract** file. This is especially critical when multiple vulnerable loops are present across contracts. The merger ensures that all fixes are accurately placed back in context, producing a fully patched, deployable version of the source code.

Benchmark Dataset: To evaluate and fine-tune GasGuard, we constructed a comprehensive benchmark based on 476 real-world Solidity contracts containing a total of 893 loops. Each loop was manually validated and categorized based on complexity, as summarized in Table 4.1. Categories ranged from simple fixed-bound loops to deeply nested and conditionally infinite ones, as illustrated in Figure 4.2. The dataset was split into training, validation, and testing subsets to support fine-tuning and performance evaluation. A manually repaired version of each contract was created to serve as ground truth for supervised learning. This dataset not only enabled GasGuard’s development but also served as a reusable benchmark for the broader smart contract research community.

GasGuard’s architecture, coupled with a carefully curated and labeled dataset, enables high-accuracy vulnerability mitigation at scale, bringing smart contract repair closer to full automation.

```

1 // Input vulnerable code section
2 while (true) {
3     if (poolUsers[_userID].a < _a) {return _userID;}
4     _userID++;}
5
6 // Output of GasGuard
7 require(poolUsers.length < (gasleft() - 6222) / (2000),
8     "Not enough gas for the loop iterations!");
9 while (true) {
10     if (poolUsers[_userID].a < _a) {return _userID;}
11     _userID++;}

```

Figure 4.3: Example of the input and output of GasGuard. GasGuard introduces a gas-aware runtime guard to prevent Out-of-Gas failures in an unbounded loop.

4.1.2 The Output of GasGuard

Figure 4.3 shows a concrete example of the type of repair that GasGuard performs on a vulnerable loop. In this case, the original `while` loop iterates over a dynamic list of users in the `poolUsers` array. The termination condition is based on a user-specific property, but there is no explicit upper bound on the number of iterations. As a result, the loop is susceptible to OOG failures when the `poolUsers` array grows too large. Manual identification and repair of such gas risks typically require a detailed inspection of both the loop body and the surrounding contract state.

GasGuard automatically detected this vulnerability and synthesized a runtime guard using Solidity’s `gasleft()` function. The inserted `require` statement introduces a gas-aware upper bound on the array length, calculated based on a conservative per-iteration gas estimate and a fixed buffer. This prevents execution of the loop in scenarios where insufficient gas would cause it to fail mid-execution. The entire repair was generated autonomously, without manual intervention, in under one minute. This example illustrates GasGuard’s ability to produce precise, contract-specific mitigations for subtle gas vulnerabilities, transforming a labor-intensive, error-prone task into a reliable, automated step in the secure development workflow.

4.2 Experimental Evaluation and Results of GasGuard

As mentioned before, a benchmark was carefully crafted to evaluate GasGuard. 500 loops were used for training and validation, while 393 loops were reserved for testing. The dataset was categorized based on loop complexity to maintain a balanced distribution across training and testing sets.

We employed a three-step verification process to ensure the correctness of the generated fixes. First, expert auditors manually created a reference set of correctly remediated contracts. Then, each LLM-generated fix was compared against the ground truth. If a match was found, the fix was marked as correct; otherwise, it was manually reviewed to determine whether they were valid alternative fixes or incorrect. The final validated contracts were executed against test cases to ensure correctness and adherence to expected Gas-Aware

optimization behavior. This multi-stage validation approach minimized false positives and ensured that GasGuard produced reliable and executable smart contract fixes.

We evaluated three methods for optimizing LLM performance: [Prompt Engineering](#), fine-tuning, and a combination of both. We systematically varied dataset size, hyperparameters, and prompt techniques to identify the most effective configuration. For [Prompt Engineering](#), we tested GPT-3.5-1106, GPT-4.0, and GPT-4o. Fine-tuning was performed exclusively on GPT-3.5-1106, as it was the latest fine-tunable model supporting Solidity during the experiments.

We experimented with multiple [Prompt Engineering](#) techniques. *System Messages* involves providing a detailed expectation explanation and specifying the appropriate scope of responses. *Few-shot Learning* utilizes a small set of training examples within prompts to provide context, aiding the model in generating more accurate responses. *Chain of Thought Prompting* encourages the model to detail its reasoning process step-by-step, promoting transparency and depth in responses. Lastly, *Providing Grounding Context* involves supplying relevant data the model can draw to enrich and substantiate its answers, ensuring responses are well-informed and relevant. These techniques were applied individually and in combination to assess their effectiveness in generating correct fixes.

For fine-tuning, we varied the dataset size, hyperparameters, and prompt techniques to optimize the model’s performance. We altered three hyperparameters. The *Number of Epochs*, which typically ranges from 1 to 5 with a default of 3, determines how many complete passes the model makes through the training dataset. The *Batch Size*, which can vary between 1 and 32 with a default of 1, dictates the number of examples processed before the model’s internal parameters are updated. Finally, the *Learning Rate Multiplier* adjusts the pretraining learning rate by 0.01 to 10, with 1 being the default. These settings are crucial for optimizing model performance and ensuring consistency in outcomes.

This setup allowed us to systematically evaluate the tool’s performance and ability to automate the repair of gas-based vulnerabilities in smart contracts.

4.2.1 Experiment 1: Prompt Engineering Techniques On GPTs

This experiment aims to optimize [Prompt Engineering](#) techniques for GPT-3.5, GPT-4.0, and GPT-4o models, focusing on smart contract repair. This involves testing three distinct approaches to assess and enhance the design’s feasibility, evaluating how different [Prompt Engineering](#) methods impact the accuracy of the models, and comparing the performance across the different GPT versions. This initial exploration laid the foundation for further refinements. The three [Prompt Engineering](#) methods are as follows:

1. **Technique 1: System Messages and Few-shot Learning:** The model was assigned a clearly defined assistant role with specific responsibilities. Six carefully selected examples representing various loop complexity levels were provided to establish context and guide the model in making appropriate contract modifications.

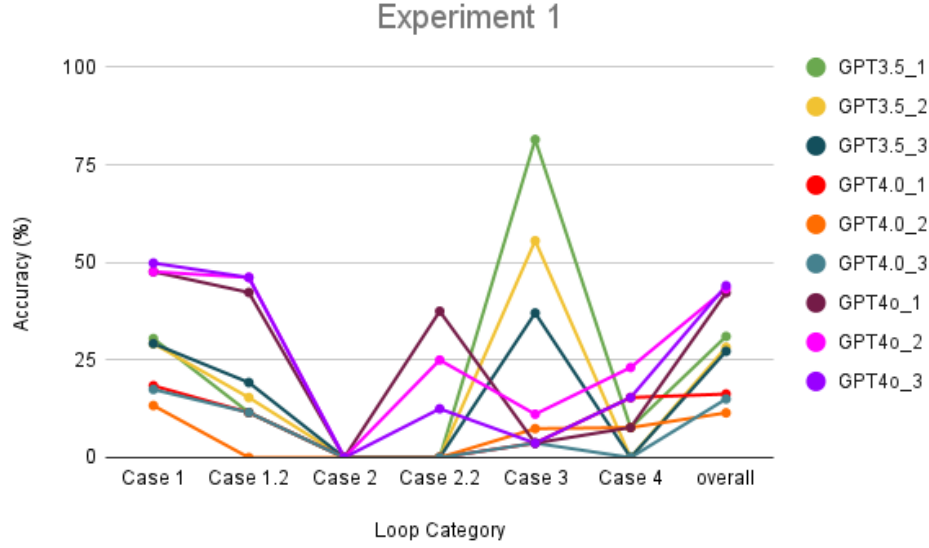


Figure 4.4: Summary of the results of Experiment 1. The format of each graph line is *LLM model_technique used*.^[2]

2. **Technique 2: System Messages, Few-shot Learning, and Chain of Thought Prompting:** In addition to Technique 1, we instructed the model to explain its reasoning step-by-step when modifying the contract. This approach aimed to ensure logical consistency, improve interpretability, and provide insights into the decision-making process.
3. **Technique 3: System Messages, Few-shot Learning, and Grounding Context:** In addition to Technique 1, we provided brief explanations of the changes made and why they were necessary as grounding context for each example. This grounding context was intended to help the model understand the rationale behind changes, potentially leading to improved accuracy.

As demonstrated in Figure 4.4, the results show that GPT4o outperforms the other models in all three techniques with the highest accuracy of 44.02% using the third technique. In contrast, GPT4.0 shows the worst performance with the lowest accuracy of 11.45% using the second technique. Although different techniques perform differently in each model, changing the technique does not significantly affect accuracy. This finding assured us that we could achieve better results by incorporating more examples and employing proper fine-tuning.

Failure Case Analysis: We analyzed cases where models generated incorrect fixes to understand the limitations of [Prompt Engineering](#) alone. Syntax-related issues were the most common cause of failure, especially in GPT-3.5 and GPT-4.0. Many incorrect fixes resulted from misplaced Solidity keywords, incorrect function definitions, or invalid **require** statements. Also, complex loops (e.g., binary search or state-dependent iteration counts) were more challenging for all models to correct, as they required deeper reasoning

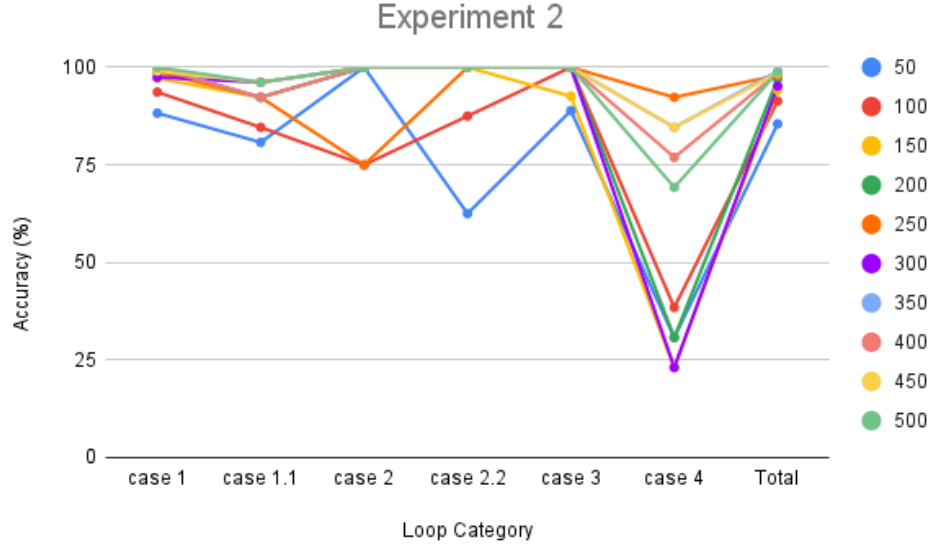


Figure 4.5: Summary of the results of Experiment 2. Each line of the graph shows the results for a fine-tuning dataset size.[2]

about the execution state. In most cases, Chain of Thought Prompting improved logical consistency in suggested fixes but did not significantly improve syntax correctness.

4.2.2 Experiment 2: Effect of Dataset Size on Fine-Tuning Accuracy

This experiment aims to evaluate the feasibility of fine-tuning and the impact of dataset size on model accuracy. The dataset size was systematically varied from 50 to 500 examples, increasing in increments of 50. Each dataset was split using an 80/20 ratio for training and validation. To ensure fairness, contracts were selected randomly while maintaining a sufficient number of loops for each complexity level. We used GPT 3.5-1106 with default parameters for consistency.

We first started with prompt testing on a small sample to identify the most effective structures, which are then applied throughout. In this experiment, the inputs consisted of a vulnerable smart contract loop and its summary, while the output was the corrected contract based on GasGauge’s findings. This approach assesses the tool’s ability to handle different dataset sizes.

Figure 4.5 shows the summary of the results. The results suggest that providing 350 examples yields the most accurate results, with an accuracy of 98.98%. Another interesting finding is that providing 250 examples has an overall accuracy of 97.96% but performs the best for complex loops (92.31% accuracy). However, compared to most other sets, it performs poorly for nested and simple loops. While this could be because of the random selection of the contracts in each dataset, it suggests that varying the number of examples

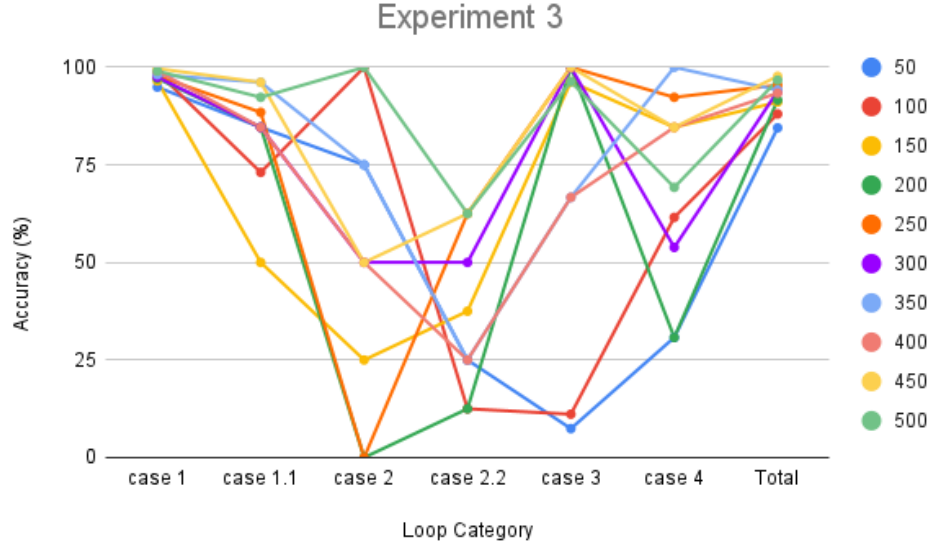


Figure 4.6: Summary of the results of Experiment 3. Each line of the graph shows the results for a fine-tuning dataset size.[2]

could also affect the accuracy for each level of loop complexity. Also, providing 50 examples results in the worst accuracy (85.5%). This suggests that even with a few examples, we may achieve an acceptable accuracy using proper techniques.

Impact of Dataset Size on Complexity Levels: The findings suggest that larger datasets improve overall accuracy, but dataset composition influences performance for specific loop complexities. For instance, complex loops require at least 250 examples for high accuracy, as they demand a deeper understanding of gas optimization patterns. Meanwhile, simple or nested loops performed best when the dataset contained a balanced mix of examples, which was more prevalent in larger datasets (350+ examples).

These results highlight that while 350 examples provide the highest accuracy overall, dataset diversity and structure are as important as size. Future work will explore optimal dataset compositions and whether data augmentation or synthetic examples can further improve generalization.

4.2.3 Experiment 3: Effect of Explanatory Context on Fine-tuning

This experiment investigates the impact of adding explanatory context to the training data during fine-tuning. While adding chain-of-thought prompting is generally not ideal for fine-tuning, we aimed to explore whether incorporating grounding context could improve performance. Specifically, we examine whether providing explanations for contract modifications helps the LLM understand the reasoning behind fixes and improves its accuracy. The setup mirrors that of Experiment 2. However, in this case, the model is expected



Figure 4.7: The charts illustrate the impact of different hyperparameters. Top: model fine-tuned with 350 instances. Center: 50 instances. Bottom: The best results are from the center chart. E: Number of Epochs, B: Batch Size, L: Learning Rate [2].

to provide both the corrected contract and reasoning for the changes.

As shown in Figure 4.6, the best result is 97.71% when we provide 450 examples. Similar to the previous experiment, the worst result is achieved when only 50 examples (84.48% accuracy) are provided. Nevertheless, the accuracy drops slightly at any point compared to the previous experiment.

Impact of Explanatory Context: Complex loops benefited from added explana-

tions, as the LLM better understood why specific gas optimizations were necessary. However, the LLM tended to overanalyze the changes for simple loops, leading to unnecessary modifications and reduced accuracy. Hence, the accuracy dropped slightly across all dataset sizes, suggesting that explanatory context can introduce unintended biases in the LLM’s repair strategy.

These findings suggest that feeding explanations during fine-tuning is only beneficial for complex loops, which are not as common. Given the added resource cost, it is generally best to avoid using explanations.

4.2.4 Experiment 4: Impact of Hyperparameter Optimization

This experiment investigates the impact of different hyperparameter tunings on the model’s performance. Based on the results of Experiment 2, we selected the best-performing point (350 examples) and the least effective one (50 examples) as the starting points. The experiment systematically adjusts the key hyperparameters. For the number of epochs, we tested 2, 4, and 5; for the batch size, we tested 10, 20, and 30; and for the learning rate, we tested 0.5, 5, and 10 to assess their impact on accuracy. The setup mirrors Experiment 2, with the goal of determining how varying these parameters affects model performance.

As shown in Figure 4.7, our findings show that for the set with 350 examples, the default parameters produce the highest accuracy. While altering the number of epochs and learning rate causes an insignificant drop in accuracy, changing the batch size has a higher impact. On the other hand, modifying the hyperparameters significantly affects the accuracy of the case with 50 examples. In this case, altering the batch size and decreasing the number of epochs and the learning rate drops the accuracy. However, increasing the number of epochs and the learning rate improves the accuracy. Additionally, our findings reveal that running the training for four epochs for the model trained on 50 examples yielded the most promising results with 88.04% accuracy.

To further refine our understanding, we conducted an additional test with 50 examples but increased the number of epochs to five. This allowed us to pinpoint how different hyperparameters affected the learning process, especially in scenarios with limited available data. The findings are similar to the previous ones. In the best case, where we set the number of epochs to 4, batch size to 1, and learning rate to 10, we can achieve an accuracy of 94.4%. This important finding reveals that high accuracy can be achieved using only 50 examples under suitable hyperparameters.

Key Findings: Our results confirm that proper hyperparameter tuning can significantly improve performance, especially when training with fewer examples. This demonstrates that high accuracy can be achieved with a small dataset when hyperparameters are carefully optimized. Meanwhile, 350 examples are inherently more stable, making hyperparameter adjustments less critical. These findings suggest that while dataset size plays a crucial role in accuracy, hyperparameter optimization can partially compensate for limited training data. This has important implications for resource-efficient fine-tuning when access to labeled data is constrained.

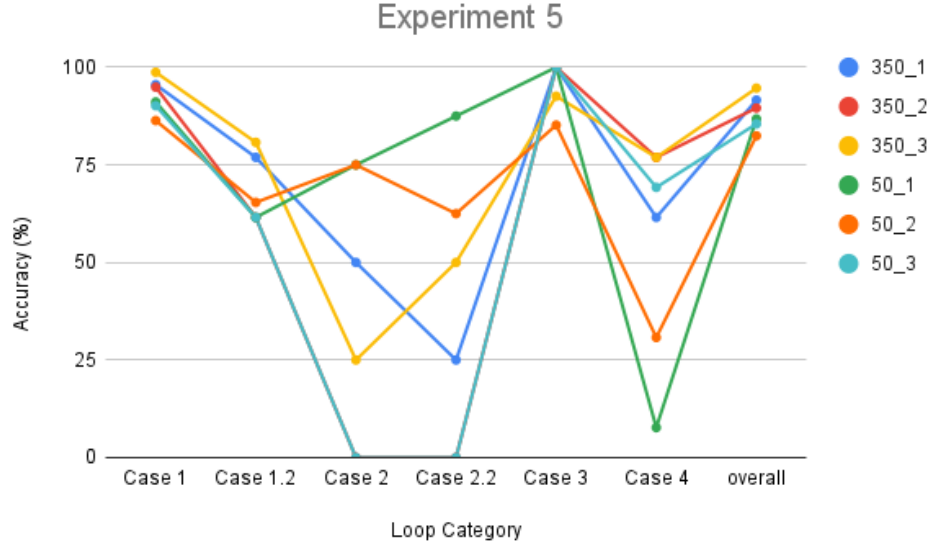


Figure 4.8: Summary of the results of Experiment 5. The format of each graph line is *Dataset size_technique used*.^[2]

4.2.5 Experiment 5: Combining Fine-tuning with Prompt Engineering

In this experiment, we aim to explore how different prompt engineering techniques affect the performance of fine-tuning models. To do so, we picked Experiment 2’s best-case scenario (350 examples) and worst-case scenario (50 examples) and applied the three prompt engineering strategies used in Experiment 1.

Figure 4.8 shows the summary of the results. Both cases show that the second prompt engineering technique performs poorly compared to the other techniques. Although prompt engineering techniques reduce the model’s accuracy with 350 examples, they slightly increase the model’s accuracy with 50 examples. In this case, the best result is achieved using the first prompt engineering technique, where the accuracy is 86.77%. This finding shows that we can improve the accuracy of a fine-tuning model with only 50 examples using proper adjustment.

Key Findings: fine-tuning models trained on larger datasets (350 examples) perform optimally without additional prompt engineering. This indicates that dataset diversity plays a more significant role than prompt refinements. However, for smaller datasets (50 examples), prompt engineering can compensate for the lack of training data. A structured approach with well-defined examples improves accuracy. These findings demonstrate that while prompt engineering alone is not a replacement for fine-tuning, it can be leveraged to improve accuracy when training data is limited. This insight is particularly valuable where access to large labeled datasets is infeasible.

Experiment	Best Case	Best Case Accuracy (%)	Worst Case	Worst Case Accuracy (%)
Prompt Engineering	GPT4o Technique 3	44.02	GPT4.0 Technique 2	11.45
Fine-tuning (Dataset Size)	350 examples	98.98	50 Examples	85.50
Experiment 2 + Explanation	450 examples	97.71	50 Examples	84.48
Fine-tuning (Hyper Parameters) - 50 examples	E4B1L10	94.4	E4B30L1	5.09
Fine-tuning (Hyper Parameters) - 350 examples	E3B1L1	98.98	E3B30L1	91.09
Experiment 1 + 2 - 50 Examples	Technique 1	86.77	Technique 2	82.44
Experiment 1 + 2 - 350 Examples	Technique 3	94.66	Technique 2	89.57

Table 4.2: Highlighted Results from the Experiments[2].

4.2.6 Summary of the Results

Table 4.2 provides a consolidated overview of the experimental outcomes across multiple dimensions, prompt engineering, dataset size, hyperparameter tuning, and hybrid configurations. Each row highlights the best and worst-case performance to capture the range of variation observed in GasGuard’s evaluation.

- **Prompt Engineering:** Among the tested methods, the third technique (system message + few-shot + grounding context) using GPT-4o yielded the highest accuracy at 44.02%. In contrast, GPT-4.0 paired with the second technique (system message + few-shot + chain-of-thought) produced the lowest result at 11.45%. These findings demonstrate that model selection and prompt strategy both play critical roles in shaping output quality.
- **Fine-tuning (Dataset Size):** The size of the fine-tuning dataset significantly influenced model performance. Using 350 training examples resulted in the highest observed accuracy of 98.98%, whereas reducing the dataset to only 50 examples caused performance to drop to 85.50%. This underscores the importance of ample and diverse training data for achieving robust accuracy.
- **Fine-tuning with Explanatory Context:** Adding grounding context during training led to mixed results. With 450 examples, the model achieved 97.71% accuracy, but with only 50 examples, accuracy fell to 84.48%. While explanations improved the handling of complex loops, they occasionally introduced ambiguity or redundancy, leading to marginally reduced overall performance.
- **Hyperparameter Optimization:** Fine-tuning hyperparameters had a significant impact, especially for smaller datasets. For 50 examples, accuracy reached as high as 94.38% with proper tuning, while the worst configuration dropped to just 5.09%. With 350 examples, even suboptimal hyperparameters maintained strong accuracy (91.09%), but optimal tuning preserved the peak of 98.98%. These results highlight that effective hyperparameter selection can partly compensate for limited data.
- **Combining fine-tuning with Prompt Engineering:** Integrating both methods produced noticeable improvements in low-data scenarios. For 50 examples, the best hybrid technique raised accuracy to 86.77%, a 1.3% increase over fine-tuning alone. For 350 examples, the gain was more modest, peaking at 94.66%. This confirms that prompt engineering can boost performance in constrained environments but offers diminishing returns when ample data is available.

