Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica
Curriculum «CURRICULUM NAME»
XXXVIII Ciclo

Tesi di Dottorato / Ph.D. Thesis

Detection and Localization of Bad Randomness Vulnerabilities in Ethereum Smart Contracts: An Empirical Systematization with Taxonomy, Semantic Taint Analysis, and Reinforcement-Learning–Enhanced Path Optimization

Hadis Rezaei

Supervisor: **Prof. Francesco PALMIERI**

PhD Program Director: **Prof. Andrea DE LUCIA**

A.A 2024/2025

To Iran, ancient yet ever young, broken yet never defeated.

You gave the world algebra and astronomy, epic poetry and enduring philosophy, the concept of human rights and the art of healing. Long before these pages were written, you taught humanity that knowledge is the highest form of devotion.

I dedicate this work to the land that shaped my identity, to the culture that sharpened my mind, and to the enduring spirit of a people who have turned seven thousand years of history into an unfinished masterpiece.

Whatever I have achieved began with you. Whatever I will become, I owe to you.

ACKNOWLEDGMENTS

No meaningful work is ever accomplished alone, and this thesis is no exception. I have been fortunate to be surrounded by remarkable individuals whose guidance, generosity, and belief in me made this journey possible.

First and foremost, I express my deepest gratitude to my supervisor, **Prof. Francesco Palmieri**, whose sharp insight, patience, and unwavering support shaped both this research and the researcher I have become. His mentorship went far beyond the technical, he taught me how to think critically, write clearly, and never settle for the easy answer.

I am grateful to **Prof. Andrea De Lucia**, the head of our department, for fostering an academic environment where curiosity and ambition could thrive.

A special chapter of this journey was written in Sweden, where **Prof. Karl Andersson** at Luleå University of Technology welcomed me as a visiting researcher with extraordinary warmth and intellectual generosity. I also thank **Dr. Ahmed Afif Monrat**, whose collaboration and daily discussions during that period enriched my perspective in ways I did not anticipate.

Another enriching chapter of this journey unfolded in England, where **Prof. Rahim Taheri** at the University of Portsmouth opened the doors of his research group to me with genuine hospitality and academic generosity. The fresh perspectives I gained during that period left a lasting imprint on the direction of my work.

I owe a particular debt of gratitude to **Dr. Mojtaba Eshghie**, postdoctoral researcher at Umeå University. His expertise, thoughtful feedback, and genuine willingness to help at every stage left a lasting mark on this work. I am proud to call him a collaborator.

I also thank the thesis committee members for their time and constructive feedback, which helped strengthen the final version of this work.

Finally, and most profoundly, I thank my **family**, especially my **mother**, whose quiet sacrifices, boundless love, and unshakeable faith in me have been the true foundation beneath every achievement of my life. And to my dear friends, who reminded me to breathe, to laugh, and to keep going when the road felt long thank you.

ABSTRACT

The *Bad Randomness* vulnerability is one of the critical security issues in smart contracts, rooted in a fundamental contradiction between the deterministic nature of blockchains and the need for randomness in decentralized applications. This vulnerability, catalogued as `SWC-120`, has enabled real-world exploits resulting in significant financial losses, including the *SmartBillions* attack that drained over 400 ETH and the *Fomo3D* exploit that caused losses exceeding $3 million. Despite being ranked as the fourth most critical smart contract vulnerability by OWASP in 2025, only two specialized detection tools exist for this vulnerability class. General-purpose tools such as Slither and Mythril demonstrate poor detection accuracy with high false positive rates, as they rely on simple syntactic pattern matching without understanding the semantic context of how blockchain values are used. These tools cannot distinguish between safe uses of block attributes for time-locks and dangerous uses for randomness generation.

To address these challenges, this research presents an approach composed of four major components. First, a combined systematic literature review (SLR) and systematization of knowledge (SoK) identifies 24 active smart contract vulnerabilities and analyzes 50 real-world attacks causing financial losses exceeding $1.09 billion, resulting in a new four-tier framework for classifying root causes of vulnerabilities. Second, we construct a risk-stratified benchmark dataset of 17,466 labeled contracts for Bad Randomness (`SWC-120`) vulnerabilities through a five-phase methodology including function-level validation, which revealed that 49% of apparently protected contracts were actually exploitable. Third, we develop **TaintSentinel**, a semantic taint analysis system based on *graduated taint propagation* and context-sensitive rules, trained and evaluated on 4,844 labeled Ethereum contracts. Fourth, we design **SmartTaintRL**, a deep reinforcement learning system that mitigates path explosion and provides precise vulnerability localization at both function and code-node levels.

The contributions of this study are evaluated at three levels. *Theoretically*, we introduce a four-tier framework grouping vulnerabilities by fundamental causes: faulty economic design, protocol lifecycle flaws, external dependency weaknesses, and implementation-level defects. We show that 26% of successful attacks exploit chains of multiple vulnerabilities. *Technically*, we introduce two advanced detection systems: **TaintSentinel**, using a dual-stream neural architecture that integrates global structural analysis with path-specific patterns; and **SmartTaintRL**, a Deep Q-Network with an attention mechanism that reduces analysis paths by 44.3% while preserving 96% recall,

combined with gradient-based attribution methods for pinpointing vulnerable code locations. *Practically*, we construct two complementary datasets: a risk-stratified benchmark of 1,758 vulnerable contracts with four-level classification (High_Risk, Medium_Risk, Low_Risk, Safe) that is 51× larger than existing datasets and serves as the first validated benchmark for `SWC-120`; and a detection-focused dataset of 4,844 real Ethereum contracts with over 1.1 million execution paths, including 252,844 high-quality paths for training machine learning models.

Experimental results on balanced datasets show that **TaintSentinel** achieves an F1-score of 0.892, representing nearly a fourfold improvement over existing tools. **SmartTaintRL** further improves performance with an F1-score of 0.930 on balanced and 0.920 on imbalanced datasets. Evaluation of existing tools on our benchmark showed that both Slither and Mythril failed to detect the majority of vulnerable contracts, confirming the limitations of pattern-based approaches. In terms of localization accuracy, **SmartTaintRL** identifies 64.3% of vulnerable functions exactly and 92.9% when including caller functions, demonstrating that precise vulnerability localization is achievable through reinforcement learning. The system also introduces a new metric, *Path Risk Accuracy*, achieving 97% accuracy on balanced data.

Analysis of real-world attacks reveals that access-control vulnerabilities (13 incidents, $417.95 million loss) and price-manipulation attacks (13 incidents, $279.75 million loss) constitute the dominant exploitation patterns, while reentrancy—despite extensive research attention—accounts for only 11% of total losses. Furthermore, 26% of successful attacks involve multi-vulnerability exploitation chains, a phenomenon largely overlooked in prior research.

The scientific impact of this research includes the first path-level analysis method for detecting Bad Randomness, the first localization approach for identifying vulnerable functions and code nodes in Bad Randomness vulnerabilities, the introduction of the *graduated taint propagation* concept, the first validated risk-stratified benchmark for `SWC-120` vulnerabilities, and demonstrating the effectiveness of deep reinforcement learning in addressing path explosion. The main limitation lies in the current model's inability to capture complex inter-contract interactions during online execution. While the system performs effectively for intra-contract analysis and mitigates class imbalance through reinforcement learning and dual-scenario evaluation, runtime multi-contract interactions remain outside the present scope. Future work will focus on runtime detection mechanisms, hybrid static–dynamic analysis, and enhanced modeling of multi-contract execution interactions.

Overall, this research bridges the gap between academic security analysis and real-world needs of smart contract ecosystems, offering

deployable technical solutions for addressing emerging blockchain security challenges.

Keywords: Smart Contracts · Bad Randomness · Semantic Taint Analysis · Reinforcement Learning · Path Optimization · Vulnerability Localization · Blockchain Security · Ethereum · Benchmark Dataset

Contents

I Introduction and Background
1 Problem Statement 3
1.1 Context and Motivation 3
1.2 Research Focus: Bad Randomness Vulnerability 4
1.3 Research Questions 6
1.4 Research Methodology 8
1.4.1 Phase 1: Systematic Landscape Analysis 8
1.4.2 Phase 2: Solution Development 8
1.4.3 Phase 3: Validation and Evaluation 9
1.5 Research Contributions 9
1.5.1 Theoretical Contributions 9
1.5.2 Technical Contributions 10
1.5.3 Practical Contributions 10
1.6 Thesis Organization 11
2 Background 13
2.1 Introduction 13
2.2 Blockchain and Ethereum Fundamentals 13
2.3 Smart Contracts 16
2.3.1 Smart Contract Lifecycle 16
2.3.2 Types of Contracts and Applications 17
2.4 Bad Randomness Vulnerability 17
2.4.1 Weak Random Sources in Ethereum 18
2.4.2 Weak Randomness Attack Patterns 18
2.5 Vulnerability Detection Techniques 20
2.5.1 Static Analysis and Pattern Matching 20
2.5.2 Symbolic Execution and Path Analysis 20
2.5.3 Taint Analysis and Data Flow Tracing 21
2.5.4 Machine Learning Approaches 22
2.6 Path Explosion Problem 23
2.6.1 Why Path Explosion Occurs in Smart Contracts 23
2.6.2 Limitations of Exhaustive Analysis 23
2.7 Summary 24
3 State of the Art in Bad Randomness Detection and Localization 25
3.1 Introduction 25
3.2 Bad Randomness Detection Approaches 25
3.2.1 Specialized Detection Tools 25

3.2.2 General-Purpose Static Analysis Tools 26
3.2.3 Machine Learning and LLM-Based Approaches 27
3.2.4 Dynamic Analysis and Fuzzing Approaches 27
3.3 Taint Analysis Methods for Smart Contracts 28
3.3.1 Taint-Based Tools Overview 28
3.3.2 Gap Analysis: Why Existing Taint Analysis Falls Short 30
3.3.3 Summary and Implications 32
3.4 Vulnerability Localization Methods 32
3.4.1 Evolution of Localization Techniques 32
3.4.2 Granularity Levels in Vulnerability Localization 33
3.5 Summary 34

II Empirical Studies
4 Systematic Literature Review of Smart Contract Vulnerabilities 39
4.1 Introduction 39
4.2 Research Protocol 41
4.2.1 Study Identification 41
4.2.2 Study Selection 41
4.3 Active Vulnerabilities Catalog 42
4.3.1 Integer Over/Underflow and Rounding Errors 43
4.3.2 Improper Handling of External Calls 45
4.3.3 Reentrancy 45
4.3.4 Dangerous Delegatecall 47
4.3.5 Unbounded Loops in Dynamic Arrays 48
4.3.6 DoS Attack via Owner Account 48
4.3.7 State-Revert 49
4.3.8 Frozen Ether 49
4.3.9 Insecure Randomness 50
4.3.10 Front-Running (Abusing Transaction-Ordering Dependency) 51
4.3.11 Authorization Using tx.origin 52
4.3.12 Additional Vulnerabilities 53
4.3.13 Hierarchical Organization of Vulnerabilities 53
4.4 Deprecated Vulnerabilities 55
4.4.1 Upgradable Contracts 55
4.4.2 Wrong Address 56
4.4.3 Erroneous Visibility 57
4.4.4 Suicidal Contract 58
4.4.5 Call-Stack Depth Limit 58
4.5 Summary 58

5 Empirical Analysis of Real-World Smart Contract Attacks 61
5.1 Introduction 61
5.2 Incident Review Protocol 62
5.2.1 Stage 1: Comprehensive Data Collection 62
5.2.2 Stage 2: Impact-Based Prioritization 63
5.2.3 Stage 3: Ecosystem Classification 63
5.2.4 Stage 4: Vulnerability Verification and Report Quality Assessment 63
5.3 Dataset Overview 64
5.4 Vulnerability Distribution Analysis 66
5.4.1 Dominant Vulnerability Patterns 67
5.4.2 Persistent Classic Vulnerabilities 68
5.4.3 Diminishing Legacy Vulnerabilities 68
5.4.4 Multi-Vulnerability Exploitation Chains 69
5.4.5 Application-Specific Vulnerability Correlations 69
5.5 Key Insights from Incident Analysis 70
5.5.1 Insight 1: Flash Loans as Attack Amplifiers 70
5.5.2 Insight 2: The Pervasiveness of Human Error 71
5.5.3 Insight 3: Composability Risks 72
5.5.4 Insight 4: Oracle Security is Important 72
5.5.5 Insight 5: Insider Threats and Privilege Abuse 73
5.5.6 Insight 6: New Attack Vectors for EVM Upgrades 73
5.5.7 Insight 7: Inadequate Emergency Response Mechanisms 74
5.5.8 Insight 8: Complex Cross-Chain Exploits 74
5.5.9 Insight 9: Multi-Vulnerability Exploit Chains 75
5.5.10 Insight 10: Business Logic Flaws is Not Specific-Enough 76
5.5.11 Insight 11: Complications of Proxy/Upgradability Patterns 77
5.6 Discussion and Implications 78
5.6.1 Gap Between Literature and Practice 78
5.6.2 Four-Tier Root-Cause Framework 79
5.7 Exploit Chain Case Study: LI.FI GasZip Facet 79
5.7.1 Protocol, Code, and On-Chain Addresses 80
5.7.2 Incident Summary 80
5.7.3 Vulnerable Code 81
5.7.4 Exploit Chain Reconstruction 81
5.7.5 Breaking the Exploit Chain 82

5.8 Summary 83

III Novel Methods
6 Risk-Stratified Benchmark Dataset for Bad Randomness 87
6.1 Introduction 87
6.2 Background and Related Work 88
6.2.1 Mitigation Mechanisms 89
6.2.2 Existing Detection Tools and Datasets 89
6.3 Dataset Construction and Validation 90
6.3.1 Phase 1: Data Collection and Keyword Filtering 90
6.3.2 Phase 2: Vulnerability Pattern Labeling 90
6.3.3 Phase 3: Risk-Level Classification 92
6.3.4 Phase 4: Function-Level Validation 94
6.3.5 Phase 5: Context-Aware Refinement 96
6.3.6 Final Dataset and Comparison 97
6.4 Discussion 98
6.4.1 Comparison with Existing Detection Tools 98
6.4.2 Implications for Detection Tools 101
6.4.3 The Prevalence Problem 101
6.4.4 Limitations 101
6.5 Summary 102
7 TaintSentinel: Semantic-Aware Detection 103
7.1 Introduction 103
7.2 System Architecture Overview 105
7.3 Phase 1: Context-Aware Taint Analysis 106
7.3.1 AST Construction and Source/Sink Identification 106
7.3.2 Semantic Graph Construction 108
7.3.3 Graduated Taint Propagation 108
7.3.4 Context-Sensitive Risk Assessment 108
7.4 Phase 2: Dual-Stream Neural Architecture 109
7.4.1 Global Context Stream 109
7.4.2 Path-Focused Stream 111
7.4.3 Adaptive Fusion Mechanism 111
7.5 Dataset Construction and Labeling 112
7.5.1 Data Collection 112
7.5.2 Labeling Methodology 114
7.5.3 Dataset Statistics 114
7.5.4 Evaluation Metrics 115
7.6 Results and Analysis 116
7.6.1 Overall Performance Analysis 116
7.6.2 Path Risk Assessment Performance 117
7.6.3 ROC Curve Analysis 117
7.6.4 Training Dynamics and Convergence 118

7.6.5 Computational Efficiency 119
7.6.6 Comparison with State-of-the-Art Tools 119
7.7 Discussion and Summary 121
7.7.1 Key Findings 121
7.7.2 Limitations 121
7.7.3 Summary 122
8 SmartTaintRL: RL-Based Detection and Localization 123
8.1 Introduction 123
8.2 System Architecture Overview 123
8.3 Phase 1: Offline Preprocessing 126
8.3.1 Path Extraction via Taint Analysis 126
8.3.2 Feature Extraction 126
8.3.3 Offline Storage and Indexing 127
8.4 Phase 2: RL-based Path Prioritization 127
8.4.1 Empirical Weight Calibration 128
8.4.2 Dynamic Pool Management 129
8.4.3 Pattern Registry 130
8.4.4 Reinforcement Learning Environment 131
8.4.5 Deep Q-Network Architecture 132
8.4.6 Hierarchical Reward Engineering 134
8.5 Phase 3: Vulnerability Localization 136
8.5.1 Function-Level Localization 137
8.5.2 Node-Level Localization 137
8.5.3 Ground Truth Dataset 140
8.5.4 Evaluation Results 141
8.6 Dataset Construction 144
8.6.1 Dataset Overview 144
8.6.2 Training Data Filtering 145
8.7 Experimental Setup 145
8.7.1 Model Architecture and Hyperparameters 146
8.7.2 Training Procedure 146
8.7.3 Hierarchical Reward Configuration 147
8.7.4 Implementation Environment 147
8.8 Results and Evaluation 148
8.8.1 Overall Performance Analysis 148
8.8.2 Training Dynamics and Convergence 149
8.8.3 Path Selection Quality Metrics 152
8.8.4 Comparison with State-of-the-Art Methods 153
8.8.5 Computational Efficiency Analysis 156
8.9 Summary 157

IV Conclusion
9 Discussion and Future Directions 161
9.1 Chapter Overview 161

9.2 Summary of Contributions 163
9.2.1 Theoretical Contributions 163
9.2.2 Technical Contributions 164
9.2.3 Practical Contributions 165
9.3 Answering the Research Questions 166
9.3.1 RQ1: Vulnerability Landscape and Misalignment 166
9.3.2 RQ2: Semantic-Aware Detection Effectiveness 168
9.3.3 RQ3: Reinforcement Learning for Path Explosion 169
9.3.4 RQ4: Vulnerability Localization 171
9.4 Key Findings and Implications 172
9.4.1 Theoretical Implications 173
9.4.2 Methodological Implications 173
9.4.3 Practical Implications 174
9.5 Comparative Analysis and Positioning 174
9.5.1 Detection Philosophies 175
9.5.2 Performance Characteristics and Trade-offs 176
9.6 Limitations 177
9.6.1 Scope Constraints 177
9.6.2 Methodological Limitations 177
9.6.3 Dataset Characteristics 178
9.6.4 Computational Considerations 179
9.7 Open Issues and Future Research Directions 179
9.7.1 Extension to Other Vulnerability Types 179
9.7.2 Cross-Contract and Compositional Analysis 180
9.8 Concluding Remarks 180

V Appendix
A Supplementary Material for Chapter 3 185
A.1 Complete Active Vulnerabilities Catalog 185
A.2 A.1 Prodigal Contract 185
A.3 A.2 Ether Lost to Orphan Address 185
A.4 A.3 Untrustworthy or Manipulable Data Feeds 186
A.5 A.4 Compiler Version Not Fixed 187
A.6 A.5 Event-Ordering Bug 187
A.7 A.6 Type Casting 187
A.8 A.7 Ponzi Scheme 188
A.9 A.8 Storage Collision 188
A.10 A.9 Business Logic Flaws 190
A.11 A.10 Dangerous Balance Inequality 191
A.12 A.11 Resource Exhaustion 191
A.13 A.12 Access Control 192
A.14 A.13 Timestamp Dependence 192

a.15 Attack Transaction Details 193
b Supplementary Material for Chapter 7 195
b.1 Complete Ground Truth Contract Information 195