In summary, these results collectively emphasize the trade-offs and synergies between prompt design, dataset size, and model tuning. While large, balanced datasets consistently deliver top-tier performance, thoughtful prompt engineering and hyperparameter tuning can meaningfully elevate accuracy in resource-constrained settings.

4.3 Optimization Theory for Model Performance of GasGuard

Achieving high performance in GasGuard requires a careful balance between model accuracy and the resources invested during training and deployment. These resources include both computational expenditures, such as GPU time and memory, and human labor for tasks like dataset curation, annotation, and prompt design. To capture this multidimensional investment, we introduce a cost-aware framework for optimizing model performance, with a focus on real-world feasibility and scalability.

We define a *Total Cost* function that encapsulates both computational and manual efforts:

$$\text{TC}(M) = \gamma \times \text{Computation}(M) + \delta \times \text{Labor}(M)$$

Here, M refers to a given model configuration, defined by the choice of hyperparameters, dataset size, fine-tuning strategies, and prompt engineering techniques. The coefficients γ and δ represent the relative weights of computational and labor costs, respectively, and can be tuned to reflect an organization’s available resources or strategic priorities.

Our overall optimization objective is to maximize the utility of the model by weighing accuracy against total cost. We express this formally as:

$$\text{Objective} = \alpha \times \text{Accuracy}(M) - \beta \times \text{TC}(M)$$

Where α and β determine the trade-off between achieving high accuracy and minimizing resource usage. This formulation provides a tunable knob for prioritizing performance or cost-efficiency depending on use case constraints.

To further formalize this goal, we cast the problem as a constrained optimization task:

$$\min_{\theta} \quad \text{TC}(M(\theta)) \quad \text{subject to} \quad \text{Accuracy}(M(\theta)) \geq \text{Threshold}$$

In this formulation, θ includes all tunable parameters and configurations, ranging from learning rate, batch size, and number of epochs to dataset curation and annotation time. The constraint enforces a minimum required accuracy for the model to be considered usable in a given application domain.

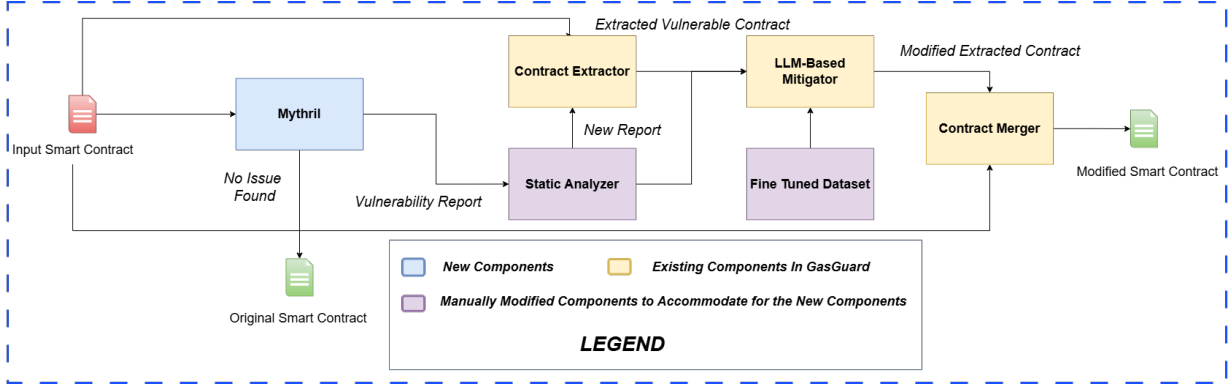


Figure 4.9: An Overview of the Extensibility of GasGuard [2]

4.3.1 Managing Trade-offs with Adaptive Strategies

This framework enables strategic decision-making tailored to different environments. For instance, in low-resource settings, one might assign a higher penalty to labor cost (δ) and prioritize prompt engineering or lightweight tuning over dataset expansion. Conversely, when the target application demands high reliability, such as in financial or legal smart contracts, accuracy can be weighted more heavily ($\alpha \gg \beta$) to justify additional effort.

As demonstrated in our experiments, increasing the dataset size from 50 to 350 examples boosts accuracy by over 13%, but also incurs substantial manual overhead and longer fine-tuning time. On the other hand, prompt engineering techniques, though yielding smaller improvements, require significantly fewer resources and offer a more cost-effective route when resource constraints are tight.

By explicitly modeling both human and computational effort, this optimization framework supports informed trade-offs and encourages efficient resource use without compromising reliability. Ultimately, this approach ensures that GasGuard remains scalable, practical, and deployable in a wide range of operational contexts.

4.4 Initial Validation and Extensibility of GasGuard

This work represents the foundation for advancing automated techniques to detect and mitigate gas-related vulnerabilities in Ethereum smart contracts. In designing GasGuard, we aimed not only to address a niche yet impactful vulnerability class but also to demonstrate that integrating LLM with guided repair pipelines can result in scalable and precise smart contract security solutions. The successful results obtained in our experiments encourage broader application of this methodology to other classes of vulnerabilities and mark a promising step toward fully automated vulnerability detection and remediation.

To further evaluate the generalizability of our pipeline, we extended the framework to a new vulnerability type, **integer overflow/underflow**. This class of arithmetic bugs can be exploited to manipulate balances or bypass critical checks, potentially resulting in

severe financial harm. While Solidity v0.8.0 and above includes built-in protections against such issues, many legacy contracts remain vulnerable. We selected this vulnerability to evaluate extensibility due to its precise semantics, existing detection tools, and relatively straightforward manual repair.

Here, instead of relying on GasGauge for vulnerability detection, we used Mythril [34], a well-known auditing tool that effectively identifies overflow and underflow issues. Since Mythril’s output format differs from GasGauge’s, we slightly modified the static analyzer to parse Mythril’s reports. However, we preserved the final structured JSON format, ensuring compatibility with downstream components like the contract extractor.

The new pipeline is shown in Figure 4.9. The most significant change was constructing a new dataset tailored to overflow/underflow vulnerabilities. Inspired by our previous experiments, which showed strong performance even with a small dataset, we manually built a set of 50 diverse vulnerable contracts. These examples spanned multiple arithmetic operations, data types, and contract structures to ensure sufficient generalization.

Using this new dataset, we fine-tuned a new LLM specifically for integer overflow/underflow remediation. The static analyzer’s report and extracted contracts were used as input, and the model was trained to generate fixed versions of the vulnerable code. Despite requiring a different LLM than GasGuard’s gas-based model, the retraining process was automated and low-effort. The final mitigation results were merged into the original contract using the same contract merger module.

Our evaluation showed that even with just 50 fine-tuning examples and default hyperparameters, the overflow/underflow model achieved over 95% accuracy. This strongly supports the hypothesis that GasGuard’s architecture is adaptable. While a reliable dataset remains the main manual bottleneck, extending the framework to new vulnerabilities requires minimal reengineering. These findings position GasGuard as a modular, scalable solution capable of evolving alongside the security needs of the Ethereum ecosystem.

4.5 Conclusion, Limitations and Next Steps

This chapter introduced GasGuard, an automated repair system for gas-based DoS vulnerabilities in Ethereum smart contracts. Built on the GasGauge toolchain, GasGuard uses a fine-tuned Large Language Model to insert runtime safeguards around vulnerable loops, significantly reducing manual triage and remediation time. Its accuracy and speed make it practical for integration into development pipelines and audit workflows.

While GasGuard represents a significant step toward automated repair of gas-based vulnerabilities, it is not without limitations. First and foremost, the tool focuses exclusively on mitigating OOG issues and does not address broader classes of smart contract vulnerabilities, such as reentrancy, access control misconfigurations, or arithmetic overflows. For comprehensive security coverage, GasGuard should be integrated into a broader suite of auditing tools.

Second, GasGuard is tightly coupled with GasGauge, meaning it inherits any limitations from the detection phase. If GasGauge fails to identify a vulnerable loop, due to

obfuscation, atypical loop structures, or static analysis limitations, then GasGuard cannot apply a repair. As such, its effectiveness is bounded by the precision and recall of the underlying detection engine.

Lastly, while GasGuard effectively patches loops governed by constants or function inputs by inserting gas-aware guards, it is less effective for loops driven by dynamic state variables. In such cases, mitigation may require a deeper semantic understanding of contract logic and the identification of auxiliary functions that reduce the loop-bound variable.

These cases are further addressed by the next tool in our pipeline, GasGaugeAI, introduced in the next chapter. GasGaugeAI extends this line of work by leveraging a multi-agent [LLM](#) collaboration and symbolic testing to fully close the repair loop and support verifiable, state-aware mitigation of gas vulnerabilities.

Chapter 5

GasGaugeAI: A Multi-LLM APR Framework for State-Aware Gas Vulnerability Mitigation

Denial of Service (DoS) vulnerabilities, particularly those caused by unbounded loops in Ethereum smart contracts, continue to pose significant challenges for secure and scalable decentralized applications. To mitigate this issue, one needs to ensure safeguards are in place to prevent gas wastage. That is when the transaction initiator pays for gas fees, but the transaction runs out, causing the transaction to revert without any state changes. A common safeguard is adding a required statement before entering the loop to prevent execution of the loop if there is not enough `gasleft()`. This helps the caller to save some gas and get an idea of how to structure the next call.

While GasGuard has made strides in detecting and mitigating gas wastage vulnerabilities, the mitigation is sufficient where constant or input-based parameters govern loops iterations. The reason is that constant variables that cause OOG can be captured in testing, and the caller can control the input variables if OOG occurs. However, many real-world contracts contain loops whose bounds depend on evolving state variables, values that change with contract execution and user interaction. Although gas wastage safeguards are still helpful, there are other steps needed to ensure there is a way to prevent a loop from getting stuck in an OOG state. These loops are generally complicated to analyze and repair because their behavior is not easily predictable at compile time.

GasGaugeAI addresses this gap by introducing a state-aware, fully automated repair framework using a novel multi-LLM architecture. Its core motivation stems from the observation that many loops in deployed contracts are only safe under some conditions and unsafe under others. Therefore, a robust mitigation system must go beyond static detection and integrate runtime testing, symbolic reasoning, and intelligent synthesis of new functionality.

GasGaugeAI was developed with five primary goals in mind. First, **comprehensive coverage** to extend detection to state-dependent loops and provide context-sensitive analysis that adapts to contract behavior. The next goal was **automation and scalability**

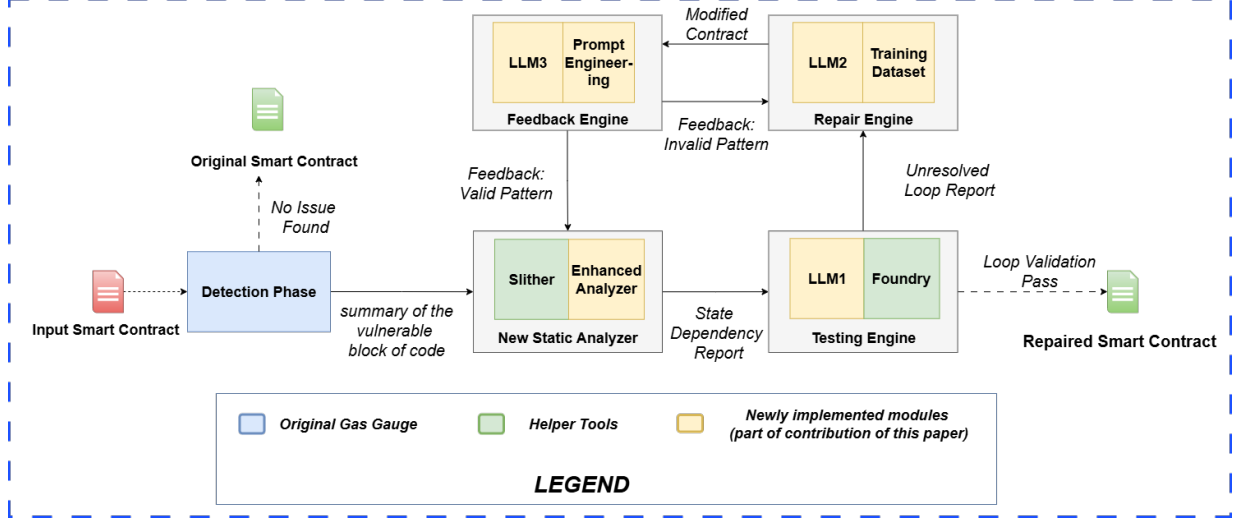


Figure 5.1: GasGuard architecture: from loop detection to automated repair for state-dependent loops [3].

to eliminate the need for manual intervention by employing three specialized LLMs that detect, validate, and fix gas vulnerabilities. Also, **modular extensibility** is another important goal to enable future adaptation to additional vulnerabilities through reusable components and a well-defined data pipeline. The fourth goal was **grounded fix generation** to combine symbolic testing and AI-generated logic to suggest and validate repairs aligned with existing contract semantics. Finally, **an auditor-friendly output** to produce clear diagnostics and fix suggestions compatible with industry review practices.

5.1 GasGaugeAI Architecture

GasGaugeAI is an advanced automated repair framework designed to address OOG vulnerabilities in Ethereum smart contracts, especially those involving loops driven by dynamic state variables. Extending the foundational capabilities of *GasGauge*, this system advances vulnerability repair by integrating static analysis, dynamic testing, and large language models (LLMs) into a unified pipeline. The result is a robust, scalable approach capable of not only identifying and bounding risky loops but also reasoning about contract state dependencies and synthesizing missing logic when necessary. The structure of GasGaugeAI is shown in Figure 5.1, and the summary of each component is provided below.

5.1.1 Phase I: Loop Detection and Risk Analysis via GasGauge

The initial phase of GasGaugeAI builds upon the *GasGauge* tool, which analyzes Solidity contracts to detect gas-intensive loops. Each loop is assessed for risk and categorized as one of the following:

- **Constant-controlled:** loop bounds are fixed and inherently safe.
- **Input-controlled:** Risky when user inputs can create unbounded iterations.
- **State-controlled:** Most hazardous, as loop bounds evolve with persistent state changes over time.

It then derives an upper bound using `gasleft()` in combination with the loop’s computational complexity, yielding a gas usage formula that reflects real-time execution conditions. However, while effective in detection, GasGauge does not offer repair capabilities for state-controlled loops, a limitation GasGaugeAI is built to address.

5.1.2 Phase II: Automated Repair for State-Controlled Loops

State-controlled loops present a unique challenge because their iteration limits are influenced by evolving contract states, making them resistant to static bounding. To address this, GasGaugeAI introduces a four-stage repair process combining static reasoning, dynamic validation, automatic code generation, and a feedback-driven, multi-LLM architecture for code synthesis and verification.

1. Static Variable Analysis and Categorization. GasGaugeAI begins by analyzing each state-controlled loop identified in Phase I. While we use Slither [1] for its fast AST traversal and dependency resolution, we found it insufficient for our needs, particularly in capturing full loop context, handling writer analysis beyond public functions, and isolating minimal logic required for repair. Therefore, in addition to Slither, we use a custom parser to extract contract boundaries, loop metadata, and state-variable dependencies.

Hence, the developed custom static analyzer augments Slither by performing:

- **Contract and function boundary isolation:** Extracts flattened contracts into function-level blocks for modular repair.
- **Loop localization:** Identifies the exact start and end lines of each loop, even in nested or multiline structures.
- **State variable dependency mapping:** Tracks which state variables influence loop bounds and classifies them as constant, input-based, or mutable.
- **Writer function resolution:** Constructs a refined call graph to identify all internal and external functions, including private, internal, or modifier-based logic, that can modify the loop-affecting variables.

This combination allows us to build a fine-grained control-flow map and dependency tree for each loop. Unlike traditional analyzers, our custom system produces clean, context-limited inputs for downstream LLMs, improving repair precision and reducing prompt noise. These outputs are also used to construct targeted Foundry test cases for runtime verification.

2. Functionality Testing via Foundry Testing (LLM-1). In Step 2, GasGaugeAI validates whether any existing writer functions can safely reduce the loop-bound state variable. This is performed empirically using the Foundry framework and automated test generation.

Given the loop context, state variable, and candidate writer functions, GasGaugeAI invokes LLM-1 to synthesize Solidity-based Foundry test cases. These tests simulate realistic call sequences that attempt to reduce the loop-bound variable, thereby preventing OOG conditions. Tests are compiled and executed to verify whether loop iterations become safely bounded under any execution path.

If Foundry fails to compile the generated test case or encounters a runtime exception, the test is considered invalid. In such cases, LLM-1 receives structured feedback and is prompted to regenerate the test case using an updated few-shot prompt and trace context. This process repeats until a valid test is produced or a configurable retry limit is reached. In our implementation, we allow up to 5 attempts per loop instance, which balances robustness with computational cost.

We evaluated multiple LLMs, including variants of GPT, and found that GPT-4.0 yields the most accurate and robust test generation. Using a few-shot prompting strategy with six representative examples, LLM-1 achieved a 93% test generation accuracy, defined as the percentage of test cases that successfully tested the existence of a function to reduce loop iterations.

If any test case confirms that an existing function safely bounds the loop, GasGaugeAI halts repair for that loop to preserve the original semantics. Otherwise, the pipeline proceeds to synthesis.

3. Secure Repair via LLM-Driven Synthesis (LLM-2). If no existing function safely reduces loop iterations, GasGaugeAI transitions to repair synthesis using LLM-2. The model generates Solidity logic that ensures bounded loop execution by modifying state variables or constraining their growth.

LLM-2 is based on GPT-4.0 (32k tokens), leveraging its extended context window to reason over large contract structures. We employ few-shot prompting with six representative repair examples and chain-of-thought (CoT) prompting to improve reasoning accuracy and consistency. This setup enables LLM-2 to generate precise, context-aware Solidity patches without exceeding token limits.

While GPT-4.1 supports fine-tuning, its 16k token limit proved restrictive for complex contracts in our pipeline. Therefore, we use GPT-4.0 (32k) with advanced prompting as our default model for repair synthesis.

Repairs typically involve adding decrement operations, capping array lengths, or introducing modifiers to bound variable growth. If no suitable writer function exists, GasGaugeAI generates a new internal utility function invoked along safe execution paths. All patches are applied additively to preserve business logic and semantic integrity. The modified contract is then prepared for validation.

4. Pattern Classification and Feedback Loop (LLM-3 and LLM-1). Once LLM-2 generates a repair, it is passed to LLM-3 for semantic validation. LLM-3 is based

Table 5.1: Summary of Common Repair Patterns Used in Fine-Tuning LLM-2

Pattern Name	Context / Vulnerability	Repair Strategy (Patch)
<i>Mapping Refactor</i>	Loop over a dynamic array stored in the state	Replace loop logic with a mapping lookup to achieve constant-time access
<i>Gas-Aware Batching</i>	Loop controlled by a growing state array	Introduce batching logic using <code>gasleft()</code> and a per-call index counter
<i>Decrement Guard</i>	Loop count controlled by an increment-only variable	Insert decrement logic in a writer function to reduce loop-bound variable
<i>Array Length Cap</i>	Loop over a modifiable list that can grow indefinitely	Add a cap on the array length using a conditional push or bounded array size
<i>Conditional Guard</i>	Loop used for reward distribution or transfer across unbounded user list	Add a <code>require</code> or conditional statement to skip or prevent execution if size exceeds a safe threshold

on GPT-4.1, trained with few-shot prompting (six examples) and chain-of-thought (CoT) reasoning on a curated taxonomy of known safe repair patterns. LLM-3 verifies whether the patch conforms to one of these patterns or represents a novel, yet valid, variant. If the patch is correct and matches an existing pattern, it proceeds to validation. If correct but novel, LLM-3 adds it to the evolving pattern set. If the patch is incorrect or unsafe, LLM-3 provides structured feedback to LLM-2, triggering a refinement cycle.

Validated patches are then passed to LLM-1 and re-engaged in validator mode with a modified prompt. In this phase, LLM-1 generates targeted Foundry test cases that assess whether the LLM-2 patch effectively reduces the loop-bound state variable and mitigates OOG risk under realistic execution paths. Unlike initial fuzzing, which explores general state space, this phase focuses on testing specific functions and call paths identified by LLM-2 and LLM-3 as critical to validate the repair. The summary of the common repair patterns is shown in Table 5.1

If Foundry validation succeeds, the repair is finalized. If validation fails due to unbounded loops, ineffective variable control, or test case failure, the pipeline enters a feedback loop. Execution traces and validation results are sent back to LLM-2 via LLM-3, which refines the patch. The cycle continues until the patch passes both LLM-3 classification and LLM-1 validation or a retry limit is reached, after which the loop is flagged for manual review.

By orchestrating this closed-loop interaction across three cooperating LLMs, GasGaugeAI delivers semantically sound and empirically validated repairs for gas-sensitive state-controlled loops while minimizing the need for human intervention.

5.1.3 Modular Design and Synergy of AI with Program Analysis

A core design goal of GasGaugeAI is modularity. Thus, each component in the pipeline operates independently, with clearly defined inputs and outputs. This modularity enables

Table 5.2: Summary of GasGaugeAI’s Design and Workflow.

Phase	Objective	Techniques / Tools	Output
Loop Detection (GasGauge)	Identify potentially unbounded loops	Static / Dynamic analysis	Loop locations, types, and gas-usage formulas
Variable Classification	Determine if the loop is controlled by constant, input, or state variables	Slither printers + custom static parser	Loop type and affecting variables
Function Extraction	Identify functions that modify loop-bound state variables	Static call graph + write analysis	Slither (Writer function list per variable)
Testing (LLM-1)	Check if state variables can be reduced via existing logic	GPT-4.0 few-shot prompting + Foundry	safe/unsafe based on runtime behavior
Patch Synthesis (LLM-2)	Synthesize missing logic for bounded execution	GPT-4.0 (32k), few-shot + CoT prompting	Secure Solidity patch to modify state variable
Feedback Loop & Pattern Classification (LLM-3)	Classify patch against known safe patterns; provide feedback if needed	GPT-4.1, few-shot + CoT prompting	Verdict: valid/invalid + feedback to LLM-2

flexible experimentation, allows components to be improved or swapped in isolation, and supports a future extension to broader vulnerability classes.

In particular, we deliberately decouple validation (LLM-1) from feedback and pattern classification (LLM-3). While LLM-1 is well-suited for empirical validation, reusing it for feedback would mix verification with semantic reasoning, two distinct tasks best handled separately. By introducing a dedicated LLM-3, trained on a curated taxonomy of known repair patterns, we ensure that pattern classification and feedback remain consistent, interpretable, and robust across multiple repair attempts. This separation also allows LLM-1 to be tuned purely for test generation and LLM-3 to evolve independently as new repair patterns are discovered.

Crucially, GasGaugeAI does not rely solely on either program analysis or [AI](#) models, but instead combines both to leverage their respective strengths. Traditional program analysis methods, such as static analysis and fuzzing, provide soundness guarantees, precise symbolic reasoning, and full coverage of potential execution paths. However, they often struggle with scalability, are brittle in the face of nontrivial code structures, and cannot synthesize missing logic when no fix is available. Also, upgrading the design requires more effort. Moreover, purely formal methods are difficult to extend to real-world contract patterns without deep domain-specific encodings.