Bibliography 197

List of Figures

Figure 2.1 Ethereum multi-layer architecture showing the application layer (smart contract types and interaction patterns), EVM execution environment (stack, memory, storage, and opcodes), consensus mechanism (Proof of Stake with validator lifecycle and state transitions), and Ethereum-related networks (Layer 2 solutions, sidechains, and cross-chain bridges). 14

Figure 4.1 Systematic review filtering process showing the reduction from 17,013 identified papers to 71 quality-validated papers. The process moved through four stages: Identification, Screening, Eligibility assessment, and Quality-based selection. Source: Adapted from [3]. 43

Figure 4.2 Vulnerable ↔ attack contract pairs and the patched contract involving in reentrancy vulnerability. 46

Figure 4.3 Delegatecall attack flow. A flawed interaction through a `delegatecall` can be initiated by the attacker calling a function in the target contract, which involves an unsafe `delegatecall` to an attacker-desired contract. This vulnerability might require a chain of exploits to pull off a successful attack. 47

Figure 4.4 Demonstration of front-running attack where higher gas incentive (50 Gwei vs 20 Gwei) allows Transaction B to preempt Transaction A in block inclusion, despite later submission time. 51

Figure 5.1 Four-stage smart contract incident review protocol to systematically identify, prioritize, and analyze relevant attacks. 62

Figure 5.2 Vulnerability chains in analyzed incidents. The left column represents entry vulnerabilities, the right column shows second-stage vulnerabilities. Numbers on flows indicate incident count for each chain pattern. 76

Figure 6.1 Dataset construction pipeline showing contract counts at each phase. The final output is categorized into four risk levels. 91
Figure 6.2 Distribution of vulnerability patterns across 1,903 candidate contracts. G1 (direct modulo) and G2 (uint cast) together account for 57.8% of all patterns. 93
Figure 6.3 Validation flow showing contract reclassification across phases. The OWNER_ONLY category decreased from 1,167 to 172 contracts (85.3% false positive rate) after function-level validation. 94
Figure 6.4 Recall comparison of vulnerability detection tools. Existing tools (Slither, Mythril) achieve 0% recall on our dataset, while our pattern-based labeler achieves 100% recall by design. 99
Figure 7.1 TaintSentinel Framework Overview: Two-phase architecture for path-level bad randomness vulnerability detection. Phase 1 performs context-aware taint analysis through four stages: role-based AST construction, semantic graph building, graduated taint propagation, and vulnerability path discovery. Phase 2 employs a dual-stream neural architecture combining bidirectional LSTM for path sequences and three-layer GCN for global graph structure, merged through adaptive fusion for final classification. 105
Figure 7.2 Confusion matrices showing prediction accuracy for (a) balanced dataset and (b) imbalanced dataset with optimized threshold. 116
Figure 7.3 Comprehensive performance comparison between balanced and imbalanced scenarios: (a) bar chart showing individual metrics, (b) radar plot visualization across five key metrics. 117
Figure 7.4 ROC curves showing model's discriminative ability for (a) balanced dataset (AUC = 0.951) and (b) imbalanced dataset (AUC = 0.940). 118
Figure 7.5 Training dynamics showing loss and F1-Score evolution during training for balanced (top) and imbalanced (bottom) datasets. 118

Figure 8.1 SmartTaintRL Architecture comprising four components. **(Top)** Path database with 100-dimensional feature vectors. **(Middle)** RL decision engine with pool management, attention-guided Q-network, and ϵ-greedy action selection. **(Bottom-Left)** Training system with hierarchical reward, pattern registry, experience replay, and dual DQN networks. **(Bottom-Right)** Localization module combining attention weights and gradient-based attribution for identifying vulnerable functions and nodes. 124

Figure 8.2 Empirical weight calibration based on 223 vulnerable contracts. Blue bars show prevalence of each entropy source in vulnerable functions. Orange bars represent normalized weights assigned to each source. 128

Figure 8.3 Source-sink combination patterns observed in 223 vulnerable contracts. Heatmap displays frequency distribution with darker colors indicating higher occurrence rates. 129

Figure 8.4 Localization evaluation results on 14 vulnerable contracts. **(a)** Contract complexity distribution showing lines of code (colored bars) and control flow paths (gray bars) for each contract, with color-coding indicating localization outcome: blue for exact matches, green for caller detection, and red for incorrect predictions. **(b)** Function-level localization distribution across exact match (64.3%), caller detection (28.6%), and incorrect predictions (7.1%), achieving strict accuracy of 64.3% and relaxed accuracy of 92.9%. **(c)** Performance metrics comparison across function-level (strict and relaxed) and node-level localization. 143

Figure 8.5 Performance metrics evolution across training episodes for balanced and imbalanced datasets. Four panels show Accuracy, Precision, Recall, and F1-Score trajectories from episode 500 to 5000. 149

Figure 8.6 Confusion matrices for balanced and imbalanced datasets at 500, 2,500, and 5,000 episodes. Color intensity indicates prediction frequency. 150

Figure 8.7 Path pruning evolution during training. The Analyze ratio decreases from 90% to 55.7% over 2,500 episodes, demonstrating that the agent learns to selectively skip low-risk paths while maintaining 96% recall. 150
Figure 8.8 Contract-level detection rates at episode 2,500 for both datasets. Dashed line indicates 95% threshold. 151
Figure 8.9 Episode reward evolution during training with moving average. Final average reaches 18.47 at episode 2,500. 151
Figure 8.10 Training loss (TD error) over training steps showing exponential convergence. 152
Figure 8.11 Q-value variance evolution showing three-phase learning dynamics: exploration, transition, and exploitation. 152
Figure 8.12 Distribution of importance scores for analyzed and skipped paths showing clear separation. 153
Figure 8.13 Q-value distribution analysis showing sensitivity to timestamp presence (left) and require density (right). 153
Figure 8.14 Cumulative unique vulnerability patterns discovered during training. Rapid initial discovery followed by gradual plateau. 154

List of Tables

Table 1.1 Vulnerability Distribution in 50 High-Impact Attacks (2022-2025) 5
Table 3.1 Comparative Analysis of Bad Randomness Detection Tools 26
Table 3.2 Comparison of Taint Analysis Features 30
Table 3.3 Generational Evolution of Vulnerability Localization Methods 33
Table 3.4 Granularity Levels in Vulnerability Localization 34
Table 4.1 Comparison of smart contract security SLRs and SoKs 40

Table 4.2 Keyword extraction method using PICOC framework 41
Table 4.3 List of exclusion criteria 42
Table 4.4 Mapping of Vulnerability Categories to 4-Tier Root Cause Framework 56
Table 4.5 Summary of deprecated smart contract vulnerabilities and their mitigation 57
Table 5.1 Overview of studied security incidents 65
Table 5.2 A multi-tier root-cause framework derived from real-world incidents. Incident IDs refer to rows in Table 5.1. 80
Table 5.3 Breaking the LI.FI GasZip exploit chain 82
Table 6.1 Existing SWC-120 Benchmark Datasets 89
Table 6.2 Vulnerability Detection Patterns Organized by Semantic Category 92
Table 6.3 Risk Levels, Attacker Capabilities, and Initial Classification Results 93
Table 6.4 Context-Aware Classification of 260 No_Pattern_In_Functions Contracts 96
Table 6.5 Final Dataset Composition 97
Table 6.6 Comparison with Existing SWC-120 Benchmark Datasets 97
Table 6.7 Detection results of Slither and Mythril on our dataset. 98
Table 7.1 Randomness Vulnerability Pattern Classification 107
Table 7.2 Context-Sensitive Classification Rules 109
Table 7.3 Distribution of Primary Bad Randomness Sources in SmartBugs-Wild 114
Table 7.4 Summary of Vulnerable and Safe Contracts from Three Primary Sources 115
Table 7.5 TaintSentinel Runtime Performance Across Contract Sizes 119
Table 7.6 Performance Comparison with Existing Tools 120
Table 8.1 Ground Truth Dataset for Bad Randomness Localization 142
Table 8.2 Localization Results on 14 Vulnerable Contracts 143
Table 8.3 Dataset Statistics: Contracts, Paths, and Label Distribution 145
Table 8.4 SmartTaintRL Training Configuration and Hyperparameters 146
Table 8.5 Performance Metrics Across Training Episodes and Datasets 148

Table 8.6 Performance Comparison with State-of-the-Art Methods 155
Table 8.7 Computational Time Breakdown 157
Table A.1 Mapping of exploit transaction hashes, compromised victim contracts, and attacker addresses (smart contracts and EOAs). 194
Table B.1 Complete Contract Information for Localization Ground Truth 195

LISTINGS

2.1 Vulnerable lottery contract using weak randomness . . 19
2.2 Smart Contract Taint Analysis Example 21
4.1 Integer overflow/underflow in Solidity 44
4.2 Lottery Contract with insecure randomness 50
4.3 Victim contract demonstrating the use of tx.origin . . . 52
4.4 Attacker contract exploiting tx.origin vulnerability . . 52
5.1 Excerpt from GasZipFacet.sol 81
6.1 Test contract with direct modulo pattern. 99
A.1 Transfer function without destination address validation 185
A.2 Wallet Contract with storage collision vulnerability . . 189
A.3 Smart Contract: Pools 190
A.4 VulnerableContract Example 192

ACRONYMS

Part I

INTRODUCTION AND BACKGROUND

1 PROBLEM STATEMENT

1.1 CONTEXT AND MOTIVATION

In 2021, approximately $14 billion was lost in the cryptocurrency space due to exploits and security incidents a trend that continued through 2024 when $2.2 billion was stolen through attacks on blockchain platforms [1, 2]. These figures reveal a fundamental paradox in blockchain technology: while its immutable and decentralized architecture provides unprecedented transparency and trust, it also creates an irreversible security problem. Once a vulnerability is exploited, the resulting financial losses are rarely recoverable. This paradox has transformed smart contract security from an academic concern into a critical industrial necessity [3, 4].

The severity of this challenge is clear from high-profile attacks that have cost millions. The Fomo3D [5] incident of 2018 is a prime example. Through a sophisticated combination of Bad Randomness exploitation and denial-of-service attacks (DoS), malicious actors drained over $3 million from the platform. This was not simply a technical exploit. It was a carefully orchestrated manipulation of the platform's flawed randomness generation mechanism, which relied on predictable blockchain values such as `block.timestamp` and `blockhash` [5].

The SmartBillions [6] hack tells a similar story. Attackers stole 400 ETH (approximately $120,000 at the time), proving that smart contract vulnerabilities are not just theoretical risks, they result in real financial losses. These incidents highlight an urgent reality: despite billions of dollars at stake, smart contract vulnerabilities continue to be exploited with devastating consequences [6].

Despite the critical importance of smart contract security, there is a notable gap between academic research and real-world exploitation patterns. The research community has made significant efforts to identify and categorize vulnerabilities, develop detection tools, and propose mitigation strategies [3, 7, 8, 9]. However, these efforts often fall short in practice.

Current state-of-the-art detection tools such as Slither [10], Mythril [11], and Oyente [12] rely on pattern-based detection mechanisms. They search for syntactic patterns in code without understanding the semantic context. Empirical evaluations on real Ethereum contracts have revealed concerning results: these tools show poor performance in detecting many vulnerability types [13, 14].

The core issue is their approach. They treat vulnerability detection as a simple matching exercise, looking for specific code constructs that fit predefined templates. This means they miss the semantic and contextual nature of many vulnerabilities [13, 14].

To understand this gap between research and real-world threats, we conducted a dual analysis that forms the foundation of this thesis.

Our investigation used two approaches. First, we performed a systematic literature review following PRISMA guidelines [3]. We analyzed 71 peer-reviewed papers published between 2018 and 2024 and built a taxonomy of 25 active vulnerability types identified by researchers [3].

Second, we conducted an analysis of 50 real-world attacks that occurred between 2022 and 2024. These attacks resulted in over $1.09 billion in losses. This analysis helped us understand which vulnerabilities are actually exploited in practice [13]. This analysis helped us understand which vulnerabilities are actually exploited in practice [13].[1]

The results revealed a key misalignment: the vulnerabilities researchers focus on are often not the ones attackers exploit. For example, reentrancy vulnerabilities received significant academic attention [3, 15] and accounted for 35% of financial losses. Access control issues contributed to 28% of losses. However, many other categorized vulnerabilities showed minimal presence in real-world attacks [13].

This gap raises an important question: why do certain high-severity vulnerabilities remain understudied despite their proven impact?

1.2 RESEARCH FOCUS: BAD RANDOMNESS VULNERABILITY

We selected Bad Randomness as the primary focus of this research based on three converging factors. First, OWASP ranks it as the fourth most critical smart contract vulnerability in 2025 [16]. Second, only one specialized detection tools exist: RNVulDet [17]. Third, historical attacks demonstrate severe financial impact, including the Fomo3D exploit that drained $3 million in 2018 [5] and SmartBillions' loss of 400 ETH (approximately $120,000) [6]. These factors suggest an urgent need for improved detection methods.

Bad Randomness occurs when smart contracts use predictable or manipulable blockchain values for random number generation [18]. The blockchain's deterministic nature essential for distributed consensus makes it fundamentally impossible to generate true randomness [19, 16, 20], creating a critical problem for applications requiring unpredictable values.

However, our systematic analysis revealed an unexpected pattern. Despite examining 50 major attacks between 2022 and 2025 that collectively resulted in $1.09 billion in losses [3], Bad Randomness did not

1 Complete attack analysis data, incident timelines, victim/attacker addresses, and transaction hashes are available at: .

appear as a primary attack vector in any incident. Table 1.1 presents the vulnerability distribution across these incidents.

Table 1.1: Vulnerability Distribution in 50 High-Impact Attacks (2022-2025)

Vulnerability Type	Incidents	Total Loss (USD)
Access Control	13	$417,950,000
Price Manipulation	13	$279,750,000
Reentrancy	7	$118,501,279
Business Logic Flaws	5	$177,400,000
Input Validation	6	$41,850,000
Rounding Error	4	$36,900,000

This absence creates a critical paradox: a vulnerability ranked fourth by OWASP with documented historical losses does not appear in recent high-impact attacks, while Access Control and Price Manipulation dominate the current threat landscape. We identified four explanations that transform this paradox into motivation.

Factor 1: Economic Target Distribution. Bad Randomness vulnerabilities concentrate in gaming and lottery applications [13]. These applications naturally require randomness for core functionality such as winner determination or outcome generation. However, they manage significantly lower total value locked (TVL) compared to DeFi protocols [21]. Sophisticated attackers rationally prioritize high-value DeFi targets where single exploits yield tens or hundreds of millions of dollars. This economic calculation explains why Bad Randomness attacks, though technically feasible, remain less attractive than exploiting vulnerabilities in high-TVL protocols.

Factor 2: Detection Tool Failure. Our empirical evaluation on 4,844 Ethereum contracts revealed severe limitations in existing tools. Slither achieved F1-score of only 0.232. Mythril reached 0.236 [13]. Even specialized tools designed for randomness detection [17] show significant limitations. These tools rely on syntactic pattern matching, searching for keywords like `block.timestamp` or `blockhash` without understanding semantic context. They flag safe uses such as timelocks as vulnerable while missing actual vulnerabilities in complex execution paths [13].

This detection failure creates a dangerous situation. Automated tools cannot reliably identify these vulnerabilities, leaving them hidden in production code. Developers receive either overwhelming false alarms that erode trust, or false confidence in contract security [22]. Vulnerabilities exist in deployed contracts but remain undetected and unexploited, not because they cannot be exploited, but because finding them requires expensive manual analysis that attackers may not consider worthwhile for lower-value targets.

Factor 3: Emerging Threat Landscape. The blockchain gaming sector experienced explosive growth in recent years. Play-to-earn platforms and NFT-based games attract substantial investment [23]. As economic value in gaming applications increases, attacker incentive structures will shift. A vulnerability currently unexploited due to target economics may become highly attractive as the gaming sector matures.

Blockchain expansion into new domains creates additional attack surfaces. Decentralized prediction markets, on-chain governance systems requiring unbiased randomness, and fair NFT distribution mechanisms all need robust randomness generation. Many developers lack awareness of how these vulnerabilities can be exploited. These emerging applications amplify the urgency of improved detection methods.

Factor 4: Research Gap and Opportunity. Our systematic analysis [3] demonstrated that Access Control and Price Manipulation have received extensive research attention. Multiple specialized tools, formal verification techniques, and established best practices exist for these vulnerability classes. In stark contrast, Bad Randomness remains severely understudied despite its OWASP ranking and proven historical impact.

This combination of high theoretical severity, weak detection capabilities, inadequate tooling, and expanding application domains creates a dangerous situation. The current detection gap leaves systems increasingly vulnerable as economic value grows. Unlike well-studied vulnerabilities where marginal improvements are difficult, Bad Randomness offers potential for transformative advances. The complete failure of existing tools establishes a low baseline, meaning significant improvements are both feasible and measurable.

Solving the Bad Randomness detection problem requires addressing fundamental challenges: context-sensitive semantic understanding, path explosion in execution trace analysis, and distinguishing safe from unsafe usage patterns. Solutions to these challenges may generalize to other vulnerability types, amplifying research impact beyond this specific vulnerability class.

These four factors validate our initial research decision. The systematic analysis reinforced rather than contradicted our choice. The absence from recent attacks reflects detection failure, not genuine security. Vulnerabilities that exist undetected represent latent threats that could materialize as application domains evolve and attack economics change.

1.3 RESEARCH QUESTIONS

RQ1: What is the current landscape of smart contract vulnerabilities in both academic literature and real-world practice, and what factors explain the misalignment between them?

This question establishes our understanding of the vulnerability ecosystem from two perspectives. We analyze which vulnerabilities have been identified and categorized by researchers through academic papers. We also examine which ones are actively exploited in practice through real-world attacks.

More importantly, we investigate what explains the gap between these two perspectives. Why do certain vulnerabilities receive extensive academic attention while others remain understudied? Why do some vulnerabilities continue to cause significant financial losses despite their documented severity?

RQ2: Can semantic-aware taint analysis with context-sensitive rules effectively detect Bad Randomness vulnerabilities that current pattern-based tools fail to identify?

This question addresses a key limitation of existing detection approaches that rely on simple syntactic pattern matching. We investigate whether graduated taint propagation combined with domain-specific context-sensitive rules can distinguish between safe and unsafe usage patterns of blockchain values.

For example, can context-aware analysis differentiate benign uses of `block.timestamp` for deadline checking from dangerous uses for randomness generation? Can path-level analysis achieve high detection accuracy while maintaining acceptable computational overhead?

RQ3: Can reinforcement learning techniques intelligently address the path explosion problem while maintaining high detection accuracy?

The path explosion problem has been a major barrier to exhaustive path analysis in smart contract verification. This question examines whether deep reinforcement learning can learn to prioritize high-risk execution paths while safely pruning low-value paths.

Can pool-based exploration strategies and per-path decision-making reduce computational complexity? Can hierarchical reward engineering effectively incorporate domain knowledge about vulnerability patterns? More importantly, can such approaches achieve significant path reduction without sacrificing detection recall?

RQ4: Can vulnerability detection systems provide precise localization that identifies not only vulnerable contracts but also the specific functions and code locations responsible for security flaws?

Current detection tools typically provide binary classifications. They flag entire contracts as vulnerable or safe without indicating where the vulnerability resides.

This question investigates whether automated analysis can achieve precise localization at two levels. First, can it identify which functions contain vulnerabilities? Second, can it pinpoint the specific code nodes responsible? We examine localization accuracy across contracts of varying sizes and complexity. Does such precision provide developers

with useful information for effective remediation, or does it still force manual inspection of entire codebases?

1.4 RESEARCH METHODOLOGY

To address these research questions, we use a mixed-methods approach that combines empirical analysis, novel algorithm development, and experimental validation. Our methodology has three phases:

1.4.1 *Phase 1: Systematic Landscape Analysis*

We conduct a dual-perspective analysis of the smart contract vulnerability ecosystem, presented in detail in Part ii. First, we perform a systematic literature review of 71 peer-reviewed papers to identify and categorize known vulnerabilities. Second, we conduct analysis of 50 real-world attacks representing over $1.09 billion in losses to understand which vulnerabilities are actually exploited in practice. This analysis reveals critical misalignments between academic focus and practical exploitation patterns (real-world), detailed in Chapters 4 and 5.

1.4.2 *Phase 2: Solution Development*

Based on these limitations, we develop a comprehensive benchmark dataset and two detection approaches, presented in detail in Part iii.

Risk-Stratified Benchmark Dataset addresses the fundamental challenge of lacking large, validated datasets for Bad Randomness vulnerabilities. We construct a dataset of 1,758 labeled contracts through a five-phase methodology: keyword filtering, pattern matching with 58 regular expressions, risk-level classification into four categories (SAFE, LOW_RISK, MEDIUM_RISK, HIGH_RISK), function-level validation, and context-aware refinement. This benchmark is 51× larger than existing datasets and provides the foundation for training and evaluating the detection methods 6.

TaintSentinel uses semantic-aware taint analysis with graduated propagation and context-sensitive rules. This allows it to distinguish between benign and vulnerable uses of blockchain values 7.

SmartTaintRL uses deep reinforcement learning with hierarchical reward engineering to address the path explosion problem. It learns to prioritize high-risk execution paths while maintaining comprehensive vulnerability coverage 8.

1.4.3 *Phase 3: Validation and Evaluation*

We validate our approaches through experiments on real-world contract datasets. We test under both balanced and imbalanced conditions that reflect production environments (iii, Chapters 7–8). Our evaluation includes several components:

- Quantitative comparison with existing state-of-the-art tools using standard metrics (precision, recall, F1-score, AUC-ROC)
- Analysis of computational efficiency
- Assessment of localization accuracy for practical remediation

This methodology directly addresses our research questions: RQ1 through systematic landscape analysis (Phase 1, detailed in Part ii); RQ2 through TaintSentinel's semantic-aware taint analysis (Part iii, Chapter 7); RQ3 and RQ4 through SmartTaintRL's reinforcement learning approach and gradient-based localization mechanisms (Part iii, Chapter 8).

1.5 RESEARCH CONTRIBUTIONS

This thesis makes several contributions to the field of smart contract security:

1.5.1 *Theoretical Contributions*

Comprehensive Vulnerability Landscape Analysis: We provide the systematic dual-perspective analysis combining academic research and real-world exploits. Using Systematic Literature Review (SLR) and Systematization of Knowledge (SoK) methodologies, we analyzed 71 papers and 50 attacks to reveal fundamental misalignments between research focus and practical threats, offering insights that can redirect future research efforts toward more impactful areas.

Four-Level Root Cause Taxonomy and Exploit Chain Discovery: Through our SoK framework, the analysis of attack patterns yields a novel hierarchical taxonomy of root causes [3]. This framework categorizes vulnerabilities based on their origin. These include flawed economic design and protocol logic, protocol lifecycle and governance failures, external dependency vulnerabilities, and implementation-level weaknesses.

Importantly, our investigation reveals that successful attacks rarely exploit a single isolated vulnerability. Instead, they employ exploit chains where multiple weaknesses are combined strategically. For example, attackers often use access control flaws as entry points to enable price manipulation. They may also leverage reentrancy vulnerabilities to amplify the impact of oracle manipulation. This chained

exploitation pattern has been largely overlooked in prior literature. Previous work typically treats vulnerabilities as independent entities.

1.5.2 *Technical Contributions*

Semantic-Aware Taint Analysis: TaintSentinel introduces a novel approach to vulnerability detection that goes beyond pattern matching to understand semantic context. By tracking tainted data through complex execution paths and applying context-aware rules, it achieves an F1-score of 0.892 on balanced datasets, representing nearly a 4× improvement over existing tools [13].

Reinforcement Learning for Path Optimization: SmartTaintRL represents the first application of deep reinforcement learning to the smart contract vulnerability detection problem. Through hierarchical reward engineering and pattern-based learning, it reduces path exploration by 45% while maintaining 96% recall, effectively solving the path explosion problem that has limited previous approaches.

Vulnerability Localization: We develop a gradient-based attribution method that not only detects vulnerabilities but pinpoints their exact location at both function and node levels. This precise localization is crucial for developers who need actionable feedback to fix vulnerabilities.

1.5.3 *Practical Contributions*

Risk-Stratified Benchmark Dataset: We constructed the largest validated benchmark dataset for Bad Randomness (SWC-120) vulnerabilities, containing 1,758 labeled contracts categorized into four risk levels: HIGH_RISK (1,543 contracts with no protection), MEDIUM_RISK (37 contracts exploitable by miners), LOW_RISK (172 contracts exploitable only by owners), and SAFE (6 contracts using Chainlink VRF or commit-reveal). The dataset was constructed through a five-phase methodology including function-level validation, which revealed that 49% of contracts initially classified as protected were actually exploitable because the mitigation was applied to a different function than the vulnerable code. This dataset is 51× larger than RNVulDet (34 contracts) and provides the first risk-level classification for this vulnerability category.[2]

Labeled Dataset for Detection Methods: For training and evaluating our detection approaches, we extracted a subset from the benchmark with more stringent filtering criteria. Initial taint analysis on 4,844 Ethereum contracts from SmartBugs-Wild extracted 1.1 million execution paths [13]. Using context-aware analysis rules, we distinguish contracts that are truly vulnerable (394) from those using

2 The benchmark dataset is publicly available at: `https://github.com/HadisRe/BadRandomness-SWC120-Dataset`

blockchain values safely (4,450), such as for time-locks or access control. Manual validation by security experts confirmed 94% labeling agreement [13]. For reinforcement learning experiments, we applied quality filtering by removing paths with fewer than 3 nodes and contracts with insufficient complexity, resulting in 252,844 high-quality paths suitable for training machine learning models [24].[3]

1.6 thesis organization

This thesis is organized into four parts:

Part I: Foundation establishes the research context through three chapters. Chapter 1 presents the problem statement and research questions. Chapter 2 provides background on blockchain technology, smart contracts, and vulnerability types. Chapter 3 reviews related work in Bad Randomness detection and localization methods.

Part II: Empirical Analysis examines the vulnerability landscape. Chapter 4 systematically reviews academic literature on smart contract vulnerabilities. Chapter 5 analyzes real-world attacks to identify exploitation patterns.

Part III: Technical Solutions and Evaluation presents our benchmark dataset, detection approaches, and their validation. Chapter 6 introduces a risk-stratified benchmark dataset for Bad Randomness vulnerabilities, constructed through a five-phase methodology including function-level validation and context-aware refinement. Chapter 7 introduces TaintSentinel with semantic-aware taint analysis and context-sensitive rules, including experimental evaluation. Chapter 8 describes SmartTaintRL using reinforcement learning for intelligent path prioritization, along with performance assessment.

Part IV: Conclusions synthesizes research contributions and outlines future directions. Chapter 9 provides comprehensive analysis of findings, discusses limitations, implications for smart contract security, and identifies opportunities for future research.

---

3 The complete labeled dataset, taint analysis results, and path extraction data are publicly available at: `https://github.com/HadisRe/TaintSentinel`

2 BACKGROUND

2.1 INTRODUCTION

This chapter establishes the technical foundation necessary to understand Bad Randomness vulnerabilities in smart contracts. We present essential concepts across five interconnected layers. These are distributed ledger technology, Ethereum architecture, smart contract mechanics, vulnerability patterns, and analysis challenges.

Section 2.2 introduces blockchain fundamentals and Ethereum's multi-layer architecture. It emphasizes the deterministic execution model that creates the randomness problem. Section 2.3 examines smart contract lifecycle and types. It highlights immutability's security implications. Section 2.4 details the Bad Randomness vulnerability, its sources, and attack patterns. Section 2.5 reviews four main detection approaches and their limitations. Section 2.6 addresses the computational challenge of exhaustive path analysis in vulnerability detection. Together, these sections provide the theoretical foundation for understanding the Bad Randomness vulnerability and current detection challenges.

2.2 BLOCKCHAIN AND ETHEREUM FUNDAMENTALS

A **blockchain** is a distributed ledger that stores transactions in a series of interconnected blocks. Each block contains the hash of the previous block, making the chain immutable and impossible to alter [25]. **Ethereum** is a second-generation blockchain platform that extends beyond simple digital currency transfers. It supports smart contract execution and provides a platform for *decentralized applications* [26].

Each block in Ethereum consists of three main components that together form an immutable unit. The **Block Header** contains metadata information, including `prevHash` to link to the previous block, `stateRoot` which holds the root of the state tree, and `timestamp` which specifies when the block was generated. The **Transaction List** contains all transactions executed in this block. The outcome of each transaction is definitively determined [27]. The **State Root** is the root of a Merkle Patricia tree. It provides a concise representation of the current state of all accounts in the network, allowing for fast verification (Figure 2.1, Consensus Mechanism layer).

Ethereum uses two distinct account types with different natures and capabilities. **Externally Owned Accounts** (EOAs) are user accounts controlled by a private key. They contain two main fields: `nonce` to

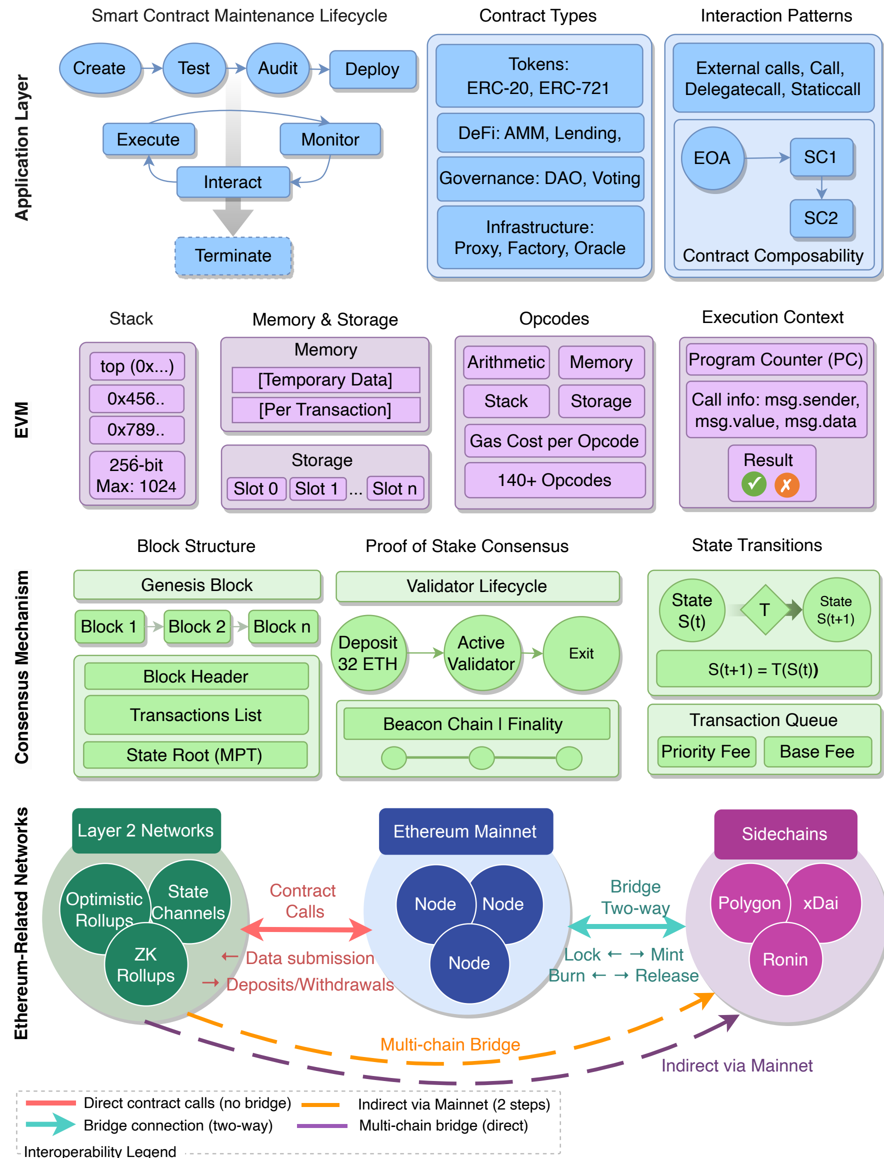

Figure 2.1: Ethereum multi-layer architecture showing the application layer (smart contract types and interaction patterns), EVM execution environment (stack, memory, storage, and opcodes), consensus mechanism (Proof of Stake with validator lifecycle and state transitions), and Ethereum-related networks (Layer 2 solutions, sidechains, and cross-chain bridges).

prevent replay attacks and `balance` to hold ETH balances. In contrast, **Contract Accounts** are smart contract accounts. In addition to `nonce` and `balance`, they have two additional fields: `code`, which contains the contract bytecode, and `storage`, which provides persistent memory for storing state variables. This fundamental difference between the two account types shapes Ethereum's security architecture [27, 28].

Ethereum successfully transitioned to a **Proof of Stake** (PoS) consensus mechanism in 2022, which consumes significantly less energy than

Proof of Work. In this mechanism, validators must deposit 32 ETH as stake to enter the validation cycle. The cycle consists of three stages. First, **Deposit**, where the validator locks up its funds. Second, **Active Validator**, where it generates and verifies blocks and receives rewards. Third, **Exit**, where the validator can leave the network and withdraw its stake. The **Beacon Chain**, which is the heart of the PoS mechanism, guarantees the finality of blocks. It ensures that validated blocks cannot be reversed (Figure 2.1, Proof of Stake Consensus section) [29].

Each transaction in Ethereum has a specific structure. It includes a sender address (`from`), a recipient address (`to`), the amount of ETH sent (`value`), a digital signature for authentication (`signature`), and a `gasLimit` that specifies the maximum computational cost allowed. These transactions are placed in the **Transaction Queue** before execution. They wait to be included in the next block by a validator. Ethereum's fee model consists of two components [30]. The **Base Fee** is automatically burned, gradually reducing the supply of ETH. The **Priority Fee** is paid directly to the validator and provides an incentive to prioritize transactions.

One of the fundamental features of Ethereum creates both power and limitations: the deterministic nature of transaction execution. Each transaction creates a state transition, represented by the formula $S(t+1) = T(S(t))$, where T is the transaction function and $S(t)$ is the current state of the system. This determinism means that executing a transaction on a given state will result in the same final state on all nodes in the network [25]. This property is essential for achieving distributed consensus. All nodes must be able to independently compute the same result (Figure 2.1, State Transitions section). However, this property makes true randomness impossible. It forms the basis of the Bad Randomness vulnerability, which will be discussed in detail in Section 2.4.

Ethereum today is a multi-layered ecosystem that extends beyond a single blockchain. **Layer 2 Networks** include solutions such as Optimistic Rollups, ZK Rollups, and State Channels. These process transactions off-chain but derive their security from the Mainnet. They send digested data to the Mainnet and allow for deposits and withdrawals of assets, significantly increasing throughput without compromising security. **Sidechains** such as Polygon, xDai, and Ronin are independent blockchains that connect to the Mainnet via two-way bridges. They use the Lock-Mint mechanism to transfer assets from the Mainnet to the sidechain and Burn-Release to return. **Multi-chain Bridges** also allow for direct connectivity between different networks (Figure 2.1, Ethereum-Related Network layer) [31].

While this multi-layer architecture provides significant scalability, it also introduces new security complexities. Cross-chain vulnerabilities account for approximately 10% of large attacks [13]. These are typically related to exploiting bridges or incompatibilities between chains [3].

This suggests that as the ecosystem expands, so does the attack surface, necessitating a more comprehensive security approach.

2.3 SMART CONTRACTS

Smart contracts are automated programs that run on a blockchain and enforce the terms of an agreement between parties without the need for an intermediary [32]. Unlike traditional software, smart contracts have two fundamental properties with important security implications. First, they are **immutable**. Once deployed on the blockchain, the contract code cannot be changed. This means that any vulnerabilities in the code remain permanent and cannot be patched [33]. Second, they are **autonomous**. Contracts execute automatically, and no central authority can stop them from executing, even if a vulnerability is exploited [34]. Together, these two properties greatly highlight the importance of pre-deployment testing and security auditing.

2.3.1 *Smart Contract Lifecycle*

Figure **??** illustrates the comprehensive lifecycle of smart contracts, organized in three distinct layers: off-chain development activities (top), on-chain blockchain operations (middle), and continuous lifecycle management (bottom).

The lifecycle begins with three critical off-chain phases. In the **Creation** phase, developers write contract code using Solidity, a high-level programming language specifically designed for Ethereum Virtual Machine (EVM) [32]. The **Testing** phase follows, where contracts are deployed on testnets like Goerli or Sepolia to verify functionality without risking real assets. The **Security Audit** phase is crucial due to blockchain's immutability—any vulnerability missed here becomes permanent once deployed [35].

The **Deployment** phase marks the critical transition from off-chain to on-chain, indicated by the red arrow in the figure. During deployment, a special transaction with an empty `to` field and contract bytecode in the `data` field is sent to the blockchain, requiring gas payment [33]. The contract receives a unique address and becomes permanently stored in a blockchain block.

Once deployed, the contract operates entirely on-chain. The **Execution** phase involves automatic function execution when triggered by transactions. Each execution is deterministic—identical inputs always produce identical outputs. The **Monitoring** phase runs continuously, capturing events emitted by the contract for debugging and tracking suspicious activities [36].