On the other hand, LLMs offer flexibility, generalization, and the ability to synthesize human-like logic even in under-specified scenarios. But LLMs lack formal guarantees and can hallucinate unsafe or irrelevant code, especially in the absence of context constraints.

This layered approach also provides robustness. If [LLM](#) synthesis fails, the static analyzer combined with fuzzing still provides interpretable artifacts. Conversely, if the [AI](#) suggests a patch that passes validation but diverges semantically, LLM-3 and test oracles detect the deviation.

In summary, GasGaugeAI’s hybrid design reflects a philosophy of layered verifiability, where no single tool is trusted in isolation, but the system as a whole becomes more accurate, interpretable, and resilient by aligning AI-driven synthesis with principled program analysis. The summary of the design of the workflow of GasGaugeAI is provided in [Table 5.2](#).

5.1.4 Dataset Overview

To support this tool, a dataset of 800 state-variable-dependent loops was curated from real-world smart contracts. From this dataset, six were picked as few-shot examples. From the remaining 794 loops, 740 loops were successfully processed by the first [LLM](#) component (LLM-1), which generated property-based test cases using the Foundry testing framework [\[39\]](#). These test cases were designed to determine whether any existing function in the contract reduced the loop-bound state variable under realistic execution conditions. For the remaining 54 functions, LLM-1 was unable to generate appropriate or compilable tests due to factors such as complex dependencies, ambiguous logic, or insufficient context. These cases were flagged as still vulnerable to state-dependent gas issues.

After executing the LLM-1-generated tests, the framework categorized each contract according to whether a reducing function for the loop-bound variable was found. For cases where such functionality was detected, the repair was considered complete. When no such function was found, the contract was passed to the second [LLM](#) component (LLM-2), which synthesized new reduction logic. Each synthesized patch was then classified by the third [LLM](#) component (LLM-3), trained on a taxonomy of known safe repair patterns. Patches classified as correct proceeded to validation, while incorrect patches triggered a refinement loop with structured feedback from LLM-3 to LLM-2.

Validated patches were passed back to LLM-1 for targeted re-testing with Foundry to ensure that the repair safely reduced loop iterations under realistic execution paths. If validation failed, the repair loop continued until the patch was successfully classified and validated, or the retry limit was reached, in which case the loop was flagged for manual review.

This multi-stage process, test-guided detection, [LLM](#)-based synthesis, pattern classification, and iterative validation, enabled GasGaugeAI to provide precise, minimally invasive repairs tailored to each loop’s structure and state dependencies. The dataset not only facilitated a robust evaluation of the multi-[LLM](#) architecture but also served as a valuable benchmark for future research on AI-assisted repair of complex, state-driven vulnerabilities in Ethereum smart contracts.

5.1.5 Limitations of GasGaugeAI

While GasGaugeAI represents a significant advancement in repairing gas-related vulnerabilities, several limitations persist. First, if the initial loop detection or variable classification fails, subsequent LLM reasoning may be misdirected or incomplete. Second, LLM-2’s performance may vary when synthesizing novel logic, especially in highly customized or obfuscated contract architectures. Finally, the current implementation only supports gas-based DoS vulnerability, limiting the applications of the tool.

Despite these constraints, GasGaugeAI lays the foundation for a new generation of smart contract repair tools, ones that are autonomous, intelligent, and capable of addressing dynamic, state-dependent vulnerabilities with minimal developer intervention.

5.2 Experimental Evaluation of GasGaugeAI

In this section, we assess the capabilities of **GasGaugeAI**, our multi-agent LLM-driven framework designed for detecting and repairing gas-related vulnerabilities in Ethereum smart contracts. The evaluation is structured around five core research questions:

- **RQ1:** How do various prompt formats and engineering techniques influence LLM-1’s ability to generate test cases for functions that affect loop behavior?
- **RQ2:** To what extent does prompt engineering impact the quality of repairs produced by LLM-2?
- **RQ3:** What is the effect of combining multiple prompt strategies on LLM-2’s overall repair accuracy?
- **RQ4:** How effectively does LLM-2 generalize to real-world loop vulnerabilities after fine-tuning?
- **RQ5:** Can iterative feedback through LLM-3 significantly enhance the success of repair attempts?

5.2.1 Preliminary and Experimental Setup

To evaluate GasGaugeAI, we assembled a benchmark of 1,000 Solidity smart contracts, each containing at least one loop previously flagged by the GasGauge static analyzer. From this dataset, we extracted 800 loops across 315 contracts, where state variables govern the loop behavior. These became the foundation for our experimental evaluation. All contracts were sourced from real-world public repositories, flattened for uniformity, and normalized to ensure compatibility with the Foundry testing framework.

All experiments were conducted on a 16-core Intel Xeon machine with 64 GB of RAM. We used Foundry CLI (v0.2.x) and Slither (v0.9.0) for static analysis and testing. **LLM-1**

was powered by GPT-4.0 (32k context window) accessed via the OpenAI API. **LLM-2** utilized both GPT-4.0 and a fine-tuned GPT-4.1 model, trained on a curated dataset of 10 to 50 annotated real-world examples. **LLM-3** was implemented using GPT-4.1. To ensure reproducibility, all models were evaluated under consistent temperature, sampling, and token settings.

We report results across several key metrics: Test generation accuracy, patch success rate, feedback loop convergence, and breakdowns of performance across different prompt strategies and loop types.

5.2.2 Experiment 1: Effectiveness of LLM-1 for Test Generation

We begin our evaluation with **LLM-1**, the component responsible for generating Foundry test cases to assess whether a contract includes logic that can reduce the number of loop iterations, particularly when those loops are governed by state variables. To guide the model, we provided six representative few-shot examples that span a diverse range of loop structures and contract designs. The remaining **794 loops** formed our primary evaluation dataset.

For each loop, LLM-1 generated a targeted test case, which was then compiled and executed using Foundry. If the test failed, due to compilation errors or incorrect behavior, the resulting feedback was returned to the model, allowing it to revise and regenerate its output. Each loop was given up to three attempts to yield a valid, passing test.

We conducted this evaluation using multiple **LLM** variants and prompting strategies, as outlined below:

- **LLMs:**
 - **GPT-4:** OpenAI’s March 2023 flagship model, known for strong reasoning, coding, and general-purpose capabilities.
 - **GPT-4o:** A more recent (May 2024) version offering faster inference, improved multi-modal reasoning, and lower latency.
 - **GPT-4o-mini (GPT-4oM):** A lightweight variant of GPT-4o, optimized for cost and speed, though with reduced depth in complex reasoning.
 - **GPT-3.5 Turbo (o1):** An earlier generation model that serves as a baseline, offering fast response times but limited reasoning capacity.
- **Prompting Techniques:**
 - **Chain-of-Thought (CoT):** Prompts encourage step-by-step reasoning, helping the model structure logical sequences during test generation.
 - **Grounding Context (grounding):** Additional information about the vulnerability, contract structure, and loop semantics is included to improve model understanding.

Table 5.3: RQ1: LLM-1 Test Generation Results [3]

Model_Prompt	Successful	Unsuccessful	Success_Rate (%)
GPT-4 COT	552	242	69.5
GPT-4 Grounding	417	377	52.2
GPT-4 Shots	740	54	93.2
GPT-4o COT	340	454	42.9
GPT-4o Grounding	60	734	7.6
GPT-4o Shots	346	448	43.6
GPT-4oM COT	145	649	18.3
GPT-4oM Grounding	30	765	3.8
GPT-4oM Shots	171	623	21.5
o1 COT	156	638	19.6
o1 Grounding	110	684	13.9
o1 Shots	250	544	31.5

- **Few-shot Examples (shots)**: The prompt includes six hand-picked examples of well-formed test cases across diverse scenarios to guide the model by analogy.

As shown in Table 5.3, we observed clear performance differences across both model types and prompting strategies. Among all approaches, few-shot prompting consistently yielded the best results, with **GPT-4 using few-shot examples achieving a 93.2% success rate**. Chain-of-Thought prompting offered moderate improvements, while Grounding alone showed limited effectiveness. In terms of model performance, larger models outperformed their lighter counterparts, with GPT-4 leading across all configurations. Based on these findings, we adopted **GPT-4 with few-shot prompting** as the default setup for LLM-1 in the final GasGaugeAI pipeline.

Interestingly, during this experiment, we found that only **one out of 800 evaluated loops** already included a mechanism that safely bounded its iteration count. We excluded this loop from further evaluation. This finding reinforces the critical need for GasGaugeAI. Such cases of pre-existing safety measures are exceedingly rare in practice, underscoring the importance of automated detection and repair.

While further gains in accuracy could likely be achieved through more advanced prompt engineering or fine-tuning, we chose to stop here. Achieving over **93% accuracy** using GPT-4 with few-shot prompting was sufficient for our pipeline’s needs. Any residual issues in test case generation are handled by the downstream components, LLM-2 and LLM-3, which perform repair synthesis and iterative verification.

5.2.3 Experiment 2: Impact of Prompt Engineering on LLM-2 Repair

In this experiment, we explored how different prompt engineering strategies affect the repair quality produced by **LLM-2**. From the original set of 800 vulnerable loops, one was eliminated by LLM-1, and we reserved 50 examples for fine-tuning and validation in later experiments. The remaining **749 loops** formed the evaluation set for this and all subsequent analyses.

Table 5.4: RQ 2: LLM-2 Prompt Engineering Results[3]

Prompt	CR	NRCL	IL	NOG	CRate (%)
Role	396	161	168	24	52.9
Grounding	418	163	154	14	55.8
Format	390	158	174	27	52.1
Instruction	476	138	127	8	63.6
Shots	449	93	193	14	59.9
COT	456	151	130	12	60.9

Given that GPT-4.0 (32k context window) was the only model available to us with extended token support, we selected it as the base LLM for this evaluation. We tested six distinct prompting strategies, each designed to influence the model’s repair behavior in a different way:

- **Role-Based Prompting (Role)**: Introduces the model as a dedicated repair agent through an explicit system role definition.
- **Grounding Context (Grounding)**: Provides detailed background on the vulnerability, contract behavior, and loop semantics to anchor the model’s understanding.
- **Format-Constrained Prompting (Format)**: Specifies a strict output structure to improve consistency and post-processing.
- **Instruction-Tuned Prompting (Instruction)**: Supplies comprehensive, step-by-step repair guidelines to directly steer the model’s actions.
- **Few-Shot Learning (Shots)**: Includes six example repairs to help the model generalize patterns and adapt to different loop contexts.
- **Chain-of-Thought Prompting (COT)**: Encourages the model to reason step-by-step through the repair process to improve logical clarity and correctness.

Correctness was evaluated through a structured multi-phase validation process. First, each generated repair was manually reviewed by the authors to assess both logical soundness and adherence to defined repair principles. In situations where manual inspection alone was inconclusive, such as complex refactorings or edge-case logic, the repair candidate was passed through the **LLM-3** verifier. If further confirmation was needed, we also used **LLM-1**-generated test cases to validate that the repaired contract was executed safely under representative conditions. This layered approach ensured high confidence in the quality and correctness of each patch.

The results of this experiment are summarized in Table 5.4, which compares how different prompt engineering strategies influenced LLM-2’s ability to repair gas-vulnerable loops. Each repair outcome was categorized as follows:

- **Correct Repair (CR)**: A successful fix that removed the vulnerability while maintaining the intended contract behavior.

Table 5.5: Experiment 3: Prompt Combination Strategies for LLM-2 Repair (15 Combinations)[3]

Name	First Approach	Second Approach	Makes Sense??	Why?
Zero-Shot Learning (Zero-Shot)	Grounding	COT	Yes	Establishes context then guides reasoning.
Immersive Grounding (Immersive)	Grounding	Role	Yes	Provides context and positions the model as an expert.
Grounding + Instruction	Grounding	Instruction	Maybe	Instruction tuning may override or ignore contextual grounding cues.
Grounding + Format	Grounding	Format	No	Context useful, but lacks reasoning or process guidance.
Grounding + Few-shot	Grounding	Few-shot	Maybe	Context helps, but static examples may not generalize well.
Expert Reasoning (ExpertR)	COT	Role	Yes	Encourages step-by-step reasoning with expert framing.
COT + Instruction	COT	Instruction	Maybe	Instruction tuning may dominate and suppress effective step-by-step reasoning.
COT + Format	COT	Format	No	Lacks concrete examples or process clarity; too rigid for nuanced repairs.
Reasoned Few-Shot (Reasoned)	COT	Few-shot	Yes	Combines reasoning and example-driven learning.
Compact Expert (Compact)	Role	Instruction	Yes	Expert tone + explicit task guidance.
Role + Format	Role	Format	No	Formatting alone does not leverage role framing; lacks reasoning.
Guided Few-Shot (Guided)	Role	Few-shot	Yes	Expert framing enhances learning from examples.
Focused Instruction (Focused)	Instruction	Format	Yes	Precise control over both repair and output.
Few-Shot Snap (Snap)	Instruction	Few-shot	Yes	Teaches model to apply guided repairs.
Structured Few-Shot (Structured)	Few-shot	Format	Yes	Examples plus structured output.

- **Not Repaired But Correct Logic (NRCL)**: The output followed good development practices and introduced no functional errors but did not directly mitigate the gas vulnerability.
- **Incorrect Logic (IL)**: The patch introduced logical errors or deviated from expected semantics, either breaking the contract or failing to apply the repair correctly.
- **No Output Generated (NOG)**: The model failed to return usable output, such as empty responses, invalid Solidity code, or unrecoverable errors.
- **Correct Rate (CRate) (%)**: The percentage of repairs that were classified as **Correct Repair** out of the 749 evaluated cases.

The data in Table 5.4 reveals clear differences in effectiveness across prompting strategies. As anticipated, techniques that provided more structured and explicit guidance, particularly **Instruction-Tuned** and **Chain-of-Thought** prompting, led to the highest success rates. By contrast, **Role-Based** and **Format-Constrained** prompts offered general framing but lacked the step-by-step direction necessary for reliably producing complex repairs. **Few-Shot Learning** performed moderately well, leveraging concrete examples, but its static nature may have limited its adaptability to unseen patterns.

Overall, these results reinforce the importance of process-oriented prompting when performing automated program repair, particularly for subtle and context-sensitive issues like gas inefficiencies in loops.

Table 5.6: RQ3: LLM-2 Prompt Combination Results

Prompt	CR	NRCL	IL	NOG	CRate (%)
Zero-Shot	443	161	132	13	59.1
Structured	462	81	186	20	61.7
Immersive	377	174	183	15	50.3
Compact	333	259	110	47	44.5
Guided	461	82	187	19	61.5
Snap	464	95	168	22	61.9
ExpertR	411	149	166	23	54.9
Reasoned	479	97	162	11	64.0
Focused	377	200	154	18	50.3

5.2.4 Experiment 3: Impact of Prompt Combination Strategies on LLM-2 Repair

This experiment investigates whether combining different prompting techniques can enhance the repair performance of **LLM-2**. Building on insights from Experiment 2, we constructed a set of **two-component prompt combinations**. Given six individual strategies, this results in $C(6, 2) = 15$ unique pairwise combinations.

Table 5.5 presents all 15 possible combinations considered. For each pair, we evaluated its conceptual soundness based on how large language models typically respond to different types of guidance. Combinations that blended complementary strategies, such as reasoning-based prompts (e.g., Chain-of-Thought), concrete examples (Few-Shot), and explicit task framing (Instruction or Role), were deemed most promising. In contrast, pairs that lacked a reasoning component or relied heavily on rigid formatting were considered less likely to yield improvements.

From this set, we selected the **9 combinations labeled as "Yes"** for full evaluation. These combinations offered the strongest synergy and were best aligned with our goal of producing robust and semantically correct automated repairs for gas-vulnerable loops.

The outcomes of this evaluation are presented in Table 5.6, following the same repair classification scheme used in earlier experiments. Overall, the results reinforce the value of combining **reasoning-based prompts** with **example-driven learning** to enhance LLM-2’s repair capabilities. Among the combinations tested, **Reasoned Few-Shot** achieved the highest correct repair rate at **64.0%**, with **Few-Shot Snap** (61.9%) and **Structured Few-Shot** (61.7%) performing nearly as well. These approaches successfully leveraged the generalization power of few-shot learning while anchoring the model’s logic through structured reasoning.

On the other hand, combinations like **Compact Expert** (44.5%), **Immersive Grounding** (50.3%), and **Focused Instruction** (50.3%) yielded comparatively lower performance. We hypothesize that while these strategies excel in clarity and task framing, they fall short in providing the concrete reasoning or analogical depth required to navigate diverse loop structures. Notably, the **Compact Expert** configuration resulted in the weakest performance, suggesting that expert-style prompts alone are insufficient without complementary reasoning support.

Table 5.7: RQ4: LLM-2 Fine-Tuning Results [3]

# of Instances	CR	NRCL	IL	NOG	CRate (%)
10	545	51	11	141	72.8
20	553	44	10	131	73.8
30	585	32	8	122	78.1
40	595	24	8	119	79.4
50	630	23	6	89	84.1

These results further validate our earlier findings: the most effective prompt designs strike a balance between abstract reasoning and grounded examples, enabling the model to adapt to varied contract structures and loop vulnerabilities with greater precision.

5.2.5 Experiment 4: Fine-Tuning GPT-4.1 for Loop Repair

In this experiment, we investigated whether fine-tuning a large language model could enhance LLM-2’s repair capabilities beyond what was achievable through prompt engineering alone. We selected **GPT-4.1** as the base model for fine-tuning, given its strong reasoning capabilities, stable performance on complex generation tasks, and support for moderate-sized domain-specific fine-tuning. This made it a suitable candidate for our use case, which demands both deep semantic reasoning and precise Solidity code synthesis.

We fine-tuned GPT-4.1 using between **10 and 50 curated examples** from our repair dataset. These examples were incrementally grouped into training sets of increasing size to evaluate how the amount of fine-tuning data affects repair accuracy. For training sizes of 20 examples or more, we adopted an **80/20 train-validation split** to ensure the model was evaluated on unseen samples. In the 10-example scenario, all examples were used for training due to the platform’s minimum fine-tuning threshold.

The fine-tuning dataset was constructed through a careful curation process. We selected examples that covered a diverse range of loop structures, including bounded loops, accumulator patterns, nested iterations, and dynamic array traversals, ensuring broad generalization potential. Additionally, we included examples where prompt-only approaches had failed, specifically targeting the model’s known weaknesses. Each example was manually reviewed and validated to ensure correctness and consistency before inclusion.

Once fine-tuned, each model variant was evaluated on the full set of **749 loop repair tasks**, using the same classification criteria as in previous experiments.

Table 5.7 summarizes the results of our fine-tuning evaluation. As the table shows, the correct repair rate steadily improved with more training examples, rising from **72.8% with 10 examples** to **84.1% with 50 examples**. This trend confirms that exposure to a wider range of repair scenarios enables the model to generalize more effectively to unseen loop patterns.

Each loop repair was given a timeout window of **240 seconds**, with up to **5 retries** permitted per task. Despite these allowances, some repair attempts still resulted in **No Output Generated**. These failures were typically caused by exceeding the model’s

Table 5.8: RQ5: Repair Results After LLM-3 Feedback [3]

Config	CR1	CR2	NRCL1	NRCL2	IL1	IL2	NOG1	NOG2
Format	614	688	68	34	37	10	30	17
Instruction	661	696	40	17	19	5	29	29
Compact	630	694	61	18	22	11	36	26
Reasoned	654	712	64	14	13	10	18	13
10 Examples	569	571	2	0	0	0	178	178
50 Examples	654	658	4	2	2	0	89	89

token limits (often due to large contract inputs or intricate reasoning chains) or encountering repeated timeouts. Notably, as the fine-tuning dataset grew, the number of no-output cases consistently decreased, from 141 cases at 10 examples to just 89 at 50 examples. We interpret this as a sign of improved model confidence and output stability: with more training data, the model internalized common repair strategies more effectively, reducing both reasoning complexity and the likelihood of long, unstable response sequences.

Overall, these results demonstrate that fine-tuning substantially enhances LLM-2’s accuracy, reliability, and resilience in real-world repair scenarios, complementing the earlier improvements achieved through prompt engineering alone.

5.2.6 Experiment 5: Multi-Round Feedback Using LLM-3

In our final experiment, we examined whether introducing a dedicated **feedback layer**, powered by **LLM-3**, could further enhance the GasGaugeAI repair pipeline. The goal was to intelligently mediate between LLM-2 and LLM-1 by reasoning about the quality of repairs and triggering appropriate follow-up actions.

To build LLM-3, we used **GPT-4.1** configured with a combination of **Few-Shot Learning** and **Chain-of-Thought (COT)** prompting. This setup was designed to help the model reason through repair attempts and classify them effectively based on correctness.

We applied LLM-3 to assess both the highest- and lowest-performing configurations from the previous experiments, spanning prompt engineering, prompt combinations, and fine-tuning. For each case, LLM-3 reviewed the generated repair and classified it as either correct or incorrect. If the repair was deemed correct, it was passed to **LLM-1** for test-based validation. If marked incorrect, LLM-3 returned structured feedback to **LLM-2**, prompting it to attempt a new repair. This **two-round feedback loop** was executed for each sample to evaluate the added value of LLM-3 in orchestrating iterative improvement.

This experiment aimed to determine whether a lightweight, reasoning-based feedback agent could bridge the remaining gap between prompting-based repairs and fully validated fixes, while also reducing reliance on manual inspection in the repair cycle.

Table 5.8 reports the outcomes of applying our two-round LLM-3 feedback loop across selected prompt and fine-tuning configurations. The table includes the following columns:

- **CR1, CR2**: Number of **CorrectRepair** outcomes after the first and second feedback rounds.

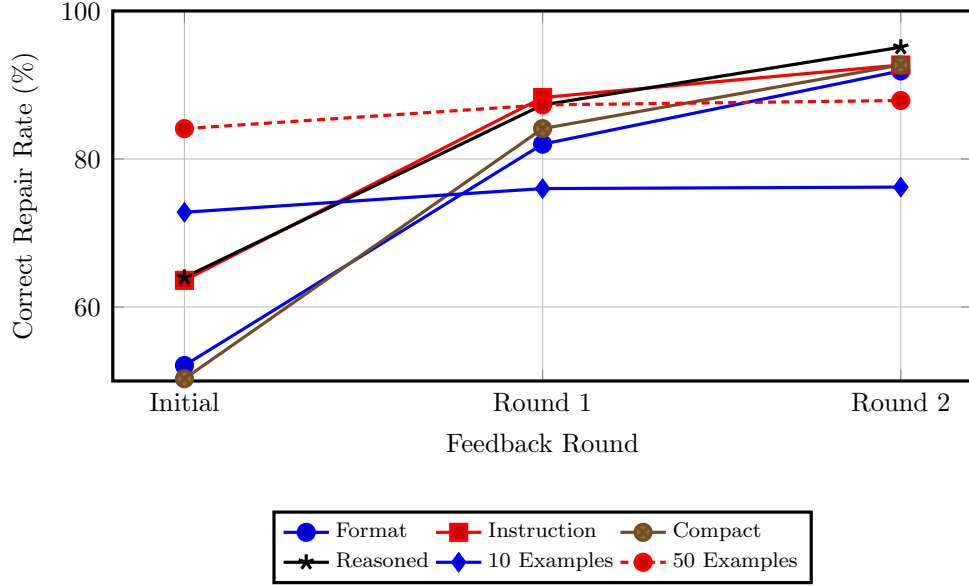


Figure 5.2: LLM-3 Feedback Impact on Repair Accuracy across 6 Configurations [3]

- **NRCL1, NRCL2**: Instances classified as `Not_Repaired_But_Correct_Logic` after each round.
- **IL1, IL2**: Number of `Incorrect_Logic` cases after each round.
- **NOG1, NOG2**: `No_Output_Generated` cases due to timeouts or token limit overflows.

Overall, the results clearly demonstrate that introducing a dedicated LLM-3 feedback mechanism significantly improved repair performance across all tested configurations. In each scenario, one or two rounds of feedback enabled LLM-2 to correct additional errors that would otherwise persist. For instance, the **Reasoned Few-Shot** configuration improved from a **64.0%** to **95.1%** correct repair rate after two rounds. Similarly, **Compact Expert** increased from **50.3%** to **92.7%**, and even configurations such as **Format** and **Instruction** surpassed the **90%** mark, as visualized in Figure 5.2.

One noteworthy observation is that for the fine-tuned models trained on 10 and 50 examples, several large loops triggered timeouts or token limit issues, limitations inherent to the underlying GPT-4.1 model rather than the repair framework itself. Despite this, in the 50-example case, for loops where the model did successfully generate output in the second feedback round, only two were incorrect; the remaining were repaired successfully. This indicates that when operational, the fine-tuned model maintained a strong internal grasp of effective repair strategies.

These findings highlight the value of integrating a lightweight, reasoning-driven feedback agent like LLM-3 into the automated repair pipeline. Not only does it substantially boost repair accuracy, but it also helps streamline the responsibilities of LLM-2 and LLM-1 by focusing their efforts more precisely on problematic cases.

5.2.7 Summary of Findings

Our experiments demonstrate that integrating prompt engineering, fine-tuning, and multi-round feedback forms a highly effective strategy for automating the repair of gas-related loop vulnerabilities in smart contracts. The following key insights emerged across our evaluations:

- **Prompt Engineering:** Structured prompts that combine reasoning, example-driven learning, and role framing significantly outperformed simpler strategies such as grounding or formatting alone. These designs provided LLM-2 with clearer guidance, resulting in more accurate and context-aware repairs.
- **Prompt Combinations:** Configurations such as `Reasoned Few-Shot`, `Snap`, `Structured Few-Shot`, and `Guided Few-Shot` proved most effective. These combinations validate the importance of blending abstract reasoning with concrete analogies to support robust repair behavior.
- **Fine-Tuning:** Fine-tuning GPT-4.1 on even modestly sized datasets led to notable improvements in both accuracy and stability. The correct repair rate increased from **72.8%** with 10 examples to **84.1%** with 50 examples. Larger fine-tuning sets also reduced the occurrence of timeouts and non-responses, indicating improved model robustness.
- **LLM-3 Feedback:** Adding a lightweight, reasoning-driven feedback layer via LLM-3 consistently elevated repair quality across all tested configurations. After two feedback rounds, the top-performing settings exceeded **95%** correct repair rates, while even the weakest initial configurations saw substantial gains. This highlights the critical role of LLM-3 in orchestrating iterative, self-correcting repairs with minimal human supervision.

Taken together, these findings affirm that a layered, multi-agent [LLM](#) framework, built on a foundation of prompt design, targeted fine-tuning, and intelligent feedback, offers a scalable and highly accurate approach to automated vulnerability repair in Ethereum smart contracts.

5.3 Summary, Discussion, and Future Work

GasGaugeAI introduces a novel multi-[LLM](#) architecture that combines vulnerability detection, repair synthesis, and behavioral validation into a fully automated pipeline for mitigating gas-related [DoS](#) vulnerabilities in Ethereum smart contracts. By specifically targeting loops whose iteration counts are governed by mutable state variables, a class of vulnerabilities often missed or incompletely addressed by existing tools, GasGaugeAI fills a critical gap in current security frameworks.

The system operates through a three-stage collaboration between language models: LLM-1 generates Foundry-based test cases to determine whether an existing function can reduce the value of a loop-bound variable, while LLM-2 steps in only when such a function is absent, synthesizing and integrating appropriate logic to ensure the loop becomes safely bounded. LLM-3 verifies the work of LLM-2 and provides feedback when needed. This closed feedback loop not only increases repair accuracy but also reduces unnecessary code injection, preserving contract semantics and minimizing unintended side effects.

The empirical results, built upon a curated dataset of 800 real-world vulnerable loops, demonstrate that GasGaugeAI achieves high accuracy, effective generalization across loop types, and operational compatibility with modern development workflows. Its extensible and modular design enables seamless integration into existing auditing pipelines, while also providing a clear upgrade path for addressing other classes of vulnerabilities.

GasGaugeAI exemplifies how emerging AI capabilities, when embedded within domain-specific workflows, can outperform static or heuristic-based tools in both accuracy and adaptability. Unlike single-shot repair models that operate without runtime feedback, GasGaugeAI leverages active testing and iterative synthesis to ensure both functional correctness and gas safety. This marks a shift toward *verifiable AI-powered repair*, which balances automation with correctness guarantees.

Moreover, the tool respects the nuanced constraints of Ethereum’s gas model, ensuring that inserted fixes are not only syntactically valid but also cost-efficient. This is particularly important in DeFi applications, where contract size and execution cost can materially impact user adoption and long-term viability.

We envision integrating GasGaugeAI with decentralized auditing networks or bug bounty platforms to automatically triage and repair contracts prior to human review. This would support continuous security monitoring and scalable, trust-minimized contract deployment, critical components for the future of secure and autonomous decentralized ecosystems.

As AI grows in power and ubiquity, blockchain’s potential to anchor it in transparent, decentralized, and incentive-compatible frameworks is becoming increasingly relevant. Whether through verifiable data sharing, decentralized computing, or trustworthy agent coordination, blockchain offers AI a path toward democratized, ethical, and auditable intelligence. This outcome is significant in adversarial or high-stakes environments.

5.4 Towards Trustworthy Decentralized Systems

As decentralized applications evolve in scale and complexity, the notion of “trustlessness” has shifted from a binary attribute to a subtle design spectrum. In practice, no blockchain system is fully trustless. Instead, it derives trust from verifiable assurances, whether in how computation is performed, how inputs are sourced, or how actors are incentivized to behave correctly. For smart contracts to function securely in adversarial environments, they must execute correctly and base their logic on reliable external data. This dual requirement underpins much of the recent progress in blockchain security infrastructure.

So far, we have explored the GasGauge toolchain, which addresses gas-based denial-of-service (DoS) vulnerabilities by bounding execution and synthesizing automated repairs. These tools operate entirely within the execution environment of Ethereum smart contracts, ensuring that loops terminate safely and that resource limits are respected.

However, execution is only part of the equation. Many blockchain applications, particularly those involving counting, pricing, or user behavior, also depend on data from the external world. This reliance introduces new challenges around liveness, integrity, and collusion resistance, especially when third parties report such data.

The next chapters will introduce CountChain and AdChain, two decentralized oracle-driven systems that support decentralized, audit-friendly counting tasks. While CountChain and AdChain systems operate in domains different from GasGauge, they are motivated by the same core problem: *how to make decentralized systems behave correctly under minimal trust assumptions.*

Chapter 6

Game-Theoretic Decentralized Oracle Network for Counting Systems

Despite the increasing sophistication of smart contracts, blockchain applications remain limited to directly accessing or verifying off-chain information. This so-called "oracle problem" creates a critical bottleneck for real-world adoption, particularly in domains that rely on accurate, high-frequency data such as advertising, voting, user engagement, and asset tracking [107][9][91]. While traditional oracle solutions like Chainlink [9] and Band Protocol [108] have demonstrated the feasibility of decentralized data feeds, they often prioritize price feeds and lack mechanisms tailored to domains that require count-based aggregation under adversarial assumptions.

To address this gap, this chapter introduces a game-theoretic decentralized oracle network (**DON**) designed to support secure and verifiable counting in adversarial environments. We begin with **CountChain** [5], a general-purpose oracle framework that enables decentralized reporting of numeric values, such as impressions, votes, or click counts, without requiring a trusted third party. CountChain ensures that the reported values are accurate and tamper-resistant by leveraging a lazy-commit-and-vote mechanism, rational incentives, and cryptographic auditability. Its design emphasizes scalability, resistance to Sybil attacks, and the use of lightweight trust assumptions. This work has been published at the *IEEE Blockchain 2024* conference.

In the next chapter, building on CountChain, we introduce **AdChain** [6], a real-world application of the protocol in the domain of digital advertising. AdChain addresses a long-standing issue in the advertising industry known as impression discrepancy between publishers and advertisers. Rather than attempting to remove intermediaries altogether, AdChain integrates seamlessly with existing infrastructure in header bidding, while introducing a decentralized layer of verifiers that independently confirm impression counts. Doing so provides a transparent framework for auditing ad campaigns and attributing discrepancies to their trustworthy source.

Both CountChain and AdChain are grounded in game-theoretic security models that align the incentives of participating nodes with truthful reporting. Through cryptographic commitments, stake-weighted voting, and slashing conditions, these protocols ensure that

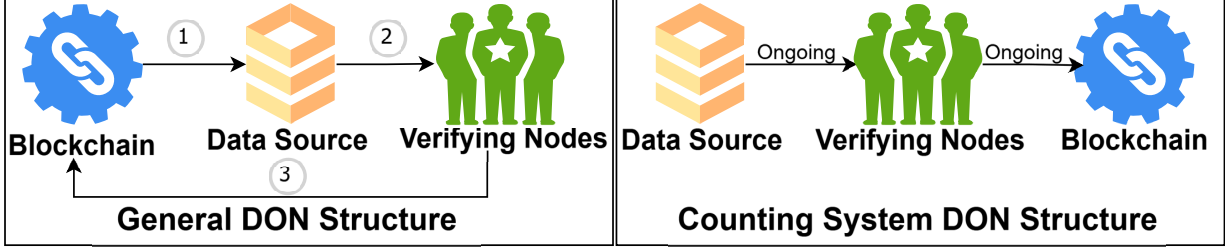


Figure 6.1: General DON vs. Counting System DON Design [5].

acting honestly is a rational strategy even in the presence of noise, latency, or malicious actors.

6.1 CountChain: A DON for Verifiable Counting

Blockchain applications increasingly rely on data that originates outside of the chain, whether for governance, finance, or user interaction. In many of these cases, what is needed is not just a scalar value (like a token price), but an accurate *count* of an event. For instance, we need to know how many people voted, how many users viewed a post, or how many ad impressions were delivered. Unfortunately, existing [DONs](#) are not well-suited to high-frequency counting tasks [5]. This is the primary motivation behind the design of CountChain.

6.1.1 Motivation and Problem Scope

Counting systems serve as foundational tools for recording the frequency of specific events, playing a critical role in high-volume domains such as online advertising. Traditional counting architectures, however, often suffer from systemic vulnerabilities, including susceptibility to errors, central points of failure, and a lack of transparency or trust. To address these shortcomings, blockchain-based solutions have emerged as promising alternatives. By integrating smart contracts with [DONs](#), these systems aim to provide tamper-resistant, transparent, and verifiable counting mechanisms [9][107].

Traditional [DONs](#) like Chainlink [9] and Band Protocol [108] are optimized for low-latency, scalar-value resolution. These systems typically rely on a small group of trusted or pre-selected nodes to fetch and submit off-chain values, such as token prices or weather conditions, which are then aggregated on-chain via majority or median functions [9, 108].

While [DONs](#) have shown promise in bridging on-chain and off-chain data, most existing architectures struggle to support the high-precision demands of large-scale counting systems, such as those used in online advertising. These shortcomings pose a substantial barrier to blockchain adoption in billion-dollar industries that depend on exact counts to ensure fair compensation and prevent fraud [109][110].

To provide more context, in typical **DONs** implementations like Chainlink, smart contracts initiate data requests, specify service parameters via Service Level Agreements (SLAs), and rely on oracles to fetch and verify off-chain data. Results are aggregated, often using statistical techniques like median selection, and participating oracles are financially incentivized [9]. This model works well for discrete, low-frequency queries, such as fetching the ETH/USD price, but is poorly suited for high-frequency, unpredictable event streams like ad impressions.

Counting systems require a fundamentally different workflow. Events can occur asynchronously and continuously, meaning smart contracts cannot preemptively request data. Instead, the data must be pushed to the contract after being observed and validated. This reverses the typical **DON** flow, requiring oracle networks to initiate interaction with the contract upon verification of external events. Figure 6.1 contrasts this reactive counting system design with conventional **DON** architectures.

The need for decentralized counting systems is more intricate. As illustrated in Figure 6.1, these systems demand a structured protocol that includes independent observation by multiple parties, commitment schemes to ensure data integrity, a transparent and verifiable voting process, and an outcome-driven reward mechanism to align participant behavior with truthful reporting [5]. These components go beyond what existing oracles offer and require a framework grounded in distributed consensus and game-theoretic modeling.

Current approaches to counting via **DONs**, including number-based SLAs (e.g., "How many views?") or binary propositions (e.g., "Did X see Y?"), introduce their own limitations. Number-based models suffer from ambiguity when oracle responses differ slightly, e.g., if two oracles report nine views while three report ten, it is unclear whether the discrepancy reflects shared or unique data losses. Binary SLAs, meanwhile, are vulnerable to single points of failure when the requester misses some data. A hybrid model, where one oracle proposes a count and others vote to agree or challenge it, presents a promising direction but requires new consensus protocols to prevent dishonest or lazy oracle behavior.

Additional concerns include scalability and cost since rewarding oracles for each interaction leads to high transaction overhead. More importantly, accurate record-keeping often requires metadata like user IDs or timestamps, which existing **DONs** are not well-equipped to handle. As a result, current **DONs** fall short of ensuring the precision, reliability, and transparency demanded by modern counting systems [109][110].

6.1.2 Design Principles of CountChain

These challenges motivate the design of **CountChain**, a decentralized oracle protocol that provides secure, auditable, and incentive-compatible *counting-as-a-service* for blockchain applications. CountChain's design is based on four core principles:

1. **Decentralized Observation:** A pool of verifiers independently observe and record counts from a public or semi-public data source (e.g., webpage visits, smart contract events, external logs) [5].

2. **Commit-and-Reveal for Integrity:** To prevent free-riding or collusion, verifiers hash-commit their observed values and reveal them only after all commitments are submitted [5].
3. **Stake-Weighted Voting:** Final values are selected through a stake-weighted majority vote, discouraging manipulation and aligning incentives for accurate reporting [5].
4. **Auditability and Slashing:** Disagreements and inconsistencies are penalized through stake slashing, while winning verifiers receive rewards from a shared prize pool [5].

The protocol is chain-agnostic, lightweight, and compatible with modular blockchain architectures, including Layer 2 solutions and sidechains [5]. Its design supports deployment in latency-sensitive environments without periodic price updates or semi-centralized data providers. CountChain lays the foundation for blockchain systems that rely on high-resolution off-chain event data by emphasizing rational-agent safety and distributed validation.

System Assumptions

The CountChain protocol relies on system-level assumptions grounded in probabilistic modeling and cryptographic integrity to support decentralized, verifiable counting at scale. These assumptions are inspired by standard models in distributed consensus and voting-based reputation systems [5].

System Assumptions. We assume the following throughout the protocol:

1. **Pseudo-random number generation** is available via smart contract functions (e.g., future block hashes) or off-chain randomness beacons, ensuring fair verifier selection.
2. **Boolean truth space:** Each proposition p submitted to the system has a binary ground truth value $v(p) \in \{T, F\}$ (True or False), which is known only implicitly based on the data source.
3. **Node accuracy model:** The system consists of N participating nodes (players), each denoted by $i \in \{1, \dots, N\}$. For every proposition p , node i holds a private belief $b_i(p) \in \{T, F\}$ regarding its truth value. Node i is associated with a belief accuracy $q_i(p) \in [0, 1]$, representing the likelihood that $b_i(p)$ aligns with $v(p)$.
4. **Belief independence and honesty:** Node beliefs are drawn independently across nodes. Honest nodes always vote according to their actual belief $b_i(p)$. Malicious nodes may deviate arbitrarily.

These assumptions ensure that CountChain can use stake-weighted voting to aggregate beliefs in a way that probabilistically favors honest reporting, even under partial adversarial participation.

Term	Definition	Set by
owner	the party interested in obtaining the result of the counting system	network
host	platform storing all information	owner/ network
player	participating nodes	network
	~ ID : the unique identifier of each player	host
	~ point : the points accumulated for each vote or proposition submission	host
proposition	a boolean request to be validated	submitter
	~ deadline : the deadline to vote for a proposition	owner/ host
	~ pool : the list of available propositions to be verified	host
	~ price : the stake used when players vote for propositions	owner
submitter	the player who submits a proposition to be verified	players/ host
verifier	randomly chosen player to verify a proposition	host
input ID	unique identifier of the input data	data source
prize pool	financial reward and incentive	owner/ players

Table 6.1: Summary of the terminology of CountChain [5].

6.1.3 Setup and Terminology

The key components and roles in CountChain are summarized in Table 6.1. This vocabulary is used consistently throughout the protocol description.

The architecture involves two core roles of the *owner* and the *host*. Together, they are responsible for configuring the parameters of the system. The *host* maintains an internal state, including the list of active *propositions* and metadata about participating nodes. These nodes, often referred to as *oracles* or *players*, serve as validators within the system and are conceptually similar to miners in conventional blockchain frameworks. To promote fairness and decentralization, the host employs a randomized selection mechanism that ensures balanced task distribution across nodes.

The *owner* is the party that initiates and funds the counting system. This entity defines the conditions for event tracking, sets the relevant parameters, and provides the initial financial resources. In practical deployments, the roles of host and owner can be implemented through separate smart contracts, either with the host represented by a dedicated contract and the owner being the contract deployer, or with distinct contracts assuming each role.

Each item or event subject to counting is assigned a unique identifier, the *Input ID*. The system operates on a set of *propositions*, each associated with a covert truth value and a corresponding *proposition price*, which serves as the basis for a gamified voting protocol. Each proposition includes a termination condition, typically a *proposition deadline*, by

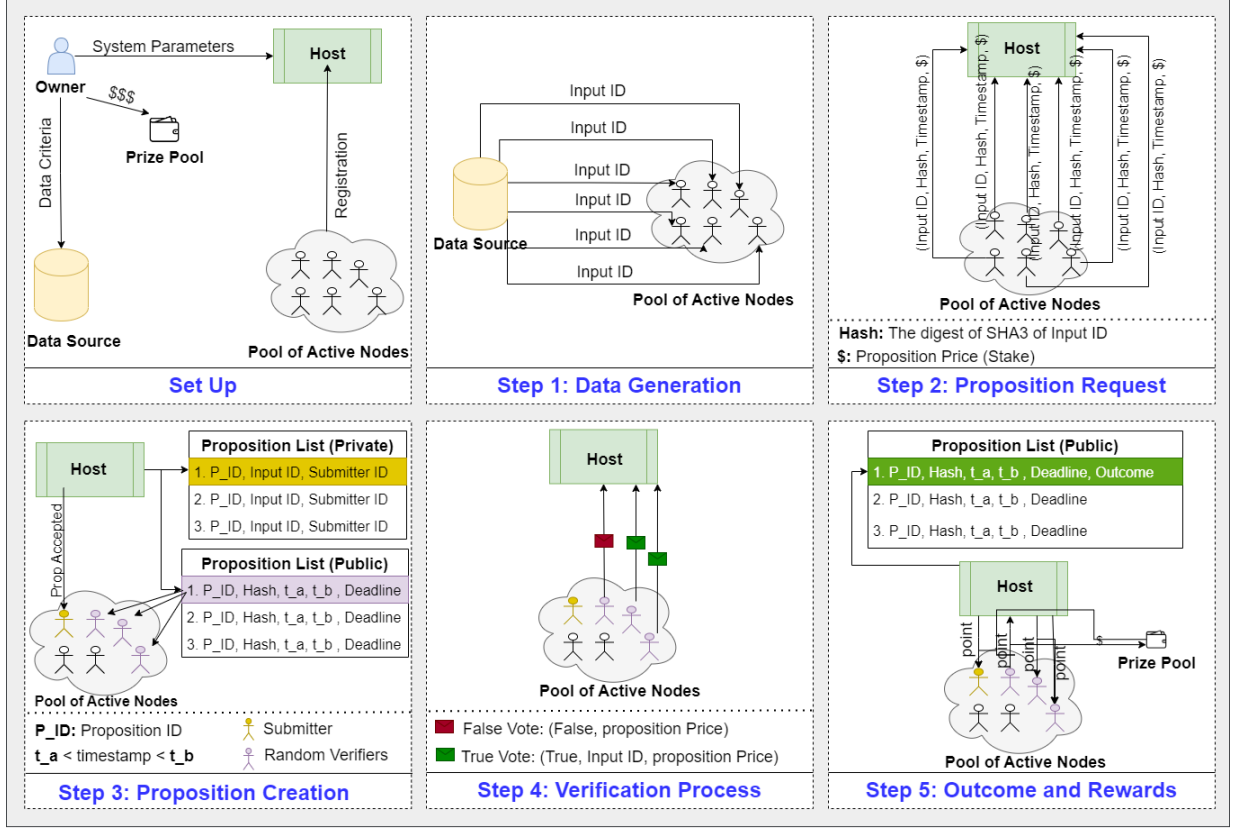


Figure 6.2: An Overview of CountChain Mechanism [5].

which all votes must be submitted. The owner defines this deadline and determines when the proposition outcome is finalized.

The financial structure of the system is supported by a *prize pool*, which consists of the initial deposit from the owner and penalty funds collected from dishonest participants. All propositions are recorded in the *Proposition Pool* for tracking and verification.

Each node, or *player*, in the system is uniquely identified by a *player ID* (e.g., their blockchain address) and accumulates *player points* through system participation, such as submitting valid propositions or casting votes. These points determine the player's final reward allocation. Players may serve in one of two roles: as a *submitter*, responsible for introducing propositions, or as a *verifier*, who evaluates the validity of assigned propositions.

This framework sets the stage for the protocol mechanics described in the following subsections, where players transition between roles (submitters, verifiers), and the system coordinates trust-minimized validation of counting propositions.

Algorithm 1 Host Operations

```
1: propList  $\leftarrow []$ 
2: function SUBMITPROP(req)
3:   if InvalidHash(req) then
4:     return WrongHash(req)
5:   end if
6:   if not PropExists(req) then
7:      $T_a, T_b \leftarrow \text{TimeSpan}(\text{req.ts})$ 
8:     prop  $\leftarrow \text{CreateProp}(\text{req}, T_a, T_b)$ 
9:     verifiers  $\leftarrow \text{AssignVerifiers}(\text{req})$ 
10:    for v in verifiers do
11:      SendProp(v, prop)
12:    end for
13:  end if
14: end function

15: function SUBMITVOTE(vote)
16:   if not IsLate(vote) then
17:     if vote.val & !Hash(vote) then
18:       return WrongHash(vote)
19:     end if
20:     AddVote(vote)
21:   end if
22: end function
```

Algorithm 2 Verifier Operations

```
1: dataStore  $\leftarrow []$ 
   // Listen for data from Host or Source
2: while Active do
3:   if data is received then
4:     if FromSource and Valid then
5:       dataStore.add(data)
6:       h  $\leftarrow \text{Hash}(\text{data.id})$ 
7:       SendPropReq(data, h, ts)
8:     else if FromHost then
9:       vote, id  $\leftarrow \text{False}, \text{NULL}$ 
10:      for entry in dataStore do
11:        if  $T_a \leq \text{e.time} \leq T_b$  &
          Hash(entry.id) = data.hash then
12:          vote  $\leftarrow \text{True}$ 
13:          id  $\leftarrow \text{entry.id}$ 
14:          break
15:        end if
16:      end for
17:      SendVote(vote, id)
18:    end if
19:  end if
20: end while
```

Figure 6.3: Pseudocode for the Host and Verifier components in the CountChain protocol. The host handles proposition submissions and vote collection, while verifiers independently validate observed data against received propositions [5].