Two parallel processes support the deployed contract. **Interaction** occurs through External Owned Accounts (EOA) or decentralized applications (dApps), enabling users to invoke contract functions.

Upgrade/Termination represents the contract's end-of-life options: either upgrading via proxy patterns to modify logic while preserving the address, or termination through migration to new contracts (as `selfdestruct` is now deprecated) [34].

The blockchain layer itself consists of interconnected blocks containing transaction data, with each contract operation recorded immutably in this distributed ledger, ensuring transparency and permanence of all contract activities.

2.3.2 *Types of Contracts and Applications*

Figure 2.1 shows four main categories of smart contracts at the Application Layer, each with its own specific applications. The first category is **Token contracts**, which include the ERC-20 standard for fungible tokens such as USDT and DAI, and the ERC-721 standard for NFTs (non-fungible tokens) to represent unique assets [32]. The second category is **DeFi contracts**, which include AMM protocols such as Uniswap for automated token exchanges, and Lending protocols such as Aave and Compound for lending. This category has the highest total value locked (TVL) and is therefore a prime target for attacks [33].

The third category is **Governance contracts**, which include decentralized autonomous organizations (DAOs) and on-chain voting systems. The fourth category is **Infrastructure contracts** that play a supporting role. These include Proxy for upgradeability, Factory for automatically creating new contracts, and Oracle for receiving off-chain data such as market prices [34]. Each of these categories has its own security challenges and requirements [36].

2.4 BAD RANDOMNESS VULNERABILITY

One of the fundamental challenges in smart contract design is the generation of random numbers. This problem stems from the deterministic nature of the blockchain, which is necessary for achieving distributed consensus but makes it impossible to generate true randomness [17]. As explained in Section 2.2, the execution of any transaction in Ethereum must result in the same output on all nodes in the network for consensus to be achieved. This determinism means that every operation performed on the smart contract, given the same input, must produce the same output [37].

This fundamental contradiction presents a design dilemma. Consensus requires certainty, but randomness requires uncertainty. This problem has no simple solution [38]. Many decentralized applications, especially games and lotteries, require random numbers for their primary function. They need randomness to determine winners, produce unpredictable outcomes, or distribute prizes fairly. This practical need

forces developers to turn to sources that appear random but are in reality completely predictable or manipulable [17].

2.4.1 *Weak Random Sources in Ethereum*

Developers typically use three blockchain sources to generate "randomness," all of which have serious vulnerabilities. The first and most common source is `block.timestamp`, which indicates the time of block generation. This value is controlled by the miner or validator and can be manipulated within a few seconds [38]. The miner that produced the block can change the timestamp over a small period to achieve a desired result. Although Ethereum has reduced the manipulation window from 900 seconds to just a few seconds after recent upgrades, this is still sufficient for malicious miners to exploit timestamp-dependent contracts [37].

The second source is `block.number`, which indicates the current block number. Although this value cannot be directly manipulated, it is completely predictable [17]. Anyone can predict the next block number with perfect accuracy. The third source is `blockhash`, which returns the hash of the previous block. This value is also visible and predictable to everyone after the block is generated. In some cases, miners can influence it by not selecting undesirable blocks [39]. Some developers attempt to combine these sources (e.g., hashing timestamp with blockhash), but this provides no additional security as all components remain predictable or manipulable [38].

2.4.2 *Weak Randomness Attack Patterns*

Attacks on weakly random contracts fall into four main categories, each using a different attack vector [17, 37].

Miner Manipulation. A miner or validator can change blockchain values to their advantage. For example, a miner can advance or delay the timestamp by a few seconds, or decide not to generate a particular block if the lottery result is not in their favor [39]. The cost of this attack is relatively low while the potential profit can be very high. This attack vector is particularly concerning as miners have direct control over block parameters.

Front-Running. This is one of the most common and destructive attacks. The attacker observes lottery transactions in the mempool and calculates the lottery result before that transaction is included in the block [37]. If the result is favorable, the attacker sends a transaction with a higher gas fee to be executed before the original transaction. This attack is very practical due to the complete transparency of the mempool and the ability to prioritize transactions via gas fees. The Fomo3D attack in 2018, which resulted in losses exceeding $3 million,

exploited this vulnerability by combining front-running with bad randomness [37].

Timestamp Dependency. This attack occurs when the contract makes decisions based solely on `block.timestamp`. Since this value is controlled by the miner, malicious miners can change it over a small period (typically a few seconds in modern Ethereum) to achieve a desired result [38].

Block Hash Prediction. Since the `blockhash` is public after the block is generated, anyone can see it and use this information to perform an attack before the contract deadline [39]. Attackers can leverage the same random number generation process used by the contract to obtain identical results, as historical blocks never change.

Listing 2.1: Vulnerable lottery contract using weak randomness

```
contract VulnerableLottery {
    address[] public players;

    function enter() public payable {
        require(msg.value == 0.1 ether);  // Line 5: Entry fee
            check
        players.push(msg.sender);
    }

    function pickWinner() public {
        // VULNERABLE: Using block.timestamp as randomness source
        uint random = uint(keccak256(abi.encodePacked(block.
            timestamp)))  // Line 11
                      % players.length;
        address winner = players[random];  // Line 13: Winner
            selection
        payable(winner).transfer(address(this).balance);  // Line
             14: Prize transfer
        delete players;
    }
}
```

In Listing 2.1, the `pickWinner` function uses `block.timestamp` as the entropy source for winner selection. Line 11 demonstrates the vulnerability: hashing `block.timestamp` does not eliminate predictability, as the timestamp value itself is manipulable by miners [17]. An attacker can easily exploit this contract by writing a malicious contract that executes in the same block where `pickWinner` is called. This malicious contract performs the same calculation at line 11 and only calls the `enter` function if it determines that it will win. The attacker can predict the outcome before committing funds, effectively guaranteeing a win at line 13 and receiving the prize at line 14.

Unlike structural vulnerabilities, the Bad Randomness vulnerability depends on the context and how the blockchain values are used [17]. Using `block.timestamp` is perfectly safe in some cases. For example,

using it for time-locks or deadline checks is acceptable. However, using the same value to pick a winner in a lottery is a serious vulnerability [**ref13**]. This semantic distinction requires a deep understanding of the data flow and context of use, which goes beyond simple pattern matching. This context-dependent nature is precisely why existing static analysis tools struggle to detect Bad Randomness vulnerabilities effectively, as they lack the semantic understanding necessary to distinguish safe from unsafe usage patterns [37].

2.5 VULNERABILITY DETECTION TECHNIQUES

Vulnerability detection in smart contracts is a complex challenge that requires a combination of different analytical techniques. Existing methods can be divided into four main categories, each with its own approach to finding security issues. The choice of the appropriate method depends on the type of vulnerability, the complexity of the contract, and the balance between accuracy and computational cost.

2.5.1 *Static Analysis and Pattern Matching*

Developers can integrate it into their Continuous Integration/Continuous Deployment (CI/CD) pipeline quickly. However, simple static analysis has a fundamental limitation. It only examines the surface structure of the code and does not consider the semantic context [10]. It cannot detect whether an operation is safe in a particular context. For example, it flags the use of `block.timestamp` in all cases, whether it is safe for time-locks or unsafe for lottery draws. This problem leads to a high number of false positives, making it difficult for developers to identify real vulnerabilities.

2.5.2 *Symbolic Execution and Path Analysis*

A more advanced approach is symbolic execution, which uses mathematical symbols instead of actual values and examines all possible paths of execution [40, 41]. This method attempts to prove whether a dangerous state is reachable. It does this by constructing an execution tree. Each branch of the tree represents a conditional decision.

Symbolic execution's ability to examine all possible paths is both its strength and its weakness. As the number of conditional branches in the code increases, the number of possible paths grows exponentially. This problem is known as the path explosion problem [42]. A contract with only ten consecutive conditions can have over a thousand different paths. Analyzing all of them can take hours or even days. This limitation makes exhaustive symbolic execution impractical for complex contracts [41].

2.5.3 *Taint Analysis and Data Flow Tracing*

Taint analysis is a static analysis technique that traces the flow of potentially hazardous data through program execution paths [43]. This method is based on the principle that certain data is "tainted." Such data originates from untrusted sources, and its misuse can lead to vulnerabilities. In smart contracts, taint analysis marks suspicious input data (taint sources). It then examines how this data propagates through contract operations to sensitive locations (sinks). Tools like teEther [44] have demonstrated the effectiveness of taint analysis for automatically identifying exploitable paths in smart contracts.

Standard taint analysis operates in three phases: *source identification*, *propagation tracking*, and *sink detection*. The first phase identifies taint sources based on the vulnerability type being analyzed. In the context of smart contracts, sources can include blockchain values such as `block.timestamp` and `blockhash`. They can also include user inputs like `msg.sender` and `msg.value`, or results from external calls. For Bad Randomness detection specifically, blockchain values serve as primary taint sources. This is because they can be predicted or manipulated by miners.

The second phase, propagation tracking, follows the flow of tainted data through various operations [43]. If a tainted variable is used in a calculation, the result becomes tainted. If assigned to another variable, the new variable inherits the taint. If passed as a function argument, the return value may be tainted. The complexity of this phase arises from the fact that some operations may "sanitize" the taint. For example, hashing a tainted value may make it safe in certain contexts but remain dangerous in others.

The third phase, sink detection, identifies whether tainted data reaches dangerous program points. A sink is a location where using tainted data could trigger a vulnerability. For Bad Randomness, sensitive sinks include random number generation, fund transfers (`transfer`, `send`), prize assignments, and critical state modifications whose security depends on unpredictability. For instance, TeEther [44] uses taint analysis to trace data from arbitrary storage reads (sources) to critical operations like fund transfers (sinks). This enables automatic exploit generation.

Listing 2.2 demonstrates the three-phase taint analysis process in a vulnerable lottery contract.

Listing 2.2: Smart Contract Taint Analysis Example

```
contract VulnerableLottery {
    uint256 public prize = 10 ether;
    function play() public payable {
        require(msg.value == 1 ether);
        // Source: Taint originates here
        uint seed = block.timestamp;
```

```
        // Propagation: Taint flows through hash operation
8       uint random = uint(keccak256(abi.encode(seed))) % 2;
        if (random == 0) {
            // Sink: Tainted data controls Ether transfer
            payable(msg.sender).transfer(prize);
        }
13    }
}
```

In this example, line 6 shows the taint source where `block.timestamp` is read. Line 8 demonstrates propagation as the tainted value flows through a hash operation and modulo calculation. Line 10 reveals the sink where tainted data ultimately controls an Ether transfer. The winner selection depends entirely on the predictable timestamp value. An attacker can exploit this by writing a malicious contract that performs the same calculation. The attacker only participates when guaranteed to win.

However, traditional taint analysis has a fundamental limitation. It cannot distinguish between safe and unsafe uses of tainted data. For instance, using `block.timestamp` to enforce a deadline (`if block.timestamp > deadline`) is perfectly safe. In contrast, using it to select a winner (`winner = players[block.timestamp % length]`) is dangerous. Simple taint analysis flags both cases identically. This results in high false positive rates. This context-insensitivity problem motivates the need for more sophisticated analysis techniques. These techniques must understand the semantic context of data usage.

2.5.4 *Machine Learning Approaches*

In recent years, the use of machine learning for vulnerability detection has grown significantly [45, 46]. Rather than relying on manual rules, this approach uses training data to learn vulnerability patterns. Deep learning models can extract complex features from code that cannot be encoded by manual rules. For example, they can learn which code patterns are commonly associated with particular vulnerabilities, without having to explicitly define these patterns [46].

However, machine learning approaches also have their own challenges. They require high-quality, labeled training data, which is rare in the smart contract security space [45]. The problem of class imbalance is also serious. There are far fewer vulnerable contracts than safe contracts, which makes it difficult to train the model [45]. Additionally, deep learning models typically act as "black boxes" and do not explain why they have detected a contract as vulnerable. This is problematic for developers who want to fix the issue.

2.6 PATH EXPLOSION PROBLEM

One of the fundamental challenges in deep static analysis of smart contracts is the **path explosion problem** [47, 48, 49]. This occurs when trying to explore all possible paths for a program to execute. This problem stems from the exponential nature of path growth as the number of decision points in the code increases [50]. In the simplest case, each `if-else` condition doubles the number of possible paths. Thus, a program with n consecutive conditions can have up to 2^n different paths.

Consider a simple Control Flow Graph (CFG) with only three branch points. At the first point, the program can take two different paths. At the second point, each of these two paths splits into two more, making a total of four paths. At the third point, each of these four paths splits into two more, giving us a total of eight paths. This exponential growth quickly reaches very large numbers [51]. Ten conditions lead to 1,024 paths, twenty conditions lead to more than a million paths, and thirty conditions lead to more than a billion paths.

2.6.1 *Why Path Explosion Occurs in Smart Contracts*

Smart contracts are prone to path explosion for several reasons [52, 49]. First, these contracts typically contain a large number of conditions for security checks, input validation, and state management [53, 54]. A simple DeFi contract may have dozens of `require` conditions for checking balances, permissions, and constraints [55].

Second, loops make the problem worse [47]. A loop with m maximum iterations and k inner conditions can produce up to $2^{k \times m}$ different paths. Third, inter-contract calls increase the complexity dramatically [56]. The paths of the called contract must also be taken into account.

A real-world example of this problem is the analysis of a Uniswap router contract that performs swap operations. This contract involves various checks for input validation, calculation of different swap paths, slippage checks, and deadline management [53]. A full analysis of all possible paths in this contract can require examining millions of paths [48]. This can take days or weeks even with powerful hardware. Many of these paths are never executed in practice because the conditions required to achieve them are not feasible in the real world [52].

2.6.2 *Limitations of Exhaustive Analysis*

Exhaustive analysis that is, examining all possible paths is ideal in theory. It can guarantee that no vulnerabilities are missed [49]. In practice, however, this approach faces several fundamental limitations [47, 48].

The first limitation is computational time [52, 50]. Even with the use of optimal algorithms and smart pruning, exhaustive analysis of complex contracts can take weeks [48]. This is unacceptable for a development process that requires rapid feedback.

The second limitation is memory [56]. Maintaining the state of all the paths being examined can require a large amount of memory. Each path must maintain its own constraints, variable values, and call stack [57]. For large contracts with millions of paths, this can require hundreds of gigabytes of memory [56]. This is problematic even on powerful servers.

The third limitation is the problem of **state space explosion**, which is independent of path explosion but exacerbates it [58]. Even in a single path, the number of possible states can be very large, especially when dealing with arrays, mappings, and external calls [57, 56]. Combining path explosion with state space explosion creates a two-dimensional problem that is practically intractable [58, 55].

2.7 SUMMARY

This chapter established the technical foundation for understanding Bad Randomness vulnerabilities in smart contracts. We examined blockchain fundamentals and Ethereum's deterministic execution model. This model creates the core randomness problem. All miners must compute identical results. We reviewed smart contract mechanics including lifecycle stages and application types. We highlighted immutability as a critical security factor.

The chapter then detailed Bad Randomness vulnerability sources and attack patterns. We analyzed weak entropy sources such as `block.timestamp`, `blockhash`, and `block.number`. We reviewed four detection approaches: static analysis, symbolic execution, taint analysis, and machine learning. Each approach has limitations in detecting context-dependent vulnerabilities. Finally, we examined the path explosion problem that limits comprehensive analysis. These foundations prepare for Chapter 3. That chapter evaluates existing detection tools and identifies gaps that motivate our proposed solutions.

3 STATE OF THE ART IN BAD RANDOMNESS DETECTION AND LOCALIZATION

3.1 INTRODUCTION

This chapter provides a survey of detection and localization techniques for Bad Randomness vulnerabilities in smart contracts. We systematically evaluate existing approaches to understand their capabilities and limitations.

We examine three aspects. First, detection tools ranging from specialized analyzers to general-purpose frameworks. Second, taint analysis methods that form the foundation of most detectors. Third, vulnerability localization techniques at various code granularities. Our analysis reveals fundamental gaps in current approaches. These gaps appear particularly in semantic understanding, cross-transaction tracking, and path-sensitive analysis. These limitations prevent effective Bad Randomness detection.

The chapter is organized as follows. Section 3.2 categorizes and evaluates detection tools. Section 3.3 examines taint analysis implementations and their shortcomings. Section 3.4 reviews localization methods and their applicability to randomness vulnerabilities.

3.2 BAD RANDOMNESS DETECTION APPROACHES

Despite the critical importance of Bad Randomness vulnerability, which is ranked as the fourth most critical smart contract vulnerability in 2025 [3], existing detection tools have significant limitations [17]. This chapter provides a comprehensive review of existing approaches for detecting and localizing Bad Randomness vulnerabilities in Ethereum smart contracts.

We examine specialized detection tools, general-purpose static analysis frameworks, machine learning-based methods, and vulnerability localization techniques. Our analysis reveals significant gaps between existing capabilities and industry requirements. These gaps motivate the need for more advanced detection and localization methods.

3.2.1 *Specialized Detection Tools*

Despite the large number of smart contract security analysis tools developed by researchers and industry, only two focus specifically on detecting Bad Randomness. These are RNVulDet [17] and TON-

Scanner [59]. Table 3.1 presents a comprehensive comparison of these existing tools. It highlights their capabilities and limitations.

Table 3.1: Comparative Analysis of Bad Randomness Detection Tools

Tool	Type	Technique	Capabilities	Limitations	Status
RNVulDet [17]	Spec.	Taint + Tax.	BR pattern	Pattern-dep.	Active
TONScanner [59]	Spec.	Taint	BR detection	TON-specific	Active
Mythril [11]	Gen.	Symbolic	SWC-120	High FP	Active
Slither [10]	Gen.	Static	BR, Time. deps.	No deep anal.	Active
SmartCheck [60]	Gen.	XPath	Time. deps.	Limited	Deprecated
Oyente [61]	Gen.	Symbolic	Time. deps.	Sol. $\leq$0.4.19	Deprecated

Abbreviations: Spec.: Specialized, Gen.: General, Tax.: Taxonomy, Time.: Timestamp, deps.: dependencies, anal.: analysis, Sol.: Solidity, FP: False Positive

RNVulDet [17] represents the first systematic Bad Randomness analysis tool. It integrates three major components. First, a taxonomy builder. Second, a taint analysis engine that considers blockchain-sourced values as tainted. Third, an attack pattern recognizer that uses static code examination.

The tool performs taint analysis where blockchain-native values such as `block.timestamp`, `block.difficulty`, and `blockhash` are marked as taint sources. Sensitive operations like random number generation and fund transfers are identified as sinks.

TONScanner [59] adapts the detection approach specifically for The Open Network (TON) blockchain. It defines TON-specific block parameters (`cur_lt`, `block_lt`) as taint sources. It applies static analysis to identify randomness vulnerabilities specific to the logical time-based consensus mechanism of TON. The tool achieves 97% accuracy in detecting Bad Randomness vulnerabilities on TON smart contracts.