6.1.4 System Description

The overview of CountChain is shown in Figure 6.2. The system contains multiple steps, as shown below:

Step 1: Data Collection: The process begins with the [data source](#) distributing information about the observed event to all active participants ([players](#)) in the system. Each player independently records the data according to predefined criteria. For instance, if the system is designed to count website visitors, the visitor’s IP address serves as the [Input ID](#), and this information may be captured through methods such as invisible tracking pixels.

Step 2: Proposition Submission: When players identify a valid event, they submit a [proposition](#) request to the [host](#) and stake an amount equal to the [proposition price](#). The request must include the [Input ID](#), a timestamp, the [player ID](#), and the $\text{Hash}(\text{Input ID})$, which is a cryptographic digest of the input identifier. This submission process is permissionless and can be executed directly on-chain.

Step 3: Proposition Generation and Assignment: Upon validating a proposition request, the host creates a formatted proposition, adds it to the [proposition pool](#), and randomly assigns a set of players to act as [verifiers](#). To preserve privacy, the Input ID is excluded from the proposition; only its hash is included. The timestamp is expanded into a time interval $[T_a, T_b]$, where the original timestamp lies at the midpoint, to accommodate possible network latency. Optionally, an additional identifier (e.g., Ad ID) may be appended to ensure uniqueness within the time window. The proposition also includes a deadline by which verifiers must cast their votes. Although all players can view propositions, only the assigned verifiers may vote. The player who submitted the request becomes the proposition’s [submitter](#), and the number of assigned verifiers, determined by the [owner](#), is fixed across all propositions.

Step 4: Vote Evaluation: Each assigned [verifier](#) scans the data recorded during $[T_a, T_b]$ and computes the hash of every [Input ID](#). If the computed hash matches the one in the [proposition](#), the [verifier](#) casts a *True* vote and submits the corresponding [Input ID](#) as evidence. Otherwise, they vote *False*. Any [verifier](#) who fails to vote by the deadline is automatically considered to have voted *False*. All [verifiers](#) must stake the [proposition price](#). This mechanism relies on the one-way nature of hash functions. While it is computationally infeasible to guess the [Input ID](#) from its hash, it is trivial to verify a match if the [Input ID](#) is already known. This ensures that only [verifiers](#) with genuine data can vote *True*.

Step 5: Outcome Determination: The [proposition](#)’s outcome is decided by a majority vote. [Verifiers](#) in the majority receive rewards, while those in the minority are penalized. In the case of an even number of [verifiers](#), the [submitter](#)’s vote is counted as *True* to break ties. The system ensures consistent accuracy by using a fixed, random subset of [verifiers](#) for each [proposition](#). It significantly reduces the potential for [Sybil attacks](#), preventing collusion or concentration of voting power on specific propositions.

The algorithms corresponding to Steps 2–5 are shown in Figure 6.3.

6.1.5 Reward and Penalty Mechanism

To promote honest behavior and defend against manipulation, the system incorporates a structured combination of incentives and penalties. When a [player](#) submits a [proposition](#) or casts a *True* vote, the [host](#) verifies whether the [Input ID](#) and $\text{Hash}(\text{Input ID})$ are consistent. If the verification fails, the [player](#) forfeits their stake and receives a penalty of two negative points, discouraging spam and [DoS](#)-style attacks.

If the [proposition](#) is accepted (majority *True* votes), the [submitter](#) receives four positive points, and each [verifier](#) who voted *True* earns one point. [Verifiers](#) who voted *False* lose their stake and receive one negative point. Conversely, if the [proposition](#) is rejected (majority *False* votes), the [submitter](#) loses their stake and is assigned two negative points, while [verifiers](#) receive no points. In scenarios where a [verifier](#) votes *True* with the correct [Input ID](#) but ends up in the minority, the lost computational cost serves as a soft penalty. As such, the system assigns greater risk and potential reward to [submitters](#), incentivizing them to propose high-quality [propositions](#).

		Majority Outcome	
		True	False
Verifier	Vote True	[1, 0]	[0, 0]
	False	[-1, -1]	[0, 0]
Submitter		[4, 0]	[-2, -1]

Table 6.2: Separated payoff matrix for CountChain. Each entry shows [Point, Stake]. Verifier payoff depends on their vote; submitter payoff depends only on the majority outcome.

Players are eligible for financial rewards once a termination condition is met. This condition could be time-based, event-based (e.g., a fixed number of **propositions**), or point-based. Only **players** with positive point balances receive a share of the **prize pool**, allocated using the formula:

$$\text{Verifier Prize} = \text{Prize Pool} \cdot \frac{p_{v_i}}{\sum_{j=1} p_{v_j} \cdot u(V_j)}$$

Here, p_{v_i} is the point total for **player i** , and $u(V_j)$ is a Heaviside function that evaluates to 1 if $p_{v_j} > 0$, and 0 otherwise. After distribution, all **player** points are reset to zero.

This point-based payout model offers several advantages. It encourages ongoing participation and fosters sustained honest behavior. **Players** who lose points are motivated to improve performance, while those who gain points are incentivized to maintain integrity. Furthermore, it minimizes the number of financial transactions, discourages lazy voting, and enhances **Sybil attack** resistance.

Mathematically, with N total **players** and n randomly chosen **verifiers** per **proposition**, a **Sybil attack** only succeeds if the adversary controls at least $N - \frac{n}{2}$ nodes, an unlikely scenario when $N \gg n$. Even partial success (manipulating at least 50% of **propositions**) would require controlling over 51% of the network. Thus, unlike most blockchain systems where 51% control ensures dominance, CountChain’s structure limits the attacker’s influence, providing enhanced resilience against coordinated attacks.

6.2 Incentive Design and Game-Theoretic Security of CountChain

The security of CountChain relies on cryptographic integrity, randomness, and properly designed incentive structures that guide rational players to behave honestly. This section analyzes the protocol’s reward mechanism through the lens of game theory and incentive-compatible mechanism design [5].

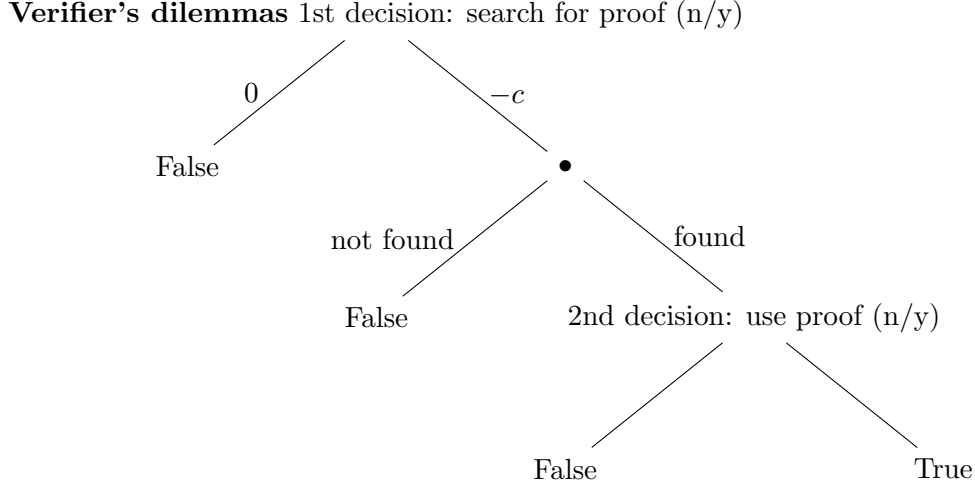


Figure 6.4: A verifier faces two sequential decisions: first, whether to incur a cost $-c$ to search for proof; and second, if successful, whether to act honestly and vote *True*, or disregard the proof and vote *False* [5].

6.2.1 Point System as a Utility Model

At the core of CountChain lies a dynamic point-based system that governs both economic incentives and reputation. Each [player](#) receives or loses points based on their [proposition](#) submissions and vote outcomes. These points serve two purposes. First, they determine the share of the [prize pool](#) a [player](#) receives upon reaching a reward threshold. Next, they act as a proxy for honest behavior since negative points reduce a [player](#)'s chance of economic gain.

This design discourages lazy behavior, spamming of invalid [propositions](#), or speculative voting. [Submitters](#) bear a greater stake and earn a higher reward only if their [proposition](#) passes majority validation, ensuring that honest event detection is prioritized.

6.2.2 Incentive Compatibility and Nash Equilibrium Analysis

A Nash Equilibrium ([NE](#)) refers to a state in strategic decision-making where no participant can unilaterally improve their utility by altering their strategy, assuming all other [players](#) maintain theirs. In the context of CountChain, verifying that the protocol design admits an [NE](#) for honest behavior is essential to ensuring its robustness and incentive compatibility.

Within each round of CountChain, [verifiers](#) are required to make two key decisions, as illustrated in Figure 6.4. The [verifier](#)'s first decision revolves around whether to expend effort to search for a valid proof. This action incurs a fixed computational cost $c > 0$, resulting in a utility of $-c$ for searching. If the [verifier](#) abstains from searching, the utility is zero. Per protocol, only those who search and find valid proof may vote *True*; otherwise, they must vote *False*.

If proof is found, the **verifier** then faces the second decision: to vote *True*, reflecting honest behavior, or to vote *False*, disregarding the valid evidence. The utilities associated with these decisions are influenced by the outcome of other **verifiers**' choices, as summarized in Table 6.2.

From the table, we observe that being in the majority is always advantageous, and voting *True* when holding a valid proof is a weakly dominant strategy. When the majority votes *True*, **verifiers** benefit strictly from doing the same. When the majority votes *False*, **verifiers** are indifferent, but not worse off by being honest.

However, the game may admit multiple equilibria. For example, if all **verifiers** default to lazy voting (i.e., voting *False* without searching), then it becomes individually rational for a **verifier** to abstain from searching as well. In such scenarios, no **propositions** are ever confirmed as *True*, and the effort expended by honest **verifiers** yields negative utility.

Thus, a central design question emerges: under what conditions is honest behavior, searching for proof and voting *True* when applicable, an **NE**? Theorem 6.2.1 addresses this question. Assuming that a fraction $p_T \in (0, 1]$ of **propositions** are indeed true, honest behavior is an equilibrium if $2p_T > c$. This assumption is relatively mild, given that the cost of hashing-based search is low.

Theorem 6.2.1 (Incentive Compatibility of CountChain [5]). *Assume that a fraction $p_T \in (0, 1]$ of propositions are True and that searching for a valid proof incurs a cost $c > 0$ to the **verifier**. Then, if all other **verifiers** are behaving honestly, i.e., search for valid proofs and vote True whenever they find one, then it is best for a remaining **verifier** also to behave honestly if and only if $2p_T > c$.*

Proof. Assuming that all **verifiers** behave honestly ensures that a fraction of p_T propositions will be decided as True, i.e., all True propositions will be decided as True and all False propositions will be decided as False. Thus, according to Table 6.2, the expected utilities of a single **verifier** from their possible actions {no-search, search-false, search-true} are the following:

$$\begin{aligned}\mathbb{E}[\text{no-search}] &= p_T(-1) + (1 - p_T) \cdot 0 = -p_T \\ \mathbb{E}[\text{search-false}] &= -c + p_T(-1) + (1 - p_T) \cdot 0 = -p_T - c \\ \mathbb{E}[\text{search-true}] &= -c + p_T \cdot 1 + (1 - p_T) \cdot 0 = p_T - c\end{aligned}$$

Clearly, the action search-false is dominated by no-search since $-p_T - c < -p_T$ for $c > 0$. Thus, to determine a **verifier**'s optimal action, we only need to consider the actions "search" and "vote True" whenever one possesses valid proof. The expected utility of a **verifier** who selects "search-true" $x \in [0, 1]$ fraction of the time and "no-search" $1 - x$ fraction of the time is

$$\begin{aligned}\mathbb{E}[x] &= x[-c + p_T \cdot 1 + (1 - p_T) \cdot 0] \\ &\quad + (1 - x)[p_T \cdot (-1) + (1 - p_T) \cdot 0] \\ &= x(2p_T - c) - p_T.\end{aligned}$$

This increases in x whenever $2p_T > c$, as claimed. In particular, the **verifier**'s utility is optimized for $x = 1$, which corresponds to always honest behavior. $\square$

In general, for any fraction $p \in [0, 1]$ of propositions that are expected to be decided as True, regardless of whether this matches the actual truth distribution p_T , the same result holds. As long as $2p > c$, honest behavior remains a best-response strategy in the verifier game.

Resistance to Lazy Voting and Free-Riding

One of the common vulnerabilities in decentralized oracle networks is lazy voting, where players abstain or guess outcomes to game the system with minimal effort. In CountChain, this is mitigated by two key mechanisms:

- **Commit-and-reveal structure:** Verifiers must submit cryptographic evidence (i.e., Input IDs) when voting True, and guessing or voting without evidence results in forfeiture of the proposition price.
- **Negative point feedback:** Unjustified or abstained votes incur point penalties, which reduce a player's future reward share or block their eligibility for future verification.

Over time, this system filters out low-quality players and encourages accurate and verifiable behavior.

6.2.3 Sybil Attack Mitigation

A Sybil attack refers to a scenario where an adversary floods the network with fake identities to subvert voting or consensus. In CountChain, verifiers for each proposition are selected randomly, making such attacks significantly more expensive.

Let N be the total number of players, n the number of verifiers selected per proposition, and k the number of nodes controlled by the attacker. An attacker will only be successful in a single round if they capture more than $n/2$ verifiers. The probability of this happening decreases rapidly if $N \gg k$ and n are chosen appropriately. Furthermore, since players accumulate points over time, an attacker must dominate verifier slots across many rounds to extract meaningful rewards, further increasing the cost of the attack.

Additionally, because the submitter vote only breaks ties (rather than overriding the majority), adversaries cannot trivially bias the system by flooding it with fake submissions unless they also dominate verification. This property preserves partial security even under majority control.

6.2.4 Game-Theoretic Robustness Summary

Overall, CountChain promotes *rational honesty* through a synergy of cryptographic guarantees, probabilistic verifier selection, and an incentive-compatible design. The delayed,

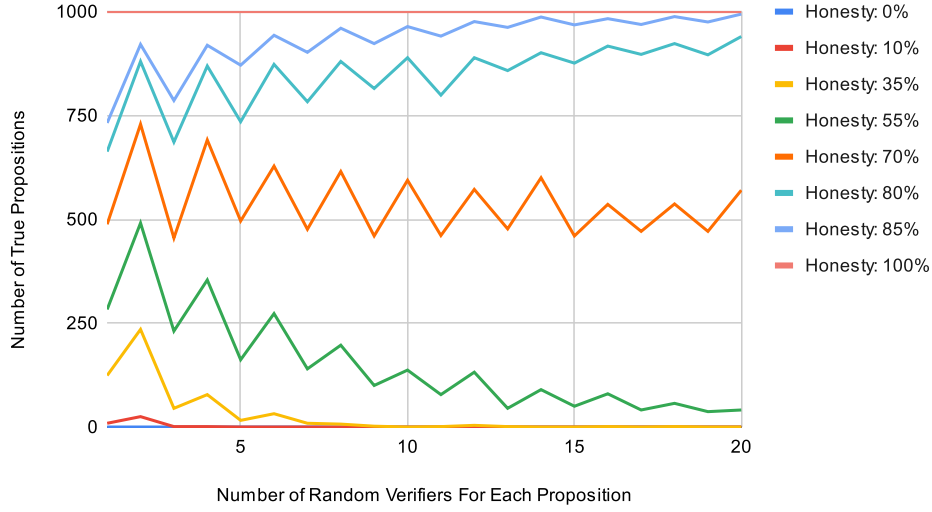


Figure 6.5: Impact of honesty rate and verifier count on proposition accuracy [5].

batch-based reward system combined with a risk-weighted point structure ensures that rational [players](#) have strong incentives to act honestly over time. Deviations from protocol honesty yield strictly lower expected utility. Attack vectors such as free-riding, [Sybil attack](#) control, and [propositions](#) spam are either penalized directly or disincentivized through structure.

This model follows the emerging trend of integrating mechanism design with decentralized security, making CountChain a practical and theoretically grounded framework for verifiable off-chain counting.

6.3 Experimental Evaluation of CountChain

To evaluate the correctness, robustness, and practical viability of CountChain, we implemented a prototype using an application in NodeJs and simulated various network conditions and adversarial behaviors. Our goal was to empirically verify that the protocol satisfies its core security and incentive objectives under realistic deployment settings. To do so, we conducted a series of experiments using an Azure E96ds v5 virtual machine (96 vCPUs, 672 GiB RAM, Ubuntu 20.04).

6.3.1 Experiment 1: Optimizing System Parameters

This experiment investigates how key parameters, namely node honesty rate, number of [verifiers](#) per [proposition](#), and total network size, affect system performance. The *honesty rate* reflects how likely [players](#) are to vote truthfully. We varied honesty rates from 0% to 100% (in 5% increments) and adjusted the [verifier](#) count from 1 to 20. For each con-

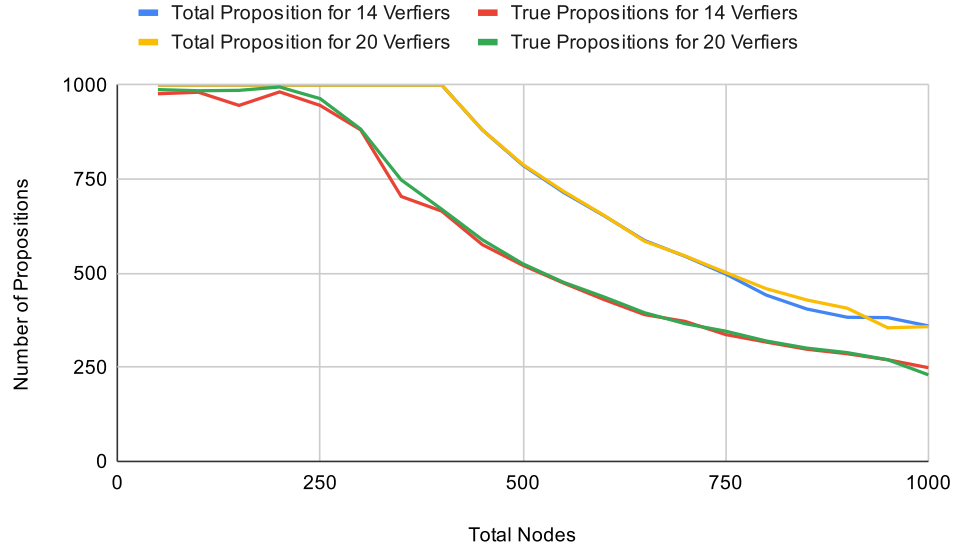


Figure 6.6: Optimal node count for maximizing accuracy [5].

figuration, 1,000 valid [Input IDs](#) were processed, yielding 420,000 [proposition](#) attempts in total.

As shown in Figure 6.5, accuracy is highest when all [players](#) are fully honest (100%) and degrades significantly when the honesty rate drops below 85%. With at least 85% honesty, precision reaches 99%, while it falls to 95% at 80%. This threshold marks a turning point where the likelihood of inaccurate majorities rises. To maintain high precision, filtering unreliable [players](#) and enforcing a minimum honesty rate of 85% is recommended.

Additionally, increasing the number of [verifiers](#) enhances precision only when the node honesty rate exceeds 80%. In contrast, adding [verifiers](#) can degrade accuracy with low honesty rates (below 60%). Interestingly, using an even number of [verifiers](#) consistently yielded better results due to tie-breaking rules involving the [submitter](#)'s vote. Based on performance and resource trade-offs, the optimal [verifier](#) count per [proposition](#) is 14.

We then evaluated how the total number of [players](#) affects system accuracy, varying network size from 50 to 1,000. As shown in Figure 6.6, performance improves up to 250 [players](#) but degrades beyond that. At 200 [players](#), accuracy was optimal, achieving 982 correctly validated [propositions](#) with 14 [verifiers](#) and 995 with 20 [verifiers](#). However, this improvement is marginal, indicating that 14 [verifiers](#) from 200 [players](#) provide the best balance.

Performance Analysis: Resource usage increases linearly, with every 50 additional [players](#) consuming 0.6% to 1% more CPU and RAM. Memory limits, not disk I/O, emerged as the primary bottleneck, especially under heap allocation constraints. Nonetheless, average processing latency (excluding network delays) remained low at 116 [ms](#), reflecting CountChain's efficiency. Fixed data formats, minimal storage overhead, and a lightweight consensus process enable this.

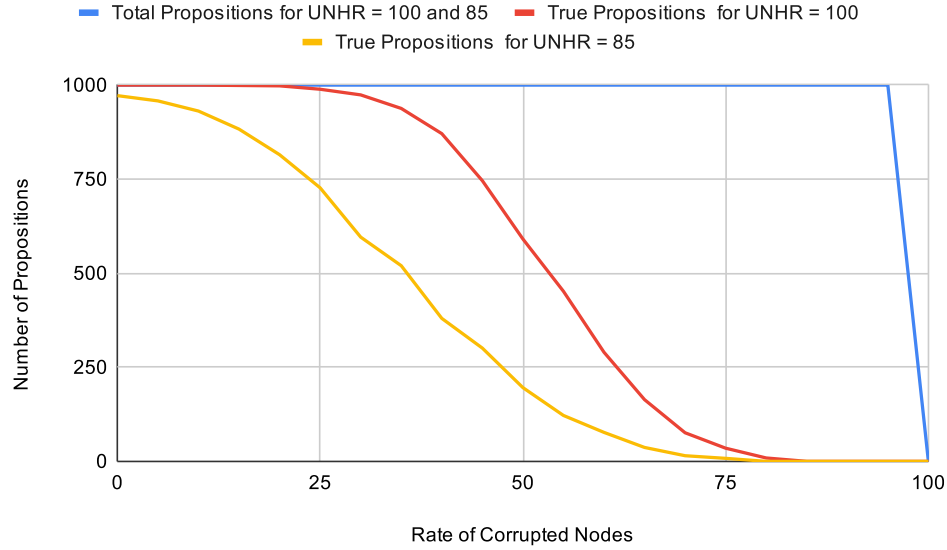


Figure 6.7: The success rate of Sybil attacks against CountChain. UNHR = Uncorrupted Node Honesty Rate [5]

6.3.2 Experiment 2: Sybil Attack Resistance

A [Sybil attack](#) occurs when a single entity controls multiple pseudonymous identities [111]. We simulated such attacks to determine how resilient CountChain is to adversarial [player](#) control. Fourteen [verifiers](#) were randomly selected from 200 [players](#), with corrupted [player](#) ratios varied from 0% to 100%. The corrupted [players](#) had 0% honesty, while the remaining [players](#) were tested with honesty rates of 85% and 100%.

As shown in Figure 6.7, when honest [players](#) had 100% honesty, attackers needed to control at least 85% of the network to compromise the system entirely. With an 85% honesty rate, this threshold dropped to around 70%. Importantly, [propositions](#) continued to be submitted successfully as long as at least one honest [player](#) remained, highlighting CountChain’s robustness in liveness, even under partial compromise.

CountChain’s resilience stems from three core mechanisms: (1) selective [verifier](#) assignment limits adversarial influence, (2) the point-stake system imposes costs for dishonest behavior, and (3) [players](#) with persistently negative scores are banned. These constraints collectively make [Sybil attacks](#) costly and impractical.