As shown in Table 3.1, only RNVulDet has been designed as a specialized tool for Bad Randomness. Other tools cover this vulnerability as a secondary feature. Among general-purpose tools, only Mythril and Slither remain active. SmartCheck and Oyente have been deprecated due to lack of updates [62].

3.2.2 *General-Purpose Static Analysis Tools*

General-purpose security analysis tools typically provide limited support for Bad Randomness detection. They rely on basic pattern matching or rule-based approaches.

Slither [10] employs a rule-based methodology with more than 92 detectors. However, Bad Randomness detection is only a minor component. The tool transforms smart contracts into an intermediate representation and performs static taint analysis to identify bugs.

Slither's weak randomness detector searches for modulo operations with block characteristics (e.g., `block.timestamp % n`). However, it fails to detect cases where values are stored in intermediate variables or used without modulo operators. Empirical studies on 4,844 real contracts from the Ethereum network showed that Slither had poor performance. It achieved an F1-score of only 0.232 [17].

Mythril [11] employs symbolic execution for vulnerability detection. The tool's predictable variables module only detects block properties when used directly in conditional expressions. It fails to capture complex data flows. Mythril achieved an F1-score of 0.236 on the same dataset, demonstrating similar limitations to Slither [17]. Further analysis revealed that these tools suffer from three fundamental problems. First, they only search for syntactic patterns without understanding the context of use. Second, they are unable to trace data flow at the storage level. Third, they lack path-sensitive analysis to distinguish between safe and unsafe paths [17].

Securify [63] performs automated analysis of Solidity contracts with basic randomness checks. The tool employs basic rule-based detection without considering true security consequences or usage context.

Oyente and Osiris [61, 64] were among the first tools to use symbolic execution for smart contract analysis. However, they focus on reentrancy and overflow vulnerabilities. Bad Randomness detection is secondary. Additionally, these tools have become deprecated as they only support older Solidity versions ($\leq$ 0.4.19).

3.2.3 *Machine Learning and LLM-Based Approaches*

Progress has been made through machine learning strategies. However, these approaches lack specialization for Bad Randomness detection.

Lightning Cat [65] achieved a 93.53% F1-score using CodeBERT, LSTM, and CNN models for general vulnerability detection. The tool employs deep learning-based solutions but does not specifically target Bad Randomness vulnerabilities.

LLM-based approaches include GPTLens [66] and SmartLLMSentry [67], which achieved 91.1% accuracy. These tools use large language models for vulnerability detection but lack specialized handling of randomness-related issues.

QuillShield [68] represents AI fusion with hybrid consensus. However, it addresses randomness broadly without deep analysis of context-dependent patterns.

3.2.4 *Dynamic Analysis and Fuzzing Approaches*

Dynamic analysis tools provide limited coverage of Bad Randomness detection.

Echidna [69] implements property-based fuzzing. The tool can detect Bad Randomness only through property violation. This requires users to specify relevant properties. Developers must manually write assertions about expected randomness behavior. This makes automated detection impossible.

Medusa [70] adds parallel processing to improve fuzzing efficiency. However, it has no built-in randomness vulnerability detection. Like Echidna, it requires manual property testing. These tools cannot provide automated Bad Randomness detection. Fuzzing tools cannot distinguish between legitimate and vulnerable uses of blockchain values without explicit guidance from developers.

3.3 TAINT ANALYSIS METHODS FOR SMART CONTRACTS

Taint analysis tracks data flow from untrusted sources to sensitive operations. This technique is well-suited for Bad Randomness detection because it can trace blockchain values from their origin to points where they affect randomness or fund distribution. Section 2.5.3 introduced the three-phase taint process: source identification, propagation tracking, and sink detection.

This section examines which tools employ taint analysis, focusing on tools not previously detailed. We assess their applicability to Bad Randomness detection and identify gaps that motivate the advanced approaches presented in Part III.

3.3.1 *Taint-Based Tools Overview*

Among the tools reviewed earlier, only RNVulDet [17] were specifically designed with taint analysis for Bad Randomness (see Table 3.1). Slither [10] includes basic taint tracking but achieves only F1=0.232 due to context-insensitivity. Beyond these, several general-purpose tools employ taint analysis for other vulnerability types. We examine three representative tools that demonstrate both the potential and limitations of taint-based approaches.

3.3.1.1 *TeEther: Exploit-Oriented Taint Analysis*

TeEther was among the first tools to apply taint analysis systematically to smart contracts [44]. It generates exploits automatically by identifying critical paths from arbitrary storage reads to fund transfers, using symbolic execution combined with taint tracking to find exploitable vulnerabilities.

The approach works in two phases. First, critical instructions that can send Ether are identified. Second, backward taint analysis is performed from these critical points to find controllable inputs. If an

attacker can control the path from input to fund transfer, a working exploit is generated automatically.

TeEther demonstrated high effectiveness for its intended purpose, discovering previously unknown vulnerabilities in deployed contracts and generating 14 exploits with a total of 23,121 USD at risk. However, it was not designed for Bad Randomness detection. Its taint model treats all attacker-controllable inputs equally without distinguishing blockchain values from user inputs. It also lacks the semantic understanding needed to identify randomness-specific patterns.

3.3.1.2 *Securify: Pattern-Based Compliance Checking*

Securify implements a compliance and violation pattern framework [63]. It uses Datalog-based analysis to check if contracts satisfy security properties, employing taint-like dependency tracking to determine which operations depend on untrusted inputs.

Patterns are defined as combinations of compliance and violation rules. For example, a compliance pattern might state that fund transfers should not depend on arbitrary inputs, while a violation pattern detects when this rule is broken. Contracts are checked against 37 predefined patterns covering various vulnerability types.

For timestamp dependencies, Securify can flag operations that depend on `block.timestamp`. However, its pattern-based approach lacks semantic context and cannot determine whether the dependency is safe (deadline checks) or dangerous (randomness generation). This limitation causes false positives, as the framework reports potential issues without understanding the actual security implications. This makes it unsuitable for accurate Bad Randomness detection where context is critical.

3.3.1.3 *Ethainter: Context-Sensitive Access Control Analysis*

Ethainter represents a significant advancement in taint analysis for smart contracts [71]. It performs context-sensitive data-flow analysis to detect access control vulnerabilities, achieving high precision by understanding the semantic meaning of data flows.

The key innovation is context-aware taint propagation that distinguishes between different uses of the same variable. For example, it recognizes that `msg.sender` used in access checks is fundamentally different from `msg.sender` used in event logging. This semantic understanding significantly reduces false positives.

Both intra-procedural flows (within functions) and inter-procedural flows (across function calls) are analyzed. A complete data-flow graph is built for each contract, and domain-specific rules are applied to identify access control violations. On a dataset of real contracts, Ethainter achieved 98.7% precision for access control vulnerabilities.

Despite its sophistication, Ethainter focuses exclusively on access control issues. Its design does not extend to Bad Randomness patterns. While its success demonstrates that context-sensitive taint analysis is both feasible and effective, it also shows that such analysis requires careful domain-specific adaptation. The rules and patterns for access control differ fundamentally from those needed for randomness detection.

3.3.2 *Gap Analysis: Why Existing Taint Analysis Falls Short*

Table 3.2 compares taint capabilities across all the reviewed tools. The analysis reveals three critical gaps that explain poor detection accuracy.

Table 3.2: Comparison of Taint Analysis Features

Feature	teEther	Securify	Ethainter	RNVulDet	TONScan.	Slither
Basic taint	✓	✓	✓	✓	✓	✓
Context-aware	×	×	✓	×	×	×
Storage track.	×	×	×	×	×	×
Path-sensitive	Partial	×	Partial	×	×	×
BR patterns	×	Minimal	×	✓	✓	Partial
Performance	Good	Good	Good	Fast	Fast	Fast
BR Detection	No	Poor	No	Good	TON only	Poor
F1-Score (BR)	—	—	—	0.68	0.97*	0.232

Note: ✓ = supported, × = not supported, BR = Bad Randomness, *TON blockchain only. F1-scores from empirical evaluation [17].

3.3.2.1 *Context-Insensitivity Problem*

Most taint-based tools treat all uses of `block.timestamp` identically, flagging every occurrence as potentially dangerous. This creates excessive false positives because many uses are perfectly safe.

Consider two cases. First, enforcing a deadline:

```
1 require(block.timestamp >= withdrawalDeadline);
```

Second, selecting a winner:

```
winner = players[block.timestamp % players.length];
```

Tools like Slither, Securify, and even RNVulDet flag both cases. In the first safe miners can delay execution slightly but cannot prevent legitimate withdrawals. In the second vulnerable miners can predict and manipulate the outcome.

Only Ethainter demonstrates true context-sensitivity, understanding semantic differences in data usage. However, its domain-specific design for access control does not transfer to Bad Randomness. The

patterns and rules differ fundamentally, showing that context-sensitive analysis requires careful adaptation for each vulnerability class.

3.3.2.2 *Storage-Level Tracking Gap*

No reviewed tool tracks taint propagation through contract storage across transactions. This is a critical limitation for Bad Randomness detection, as many contracts store random seeds in one transaction and consume them later. Consider this common pattern in gaming contracts:

```
// Transaction 1: Store seed
function initGame() public {
    gameSeed = block.timestamp;
}

// Transaction 2: Use seed
function revealWinner() public {
    winner = players[gameSeed % players.length];
}
```

Traditional taint analysis operates within single transactions and cannot connect the storage write in `initGame` to the read in `revealWinner`. The taint chain breaks at storage boundaries, allowing vulnerabilities to remain completely undetected.

TeEther performs backward analysis from critical operations but still operates within single execution traces. Securify's Datalog analysis is transaction-bound. Even RNVulDet, despite being BR-specific, lacks cross-transaction tracking. This represents a fundamental gap in all current approaches.

3.3.2.3 *Path-Insensitivity*

Most tools operate at the contract level without path-specific reasoning. They report that tainted data reaches a sink without checking if that path is actually reachable. Many flagged vulnerabilities exist only in dead code or unreachable paths.

TeEther makes some progress here, using symbolic execution to verify path feasibility before generating exploits. However, this approach faces the path explosion problem. Complex contracts have millions of possible paths, making exhaustive analysis intractable.

Path-sensitive analysis could dramatically reduce false positives by filtering unreachable paths automatically and prioritizing high-risk paths for manual review. However, this requires solving path explosion while maintaining detection recall. We address this challenge in Chapter 8 using reinforcement learning.

3.3.3 *Summary and Implications*

Our analysis reveals a fundamental tension in taint-based vulnerability detection. Simple approaches like Slither achieve fast analysis but suffer from high false positive rates (F1=0.232). Sophisticated approaches like Ethainter achieve high precision but require domain-specific adaptation.

For Bad Randomness, existing tools face three critical limitations:

1. **Context-insensitivity:** Cannot distinguish safe from unsafe usage of blockchain values
2. **Storage-tracking gap:** Cannot follow taint propagation across transactions
3. **Path-insensitivity:** Cannot determine if flagged issues are actually exploitable

These gaps explain why even specialized tools like RNVulDet achieve only moderate accuracy (F1=0.68). The problems are not fundamental to taint analysis itself. Ethainter proves that context-sensitive analysis works well when properly designed. Rather, the issue is that no tool combines the necessary features for accurate Bad Randomness detection.

Part iii of this thesis addresses these limitations. Chapter 7 introduces TaintSentinel, which implements context-aware taint propagation with graduated sensitivity levels and incorporates domain-specific rules that distinguish safe from unsafe patterns. Chapter 8 presents SmartTaintRL, which tackles path explosion through reinforcement learning. Together, these contributions enable accurate Bad Randomness detection with low false positive rates.

3.4 VULNERABILITY LOCALIZATION METHODS

3.4.1 *Evolution of Localization Techniques*

Vulnerability localization methods in smart contracts have evolved through four distinct generations, with each generation showing advancements over the previous one [17]. While these methods have been primarily developed for general vulnerabilities, their application to Bad Randomness localization remains unexplored.

Table 3.3 presents the generational evolution of vulnerability localization methods.

The analysis of Table 3.3 shows that while the first generation relied mainly on static rules, subsequent generations gradually adopted more sophisticated machine learning techniques. The third generation marks a turning point with the introduction of Graph Neural Networks (GNN), achieving F1-scores above 90%. The fourth generation, by

Table 3.3: Generational Evolution of Vulnerability Localization Methods

Generation	Approach	Representatives	Innovations	Performance
G1 (20-21)	Rule + ML	sGuard [72], EVMPatch [73], SmartShield [74]	Auto-repair, Bytecode	Limited rules
G2 (20-22)	Static + ML	SmartState [75], MVD-HG [76], sGuard+ [77]	State, AST+ CFG+DFG	87.23% precision
G3 (22-24)	GNN	G-Scan [78], HGAT [79], Lightning Cat [65]	Hierarchical, End-to-end	93% F1
G4 (24-25)	AI + LLM	MCR-VD [80], GPTScan [81]	CPG, GPT	14-90% improvement

combining Large Language Models (LLM) and static analysis, offers unprecedented accuracy but faces computational cost challenges.

3.4.2 *Granularity Levels in Vulnerability Localization*

Different localization approaches operate at various levels of granularity, each with specific trade-offs between precision and computational cost. Table 3.4 presents the granularity levels applicable to Bad Randomness localization.

Table 3.4 demonstrates that line-level methods such as G-Scan and MCR-VD provide the highest accuracy and are suitable for developers who need precise identification of vulnerability locations. In contrast, bytecode-level methods like EVMPatch enable immediate repair of deployed contracts, which is critical for emergency situations.

Line-level localization achieves the highest precision by pinpointing the exact source code line containing the vulnerability. Tools like MCR-VD and G-Scan achieve 93.69% F1-score, making them ideal for precise identification of weak RNG sources [80, 78].

Function-level localization identifies vulnerable functions with 85.64% accuracy, providing a good balance between precision and computational efficiency [79]. This level is particularly useful for Bad Randomness as vulnerabilities often span multiple statements within a function.

Statement and AST node level localization enables structural pattern matching, which can identify complex vulnerability patterns across multiple code constructs [77].

Bytecode-level methods operate on deployed contracts, enabling runtime patching without access to source code. This is particularly

Table 3.4: Granularity Levels in Vulnerability Localization

Level	Tools	Accuracy	Speed	BR Application
Line	MCR-VD [80], G-Scan [78]	93.69% F1	1.2-2.3s	Precise weak RNG source identification
Function	HGAT [79]	85.64%	∼1s	Vulnerable function detection
Statement	HGAT [79]	—	—	`block.timestamp` usage analysis
AST node	sGuard+ [77]	Variable	—	Structural pattern matching
Bytecode	EVMPatch [73]	—	Immediate	Runtime BR patching
Source	sGuard [72]	—	—	Source code randomness repair

valuable for emergency response to actively exploited vulnerabilities [73].

A careful examination of Tables 3.3 and 3.4 reveals a critical gap: none of the advanced localization methods have been specifically designed for Bad Randomness. While MCR-VD and G-Scan show excellent performance in localizing reentrancy and integer overflow vulnerabilities, their capability to precisely identify the location of weak randomness sources remains unexplored.

3.5 SUMMARY

This chapter provided a survey of detection and localization techniques for Bad Randomness vulnerabilities. We categorized detection approaches into specialized tools such as RNVulDet, general-purpose static analyzers such as Slither and Mythril, and dynamic analysis methods including fuzzing. Our evaluation revealed that existing tools achieve poor performance. Slither reaches F1-score of only 0.232. Mythril achieves 0.236. These limitations stem from reliance on syntactic pattern matching without semantic understanding.

We examined taint analysis methods that form the foundation of most detectors. Three fundamental gaps were identified. First, existing tools lack semantic context to distinguish safe from unsafe usage patterns. Second, no tool tracks taint propagation across storage and transactions. Third, path-sensitive analysis remains absent. We also reviewed localization techniques at contract, function, and line levels. Current methods provide insufficient granularity for actionable

remediation. These gaps directly motivate our proposed solutions: TaintSentinel's semantic-aware taint analysis in Chapter 7 and SmartTaintRL's path-level detection in Chapter 8.

Part II

EMPIRICAL STUDIES

4 SYSTEMATIC LITERATURE REVIEW OF SMART CONTRACT VULNERABILITIES

4.1 INTRODUCTION

While catastrophic attacks on Ethereum persist, vulnerability research remains focused on implementation-level smart contract bugs, creating a gap between academic understanding of vulnerabilities and the true root causes of high-impact real-world incidents. The Ethereum blockchain and its smart contract ecosystem have grown to manage hundreds of billions of dollars [82, 83]. Along with this growth, in recent years, a relentless series of successful high-profile hacks has occurred, resulting in staggering financial losses [7, 84]. Some reports suggest up to $2.2 billion stolen from protocols in 2024 [**chainalysis**]. In response to such attacks, a substantial body of research has been performed to classify and detect smart contract vulnerabilities. These efforts focused on implementation-level bugs such as reentrancy and integer overflows [41, 39]. This led to the development of numerous static and dynamic analysis tools like Slither [10] and Echidna [69] or even execution frameworks to run a swarm of analysis tools on a contract, such as SmartBugs [85]. Despite their usefulness in detecting certain classes of vulnerabilities [86], both the severity (total funds stolen) and number of successful attacks have increased in recent years [**chainalysis**].

Table 4.1 presents a comprehensive review of three Systematization of Knowledge (SoK) and ten Systematic Literature Review (SLR) papers on Decentralized Application (DApp) security. Surveys and SoKs classify tens of vulnerabilities across different layers of the blockchain stack (e.g., network, smart contract, and protocol layers) [87, 88, 35]. These taxonomies have become increasingly granular in classifying weaknesses [88] and mapping them to formal verification properties [89], or aligning them with frameworks such as NIST bugs [90]. However, our review unveils a few gaps in this literature: First, **prevention gap**: only 4 of 13 works (S4, S9, S12, and S13) address prevention methods, most of which focus exclusively on detection. Second, the **validation gap**: merely 3 works validate their findings against real-world exploit S3 (case studies), S9 (181 exploits) and S12 (92 exploits), while others rely on synthetic datasets. Third, the **taxonomy evolution** shows a shift from simple categorization to sophisticated multi-layer frameworks (4L, 5L), yet none adopt a root-cause approach based on real incidents.

Table 4.1: Comparison of smart contract security SLRs and SoKs

Paper	Yr	Vul	Prv[b]	Tax[a]	Wld[c]	Key Strength
S1	'20	40	P	4L	–	Vuln. ↔ attack mapping
S2	'21	20	×	3C	–	Comprehensive tool analysis
S3	'21	8	×	NI	C	Maps vuln.→attack→detection
S4	'21	202	✓	FM	–	Formal methods for SC verification
S5	'22	6	×	DT	–	Detection tools quality gaps
S6	'22	24	×	RS	–	Causal hierarchy for prevention
S7	'23	12	×	3L	–	Solidity/EVM/Blockchain layers
S8	'23	6	×	TB	–	6 vulns + tool assessment
S9*	'23	50+	✓	5L	181	181 exploits across 5 layers
S10	'24	94	×	HR	E	Hierarchical taxonomy + ODC
S11*	'24	V	×	UN	–	ER model for unifying taxonomies
S12*	'24	45	✓	4L	92	Vulnerabilities in real attacks
S13	'25	20+	✓	5L	–	5-layer attack classification
Ours	**'25**	**29**	✓	**RC**	**50**	**Root-cause framework + incidents**

Legend: *=SoK. [a]Tax: xL=x-Layer, xC=x-Category, FM=Formal, DT=Detection, RS=Root-cause, TB=Tool, HR=Hierarchical, UN=Unified, RC=Root-Cause. [b]Prv: ✓=comprehensive, P=partial, ×=none. [c]Wld: Number=exploit count, C=case study, E=examples, –=none.

These observations motivated our dual-pronged approach. First, we conduct a SLR to establish a comprehensive baseline of academically recognized vulnerabilities, their manifestation patterns, and evidence-based mitigation strategies. This addresses the need for an up-to-date catalog that consolidates knowledge from 71 seminal papers while distinguishing between active threats and deprecated vulnerabilities that have been mitigated through protocol upgrades or compiler improvements. Second, recognizing the validation gap in existing literature, subsequent chapters will analyze real-world incidents to bridge the gap between academic classifications and practical attack vectors observed in the wild.

The remainder of this chapter is organized as follows. Section 4.2 details our systematic literature review methodology, including study identification, selection criteria, and quality assessment procedures following Kitchenham's guidelines [91]. Section 4.3 presents our comprehensive catalog of 25 active smart contract vulnerabilities, organized by mechanism families, with detailed code examples and evidence-based mitigation strategies for each. We prioritize vulnerabilities with highest severity in real-world attacks, greatest prevalence, or strongest relevance to data-flow and taint analysis methodologies central to this thesis. Section 4.4 examines five vulnerabilities that have been deprecated through protocol-level upgrades (EIP-6780, EIP-150), compiler improvements (Solidity 0.5.0+, 0.8.0+), or re-contextualization based on widespread usage, demonstrating the evolution of the Ethereum security landscape.

4.2 RESEARCH PROTOCOL

We followed a rigorous protocol to ensure comprehensive coverage and quality control. Our review process consisted of two main phases: study identification and study selection. This approach follows Kitchenham's established guidelines [91] for conducting systematic literature reviews in software engineering.

4.2.1 *Study Identification*

The PICOC framework guided our search strategy development [91]. PICOC stands for Population, Intervention, Comparison, Outcome, and Context. This framework enabled systematic keyword definition. Table 4.2 shows our keyword extraction.

Table 4.2: Keyword extraction method using PICOC framework

Category	Keyword	Synonyms
P	Ethereum smart contracts	Ethereum contracts, Solidity
I	Vulnerabilities	Security Flaws OR Security Weaknesses
O	Mitigation	Classification, Mitigation, Prevention, Categorization, classification, taxonomy

The Population category defined our technology focus: smart contracts and blockchain. The Intervention category specified the problem under investigation. Synonyms were added to main keywords to ensure comprehensive search coverage. Our final search query combined all keywords from P, I, and O categories along with their synonyms.

Five major databases were searched: ACM Digital Library, IEEE Xplore, Scopus, Springer, and Google Scholar. The latter helped capture pre-prints, theses, and gray literature. Our initial search returned 17,013 papers with the following breakdown by source: ACM (395), IEEE Xplore (541), Scopus (74), Springer (7,113), and Google Scholar (8,890).

4.2.2 *Study Selection*

Systematic filtering was applied to the 17,013 initially identified papers. Our exclusion criteria are listed in Table 4.3. These criteria ensured selection of only relevant, high-quality studies.

The selection process went through four distinct stages, as illustrated in Figure 4.1.

In the first stage, duplicates were removed and language filters applied using EC1 (year filter) and EC2 (English language) criteria, reducing the corpus to 14,328 papers.

Table 4.3: List of exclusion criteria

ID	Exclusion Criteria
EC1	The article was published before 2018
EC2	The article is written in a language other than English
EC3	The article focuses on vulnerabilities in platforms other than Ethereum
EC4	There is no full-text version of the article available
EC5	The article is a book chapter or editorial notes
EC6	The article does not concern any of these topics: survey of smart contract vulnerabilities, security of smart contracts, vulnerability detection or mitigation methods for smart contracts, demonstrates findings for smart contracts through simulation or empirical evidence

Note: The vulnerabilities before 2018 are included via more recent papers.

The second stage involved title and abstract screening using EC3 (Ethereum platform focus), EC5 (excluding book chapters and editorials), and EC6 (topic relevance). This screening phase yielded 2,186 papers.

Full-text assessment was performed in the third stage, checking EC4 (availability criterion). Only papers with accessible full-text versions proceeded, refining the selection to 350 papers.

The fourth and final stage conducted quality-based evaluation through manual review of each paper. We examined empirical validation and novel contributions. This rigorous assessment resulted in 71 papers for in-depth analysis, which form the foundation of our vulnerability catalog.

4.3 ACTIVE VULNERABILITIES CATALOG

Our systematic literature review of 71 seminal papers identified 29 distinct smart contract vulnerabilities. Through rigorous analysis, we determined that 5 of these have become obsolete. This obsolescence stems from three factors. First, protocol-level upgrades (EIP-6780 for `selfdestruct`, EIP-150 for call-stack depth). Second, compiler improvements (Solidity 0.5.0 mandating visibility, Solidity 0.8.0 adding overflow checks). Third, reconceptualization based on widespread usage (upgradability is now viewed as a design pattern rather than an inherent vulnerability). This refinement yielded a catalog of **24 active smart contract vulnerabilities** that continue to pose demonstrable threats to deployed contracts as of 2025.

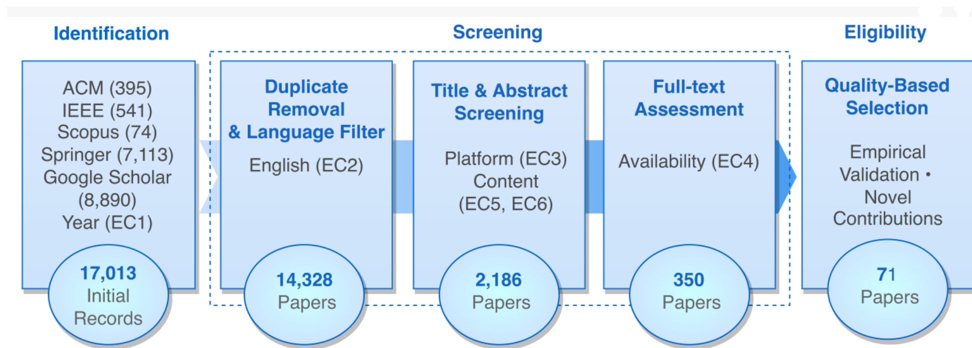

Figure 4.1: Systematic review filtering process showing the reduction from 17,013 identified papers to 71 quality-validated papers. The process moved through four stages: Identification, Screening, Eligibility assessment, and Quality-based selection. Source: Adapted from [3].

For each vulnerability in our catalog, we provide three essential components. First, real-world incident examples or proof-of-concept demonstrations. Second, the vulnerable code patterns that enable exploitation. Third, evidence-based prevention and mitigation strategies synthesized from academic literature and validated through expert review.

Given the scope of this thesis and our focus on Bad Randomness detection, we present detailed analysis of the 11 most critical vulnerabilities below. These represent vulnerabilities with the highest severity in real-world attacks (based on our 50-incident analysis in Chapter 5), greatest prevalence, or strongest relevance to our research on taint analysis and data-flow detection methodologies. The complete catalog including all 25 active vulnerabilities with comprehensive code examples and mitigation strategies is available in our companion repository[1] and detailed in our prior work [3].

4.3.1 *Integer Over/Underflow and Rounding Errors*

These vulnerabilities stem from Solidity's limitations in handling numerical data types. Integer overflow occurs when an arithmetic operation produces a result that exceeds the maximum value of the variable's type. Similarly, underflow occurs when the result falls below the minimum value. In both cases, the value wraps around rather than being capped at the type's bounds.

This issue commonly affects unsigned integers. For instance, in Listing 4.1, the balance stored in the `balances` mapping will wrap to zero if an operation causes it to exceed the maximum value of $2^{256} - 1$.

Rounding errors represent a related problem. Solidity performs integer division, which truncates fractional results. This truncation leads

1 https://github.com/HadisRe/SoK-Root-Cause-of-Smart-Contract-Incidents

to inconsistent calculation results, especially in operations requiring precision [64].

Listing 4.1: Integer overflow/underflow in Solidity

```
contract IntegerOverflowExample {
    mapping(address => uint256) public balances;
    uint256 public constant MAX_UINT = type(uint256).max;

    function deposit(uint256 amount) public {
        require(amount > 0, "Amount must be > zero");
        balances[msg.sender] += amount;
    }

    function withdraw(uint256 amount) public {
        require(amount > 0, "Amount must be greater than zero");
        require(balances[msg.sender] >= amount, "Insufficient");
        balances[msg.sender] -= amount;
    }

    function forceOverflow() public {
        balances[msg.sender] = MAX_UINT;
        balances[msg.sender] += 1;
    }
}
```

In Listing 4.1, the function `forceOverflow` attempts to assign the maximum possible value to a user's balance and then increase it by one. In earlier Solidity versions (<0.8.0), this would have resulted in an integer overflow. Similarly, the `withdraw` function reduces the user's balance. In older versions, if unchecked, this operation could lead to an integer underflow. Solidity 0.8.0 and later versions mitigate these issues by introducing built-in overflow and underflow checks. These checks cause such operations to revert automatically.

Mitigation/Prevention. To prevent arithmetic vulnerabilities in Solidity smart contracts, developers have several options. Prior to Solidity 0.8.0, the SafeMath library [92] was commonly used to detect and prevent overflows and underflows. When an overflow or underflow occurs, SafeMath reverts the transaction. However, this approach has limitations. Developers may apply it inconsistently, and it increases gas costs.

Solidity 0.8.0 and later versions eliminate the need for SafeMath by performing these checks at the compiler level [64, 93]. This built-in protection is automatic but still introduces some runtime overhead. To prevent precision loss from integer division, developers can use the FullMath library [94].

Another prevention strategy involves using appropriate data types [93]. Choosing data types with sufficient capacity helps prevent overflow and underflow issues. For example, using `uint256` instead

of `uint8` provides greater capacity and reduces the risk of these vulnerabilities.

4.3.2 *Improper Handling of External Calls*

Also known as "Exception Disorder" or "Unchecked Call Return Value", this vulnerability arises when a contract ignores the boolean success flag returned by a low-level external call. In the EVM, several operations handle external calls: `call`, `delegatecall`, `staticcall`, `callcode`, and the wrappers `send` and `transfer`. These operations never propagate a revert from the callee. Instead, they return a tuple `(success, data)`, where `success` is true on completion and false if the callee runs out of gas, deliberately reverts, or encounters any other error. Failure to test this flag allows the caller to proceed with an invalid state, potentially locking Ether or corrupting storage.

Two particularly common manifestations are documented in the literature [95, 96]. First, when Ether is transferred with `send` or `transfer`, only a 2,300-gas stipend is forwarded. Any recipient whose fallback or receive function requires more gas will revert. However, the sender sees only a false return value. If the caller ignores that flag, the Ether remains trapped in the sending contract. In some cases, this becomes irreversible.

Second, any low-level call that proceeds without checking `success` suffers the same silent-failure risk. This applies whether the call uses `call`, `delegatecall`, `staticcall`, or `callcode`. The callee's error is suppressed, control returns to the caller, and subsequent logic executes under false assumptions.

Prevention/Mitigation. Developers should verify the return value of external calls. For example, using `require(sendResult, "Transfer failed")` after `send` ensures that failures are caught. Additionally, `transfer` is recommended over `send`, as `transfer` automatically reverts the entire transaction in case of failure [95].

Since Solidity 0.6.0, the `try/catch` structure provides a more robust way to handle errors in external calls. Furthermore, adopting well-established standards such as ERC20 and ERC721 is advised. These standards implement secure transaction mechanisms [97].

4.3.3 *Reentrancy*

The reentrancy vulnerability occurs when functions are maliciously reentered (e.g., through fallback functions) [98]. If the attacker is able to bypass the validity checks in the target contract, they can reenter the callee function maliciously. The vulnerability arises due to two main reasons: first, a contract's control flow decision relies on some of its state variables that should be updated by the contract itself, but are not, before calling another contract; and second, there is no gas

limit when handing over the control flow to another contract. The Decentralized Autonomous Organization (DAO) attack exploited this vulnerability in 2016. As shown in Figure 4.2, the original contract includes functions for depositing and withdrawing Ether. The primary issue lies in the `withdraw` function, where the user's balance is set to zero after Ether has been sent to their address. This sequence allows an attacker to repeatedly call `withdraw` before the balance update occurs for multiple unauthorized withdrawals. In the attack contract, the attacker leverages the fallback function to invoke `withdraw` repeatedly before the balance is adjusted, receiving multiple illegal Ether payments.

```
1  contract DepositFunds {
2   mapping(address => uint)
3    public balances;
4   function deposit()
5    public payable {
6    balances[msg.sender]
7     += msg.value;
8   }
9   function withdraw()
10   public {
11   uint bal =
12    balances[msg.sender];
13   require(bal > 0);
14   (bool sent,) =
15    msg.sender.call
16    {value: bal}("");
17   require(sent,
18    "Failed to send!");
19   balances[msg.sender]=0;
20  }
21 }
```

(a) Vulnerable Contract

```
1  contract Attack {
2   DepositFunds public
3    depositFunds;
4   constructor(address a){
5    depositFunds =
6     DepositFunds(a);
7   }
8   fallback()
9    external payable {
10   if(address(
11    depositFunds).balance
12    >= 1 ether) {
13    depositFunds
14     .withdraw();
15   }
16  }
17  function attack()
18   external payable {
19   depositFunds.deposit
20    {value: 1 ether}();
21   depositFunds.withdraw();
22  }
23 }
```

(b) Attack Contract

```
1  contract SecureDeposit {
2   mapping(address => uint)
3    public balances;
4   function deposit()
5    public payable {
6    balances[msg.sender]
7     += msg.value;
8   }
9   function withdraw()
10   public {
11   uint bal =
12    balances[msg.sender];
13   require(bal > 0);
14   balances[msg.sender]=0;
15   (bool sent,) =
16    msg.sender.call
17    {value: bal}("");
18   require(sent,
19    "Failed to send!");
20  }
21 }
```

(c) Fixed Contract

Figure 4.2: Vulnerable ↔ attack contract pairs and the patched contract involving in reentrancy vulnerability.

The reentrancy vulnerability occurs when functions are maliciously reentered, typically through fallback functions [98]. If an attacker can bypass the validity checks in the target contract, they can reenter the function before the contract's state is properly updated. The vulnerability arises from two main factors. First, a contract's control flow decision relies on state variables that should be updated before calling another contract, but are not. Second, when control flow is transferred to another contract through low-level calls, there is no inherent gas limit that prevents complex operations in the callee.

The Decentralized Autonomous Organization (DAO) attack in 2016 exploited this vulnerability. As shown in Figure 4.2, the original contract includes functions for depositing and withdrawing Ether. The primary issue lies in the `withdraw` function. In this function, the user's balance is set to zero only after Ether has been sent to their address. This sequence creates a vulnerability. An attacker can repeatedly call `withdraw` before the balance update occurs, enabling multiple unauthorized withdrawals. In the attack contract, the attacker uses the fallback function to invoke `withdraw` repeatedly. Each call occurs be-

fore the balance is adjusted. This allows the attacker to receive multiple unauthorized Ether payments.

4.3.4 *Dangerous Delegatecall*

This vulnerability gained attention after the Parity hack in 2017, which exploited the EVM opcode `delegatecall`. Unlike a regular call, `delegatecall` executes the callee's bytecode while retaining the caller's storage, `msg.sender`, and `msg.value` (Figure 4.3). Consequently, if the address provided to `delegatecall` points to untrusted code (code section of contract B in Figure 4.3), that code runs with full write access to the caller's state and can arbitrarily modify storage or even trigger self-destruction, leading to loss of funds or contract functionality [99, 87].

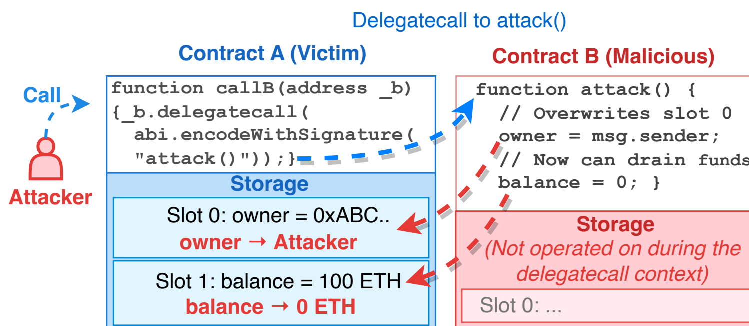

Figure 4.3: Delegatecall attack flow. A flawed interaction through a `delegatecall` can be initiated by the attacker calling a function in the target contract, which involves an unsafe `delegatecall` to an attacker-desired contract. This vulnerability might require a chain of exploits to pull off a successful attack.

A critical flaw was disclosed in December 2023 after several projects adopted the meta-transaction standard ERC-2771 [100]. Many implementations used `delegatecall` to let the forwarder library determine `_msgSender()`, but failed to validate the forwarded payload. An attacker could therefore wrap arbitrary calldata inside a trusted forwarded request and trick the callee into believing that the transaction originated from any address of the attacker's choosing. The technique was observed in the wild, causing losses of 84.59 ETH and 17,394 USDC in separate incidents [101, 102].

Mitigation/Prevention. Two well-established techniques are commonly used to mitigate reentrancy: the Check-Effects-Interactions (CEI) pattern [103, 104] and reentrancy guard locks. The CEI pattern establishes a strict execution order. First, the contract verifies all conditions (Checks). Second, it updates the internal state (Effects). Third, it executes external interactions (Interactions). In the vulnerable `DepositFunds` contract (Figure **??**), the balance is modified after the external call returns, which enables malicious reentrancy. The improved

`SecureDepositFunds` contract (Figure **??**) implements CEI correctly by resetting the user's balance to zero before transferring Ether. This sequence ensures that state updates precede external calls. By following this pattern verifying conditions, zeroing the balance, then making the external call the contract prevents repeated withdrawals even when an attacker attempts to exploit reentrancy [105].