Comparative Defenses: Conventional systems rely on Proof-of-Work (PoW), Proof-of-Stake (PoS), or Delegated PoS (DPoS) to mitigate [Sybil attacks](#) [112, 15, 113, 114, 115]. While PoW is energy-intensive, PoS ties control to the economic stake, and DPoS risks centralization through vote delegation. CountChain achieves a similar deterrent effect by tuning system parameters, using random [verifier](#) selection, enforcing penalties, and optionally requiring authentication.

For example, if 66 of 200 [players](#) are malicious (33%), the probability that eight or more

of the 14 randomly selected [verifiers](#) are dishonest (i.e., the attacker gains a majority) is approximately 5%:

$$\frac{\binom{66}{8}\binom{134}{6} + \binom{66}{9}\binom{134}{5} + \cdots + \binom{66}{14}\binom{134}{0}}{\binom{200}{14}}$$

This probability aligns closely with the empirical results presented in Figure 6.7, further validating CountChain’s probabilistic resilience.

CountChain presents a robust and adaptable solution to the problem of decentralized, verifiable counting. Combining cryptographic commitment, stake-weighted voting, and point-based incentives enables secure data validation without relying on centralized authorities. While CountChain was designed as a general-purpose protocol applicable to a wide range of scenarios, from voter turnout tracking to decentralized sensor networks, it also serves as the foundation for more specialized systems. Next, we introduce *AdChain*, a real-world application of CountChain tailored to the online advertising ecosystem, where impression discrepancy and unverifiable reporting remain pervasive challenges. AdChain demonstrates how CountChain’s design principles can be extended to domain-specific oracle requirements while preserving compatibility with existing infrastructure.

6.4 Conclusion and Discussion

This chapter presented CountChain, a game-theoretic [DON](#) tailored for verifiable counting in adversarial environments. CountChain addresses a critical gap in existing oracle infrastructures by introducing a secure, scalable, and economically rational mechanism for aggregating high-frequency count data, such as ad impressions, votes, or user interactions, without relying on a trusted third party.

Through its lazy-commit-and-vote mechanism, point-based rewards, and slashing penalties, CountChain aligns the incentives of verifiers with honest reporting, even under probabilistic belief models. Our theoretical analysis showed that honest behavior constitutes a Nash Equilibrium under mild assumptions, and empirical evaluation confirmed the protocol’s robustness to [Sybil](#) attacks and node dishonesty.

While CountChain is a general-purpose solution, its design is particularly impactful in domains where real-time, tamper-proof counting is crucial. The next chapter demonstrates this through AdChain, a production-level application of CountChain within the digital advertising ecosystem. This transition highlights how foundational protocol primitives can be extended to industry-specific problems while preserving decentralization and auditability.

Chapter 7

Towards Decentralized Header Bidding Solutions

The online advertising ecosystem relies heavily on real-time bidding. However, significant challenges remain around *ad impression discrepancy*, the mismatch between impressions reported by the parties involved. These discrepancies lead to disputes, revenue loss, and distrust across participants. Existing solutions rely on centralized verification or fragmented third-party measurement services, which fail to provide cryptographically verifiable guarantees or decentralized auditability.

In this chapter, we present **AdChain** [6], a novel decentralized header bidding solution that leverages our previously developed **CountChain** [5] framework. AdChain integrates a lightweight consensus protocol and token-incentivized proposition validation mechanism to enable verifiable counting of ad impressions, clicks, and related metrics in real-time. This approach ensures that discrepancies can be detected and resolved transparently, reducing fraud and increasing trust between the involved parties. This work was published at the *MARBLE 2024 conference*, underscoring its relevance to both academic and applied blockchain research communities.

7.1 Online Advertising and the Need for Verifiable Impressions

The digital advertising industry is a multi-billion-dollar ecosystem that drives the monetization of most modern web content [116][117]. Advertisers pay to display content to users, while publishers sell ad inventory, typically in the form of impressions. These transactions happen at high speed and scale through automated systems, yet impression delivery remains one of the least transparent aspects of the process. A particularly pervasive issue is *impression discrepancy*, where advertisers and publishers disagree on the number of ads delivered or seen (impression) [118].

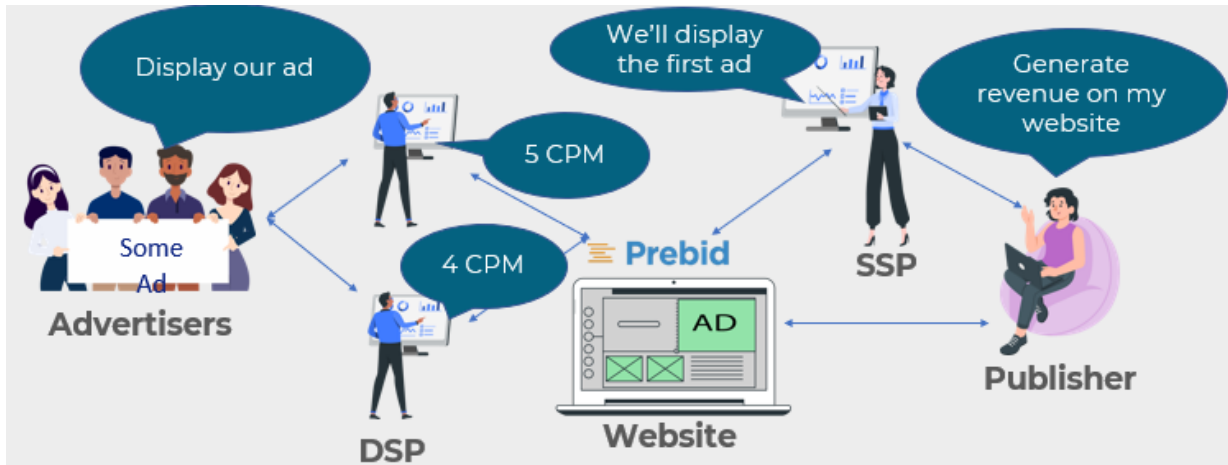


Figure 7.1: Illustration of the header bidding process in online advertising. Multiple DSPs bid through SSPs using a Prebid.js integration on the publisher’s website [6].

7.1.1 The Programmatic Stack and Header Bidding

Digital advertising operates through a real-time programmatic pipeline involving several key intermediaries:

- **Advertisers** initiate campaigns and fund ad delivery.
- **Demand Side Platforms (DSPs)** represent advertisers and bid on their behalf.
- **Supply Side Platforms (SSPs)** represent publishers and auction their ad inventory.
- **Publishers** embed advertising code on their websites to generate revenue.

Traditionally, **SSPs** conducted sequential auctions, favoring a single provider (e.g., Google’s AdX). However, this approach lacked transparency and limited price discovery. To address this, the industry has shifted toward *header bidding*. In this mechanism, the publisher’s website executes a client-side script (often through Prebid.js) that simultaneously requests bids from multiple **DSPs** before passing the highest bid to the ad server [119, 120]. This enables fairer auctions and higher publisher revenue.

7.1.2 Impression Discrepancy and Data Fragmentation

Despite the architectural improvements of header bidding, impression tracking remains fragmented. Both **DSPs** and **SSPs** independently log impressions using scripts, cookies, or server-side events. Browser caching, ad blockers, slow page loads, and network latency can result in one party counting an impression while the other does not [121, 122].

Figure 7.1 illustrates this process. Advertisers submit campaigns to DSPs, which bid for inventory when a page loads. The Prebid script on the website is the entity responsible for the auctions. The SSPs respond to bids (e.g., 5 CPM, 4 CPM), and the highest is selected. The publisher then displays the winning ad. Each party collects ad data using its invisible pixels and logs the impression count independently. After a predefined time, the DSP makes a payment to the SSP based on the impression count. This is where the ad impression discrepancy becomes problematic.

Discrepancy rates commonly range between 5% and 20%, depending on device, browser, or traffic source [118]. These mismatches lead to billing disputes, lost revenue, and reduced trust across the ecosystem. Moreover, fraud and misreporting can go undetected due to the lack of a shared verifiable log.

7.1.3 Blockchain’s Attempts to Save Ad Supply Chain

Several blockchain-based advertising platforms attempt to resolve these problems through decentralized ledgers and transparent reporting. Notable efforts include AdEx [123], a decentralized ad exchange focused on privacy and fraud prevention; AdHash, a protocol that removes intermediaries and enables direct ad contracts; Alkimi [98], a decentralized exchange designed to replace legacy ad platforms; and Adwatch [99], a blockchain system for advertisers to track campaign rules and delivery.

While these platforms offer varying degrees of transparency, they often require a complete overhaul of the existing infrastructure, replacing SSPs and DSPs, or moving all data on-chain, which can violate privacy norms and introduce scalability bottlenecks.

7.1.4 The Case for a Lightweight Oracle Layer

What the ecosystem lacks is a verifiable, incentive-aligned protocol that can be layered on top of current systems without disrupting existing integrations. Specifically, a decentralized oracle system capable of validating impressions in near real-time, without storing user data or requiring advertisers to leave their DSPs, would offer a practical middle ground.

CountChain provides the theoretical framework for this capability. In the next section, we present AdChain, a practical instantiation of CountChain for resolving impression discrepancies. AdChain introduces an oracle layer that can interface with Prebid.js, DSPs, and publishers websites to cryptographically verify impression events without compromising existing programmatic workflows.

7.2 AdChain: A Decentralized Header Bidding Solution for Advertising Discrepancy Using CountChain

Building on the general-purpose architecture of CountChain, AdChain adapts its decentralized oracle workflow to the specific needs of the online advertising ecosystem. Designed to

Term	Definition	Set by
owner	the party interested in obtaining the result of the counting system	network
host	platform storing all information	owner / network
user_ID	unique identifier of the website visitor	invisible pixels
ad_ID	unique identifier of an ad	invisible pixels
player	participant nodes that intend to validate ad data	network / nodes
	~ ID : the unique identifier of each player	host
	~ point : points accumulated for each vote or proposition submission	host
proposition	a Boolean request containing ad data to be validated	submitter
	~ countdown : minimum number of propositions a player must verify to end the game	owner
	~ deadline : the deadline to vote for a proposition	owner / host
	~ pool : the list of available propositions to be verified	host
	~ price : the stake used when players vote for propositions	owner
submitter	the player who submits a proposition to be verified	players / host
verifier	randomly chosen player to verify a proposition	host
prize pool	financial reward and incentive	owner / players

Table 7.1: The terminology used in the AdChain protocol [6].

address impression discrepancy, AdChain offers a verifiable, incentive-aligned system that integrates with modern ad pipelines, particularly those using header bidding, to validate ad delivery events without overhauling existing infrastructure [6].

7.2.1 Overview and Objectives

AdChain’s primary goal is to establish a neutral, cryptographically verifiable layer that monitors and resolves discrepancies in impressions reported by advertisers and publishers. Unlike previous blockchain advertising solutions that attempt to replace DSPs, SSPs, or ad exchanges, AdChain complements the current stack by introducing minimal instrumentation via Prebid.js or similar header bidding wrappers [119].

It mainly has four core functions. The first function is validating whether a particular impression occurred based on cryptographic input evidence. Then, resolving discrepancies using decentralized verification rather than bilateral log reconciliation. Next, incentivizing honest participation from a decentralized pool of verifiers using a stake and reward model similar to CountChain [5] and finally, enabling verifiable audits without exposing sensitive user-level data.

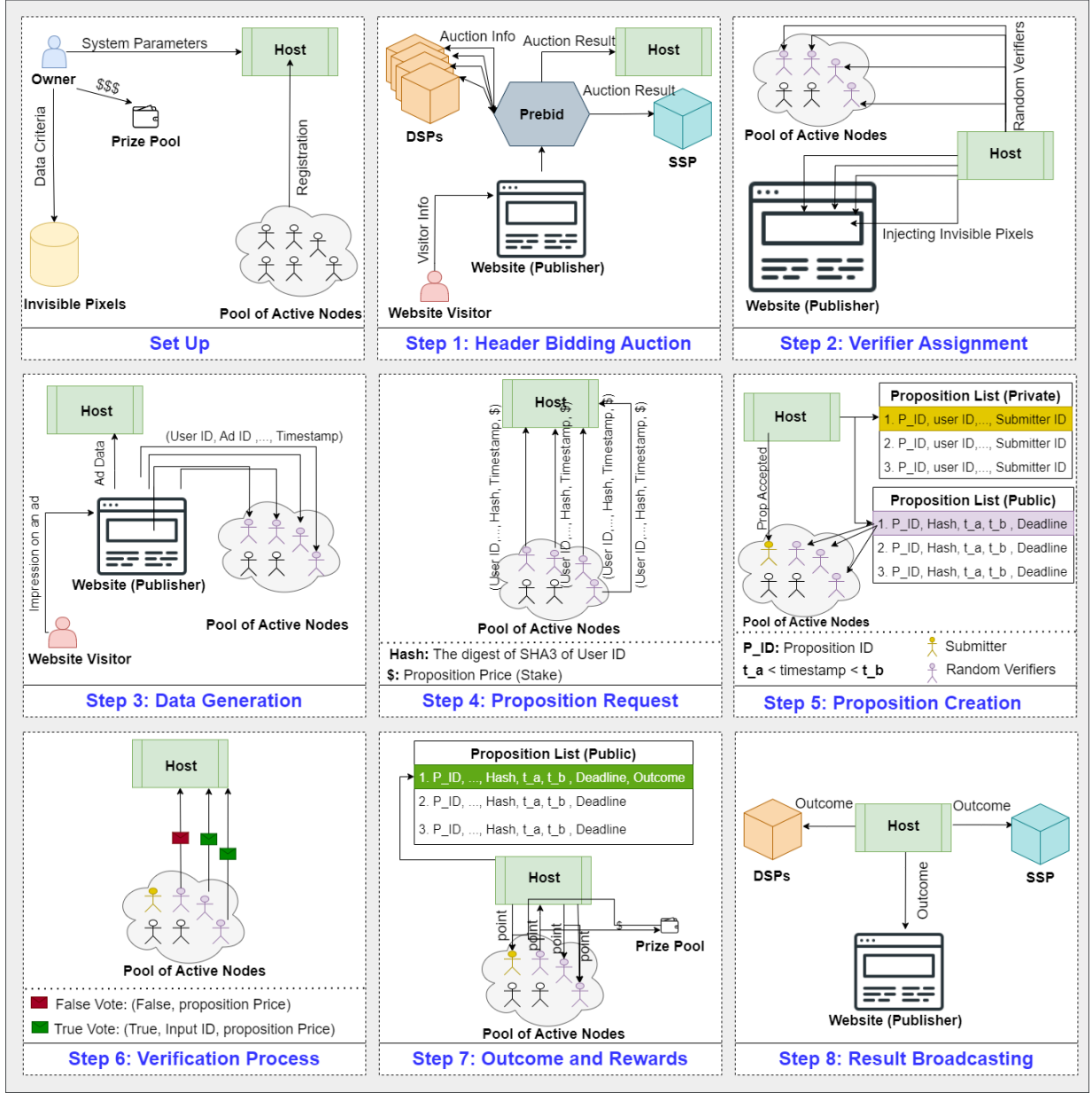


Figure 7.2: AdChain overview: adapting CountChain to verify ad impressions in a header bidding environment [6].

7.2.2 Mapping CountChain Concepts to AdChain Context

A detailed explanation of the terminology is shown in Table 7.1, and the summary of the roles adopted from the CountChain concept is as follows:

- **Input ID:** Unique impression ID generated per ad load [6].
- **Submitter:** The node that observes the impression and raises a proposition request.

Algorithm 3 Pseudocode for the Host

```
1: propositions  $\leftarrow []$ 
2: function HEADERBIDDINGAUCTION
3:   endPoints  $\leftarrow$  PickRandomVerifiers()
4:   NotifyVerifiers(endPoints)
5:   InjectInvisiblePixels(endPoints)
6: end function
7: function PROPSUBMISSION(req)
8:   if InvalidHash(req) then
9:     return WrongHash(req)
10:  end if
11:  if !PropositionExists(req) then
12:     $T_a, T_b \leftarrow$  GetTS(req[TS])
13:    RemoveID(req)
14:    prop  $\leftarrow$  makeProp(req,  $T_a, T_b$ )
15:    for verifier in endPoints do
16:      SendProp(verifier, prop)
17:    end for
18:  end if
19: end function
20: function VOTESUBMISSION(voteReq)
21:   if !PassedDeadline(voteReq) then
22:     if voteReq[vote] &
      !Hash(voteReq) then
23:       return WrongHash(voteReq)
24:     end if
25:     AddVoteToProp(voteReq[vote])
26:   end if
27: end function
```

Algorithm 4 Pseudocode for Verifiers

```
1: storedData  $\leftarrow []$ 
   // Triggers by Host or Data Source
2: while Active do
3:   if data is received then
4:     if data from InvisiblePixel &
      Valid then
5:       storedData.add(data)
6:       hash  $\leftarrow$  GetHash(data[ID])
7:       SendPropReq(data, hash, TS)
8:     end if
9:     if data from Host then
10:      vote, UserID  $\leftarrow$  False, NULL
11:      for entry in storedData do
12:        if data[ $T_a$ ]  $\leq$  entry[time]  $\leq$ 
          data[ $T_b$ ] then
13:          if GetHash(entry[ID])
            = data[hash] & entry[AdID] = data[Ad]
            then
14:            vote  $\leftarrow$  True
15:            UserID  $\leftarrow$  entry[ID]
16:            break
17:          end if
18:        end if
19:      end for
20:      SendVote(vote, UserID)
21:    end if
22:  end if
23: end while
```

Figure 7.3: Pseudocode for the Host and Verifier components in the extended CountChain protocol. The Host manages verifier selection, proposition creation, and vote collection. Verifiers validate pixel-based input and vote based on hashed proofs and time-bounded constraints.

- **Verifiers:** Decentralized nodes that check impressions records (e.g., logs or pixel traces).
- **Proposition:** “This impression occurred” , to be voted on by verifiers.
- **Stake:** A small value staked by both the submitter and verifiers to discourage manipulation.

7.2.3 System Architecture

The architecture of AdChain closely mirrors CountChain but maps each concept to advertising-specific roles and events. Figure 7.2 provides a high-level overview of the process.

AdChain’s operation follows the same five-step structure as CountChain [5] but is applied to impression validation:

Setup: A host contract is deployed with parameters set by the owner (e.g., stake amount, verifier count).

Verifier Selection: When the header bidding auction is in process, the host picks some nodes at random and injects some invisible pixels into each ad. The endpoint of these invisible pixels becomes the selected nodes.

Impression Observation: When a user makes an impression (click or view), invisible pixels capture an impression event (Input ID) and send a commitment to their endpoints.

Proposition Request: The nodes that receive the data can send a request to the host, notifying it of a potential impression.

Proposition Creation: The host validates the request and if correct, broadcasts it to the assigned verifiers.

Verification: Verifiers independently check their observation logs to determine if the impression occurred in the specified time window.

Outcome and Reward: A majority vote determines the outcome. Submitters and verifiers in the majority are rewarded; dissenting votes are penalized. The updated algorithms for the host and the verifiers are shown in Figure 7.3

This mechanism allows both advertisers and publishers to agree on verified impression events without requiring data sharing, manual log reconciliation, or reliance on trusted intermediaries [99].

7.2.4 Summary and Integration with Existing Infrastructure

Unlike solutions such as AdHash [97], Alkimi [98], or AdEx [123], AdChain does not aim to remove SSPs or alter exchange protocols. Instead, it offers a lightweight, opt-in accountability mechanism with low integration overhead.

Therefore, a significant advantage of AdChain is its compatibility with the existing ad stack. AdChain can integrate with Prebid’s event hooks to automatically trigger propositions upon impression events [119]. Also, submitters and verifiers may include SSPs, publishers, advertisers, DSPs or specialized nodes with access to logs, cookies, or impressions pixels. Moreover, AdChain can operate as an optional validation layer, used only for campaigns or impressions where transparency is critical [6].

AdChain demonstrates how a decentralized oracle framework like CountChain can be adapted to real-world domains with stringent transparency needs. It provides verifiable ad delivery without disrupting the current advertising ecosystem. It offers a path toward

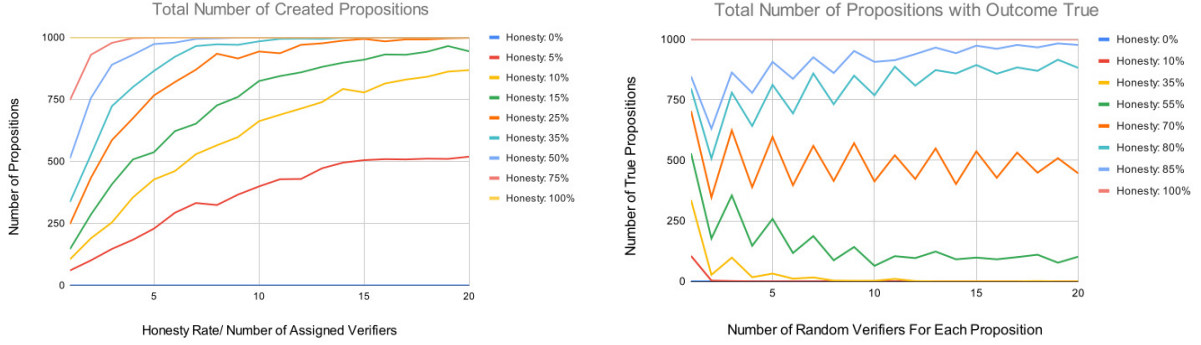


Figure 7.4: Impact of honesty rate and verifier count on proposition accuracy of AdChain[6].

resolving one of the most long-standing issues in digital marketing: the lack of reliable impression consensus [121, 122].

7.3 Experimental Evaluation of AdChain

To assess the feasibility of AdChain and understand how its core parameters influence performance and accuracy, we conducted a series of experiments on an Azure Standard E96ds v5 virtual machine equipped with 96 VCPUs and 672 GiB of RAM, running Ubuntu 20.04. The environment was configured with up-to-date versions of Node.js, Docker, and PostgreSQL (as of July 2023).

7.3.1 Impact of Honesty Rate and Verifier Count

The first experiment investigates how the honesty rate of participating nodes and the number of verifiers per advertisement affect the accuracy of AdChain in a way similar to the ones we accomplished for CountChain. Here, the honesty rate represents the likelihood that a node will vote correctly on a given proposition. This rate was varied from 0% to 100% in increments of 5 percentage points, while the number of verifiers per proposition was adjusted from 1 to 20.

For each setting, the system processed 1,000 simulated ad records, amounting to 420,000 total entries at a throughput of 10 ads per second. Verifiers were randomly selected from a pool of 100 nodes. The ground truth assumed that all propositions should be validated as *True*. The configuration parameters included an initial player balance of 10,000, a proposition price of 1, a proposition deadline of 2 seconds, a delay of 1 second, and a threshold point of -5.

Figure 7.4 illustrates that as long as nodes exhibit an honesty rate of at least 35% and 11 or more verifiers are assigned to each ad, all propositions are reliably submitted. This occurs because, even with modest honesty, at least one of the verifiers is likely to

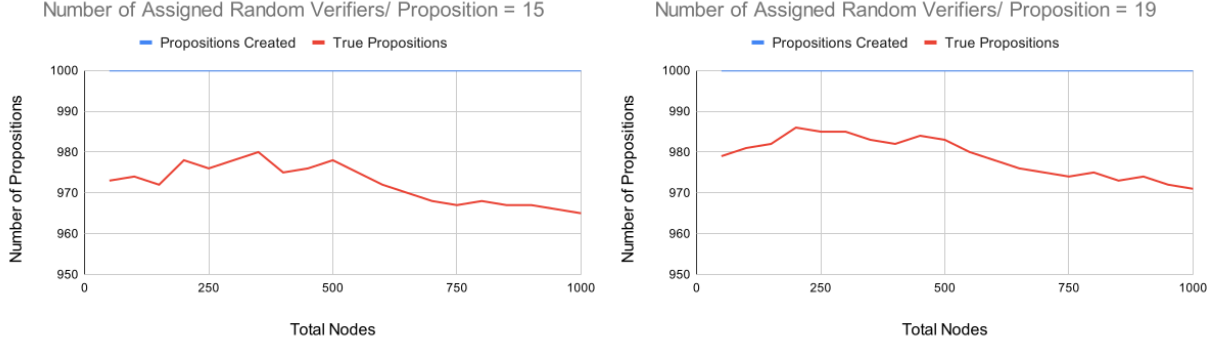


Figure 7.5: Optimal node count for maximizing the accuracy of AdChain[6].

act correctly and trigger a valid submission. This highlights the robustness of AdChain’s decentralized mechanism in capturing and validating ad events.