The second approach uses a lock variable to prevent concurrent executions of sensitive functions. When `withdraw` is called, the lock is activated to block reentry until execution completes. Once the function finishes, the lock is released. This mechanism effectively neutralizes reentrancy attacks by preventing repeated calls to the function within the same transaction cycle. Both techniques provide robust protection, though CEI is generally preferred due to its simplicity and lower gas overhead.

4.3.5 *Unbounded Loops in Dynamic Arrays*

When a loop in a smart contract iterates over all elements of a dynamic array, the gas cost scales directly with the number of elements. If the array grows over time or contains a large number of elements, the gas cost may exceed the block gas limit. This prevents the function from executing successfully and can be exploited to create a denial-of-service (DoS) attack. An attacker could intentionally populate the array with numerous entries, making the function prohibitively expensive or impossible to execute.

Mitigation/Prevention. Several best practices can prevent unbounded loop vulnerabilities [106]. Developers should limit loop iterations by implementing maximum bounds or pagination. Operations can be divided into smaller transactions that process subsets of data. Pull payment patterns should be employed rather than pushing payments to multiple recipients in loops, as this allows recipients to withdraw funds individually. Finally, developers should profile gas consumption during testing to identify and optimize expensive operations before deployment [107, 108, 104].

4.3.6 *DoS Attack via Owner Account*

Many smart contracts have an owner account that controls the contract. If this account is not adequately protected, attackers may compromise it. This could lead to severe consequences, such as permanently locking Ether in the contract or draining funds [109].

Mitigation/Prevention. Contracts should restrict access to critical functions using multi-signature approvals. This approach avoids reliance on a single account for key operations [110].

4.3.7 *State-Revert*

A contract is vulnerable to state-revert abuse when it reveals the result of a state-changing operation before the change becomes irrevocable. This allows a caller to decide whether to let the transaction stand. The classic setting is a probabilistic reward dApp that pays one of several prizes. An attacker routes the call through an auxiliary contract and records their own balance. They then invoke the reward function with `(bool ok, bytes data) = address(dApp).call(payload)` and inspect `data` to see which prize was drawn. If the prize is unsatisfactory, the attacker immediately executes `require(false)`. Because the revert propagates upward, every write inside the dApp is rolled back, including token transfers. However, the attacker has already learned the outcome. By repeating this cherry-picking loop until a desirable branch appears, the adversary collects only high-value rewards and skews the game's expected value [111].

Prevention/Mitigation. Effective countermeasures postpone disclosure or make it irreversible. Commit-and-reveal schemes write a sealed commitment (`keccak256(seed, user)`) in one transaction and reveal the seed in a later block. This fixes the outcome before either party knows it. When genuine randomness is needed, verifiable random-function services (e.g., Chainlink VRF) provide entropy that cannot be observed in time to trigger a revert. Finally, the reward function can burn gas or charge a non-refundable fee before emitting any revealing data. This eliminates the economic incentive to revert [104].

4.3.8 *Frozen Ether*

Frozen Ether (also known as "locked money") occurs when a contract can accept ETH but cannot release the funds. This happens when the contract's withdrawal logic lives in an external library reached via `delegatecall`, and that library becomes absent. If the library address is wiped with `selfdestruct`—intentionally or by accident—the forwarder still receives deposits (no code is needed for that). However, every withdrawal attempt reverts, leaving the balance permanently inaccessible. This was seen in the 2017 Parity incident where 513,774 ETH became permanently locked [112]. Despite EIP-6780 [113] limiting `selfdestruct` functionality, contracts without internal fallback mechanisms remain vulnerable [114, 99].

Mitigation/Prevention. Any contract designed to hold ETH should embed an internal withdrawal function, such as using `call{value: amount}("")`, or hard-code an immutable library address that cannot be destroyed. Without such a path, deposits may become irreversible [115].

4.3.9 *Insecure Randomness*

This vulnerability affects contracts that require randomness (e.g., gambling and lottery contracts). These contracts often generate pseudo-random numbers using seeds such as `block.number`, `block.timestamp`, `block.difficulty`, or `blockhash`. Since these seeds can be influenced by blockchain network participants (validators or miners), malicious actors can manipulate these values to alter the outcome in their favor [19, 116].

Listing 4.2: Lottery Contract with insecure randomness

```
contract VulnerableLottery {
    uint256 private seed;

    function play() public payable {
        require(msg.value == 1 ether);
        uint256 random = uint256(keccak256(
            abi.encodePacked(block.timestamp, seed)
        ));
        if(random % 10 == 0) {
            msg.sender.transfer(9 ether);
        }
        seed = random;
    }
}
```

For example, the `VulnerableLottery` [17] contract depends on `block.timestamp` (Listing 4.2) and a private seed for random number generation. This results in unsafe randomness. Miners can alter `block.timestamp` within a narrow range, allowing an attacker to affect the result by sending transactions at particular times. Moreover, using a predictable seed does not generate enough entropy to prevent the prediction of future outcomes.

Prevention/Mitigation. Using blockchain-internal sources like `block.timestamp` and `block.difficulty` for random number generation is insecure due to their transparency and predictability [117]. Instead, developers can utilize external oracles such as Chainlink VRF (Verifiable Random Function), which provides more tamper-resistant randomness [118, 119].

Another approach is the commit-reveal scheme. This enhances randomness security by requiring participants to submit a concealed value first and reveal it later. This prevents attackers from predicting or manipulating results [120, 104]. Additionally, employing public-key cryptosystems like the Signidice algorithm [121], originally developed for EOS blockchain and adopted in projects like DAO.Casino, can generate random numbers in two-party contracts.

Furthermore, cross-chain oracles such as BTCRelay [122] provide a trustless mechanism for Ethereum smart contracts to verify Bitcoin

transactions. BTCRelay stores Bitcoin block headers on Ethereum and allows contracts to check Bitcoin transaction inclusion proofs. This enables the use of Bitcoin's Proof-of-Work-derived entropy as a source of randomness in Ethereum contracts. When combined with other protections against validator manipulation, such as time-delay and multi-source entropy, this strategy is useful.

4.3.10 *Front-Running (Abusing Transaction-Ordering Dependency)*

Front-running is a common attack vector that takes advantage of the public and sequential nature of blockchain transaction processing. It allows an adversary to profit by preempting a victim's pending operation. Miners (or validators) decide the execution order of pending transactions. An adversary who foresees a profitable state change can insert their own call ahead of the victim's. This is done either by rebroadcasting the same call with a higher gas price, which miners typically favor [87], or, if the adversary controls the block proposer, by unilaterally reordering transactions regardless of fees [123]. Such reordering can alter contract outcomes and extract value from honest users.

In a simple front-running scenario, Alice broadcasts transaction T_1 with gas price G_1 at time t_1. Bob observes T_1 in the mempool and immediately submits T_2 with a higher gas price $G_2 > G_1$ at $t_2 > t_1$. Validators generally sort pending transactions by gas price. They include T_2 first in block B_x, while T_1 is confirmed later (in the same or a subsequent block). By preempting the state change encoded in T_1 (e.g., a profitable DEX swap), Bob captures its economic value. Figure 4.4 illustrates this attack, showing how Transaction B with 50 Gwei gas price overtakes Transaction A with 20 Gwei, despite being submitted later. This is an archetypal front-running attack that exploits transaction-ordering dependency.

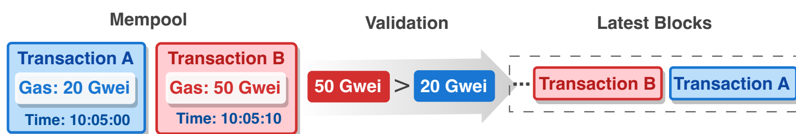

Figure 4.4: Demonstration of front-running attack where higher gas incentive (50 Gwei vs 20 Gwei) allows Transaction B to preempt Transaction A in block inclusion, despite later submission time.

Prevention/Mitigation. To address this vulnerability, developers can use techniques like batch auctions, which process several transactions at once to limit manipulation of transaction ordering [124]. Commit-reveal schemes divide transactions into two phases. They initially conceal sensitive information and reduce front-running risk [124].

4.3.11 *Authorization Using tx.origin*

The global `tx.origin` holds the externally owned account (EOA) that initiated the entire transaction chain, whereas `msg.sender` refers to the immediate caller of a function. Using `tx.origin` in access control checks rather than `msg.sender` lets an attacker insert a malicious contract into the call flow and pass the check as the original EOA.

Listing 4.3: Victim contract demonstrating the use of tx.origin

```
contract Victim {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function withdrawAll() public {
        require(tx.origin == owner, "Not authorized");
        payable(msg.sender).transfer(address(this).balance);
    }

    receive() external payable {}
}
```

Listing 4.4: Attacker contract exploiting tx.origin vulnerability

```
contract Attacker {
    Victim public victimContract;

    constructor(address _victimContractAddress) {
        victimContract = Victim(_victimContractAddress);
    }

    function attack() public {
        victimContract.withdrawAll();
    }

    receive() external payable {}
}
```

In Listing 4.3, the `Victim.withdrawAll` function authorizes withdrawal with `require(tx.origin == owner)`. The attacker contract in Listing 4.4 simply calls `withdrawAll`. Because `tx.origin` still equals the owner's EOA, the transfer succeeds and forwards funds to the attacker. Owner approval does not occur.

Prevention/Mitigation. Developers should avoid using `tx.origin` for authorization and instead rely on `msg.sender`, which represents the immediate caller of the function [125]. Additionally, restricting function access to approved users [126] by implementing strong access

control mechanisms such as OpenZeppelin's Ownable contract is recommended.

4.3.12 *Additional Vulnerabilities*

The remaining 13 vulnerabilities collectively account for the remaining attack surface identified in our systematic review. These include various categories from Ether leakage to timing manipulations, and play critical roles in the multi-tier root-cause framework developed in Chapter 5. Complete descriptions with code examples, real-world incidents, and mitigation strategies are documented in Appendix A.1.

4.3.13 *Hierarchical Organization of Vulnerabilities*

Our systematic review revealed that the 25 active vulnerabilities exist at varying levels of abstraction. Some represent broad vulnerability classes (e.g., Reentrancy, Access Control), while others describe specific manifestations of these classes (e.g., `tx.origin` authentication is a specific form of Access Control weakness). To address this inconsistency and align with established taxonomies [127, 128], we organized vulnerabilities into a two-level hierarchy and mapped them to our 4-Tier Root Cause Framework with their corresponding sub-categories (see Table 5.2 in Chapter 5).

We identified seven granular vulnerabilities that are better understood as subtypes of broader categories:

1. **Authorization via tx.origin** → subtype of *Access Control*, mapped to T4/Flawed Access Control (SWC-115 [127])
2. **Ponzi Scheme** → subtype of *Business Logic Flaws*, mapped to T1/Perverse Incentive [129]
3. **Prodigal Contract** → subtype of *Business Logic Flaws*, mapped to T1/Incorrect Pricing [130]
4. **DoS via Unbounded Loops** → subtype of *Denial of Service*, mapped to T4/State & Data Handling (SWC-128 [127])
5. **Resource Exhaustion** → subtype of *Denial of Service*, mapped to T4/State & Data Handling [131]
6. **Dangerous Balance Inequality** → subtype of *Denial of Service*, mapped to T4/State & Data Handling [132]
7. **Storage Collision** → subtype of *Proxy/Storage Issues*, mapped to T4/State & Data Handling [133]

Furthermore, we observed that several general vulnerability categories have well-documented variants in the academic literature

and security registries. To provide a comprehensive taxonomy, we incorporated these subtypes with their authoritative references. This enrichment ensures that our classification captures the full spectrum of attack vectors within each category.

Reentrancy (T4/Interaction Pattern) has four documented subtypes [127, 134, 135, 136]: Single-Function Reentrancy occurs when the attacker re-enters the same function (SWC-107); Cross-Function Reentrancy involves calling a different function that shares the same state; Cross-Contract Reentrancy spans multiple contracts with shared state; and Read-Only Reentrancy exploits view functions during reentrancy to obtain stale data.

Access Control (T4/Flawed Access Control, T2/Compromised Keys) encompasses four subtypes from the SWC Registry [127]: Default Visibility where functions without explicit visibility default to public (SWC-100); Unprotected Functions lacking proper access modifiers on critical operations (SWC-105); tx.origin Authentication using tx.origin instead of msg.sender for authorization checks (SWC-115); and Incorrect Constructor Name affecting contracts written before Solidity 0.4.22 (SWC-118).

Arithmetic Errors (T4/Arithmetic Errors) comprise three subtypes [128, 64]: Integer Overflow/Underflow when arithmetic operations exceed type bounds (SWC-101); Rounding Errors from precision loss in division operations; and Type Casting Vulnerabilities arising from unsafe conversions between integer types.

Denial of Service (T4/State & Data Handling, T2/Compromised Keys) includes six subtypes [127, 131, 132]: Unexpected Revert where external calls that revert block contract execution (SWC-113); Unbounded Loops causing gas exhaustion from iterations over dynamic arrays (SWC-128); Block Stuffing where attackers fill blocks to prevent victim transactions; Owner Account issues when lost owner keys prevent administrative functions (mapped to T2/Compromised Keys); Balance Inequality from strict equality checks on contract balance; and Resource Exhaustion depleting computational or storage resources.

Oracle Manipulation (T3/Oracle & Price Feed) has four subtypes [137, 138, 139, 140]: Spot Price Manipulation affects on-chain reserves for instant price impact; Flash Loan Attacks use uncollateralized loans to manipulate prices within a single transaction; TWAP Manipulation exploits time-weighted average price calculations; and Stale Data vulnerabilities arise from using outdated oracle responses.

Front-Running/MEV (T1/Incorrect Pricing) encompasses three subtypes [137, 141, 142]: Displacement where the attacker's transaction replaces the victim's transaction; Sandwich Attacks where the victim transaction is sandwiched between attacker's buy and sell orders; and Suppression involving delaying or preventing victim's transaction execution.

Business Logic Flaws (T1/Incorrect Pricing, T1/Perverse Incentive) include four subtypes [129, 130, 128]: Ponzi Schemes are contracts that pay earlier investors from later investments; Prodigal Contracts send Ether to arbitrary addresses without proper validation; Accounting Errors involve incorrect balance or share calculations; and Token Unit Mismatch arises from confusion between token decimals.

Blockchain Environment Dependency (T3/Unvalidated Inputs) has two subtypes [127]: Insecure Randomness from using predictable on-chain values for random number generation (SWC-120); and Timestamp Dependence from relying on manipulable block timestamps (SWC-116).

Proxy/Storage Issues (T4/State & Data Handling) comprise two subtypes [127, 133]: Dangerous Delegatecall to untrusted contracts (SWC-112); and Storage Collision from overlapping storage slots in proxy patterns.

Ether Locking/Loss (T4/State & Data Handling) includes two subtypes [130, 128]: Frozen Ether in contracts that can receive but cannot send Ether; and Orphan Address vulnerabilities from sending Ether to addresses without owners.

This hierarchical organization yields 14 distinct vulnerability categories: 10 general categories with documented subtypes as detailed above, and 4 standalone vulnerabilities that represent atomic security issues without further subdivision. Table 4.4 summarizes the complete mapping of these categories to our 4-Tier Root Cause Framework along with their corresponding sub-categories.

4.4 DEPRECATED VULNERABILITIES

This section discusses vulnerabilities in smart contracts that were formerly seen as security issues but have been addressed by protocol enhancements and governance systems. Table 4.5 presents these deprecated Ethereum smart contract vulnerabilities and their deprecation reasons.

4.4.1 *Upgradable Contracts*

Smart contracts often incorporate upgradability mechanisms through logic contracts and proxies to manage immutability constraints of the contract's code section [143]. Although early analyses classified upgradability as a potential vulnerability [87], recent advancements in upgradability patterns and governance mechanisms have demonstrated that this is better understood as a technical challenge rather than a security weakness.

On-chain governance proposals [144] now enable decentralized control over upgrades, reducing the risk of unilateral malicious actions. Moreover, research efforts such as SoliNomic [145] have introduced

Table 4.4: Mapping of Vulnerability Categories to 4-Tier Root Cause Framework

#	Category	Tier	Sub-Category	Subtypes
1	Reentrancy	T4	Interaction Pattern	Single-Function, Cross-Function, Cross-Contract, Read-Only
2	Access Control	T4 (T2)	Flawed Access Control	Default Visibility, Unprotected Functions, tx.origin, Incorrect Constructor
3	Arithmetic Errors	T4	Arithmetic Errors	Overflow/Underflow, Rounding, Type Casting
4	Denial of Service	T4 (T2)	State & Data Handling	Unexpected Revert, Unbounded Loops, Block Stuffing, Owner Account, Balance Inequality, Resource Exhaustion
5	Oracle Manipulation	T3	Oracle & Price Feed	Spot Price, Flash Loan, TWAP, Stale Data
6	Front-Running/MEV	T1	Incorrect Pricing	Displacement, Sandwich, Suppression
7	Business Logic	T1	Incorrect Pricing	Ponzi, Prodigal, Accounting Errors, Token Unit Mismatch
8	Blockchain Env.	T3	Unvalidated Inputs	Insecure Randomness, Timestamp Dependence
9	Proxy/Storage	T4	State & Data Handling	Dangerous Delegatecall, Storage Collision
10	Ether Lock/Loss	T4	State & Data Handling	Frozen Ether, Orphan Address
11	State-Revert	T4	Interaction Pattern	—
12	External Call	T4	Interaction Pattern	—
13	Event-Ordering	T4	State & Data Handling	—
14	Compiler Version	T2	Clumsy Deployment	—

Note: Tier definitions: T1=Flawed Economic Design, T2=Protocol Lifecycle/Governance, T3=External Dependencies, T4=Implementation-Level. Categories marked with (T2) have subtypes spanning multiple tiers. For complete tier descriptions and incident mappings, see Table 5.2 in Chapter 5.

formal refinement-based approaches to structuring the evolution of decentralized organizations for correct and consistent upgrades. Therefore, when properly designed, upgradability enhances the adaptability of DApps without compromising security.

4.4.2 *Wrong Address*

This vulnerability arises due to improper handling of function arguments. The EVM expects input arguments to be encoded in chunks of 32 bytes. However, if an attacker deliberately sends a truncated address (an address with missing bytes at the end), the EVM automatically pads the missing bytes with zeros. This potentially alters the intended

Table 4.5: Summary of deprecated smart contract vulnerabilities and their mitigation

Vulnerability	Deprecation Reason(s)
Upgradable Contracts	Re-contextualized from a vulnerability to a technical feature. Its risks are manageable through robust on-chain governance and verifiable evolution patterns.
Wrong Address	Mitigated by ABI-decoding requirements in Solidity v0.5.0. Calls with calldata shorter than the expected arguments will revert, preventing the padding that enabled the attack.
Erroneous Visibility	Addressed by Solidity v0.5.0, which mandated explicit function visibility declarations (`public`, `private`, etc.).
Suicidal Contract	The behavior of the `selfdestruct` opcode was severely curtailed by EIP-6780. It now only functions if executed in the same transaction as contract creation, neutralizing its use in common attack patterns.
Call-stack Depth Limit	Rendered economically infeasible by the Tangerine Whistle hard fork (EIP-150), which introduced the "63/64 gas rule." This rule limits the gas forwarded in a call, preventing deep recursion attacks.

values of other arguments. This misalignment can cause an attacker to manipulate transaction amounts or other critical parameters in smart contracts [114, 87].