The results also confirm that maximum accuracy is achieved when all nodes are fully honest (100%), and minimum accuracy occurs when all are dishonest (0%). Notably, the accuracy drops sharply between 85% and 80% honesty, reflecting the increasing likelihood that a dishonest majority may skew proposition outcomes. With a node honesty rate of 85%, accuracy peaks at 98%, especially when using 15 or 19 verifiers. Therefore, 15 verifiers per ad is recommended to strike a balance between accuracy and cost since each verifier adds resource overhead due to an additional invisible pixel.

These findings confirm that AdChain can dramatically reduce discrepancy rates in digital advertising from as high as 20% to just 2%, validating both the practicality and precision of the design.

7.3.2 Optimal Network Size

This experiment evaluates the impact of total network size on system performance. The number of nodes varied from 50 to 1,000 in increments of 50, with 15 or 19 verifiers selected per proposition, each possessing an honesty rate of 85%. Other configuration parameters mirrored those in the previous experiment.

As shown in Figure 7.5, the number of nodes in the network has only a marginal effect on overall accuracy. This is because data in AdChain is routed to a small, randomly selected subset of nodes, and only they can initiate proposition requests. Hence, scaling the total number of nodes does not significantly increase computation or storage demands.

While a minor performance drop occurs beyond 550 nodes, this is likely attributed to hardware limitations in the experimental setup. In real-world distributed deployments where each node operates independently, this degradation would likely disappear.

Moreover, using 15 instead of 19 verifiers leads to a negligible 0.5% drop in accuracy but reduces system costs. Based on this trade-off, we recommend maintaining at least 200

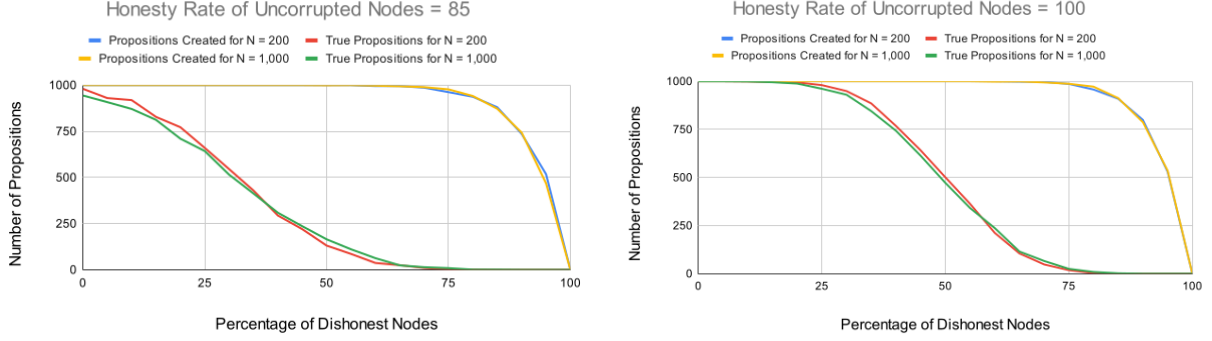


Figure 7.6: The success rate of Sybil attacks against AdChain. UNHR = Uncorrupted Node Honesty Rate [6]

active nodes and using 15 verifiers per proposition to achieve an optimal balance between scalability, accuracy, and resource efficiency.

On average, every additional 50 nodes results in just 0.4–0.8% higher CPU and RAM usage, with no significant disk I/O impact. The system exhibited a low average latency of 89 ms (excluding network delays), suggesting that AdChain can scale effectively to meet real-time demands in online advertising environments.

7.3.3 Sybil Attack Resilience

To assess AdChain’s resilience to Sybil attacks, we simulated scenarios where a fraction of the nodes were malicious. In a classical Sybil attack, a single entity controls multiple fake identities and attempts to undermine consensus by either blocking proposition creation or skewing proposition outcomes.

In this experiment, 15 verifiers were selected from a pool of 1,000 nodes. The proportion of corrupted nodes was increased from 0% to 100%, and their honesty rate was set to 0%. Two settings were tested for the honesty rate of the remaining (uncorrupted) nodes: 85% and 100%. The experiment was repeated for networks with 200 and 1,000 total nodes.

Figure 7.6 shows that AdChain maintains high accuracy even with significant network corruption. When honest nodes have a 100% honesty rate, a successful Sybil attack (i.e., preventing or falsifying propositions) only occurs when adversaries control at least 80% of the network. When the honesty rate drops to 85%, the threshold decreases to around 70%.

These results show significant improvement over traditional systems, where control of just 51% of the network often suffices. Additionally, since increasing the number of nodes in AdChain has minimal effect on system overhead, adding nodes is an effective strategy to increase attack costs.

7.3.4 Comparative Experimental Analysis of CountChain and AdChain

Both *CountChain* and *AdChain* underwent extensive experimental evaluations to assess their accuracy, scalability, and Sybil attack resistance under various network and adversarial conditions. While CountChain was designed as a general-purpose decentralized [Oracle](#) network ([DON](#)) for verifiable counting tasks, AdChain adapts its architecture specifically to the online advertising domain, with a focus on resolving ad impression discrepancies.

Accuracy. CountChain achieved up to **99% precision** when the [verifier](#) honesty rate exceeded 85% and 14–20 verifiers were selected per [proposition](#) [5]. In comparison, AdChain reached **98% accuracy** under a similar honesty rate using 15 verifiers [6]. While the difference is minor, AdChain’s performance highlights its robustness in environments characterized by fragmented data, inconsistent user behavior, and higher noise factors typical in digital advertising.

Verifier Optimization. CountChain experiments showed that 14 verifiers balanced accuracy and efficiency optimally. AdChain exhibited similar behavior, with 15 verifiers offering the best trade-off between gas cost and validation coverage [6].

Sybil Resistance. Both systems incorporate randomized verifier selection to mitigate Sybil attacks. CountChain resists attacks unless the adversary controls over 70% of the network. AdChain demonstrated even stronger resilience, requiring at least **80% network compromise** for consistent manipulation, thanks to its stricter majority enforcement and incentive alignment [6, 5].

Scalability and Cost. CountChain maintained low latency (approximately 116 [ms](#) and low memory usage, making it suitable for general decentralized counting. AdChain sustained similar throughput and maintained an average latency of **89 ms**, aided by selective verifier targeting and the use of lightweight on-chain interactions [6].

In summary, *CountChain* provides a flexible and secure framework for verifiable off-chain event validation, while *AdChain* extends this framework to a real-world use case with enhanced adversarial resistance and integration with legacy infrastructure. This comparative evaluation validates the extensibility and practical applicability of the CountChain design across domains.

7.4 Economic Model of AdChain

This section presents a cost-revenue analysis of AdChain to assess the economic feasibility of running a node in the network. The model is grounded in empirical bandwidth, computation, and storage considerations alongside realistic market and reward allocation assumptions [6].

7.4.1 Assumptions

The analysis is based on the following assumptions:

1. Each invisible pixel transmits approximately 1 [KB](#) of data per ad transaction to every assigned [verifier](#) ($n = 1KB$).
2. Due to the lightweight nature of hash-based verification, the primary performance constraint is the [verifiers' bandwidth](#) rather than computational capacity.
3. [Verifiers](#) are assumed to dedicate at least 10% of their internet [bandwidth](#) to transmitting and receiving ad-related data.
4. Each advertisement is independently verified by $V = 15$ randomly selected [verifiers](#), as identified in our experiments as the optimal number.
5. Publishers are expected to allocate 1% of their annual ad revenue to the [verifier](#) incentive pool.

7.4.2 Revenue Estimation

According to Speedtest Global Index data, the average global download speed (across fixed broadband and cellular connections) is approximately 70 [Mbps](#). Allocating 10% of this [bandwidth](#) allows a [verifier](#) to process roughly 7,000 ad transactions per second, given the [KB](#) data size per transaction.

With an estimated 300 billion impressions occurring globally each day, equivalent to 3.5 million transactions per second, the required number of [verifiers](#) (N) to process all transactions can be estimated as:

$$N = \frac{3.5 \cdot 10^6 \cdot V}{7 \cdot 10^3} = 500V$$

Substituting $V = 15$, we find that at least 7,500 verifiers are needed. However, as discussed earlier, each verifier should be selected from a pool of at least 200 nodes to ensure system robustness. This results in a recommended minimum of 100,000 active nodes to support global-scale deployment.

[header bidding](#) revenue exceeded \$414 billion in 2022 [[117](#), [124](#)]. If just 1% is allocated to verifier incentives, the annual prize pool totals \$4 billion. Distributing this evenly across 100,000 nodes yields an average revenue of \$40,000 per node annually. The remaining \$0.14 billion, along with any [stakes](#) forfeited by dishonest nodes, can be retained by the system host.

7.4.3 Cost Estimation

The average global cost for a stable internet connection capable of supporting a [verifier's](#) workload is approximately \$380/year. Additional costs arise from computation and storage requirements:

- **Computation:** A [verifier](#) processes 7 [Mbps](#) of data (7,000 transactions/sec at [KB](#) each). Most modern machines, such as Azure’s B16ms VM, can compute [SHA3](#) hashes at this throughput and cost roughly \$4,000/year.
- **Storage:** At the stated transaction rate, each [verifier](#) handles around 18 [TB](#) of monthly data. As each ad transaction remains relevant for approximately one month, [verifiers](#) only need to store one month’s worth of data at any time. Azure’s P70 tier offers 18 [TB](#)/year of storage at around \$18.7K/year.

7.4.4 Profitability Analysis

Combining the costs yields an estimated annual expenditure of approximately \$23,000 per node. With an average income of \$40,000, each node can expect a net profit of \$17,000/year.

For comparison, Bitcoin miners earn an average of \$21,000/year from mining 0.33 [BTC](#) annually (given the current price of \$63,000) but incur roughly \$9,000 in costs, leading to a net profit of about \$12,000/year [[125](#), [126](#), [127](#), [128](#)]. Unlike Bitcoin mining, which requires specialized hardware and high energy consumption, running a node on AdChain is far more accessible and environmentally sustainable. Additionally, while Bitcoin’s finite supply will cap its growth by 2028, the demand for online advertising continues to grow [[116](#)], making AdChain a more sustainable and scalable opportunity.

7.5 Discussion, Limitations, and Future Work

While both CountChain and AdChain designs demonstrate the viability of decentralized [Oracles](#) for verifiable counting and impression validation, several practical and theoretical limitations remain.

First, both protocols assume a sufficiently large and honest pool of participants to ensure the correctness of majority votes. In adversarial settings where the majority of [verifiers](#) are colluding or inactive, the outcome may be compromised. Although [stake-weighted](#) voting and slashing reduce incentives for dishonest behavior, the system may still be vulnerable to targeted [Sybils](#) attacks in low-liquidity deployments. Future work could explore adaptive verifier selection or dynamic quorum thresholds to mitigate these risks under shifting verifier populations.

Next, the protocols rely on verifiers having timely and accurate access to external data (e.g., pixel-based impression records and off-chain logs). In practice, variability in data collection (e.g., due to ad blockers or network latency) may lead to honest but inconsistent votes. This is particularly salient in AdChain, where real-time web dynamics can affect observation quality. Future improvements could incorporate redundancy through multiple data sources or employ lightweight off-chain coordination to reconcile noise.

Moreover, although AdChain integrates with existing infrastructure like Prebid.js, adoption depends on the willingness of publishers and advertisers to stake funds and expose

campaign metadata for verification. Further economic modeling is needed to calibrate the incentive structure for long-term sustainability, especially in low-margin ad ecosystems.

Additionally, while CountChain was evaluated for generic counting and AdChain for online advertising, the underlying framework is applicable to other event-heavy domains such as sensor networks, vote auditing, or content moderation. Each domain may require slight adjustments to trust models, data formats, or timing assumptions. A promising direction for future work is to create a library of modular CountChain "adapters" that encode domain-specific logic while reusing the same [Oracle](#) core.

In summary, CountChain and AdChain represent a first step toward domain-aware, incentive-compatible oracle systems. Future research can build on their foundations to create even more robust, privacy-preserving, and widely deployable decentralized verification networks.

Chapter 8

Conclusion and Future Work

This thesis aimed to address the challenges of secure execution of smart contracts and enabling scalable, trustworthy connectivity with external data sources. Though often treated independently, these challenges intersect at the core of blockchain’s promise to support decentralized, verifiable computation in adversarial environments. By bridging security and connectivity through techniques drawn from program analysis, [LLMs](#), and game theory, this work contributes toward the vision of blockchain systems that are not only self-correcting but also structurally reliable by design.

To achieve this, this thesis presents a cohesive set of tools, frameworks, and theoretical models that collectively advance the security and scalability of blockchain infrastructures. **GasGauge**, **GasGuard**, and **GasGaugeAI** introduced a hybrid static-dynamic analysis tool capable of identifying gas-intensive loops, an automatic [LLM](#)-based mitigator that inserts ‘**require**’ guards in vulnerable code regions, and a multi-[LLM](#) framework for state-aware gas vulnerability mitigation, respectively.

Additionally, **CountChain** and **AdChain** tackled the problem of external data reliability through a game-theoretic [DON](#). CountChain ensures verifiable aggregation without centralized authorities by incentivizing honest participation in counting-based protocols. AdChain shows how introducing a decentralized header bidding mechanism can reduce discrepancy fraud.

Together, these contributions form a vertically integrated body of work that addresses blockchain’s foundational gaps, from bytecode-level execution to system-wide data trust.

8.1 Summary of the Insights

Several key insights emerged from the integration of seemingly disparate domains, automated program repair, [LLMs](#), and decentralized system design. First, smart contract vulnerabilities often stem from implicit design assumptions, such as loop safety or bounded gas costs. Symbolic analysis paired with neural synthesis can formalize and systematically mitigate these. Next, [LLMs](#) are effective for code generation and reasoning agents. [LLMs](#)

can conditionally fix contract logic, identify vulnerable patterns, and participate in validation cycles when paired with structured feedback from tools like Foundry.

Moreover, security and scalability are not isolated problems. Mitigating gas-based DoS vulnerabilities improves the availability and composability of smart contracts, which in turn is necessary for scalable, interconnected dApps. Meanwhile, decentralized oracle systems require more than just cryptographic guarantees. They also need game-theoretic alignment. Incentive compatibility, reputation management, and voting mechanisms are critical for building trust in open networks. Therefore, a full-stack approach to blockchain security is necessary. From EVM-level gas metering to application-layer fraud in advertising, vulnerabilities can cascade. Only by analyzing the system holistically can we prevent subtle yet devastating attacks.

8.2 Limitations

While this thesis’s contributions are extensive, several limitations merit discussion. The three gas-based tools focus primarily on gas-related denial-of-service vulnerabilities in loop structures. While the methods are extensible, they do not currently generalize to all classes of smart contract vulnerabilities, such as reentrancy or logical flaws in access control. Also, the success of GasGuard and GasGaugeAI relies on LLMs, which can be non-deterministic and challenging to interpret. While prompt engineering and fine-tuning improve performance, ensuring consistent behavior across edge cases remains a challenge.

Additionally, CountChain and AdChain assume rational participation and may require reputation systems or staking mechanisms to guard against collusion or coordinated attacks. While the protocols are incentive-compatible under normal conditions, adversarial modeling beyond Nash equilibria remains open.

Finally, the tools and protocols developed here have been evaluated on real-world contracts and large-scale simulations. However, additional longitudinal studies in production environments would further validate their resilience and practicality.

8.3 Future Work

This thesis opens several promising directions for future research. First, the GasGauge series can be extended to address broader vulnerabilities, including access control misconfigurations, unsafe storage patterns, and reentrancy. Future tools could provide contract-wide audits and multi-function repair suggestions by integrating symbolic condition generation with specification mining. Also, reinforcement learning or fine-grained token-level scoring can enhance LLM feedback loops to reduce failure cases.

Additionally, beyond static repair, there is an opportunity to explore runtime security where lightweight on-chain monitors enforce invariants or catch violations before execution completes. Combining AI-generated patches with on-chain verifiers could yield hybrid

safety mechanisms protecting code and state. Also, there is immense potential in building [AI](#) copilots for smart contract developers and auditors. Such systems could incorporate GasGuard-like modules into IDEs, enabling real-time fix suggestions or even test-case generation. Using real audit data and community-based fine-tuning could make these systems more context-aware.

Moreover, for oracle-based systems like CountChain and AdChain, further exploration into stochastic security models or delayed verification can enhance robustness. Integrating economic defense mechanisms, such as slashing, fallback quorum layers, or dynamic reputation, could help protect against rare but catastrophic coordinated attacks.

Finally, given the convergence of [DeFi](#), advertising, gaming, and [AI](#) within blockchain ecosystems, there is a growing need for cross-domain formal verification. Future work could explore modular verification methods that reason about interactions across contracts, layers, or economic domains, where vulnerabilities can emerge from the interplay between systems rather than isolated code paths.

8.4 Closing Remarks

From vulnerability detection to decentralized connectivity, this thesis has presented a unified framework for making blockchain systems more secure, self-healing, and scalable. It shows that combining formal methods, game theory, and artificial intelligence is feasible and necessary for the next generation of blockchain infrastructure. As blockchains evolve to support increasingly complex and critical applications, security must evolve from reactive to proactive, from human-driven to AI-augmented. This work contributes toward that vision and offers foundational approaches that others may build upon in the years ahead.

References

- [1] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts,” in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.
- [2] B. Nassirzadeh, A. Hasan, and V. Ganesh, “Gasguard: An LLM-based automated gas vulnerability detection and mitigation system,” in *6th International Conference on Mathematical Research for Blockchain Economy, MARBLE 2025, Athens, Greece, July 1–3, 2025*, ser. Lecture Notes in Operations Research. Springer, 2025.
- [3] —, “GasGaugeAI: A multi-LLM APR framework for state-aware gas vulnerability in ethereum,” 2025.
- [4] B. Nassirzadeh, H. Sun, S. Banescu, and V. Ganesh, “Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities,” in *3rd International Conference on Mathematical Research for Blockchain Economy, MARBLE 2022, Vilamoura, Portugal, July 12–14, 2022*, ser. Lecture Notes in Operations Research, P. M. Pardalos, I. S. Kotsireas, Y. Guo, and W. J. Knottenbelt, Eds. Springer, 2022, pp. 143–167. [Online]. Available: https://doi.org/10.1007/978-3-031-18679-0_9
- [5] B. Nassirzadeh, A. Heinle, S. Leonardos, M. A. Hasan, and V. Ganesh, “Countchain: A decentralized oracle network for counting systems,” in *IEEE International Conference on Blockchain (Blockchain 2024), Copenhagen, Denmark, August 19–22, 2024*. IEEE, 2024, pp. 324–331. [Online]. Available: <https://doi.org/10.1109/Blockchain62396.2024.00049>
- [6] —, “Adchain: Decentralized header bidding,” in *Mathematical Research for Blockchain Economy: 5th International Conference, MARBLE 2024, Malaga, Spain, July 9–11, 2024*, ser. Lecture Notes in Operations Research, S. Leonardos, E. Alfieri, W. J. Knottenbelt, and P. M. Pardalos, Eds. Springer, 2024, pp. 265–283. [Online]. Available: https://doi.org/10.1007/978-3-031-68974-1_13
- [7] D. Siegel, “Understanding The DAO attack,” *CoinDesk*, jun 2016. [Online]. Available: <https://www.coindesk.com/learn/understanding-the-dao-attack>
- [8] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” <https://ethereum.org/en/whitepaper/>, 2014.

- [9] S. Ellis, A. Juels, and S. Nazarov, “ChainLink: A decentralized oracle network,” <https://chain.link/whitepaper/>, 2017, version 1.0, September 4.
- [10] E. Albert, S. Gordillo, B. Livshits, A. Rubio, and I. Sergey, “Gastap: Gas analysis tool for smart contracts,” *arXiv preprint*, vol. arXiv:1802.05932, 2018. [Online]. Available: <https://arxiv.org/abs/1802.05932>
- [11] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 653–663.
- [12] N. Grech, V. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, “Madmax: Surviving out-of-gas conditions in ethereum smart contracts,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.
- [13] F. Zhang, P. Daian, A. Juels, and E. Shi, “Towards a theory of decentralized oracles,” *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.
- [14] J. Xu, L. Kong, H. Li, Y. Wang, L. Gudgeon, Y. Zhou, Z. Liu, F. Zhang, and Z. Zhao, “An empirical study of defi liquidations: Incentives, risks, and instabilities,” *arXiv preprint*, vol. arXiv:2109.13598, 2021. [Online]. Available: <https://arxiv.org/abs/2109.13598>
- [15] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” <https://bitcoin.org/bitcoin.pdf>, 2008.
- [16] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, “Sok: Research perspectives and challenges for Bitcoin and cryptocurrencies,” *2015 IEEE Symposium on Security and Privacy*, pp. 104–121, 2015. [Online]. Available: <https://doi.org/10.1109/SP.2015.14>
- [17] V. Buterin, “On public and private blockchains,” <https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/>, 2015.
- [18] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, “An overview of blockchain technology: Architecture, consensus, and future trends,” *Proceedings of the IEEE*, vol. 104, no. 11, pp. 2047–2066, 2017. [Online]. Available: <https://doi.org/10.1109/JPROC.2017.2762790>
- [19] Ethereum Foundation, “The Merge,” <https://ethereum.org/en/upgrades/merge/>, 2022.
- [20] D. Larimer, “Delegated proof-of-stake (DPoS),” <https://bitshares.github.io/dpos.pdf>, 2014.
- [21] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, 1999, pp. 173–186.

- [22] Ethereum Wiki Contributors, “Proof-of-authority (PoA),” <https://ethereum.org/en/developers/docs/consensus-mechanisms/poa/>, 2024, last updated July 3, 2024; accessed June 23, 2025.
- [23] X. Xu, X. Chen, G. Kou, S. Yu, and J. Wu, “A taxonomy of blockchain-based applications for industrial internet of things,” *IEEE Transactions on Industrial Informatics*, vol. 17, no. 11, pp. 7489–7500, 2021. [Online]. Available: <https://doi.org/10.1109/TII.2020.3034186>
- [24] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, W. Knottenbelt, and B. Livshits, “SoK: Decentralized finance (DeFi),” *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.
- [25] I. Eyal and E. G. Sirer, “Majority is not enough: Bitcoin mining is vulnerable,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2014, pp. 436–454.
- [26] P. Daian, S. Goldfeder, T. Kell, I. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, “Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability,” in *IEEE Symposium on Security and Privacy*, 2020.
- [27] S. Azouvi, M. Maller, and S. Meiklejohn, “A survey of attacks on ethereum smart contracts (SoK),” *arXiv preprint*, vol. arXiv:1908.08558, 2019. [Online]. Available: <https://arxiv.org/abs/1908.08558>
- [28] Y. Liu, Y. Zhang, Q. Liu, T. Chen, K. Li, and C. Wang, “Security analysis of cross-chain protocols: A survey,” *arXiv preprint*, vol. arXiv:2206.11156, 2022. [Online]. Available: <https://arxiv.org/abs/2206.11156>
- [29] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269, 2016. [Online]. Available: <https://doi.org/10.1145/2976749.2978316>
- [30] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (SoK),” *Proceedings of the International Conference on Principles of Security and Trust (POST)*, pp. 164–186, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-57620-5_8
- [31] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “Smartcheck: Static analysis of ethereum smart contracts,” in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. Gothenburg: IEEE, 2018, pp. 9–16. [Online]. Available: <https://doi.org/10.1145/3194113.3194115>
- [32] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, “Securify v2.0,” <https://github.com/eth-sri/securify2>, 2021, accessed January 29, 2021.

- [33] C. Torres, B. Steffen, and G. Vidal, “Osiris: Hunting for integer bugs in ethereum smart contracts,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 283–294.
- [34] C. Diligence, “Mythril: Security analysis tool for ethereum smart contracts,” <https://github.com/ConsenSys/mythril>, 2018.
- [35] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “Zeus: Analyzing safety of smart contracts,” *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*, pp. 1–12, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf
- [36] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. California: IEEE, 2019, pp. 1186–1189. [Online]. Available: <https://doi.org/10.1109/ASE.2019.00133>
- [37] M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and S. P. Jones, “Symbolic haskell: First-class symbolic values for verifying haskell programs,” in *Proceedings of the 2008 ACM SIGPLAN Workshop on Haskell*. ACM, 2008, pp. 1–12.
- [38] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, “Echidna: Effective, usable, and fast fuzzing for smart contracts,” *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020. [Online]. Available: <https://doi.org/10.1145/3395363.3404366>
- [39] Paradigm, “Foundry book,” <https://book.getfoundry.sh/>, 2023, accessed May 2025.
- [40] B. Jiang, Y. Liu, and W. K. Chan, “Contractfuzzer: Fuzzing smart contracts for vulnerability detection,” *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018. [Online]. Available: <http://dx.doi.org/10.1145/3238147.3238177>
- [41] V. Wüstholtz and M. Christakis, “Harvey: A greybox fuzzer for smart contracts,” *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020. [Online]. Available: <https://doi.org/10.1145/3368089.3417064>
- [42] T. of Bits, “Medusa: Ethereum smart contract fuzzer,” <https://github.com/trailofbits/medusa>, 2023, accessed May 2025.
- [43] X. Zhang, D. Liu, T. Wang, X. Liu, Q. Zhang, H. Duan, and Z. Liang, “Txspector: Uncovering attacks in ethereum from transactions,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.
- [44] Y. Liu, Q. Li, L. Wu, Z. Zuo, and Z. Lin, “Contractward: Automated vulnerability detection models for ethereum smart contracts,” in *Proceedings of the 30th ACM*

- SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2023. [Online]. Available: <https://doi.org/10.1145/3576915.3623163>
- [45] B. Kamali, M. Valizadeh, S. A. Mirheidari, and A. Aziz, “Smarteval: Smart contract evaluation using large language models,” <https://arxiv.org/abs/2311.16474>, 2023, accessed April 30, 2025.
 - [46] X. Liu, Y. Zhang, J. Shen, and X. Zhang, “Smartinv: Invariant generation for smart contracts using large language models,” *CoRR*, vol. abs/2401.06954, 2024. [Online]. Available: <https://arxiv.org/abs/2401.06954>
 - [47] B. Gao, Q. Wei, Y. Liu, and R. S. M. Goh, “Unveiling the potential of chatgpt in detecting machine-unauditable bugs in smart contracts: A preliminary evaluation and categorization,” in *2024 IEEE Conference on Artificial Intelligence (CAI)*. IEEE, 2024, pp. 1481–1486.
 - [48] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, “Introduction to runtime verification,” in *Lectures on Runtime Verification*. Springer, 2018, pp. 1–33.
 - [49] I. Grishchenko, M. Maffei, and C. Schneidewind, “A semantic framework for the security analysis of ethereum smart contracts,” in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 243–257.
 - [50] N. Amin, A. Lal, S. Qadeer, and A. Shah, “Verisol: The power of interprocedural analysis for smart contract verification,” in *International Conference on Computer Aided Verification (CAV)*. Springer, 2021, pp. 246–270.
 - [51] J. Wu, Z. Li, Y. Liu, Z. Wu, and Q. Liu, “Defiranger: Detecting price oracle manipulation attacks in defi protocols,” in *USENIX Security Symposium*, 2023.
 - [52] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, and . . . , “Language models are few-shot learners,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020. [Online]. Available: <https://arxiv.org/abs/2005.14165>
 - [53] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” *arXiv preprint*, vol. arXiv:2107.03374, 2021. [Online]. Available: <https://arxiv.org/abs/2107.03374>

- [54] Y. Wang, S. Lu, Z. Tu, Z. Du, S. Xia, J. Zhang, and Y. Liu, “Codet5: Identifier-aware unified pre-trained encoder–decoder models for code understanding and generation,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 8696–8708.
- [55] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2018, pp. 653–663. [Online]. Available: <https://doi.org/10.1145/3274694.3274743>
- [56] M. Rodler, T. Frassetto, F. Lindner, and A.-R. Sadeghi, “Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts,” in *Proceedings of the 30th USENIX Security Symposium*. USENIX, 2021, pp. 2537–2554. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/rodler>
- [57] T. Chen *et al.*, “Solgpt: Automated smart contract repair using generative pretrained transformers,” 2024. [Online]. Available: <https://arxiv.org/abs/2403.12345>
- [58] T. D. Nguyen, L. H. Pham, and J. Sun, “sguard: Towards fixing vulnerable smart contracts automatically,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1215–1229, also released as arXiv:2101.01917.
- [59] C. F. Torres, H. Jonker, and R. State, “Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts,” in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2022, pp. 115–128. [Online]. Available: <https://dl.acm.org/doi/10.1145/3545948.3545975>
- [60] H. Zhao *et al.*, “ContractTinker: LLM-Guided context-aware smart contract repair,” 2024. [Online]. Available: <https://arxiv.org/abs/2409.09661>
- [61] A. Tsegaye *et al.*, “TRT: LLM-Driven smart contract repair using solidity vulnerability reports,” 2023.
- [62] X. Chen *et al.*, “Repaircat: Code-aware transformer for automated program repair,” arXiv preprint arXiv:2312.10035, 2023.
- [63] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, “Codegen2: Lessons for training LLMs on programming and natural languages,” *arXiv preprint*, vol. arXiv:2305.02309, 2023. [Online]. Available: <https://arxiv.org/abs/2305.02309>
- [64] J. Xu, L. Wang, W. Zhou, Y. Zhang, and K. R. Choo, “A survey on blockchain oracles: Progress, challenges, and opportunities,” *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1616–1653, 2021. [Online]. Available: <https://doi.org/10.1109/COMST.2020.3045609>
- [65] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, “Town crier: An authenticated data feed for smart contracts,” in *Proceedings of the 2016 ACM*

- SIGSAC Conference on Computer and Communications Security*, 2016, pp. 270–282. [Online]. Available: <https://doi.org/10.1145/2976749.2978361>
- [66] L. Breidenbach, C. Cachin, B. Chan, A. Coventry, S. Ellis, A. Juels, F. Koushanfar, A. Miller, B. Magauran, D. Moroz, S. Nazarov, A. Topliceanu, F. Tramèr, and F. Zhang, “Chainlink 2.0: Next steps in the evolution of decentralized oracle networks,” <https://research.chain.link/whitepaper-v2.pdf>, 2021, version 1.0, April 15.
 - [67] G. Cormode and S. Muthukrishnan, “Counting, summing, and averaging in data streams: Models and algorithms,” *Synthesis Lectures on Data Management*, vol. 4, no. 1, pp. 1–83, 2011.
 - [68] D. McCoy, K. Thomas, A. Vasudevan, and E. Bursztein, “The economics of digital ad fraud,” *Google Security Blog*, 2018, <https://security.googleblog.com/2018/12/the-economics-of-digital-ad-fraud.html>.
 - [69] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.
 - [70] J. F. Nash, “Non-cooperative games,” *Annals of Mathematics*, vol. 54, no. 2, pp. 286–295, 1951.
 - [71] W. Zhang, S. Banescu, L. Passos, S. Stewart, and V. Ganesh, “Mpro: Combining static and symbolic analysis for scalable testing of smart contracts,” in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. Berlin: IEEE, 2019, pp. 456–462. [Online]. Available: <https://doi.org/10.1109/ISSRE.2019.00052>
 - [72] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, “Securify: Practical security analysis of smart contracts,” *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 67–82, 2018. [Online]. Available: <https://dl.acm.org/doi/10.1145/3243734.3243780>
 - [73] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, “Verx: Safety verification of smart contracts,” in *2020 IEEE Symposium on Security and Privacy (SP)*. California: IEEE, 2020, pp. 1661–1677. [Online]. Available: <https://doi.org/10.1109/SP40000.2020.00024>
 - [74] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen, “Automatically finding patches using genetic programming,” *ICSE '09 Conference*, 2009.
 - [75] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 298–312, 2016.
 - [76] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” *ICSE '13 Conference*, 2013.

- [77] J. Bader and M. Pradel, “Getafix: Learning to fix bugs automatically,” ESEC/FSE ’19 Conference, 2019.
- [78] C. Yin *et al.*, “Chatrepair: Fixing software bugs via LLM-guided conversations,” <https://arxiv.org/abs/2401.08796>, 2024, arXiv preprint arXiv:2401.08796.
- [79] Y. Zhou *et al.*, “Codereviser: Revising source code for automated program repair,” ICSE ’24 Conference, 2024.
- [80] J. Yu *et al.*, “Screpair: Automated smart contract repair using genetic programming,” arXiv preprint arXiv:2005.07875, 2020.
- [81] X. He *et al.*, “Contractfix: Automatic repair of vulnerable smart contracts,” arXiv preprint arXiv:2307.08912, 2023.
- [82] G. Security, “A comprehensive understanding on smart contract automated repair,” <https://goplussecurity.medium.com/a-comprehensive-understanding-on-smart-contract-automated-repair-c97ac224f532>, 2023, accessed May 2025.
- [83] S. Zhang *et al.*, “Smartfix: Speeding up smart contract repair with statistical guidance,” ASE ’23 Conference, 2023.
- [84] J.-R. Giesen, S. Andreina, M. Rodler, G. O. Karame, and L. Davi, “Practical mitigation of smart contract bugs,” 2022. [Online]. Available: <https://arxiv.org/abs/2203.00364>
- [85] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, “Aroc: An automatic repair framework for on-chain smart contracts,” *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4611–4629, 2021. [Online]. Available: <https://doi.org/10.1109/TSE.2021.3072919>
- [86] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, “Smartshield: Automatic smart contract protection made easy,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 23–34. [Online]. Available: <https://doi.org/10.1109/SANER48275.2020.9054890>
- [87] P. Tolmach, Y. Li, and S. W. Lin, “Property-based automated repair of defi protocols,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.
- [88] C. Gao, W. Yang, J. Ye, Y. Xue, and J. Sun, “sguard+: Machine learning-guided rule-based automated vulnerability repair on smart contracts,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–55, jun 2024. [Online]. Available: <https://doi.org/10.1145/3641846>
- [89] D. Hong-Fei and E. Zhang, “Neo oracle service,” <https://docs.neo.org/docs/en-us/advanced/oracle.html>, 2014, last accessed: July 12, 2023.

- [90] N. Durov, “The open network,” <https://ton.org/whitepaper.pdf>, 2021, last accessed: April 26, 2022.
- [91] J. Peterson, J. Krug, M. Zoltu, A. K. Williams, and S. Alexander, “Augur: A decentralized oracle and prediction market platform (v2.0),” <http://www.augur.net/whitepaper.pdf>, 2022, last accessed: April 26, 2022.
- [92] Gnosis, “Gnosis whitepaper,” <https://www.allcryptowhitepapers.com/gnosis-whitepaper/>, 2017, last accessed: April 26, 2022.
- [93] J. Adler, R. Berryhill, A. Veneris, Z. Poulos, N. Veira, and A. Kastania, “Astraea: A decentralized blockchain oracle,” in *2018 IEEE International Conference on Internet of Things (iThings), GreenCom, CPSCoM, and SmartData*, 2018, pp. 1145–1152. [Online]. Available: <https://doi.org/10.1109/Cybermatics.2018.00207>
- [94] S. Panja, S. Bag, F. Hao, and B. Roy, “A smart contract system for decentralized borda count voting,” <https://www.dcs.warwick.ac.uk/~fenghao/files/dbc.pdf>, 2020, last accessed: July 12, 2023.
- [95] L. Lys and M. Potop-Butucaru, “Distributed blockchain price oracle,” <https://eprint.iacr.org/2022/603>, 2022, last accessed: July 12, 2023.
- [96] R. Gouiaa, F. Hdhili, and M. Jansen, “A dag-based decentralized oracle model: Implementation and evaluation,” https://link.springer.com/chapter/10.1007/978-3-030-86162-9_31, 2021, last accessed: July 12, 2023.
- [97] M. Stoev, “A fundamental rethinking of rtb advertising,” https://adhash.com/static/documents/AdHash_-_A_Fundamental_Rethinking_of_RTb_Advertising.pdf, 2022, last accessed: June 22, 2023.
- [98] Alkimi Exchange, “The decentralized ad exchange,” <https://alkimi.org>, 2022, accessed May 2025.
- [99] Adwatch, “Blockchain-powered ad tracking,” <https://adwatch.io>, 2022, accessed May 2025.
- [100] J. R. Douceur, “The sybil attack,” in *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002, pp. 251–260.
- [101] E. Foundation, “On finality in ethereum,” <https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/>, 2019.
- [102] B. Nassirzadeh, “Security analysis methods for detection and repair of dos vulnerabilities in smart contracts,” <https://uwspace.uwaterloo.ca/items/6cecf910-0fc1-4341-860c-dd7c48a20440>, 2021, last accessed May 3, 2025.
- [103] S. Community, “Swc registry: Smart contract weakness classification and test cases,” <https://swcregistry.io/docs/SWC-128>, accessed April 19, 2020.

- [104] OWASP Foundation, “Owasp smart contract top 10,” <https://owasp.org/www-project-smart-contract-top-10/>, 2025.
- [105] J. Fox, “Smart contract security risks: Today’s 10 top vulnerabilities and mitigations,” <https://www.cobalt.io/blog/smart-contract-security-risks>, 2024.
- [106] S. Seher, “List of smart contract vulnerabilities & how to mitigate them,” <https://hacken.io/discover/smart-contract-vulnerabilities/>, 2022.
- [107] G. Greenspan, “Why many smart contract use cases are simply impossible,” <https://www.coindesk.com/markets/2016/04/17/why-many-smart-contract-use-cases-are-simply-impossible/>, 2023, last accessed April 26, 2022.
- [108] B. Protocol, “Band protocol: Secure and scalable decentralized oracle for web 3.0,” <https://bandprotocol.com/whitepaper>, 2020, accessed May 2025.
- [109] Singular, “Ad fraud,” <https://www.singular.net/glossary/ad-fraud/>, 2022, last accessed October 4, 2022.
- [110] Google, “Click and impression counting discrepancies,” <https://support.google.com/campaignmanager/answer/2835377?hl=en>, 2023, last accessed June 22, 2023.
- [111] J. Fáwolé and L. Ciattaglia, “Sybil attack in blockchain: Examples & prevention,” <https://hacken.io/insights/sybil-attacks/>, 2023, last accessed April 26, 2023.
- [112] J. Che, Y. Zhao, and X. Chen, “Oscms: A decentralized open-source coordination management system using a novel triple-blockchain architecture,” <https://www.mdpi.com/2076-3417/13/11/6580>, 2023, last accessed July 12, 2023.
- [113] RADIX, “What are proof of work and proof of stake?” <https://learn.radixdlt.com/article/what-are-proof-of-work-and-proof-of-stake>, 2023, last accessed July 12, 2023.
- [114] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies*. Princeton University Press, 2016.
- [115] Crypto.com, “What is delegated proof of stake?” <https://crypto.com/university/what-is-dpos-delegated-proof-of-stake>, 2022, last accessed July 12, 2023.
- [116] D. MacKenzie, K. Caliskan, and C. Rommerskirchen, “The longest second: Header bidding and the material politics of online advertising,” *Economy and Society*, vol. 52, no. 3, pp. 554–578, 2023. [Online]. Available: <https://doi.org/10.1080/03085147.2023.2238463>
- [117] S. R. Department, “Digital advertising spending worldwide from 2021 to 2026,” <https://www.statista.com/statistics/237974/>, 2023, last accessed June 4, 2023.
- [118] IAB, “Digital advertising soared 35

- [119] Prebid, “Why prebid,” <https://prebid.org/why-prebid/>, last accessed October 4, 2022.
- [120] A. Levy, “Header bidding demystified,” *AdMonsters*, 2017, <https://www.admonsters.com/header-bidding-demystified/>.
- [121] Admetrics, “Impression discrepancy benchmarking report,” <https://www.admetrics.io/blog/impression-discrepancy-report>, 2021, last accessed May 22, 2025.
- [122] Association of National Advertisers (ANA), “Understanding ad delivery transparency and discrepancy,” <https://www.ana.net/miccontent/show/id/ii-ad-transparency>, ANA White Paper, Tech. Rep., 2020.
- [123] ADEX, “Ad discrepancy: When your stats don’t match up,” <https://adex.com/blog/ad-discrepancy-when-your-stats-dont-match-up/>, 2022, last accessed December 22, 2023.
- [124] A. Team, “Header bidding facts and statistics 2021,” <https://headerbidding.co/header-bidding-statistics/>, 2021, last accessed October 4, 2022.
- [125] J. Tuwiner, “How many bitcoins are there?” <https://buybitcoinworldwide.com/how-many-bitcoins-are-there/>, 2023, last accessed January 3, 2024.
- [126] D. Tandan, “What is bitcoin mining? how it works and what it takes to make it pay,” <https://www.toptal.com/finance/blockchain/what-is-bitcoin-mining>, 2023, last accessed January 3, 2024.
- [127] N. Studios, “The cost of mining bitcoin in 198 different countries,” <https://www.visualcapitalist.com/cp/the-cost-of-mining-bitcoin-in-198-different-countries/>, 2022, last accessed January 3, 2024.
- [128] A. Kohnhorst, “How much can you make mining bitcoin in 2022,” <https://miningstore.com/understanding-the-bitcoin-mining-rig-market/how-much-can-you-make-mining-bitcoin-in-2022/>, 2022, last accessed January 3, 2024.
- [129] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.
- [130] H. A. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 772–781.
- [131] S. Zamfira, R.-G. Roman, and C. Dobre, “Blockchain for artificial intelligence: Review and open research challenges,” *Information*, vol. 12, no. 8, p. 318, 2021. [Online]. Available: <https://doi.org/10.3390/info12080318>

- [132] O. P. Foundation, “Ocean protocol: A decentralized data exchange protocol to unlock data for ai,” <https://oceanprotocol.com>, 2020, accessed May 2025.
- [133] D. Network, “Datum: Decentralized data marketplace,” <https://datum.org>, 2020, accessed May 2025.
- [134] B. Team, “Bittensor: A decentralized machine learning network,” <https://bittensor.com>, 2023, accessed May 2025.
- [135] A. Network, “Akash: The decentralized cloud,” <https://akash.network>, 2023, accessed May 2025.
- [136] G. Project, “Golem: The decentralized supercomputer,” <https://golem.network>, 2023, accessed May 2025.

Glossary

gasleft() A built-in Solidity function that returns the amount of gas remaining in the current execution context [26](#), [30](#), [49](#), [51](#)

Abstract Syntax Tree (AST) A tree representation of the abstract syntactic structure of code [11](#), [28](#)

Block Gas Limit The maximum total gas allowed for all transactions in a single Ethereum block [24](#), [28](#)

block production The process by which new blocks are created and added to a blockchain, usually involving leader election and validation [22](#)

Control Flow Graph (CFG) A representation of all paths that might be traversed through a program during execution [11](#), [23](#), [28](#)

Data Dependency Graph (DDG) A graph showing data dependencies between instructions or variables in a program [11](#)

data source External origin of the data being counted [74](#)

Denial-of-Service An attack that aims to disrupt the normal functioning of a system, often by overwhelming it with resource consumption [23](#)

DSP Demand-Side Platform; a system that allows advertisers to bid on and purchase digital ad inventory in real-time [85–87](#), [90](#)

finality reversion The rollback of previously finalized blocks due to forks or consensus failures, undermining the immutability of past decisions [22](#)

Flattened Contract A Solidity source file that combines all dependencies and imports into a single file for easier deployment or analysis [34](#), [35](#)

Gas Limit The maximum amount of gas units that can be consumed by a transaction or block [23](#)

Gas-Aware Refers to software that considers gas consumption and constraints when executing or analyzing operations [25](#), [34](#), [36](#)

Governance The mechanisms and rules by which decisions are made and enforced within a decentralized protocol [23](#)

header bidding A programmatic technique that allows multiple ad exchanges to bid on advertising inventory simultaneously [95](#)

host Platform storing all information [74, 75](#)

Input ID Unique identifier of the input data [74, 75, 79, 81](#)

JSON JavaScript Object Notation, a lightweight format for structuring data [34](#)

mempool A temporary storage area where unconfirmed transactions wait before being added to the blockchain [7, 22](#)

MEV Maximal Extractable Value, referring to the potential profit a validator can extract by ordering, including, or excluding transactions in a block [8, 22](#)

network partition A failure scenario where the network splits into disjoint segments, potentially preventing consensus among nodes [22](#)

owner The party interested in obtaining the result of the counting system [75](#)

player A participating node in the system [74–77, 79–82](#)

player ID Unique identifier of each player [74](#)

prize pool The financial reward distributed among eligible players at the end of a round or epoch [76, 77](#)

Prompt Engineering The practice of designing effective input prompts to elicit accurate or useful responses from language models [37, 38](#)

proposition A boolean request to be validated [74–82, 94](#)

proposition pool The list of available propositions to be verified [75](#)

proposition price The stake used when players vote for propositions [74, 75, 79](#)

require() A Solidity function that aborts execution and reverts state if a specified condition is not met [34](#)

SHA3 A cryptographic hash function standardized by NIST, used to verify data integrity [96](#)

SMT Solver Satisfiability Modulo Theories solver, used to check logical formulas for satisfiability [11](#)

SSP Supply-Side Platform; a system used by publishers to manage, sell, and optimize available ad inventory [85–87](#), [90](#)

stake A deposit made by a participant in a blockchain protocol as collateral to incentivize honest behavior [95](#), [96](#)

State Bloat The growth of on-chain storage state due to accumulating data, potentially causing performance and cost issues [23](#)

State Change A modification to the contract’s storage variables during execution [24](#)

submitter The player who submits a proposition to be verified [75](#), [77](#), [79](#), [81](#)

Sybil An identity controlled by an adversary in a Sybil attack; often part of a network of fake identities [83](#), [96](#)

Sybil attack An attack where a single entity pretends to be multiple nodes to gain influence [75](#), [76](#), [79](#), [80](#), [82](#)

validator An entity or node in a decentralized system responsible for proposing and validating blocks or state transitions [8](#), [22](#)

verifier Randomly chosen player to verify a proposition [75–83](#), [93–96](#)

Web3 A new iteration of the World Wide Web that aims to be decentralized, transparent, and user-centric, leveraging technologies like blockchain and token-based economics [16](#)