The attacker could craft a transaction with a shortened address. When processed, this causes the EVM to pad the missing bytes with zeros, leading to a shift in memory alignment. Other function arguments, such as transfer amounts, are then misinterpreted (e.g., excessive transfers to the attacker's address).

To prevent this vulnerability, developers should implement strict validation to check that the input data matches the expected size. The contract should explicitly check `msg.data` length to ensure proper alignment of the arguments. In modern Solidity versions ($\geq$0.5.0), with its standard ABI decoding, the short address attack is prevented by design [146]. In these versions, if an attacker tries to supply a truncated address (resulting in insufficient calldata), the function call will revert before any parameter decoding or auto-padding occurs.

4.4.3 *Erroneous Visibility*

Erroneous visibility vulnerability emerged from Solidity's default function visibility behavior prior to version 0.5.0. Functions without explicit visibility modifiers automatically became public. This exposed critical operations to unauthorized external calls [147]. Implicit behavior often led to unintentional access to state-modifying, fund management,

or administrative functionality. Missing visibility specifiers allowed attackers to execute privileged functions in the Parity multisig wallet incident [148].

Solidity version 0.5.0, released in November 2018, fixed this vulnerability by requiring explicit visibility declarations for all functions [146]. Contracts without visibility specifiers are now rejected by the compiler. This turns a runtime vulnerability into a compile-time error. Functions must explicitly declare `public`, `private`, `internal`, or `external` visibility.

4.4.4 *Suicidal Contract*

The EVM's `selfdestruct` opcode allows a contract to destroy itself and send the remaining Ether to a target address. Since the Dencun upgrade (EIP-6780, March 2024) [113], `selfdestruct` no longer removes code or storage and no longer grants a gas refund, unless invoked in the same transaction that deployed the contract. Outside that narrow case, it merely transfers the contract's Ether balance to the specified address. This effectively disables contract "suicide" and the attacks it enabled.

4.4.5 *Call-Stack Depth Limit*

The EVM caps a call chain at 1,024 frames. Before EIP-150 [149], an attacker could, for about 40,000 gas, create 1,024 nested calls and cause every subsequent call from a victim contract to fail (return 0). EIP-150 raised the sub-call base cost to 700 gas. More importantly, it restricted each call to forwarding at most 63/64 of its remaining gas. Because the available gas shrinks geometrically, filling the stack now exceeds the block gas limit. This renders the "stack-depth attack" economically infeasible.

4.5 SUMMARY

This chapter presented a systematic literature review of smart contract vulnerabilities. We analyzed 71 academic papers following a rigorous research protocol. The filtering process reduced 17,013 initially identified papers to 71 quality-validated studies. From this analysis, we identified 24 active vulnerability types documented in academic research. These include reentrancy, integer overflow, dangerous delegatecall, front-running, and insecure randomness among others.

We also identified five deprecated vulnerabilities that compiler updates or protocol changes have mitigated. These include call-stack depth limit and erroneous visibility. The analysis revealed that academic literature concentrates heavily on implementation-level bugs. Reentrancy and integer overflow receive extensive attention. How-

ever, this academic focus may not reflect real-world threat priorities. Chapter 5 addresses this gap by analyzing actual attack incidents to compare academic emphasis with practical exploitation patterns.

5 EMPIRICAL ANALYSIS OF REAL-WORLD SMART CONTRACT ATTACKS

5.1 INTRODUCTION

The systematic literature review in Chapter 4 identified 25 active smart contract vulnerabilities documented in academic research. However, the relationship between these theoretically identified weaknesses and the vulnerabilities exploited in high-impact, real-world attacks remains unclear.

Existing taxonomies organize vulnerabilities by implementation patterns reentrancy, integer overflow, access control flaws. However, catastrophic incidents demonstrate a different reality. The $197 million Euler Finance exploit [150] and the $190 million Nomad Bridge hack [151] show that financially devastating attacks often stem from factors beyond isolated coding mistakes.

To address this gap, we conduct an in-depth empirical analysis of 50 significant smart contract incidents. These incidents occurred between May 2022 and April 2025. Collectively, they incurred over $1.09 billion in losses. Our analysis moves beyond vulnerability labels to examine the exploit methods, root causes, and enabling conditions that allowed these attacks to succeed.

This chapter complements the vulnerability analysis from Chapter 4 by examining how these theoretical weaknesses manifest in real-world attacks.

We begin by describing our rigorous incident review protocol in Section 5.2. This section details how we selected and analyzed 50 incidents from an initial pool of 337 documented attacks. Section 5.3 presents descriptive statistics of our dataset. These include temporal distribution, network classification (Ethereum mainnet, Layer 2, sidechains), and application types.

Section 5.4 analyzes which vulnerabilities from our systematic review appear most frequently in real-world exploits. This analysis reveals that access control failures and price manipulation dominate the attack landscape.

Section 5.5 synthesizes our findings into eleven key insights that characterize modern smart contract attacks. These insights reveal patterns that are largely absent from the academic literature. They include flash loans as attack amplifiers (Insight 5.5.1), the pervasiveness of human operational errors (Insight 5.5.2), composability risks (Insight 5.5.3), critical oracle security issues (Insight 5.5.4), insider threats (Insight 5.5.5), new EVM upgrade attack vectors (Insight 5.5.6),

inadequate emergency response mechanisms (Insight 5.5.7), complex cross-chain exploits (Insight 5.5.8), multi-vulnerability exploit chains where 62% of incidents involve combinations of two or more weaknesses (Insight 5.5.9), the overly broad categorization of business logic flaws (Insight 5.5.10), and complications of proxy/upgradability patterns (Insight 5.5.11).

Section 5.6 bridges the gap between academic vulnerability classifications and empirical reality. We propose a novel four-tier root-cause framework that organizes exploits by their fundamental causes rather than implementation patterns.

Finally, Section 5.7 demonstrates this framework through a detailed analysis of the LI.FI GasZip exploit. This case study illustrates how vulnerabilities across multiple tiers combine to enable a successful $11.6 million attack. The insights from this empirical analysis reveal critical patterns in how smart contract attacks succeed in practice.

5.2 INCIDENT REVIEW PROTOCOL

To systematically analyze real-world smart contract exploits, we established a rigorous four-stage incident review protocol. This protocol ensures relevance, quality, and comprehensive coverage of high-impact attacks. Figure 5.1 illustrates this systematic approach, which progresses from broad data collection to targeted analysis of well-documented, significant incidents.

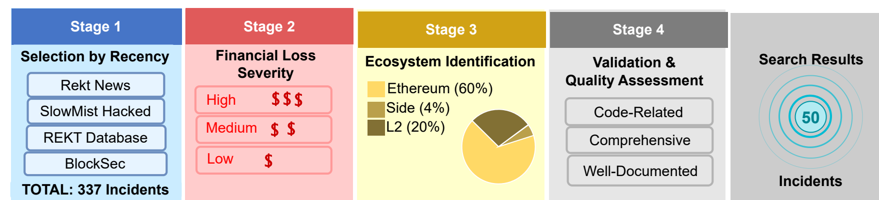

Figure 5.1: Four-stage smart contract incident review protocol to systematically identify, prioritize, and analyze relevant attacks.

5.2.1 *Stage 1: Comprehensive Data Collection*

We aggregated incident reports from four reliable blockchain security sources: Rekt News [152], REKT database [153], SlowMist Hacked database [154], and BlockSec reports [155]. From each source, we extracted up to 100 of the most recent attacks based on occurrence date. These sources were selected for their quality and analytical depth.

For BlockSec, our temporal parameters (April 2022 to April 2025) yielded only 37 documented attacks meeting our criteria. All of these

were incorporated into our analysis. Our initial dataset comprised 337 attacks for analysis.

5.2.2 *Stage 2: Impact-Based Prioritization*

Incidents underwent prioritization based on financial impact severity. We emphasized attacks resulting in substantial monetary losses. This criterion ensured our analysis concentrated on vulnerabilities manifesting the most significant ecosystem consequences. By focusing on high-impact incidents, we capture vulnerabilities that pose the greatest economic threat to the DeFi ecosystem and are most likely to inform meaningful security improvements.

5.2.3 *Stage 3: Ecosystem Classification*

We focused on incidents occurring within the Ethereum blockchain and its compatible networks. Each case was evaluated for relevance. This resulted in a final dataset of 50 distinct incidents that offer sufficient breadth while maintaining ecosystem coherence. We then mapped each incident to its respective blockchain environment, as shown in the network column of Table **??**. We established three network categories:

1. **Ethereum Mainnet:** The Ethereum network where native consensus exists, identified by transactions on etherscan.io (31 attacks).
2. **Ethereum Layer 2 Solutions:** These include Arbitrum, Optimism, Base, zkSync, Linea, Scroll, StarkNet, and Mode (17 attacks). L2s inherit security from Ethereum while processing transactions off-chain [156]. They use either Optimistic or Zero-Knowledge rollups to post transaction data back to Layer 1.
3. **Ethereum Side-Chains:** Networks like Polygon, BSC (Binance Smart Chain), and Ronin that maintain independent consensus mechanisms while supporting Ethereum Virtual Machine (EVM) compatibility [157] (2 attacks).

This classification follows the technical distinction that true L2 solutions derive their security from Ethereum itself, while side-chains operate with separate validator sets and consensus mechanisms. For example, Base is considered an L2 because it uses the OP Stack to inherit Ethereum's security properties. In contrast, Polygon PoS uses its own validator set for consensus, making it a side-chain.

5.2.4 *Stage 4: Vulnerability Verification and Report Quality Assessment*

In the final step, we applied two strict checks. First, we ensured the incident involved at least one smart contract vulnerability not merely

infrastructure failure, human error, or stolen private keys. Second, we verified that the incident was well-documented.

To be included, a report had to clearly explain the vulnerability (including localization information), how it was exploited, and what caused it. We excluded any cases that lacked sufficient technical detail. This process yielded a final dataset of 50 well-documented, high-impact incidents. Each incident involved at least one exploited smart contract vulnerability. We use this dataset as the basis for our analysis (Section 5.4), which later resulted in a root cause-based four-tier classification.

5.3 DATASET OVERVIEW

Our final dataset comprises 50 high-impact smart contract incidents that occurred between May 2022 and April 2025. These incidents collectively resulted in losses exceeding $1.093 billion. Table **??** presents a comprehensive overview, detailing the attack date, financial loss in USD, ecosystem classification, application type, exploited vulnerabilities, and target network for each case.

The incidents span diverse DeFi application categories. Lending protocols were the most frequently targeted (21 incidents), followed by decentralized exchanges (DEXs) with 12 incidents and cross-chain bridges with 4 incidents. This distribution reflects the concentration of value and complexity in these protocol types, making them attractive targets for sophisticated attackers. Other affected categories include yield aggregators, stablecoins, liquidity pools, NFT platforms, and DAO governance systems.

In terms of network connectivity, Ethereum Mainnet suffered the brunt of attacks with 31 incidents, accounting for 62% of all cases. Layer 2 solutions experienced 17 attacks (34%), while side-chains saw only 2 incidents (4%). This distribution correlates with the total value locked (TVL) across these networks, as Ethereum Mainnet continues to host the majority of DeFi protocols despite the growth of L2 ecosystems.

Temporally, our dataset reveals interesting patterns. The year 2024 witnessed the highest number of incidents (20 cases), followed by 2023 (18 cases) and 2022 (5 cases). Early 2025 data (through April) already shows 7 major incidents, suggesting a continuing trend of high-impact attacks. The average loss per incident stands at approximately $21.9 million. However, this figure is heavily skewed by outliers such as the Euler Finance ($197M) and Nomad Bridge ($190M) exploits.

Each incident in our dataset underwent rigorous verification through multiple independent sources. We cross-referenced several types of documentation. These included post-mortem reports from protocol teams, security firm analyses (BlockSec, CertiK, SlowMist, Immunebytes, etc.), on-chain transaction data via block explorers, and

verified contract source code. This multi-source validation ensures the accuracy of our vulnerability classifications and root-cause attributions.

The complete dataset is presented in Table **??** (Appendix A.15). This includes victim contract addresses, attacker addresses (both EOAs and smart contracts), and attack transaction hashes. This granular data enables reproducibility and facilitates further research on attack patterns and vulnerability exploitation techniques.

Table 5.1: Overview of studied security incidents

ID	Incident	Date	Loss	Eco	Application	Vulnerability	Net
1	Euler [158, 159, 160]	03/23	197	DeFi	Lending	Access, Logic	M
2	Nomad [161, 151]	08/22	190	Bridge	X-chain	Access Ctrl	M
3	BonqDAO [162]	02/23	120	DeFi	Stablecoin	Price Manip.	S
4	WooFi [163, 164]	03/24	85	DeFi	DEX	Price Manip.	L2
5	Fei Rari [165, 166]	05/22	80	DeFi	Lending	Reentrancy	M
6	Infini [167, 168]	02/25	49.5	DeFi	Stablecoin	Access Ctrl	M
7	KyberSwap [169, 170]	11/23	48	DeFi	DEX	Biz Logic	M
8	Penpie [171, 172]	09/24	27	DeFi	Yield Agg.	Reentrancy	M
9	Sonne [173, 174]	05/24	20	DeFi	Lending	Rounding	L2
10	Inverse [175, 176, 177]	04/22	15.6	DeFi	Lending	Price Manip.	M
11	Holograph [178, 179]	06/24	14.4	Token	NFT	Access Ctrl	M
12	Deus DAO [180, 181]	05/23	6.5	DeFi	Token	Access Ctrl	L2
13	Ionic [182, 183]	02/25	6.9	DeFi	Lending	Input Valid.	L2
14	Ronin [184, 185]	08/24	12	Bridge	X-chain	Access Ctrl	M
15	LI.FI [186, 187]	07/24	11.6	DeFi	Bridge	Access Ctrl	M
16	Yearn [188, 189]	04/23	11.4	DeFi	Yield Agg.	Misconfig	M
17	zkLend [190, 191]	02/25	9.6	DeFi	Lending	Rounding	L2
18	LI.FI Prot. [192, 193]	07/24	11.6	DeFi	X-chain	Unchk, Access	M
19	Hundred [194, 195]	04/23	7.4	DeFi	Lending	ExchRate, Rnd	L2
20	Abracadabra [196]	01/24	6.5	DeFi	Lending	Logic, Round	M
21	Lodestar [197, 198]	12/22	6.5	DeFi	Lending	Price Manip.	L2
22	1Inch [199, 200]	03/25	5	DeFi	DEX	Int Underflow	M
23	Shezmu [201, 202]	09/24	4.9	DeFi	Lending	Access Ctrl	M
24	Gamma [203, 204]	01/24	4.5	DeFi	Liquidity	Price Manip.	L2
25	Conic [205, 206]	07/23	4.2	DeFi	Lending	ROReent, Price	M
26	KiloEx [207, 208]	04/25	7.5	DeFi	DEX	Access, Price	L2
27	SIR [209, 210]	03/25	0.36	DeFi	Synthetics	Storage Coll.	M
28	Abracadabra [211]	03/25	13	DeFi	Lending	Biz Logic	L2

Table 5.1 – *Continued*

ID	Incident	Date	Loss	Eco	Application	Vulnerability	Net
29	DeltaPrime [212, 213]	11/24	4.85	DeFi	Lending	Input Valid.	L2
30	Onyx [214, 215]	09/24	3.8	DeFi	Lending	ExchRate, Acc	M
31	Dexible [216, 217]	02/23	2	DeFi	DEX Agg.	Access Ctrl	M
32	Dough [218, 219]	07/24	2.1	DeFi	Lending	Input Valid.	M
33	CloberDEX [220, 221]	12/24	0.5	DeFi	DEX	Reentrancy	L2
34	Tornado [222, 223]	05/23	2.17	DeFi	Privacy	Governance	M
35	Team Fin. [224, 225]	10/22	15.8	DeFi	Token Lock	Input Valid.	M
36	Raft [226, 227]	11/23	3.6	DeFi	Stablecoin	Rounding	M
37	Rho [228, 229]	07/24	7.5	DeFi	Lending	PrivAb, Price	L2
38	UwuLend [230, 231]	06/24	19.4	DeFi	Lending	Price Manip.	M
39	Velocore [232, 233]	06/24	6.8	DeFi	AMM	Underflow	L2
40	Curio [234, 235]	03/24	16	DeFi	DAO	Access, Deleg	M
41	Unizen [236, 237]	03/24	2.1	DeFi	DEX	Delegatecall	M
42	Seneca [238, 239]	02/24	6.4	DeFi	CDP	Input Valid.	M
43	OKX DEX [240, 241]	12/23	2.7	DeFi	DEX	Access, Deleg	M
44	Zunami [242, 243]	08/23	2.1	DeFi	Stablecoin	Price Manip.	M
45	EraLend [244, 245]	07/23	3.4	DeFi	Lending	ROReent, Price	L2
46	Atlantis [246, 247]	06/23	2.5	DeFi	Lending	Governance	S
47	Sturdy [248, 249]	06/23	0.8	DeFi	Lending	ROReent, Price	M
48	Jimbo's [250, 251]	05/23	7.5	DeFi	Lend/DEX	Access, Price	L2
49	Swaprum [252, 253]	05/23	3	DeFi	DEX	Access Ctrl	L2
50	Orion [254, 255]	02/23	3	DeFi	DEX Agg.	Reentrancy	M

Net: M=Mainnet, L2=Layer 2, S=Sidechain. **Loss** in $M.

Abbrev: Ctrl=Control, X-chain=Cross-chain, Biz=Business, Rnd=Rounding, Valid.=Validation, Manip.=Manipulation.

5.4 VULNERABILITY DISTRIBUTION ANALYSIS

After reviewing our dataset of 50 high-impact incidents, we analyzed the distribution of vulnerabilities that attackers exploited. We previously presented the complete vulnerability breakdown in Table 1.1 (Section 1.2). This section provides detailed analysis of each vulnerability category, their exploitation patterns, and their correlation with specific application types. Note that a single attack can exploit multiple vulnerabilities simultaneously. This is why the total vulnerability count exceeds the number of incidents.

The complete dataset is presented in Table **??** (Appendix A.15). This includes victim contract addresses, attacker addresses (both EOAs and smart contracts), and attack transaction hashes.

5.4.1 *Dominant Vulnerability Patterns*

The most prevalent vulnerability in our dataset is **access control**, appearing in 13 incidents (26%). These flaws allow attackers to invoke privileged functions without proper authorization. The severity ranges from simple missing modifier checks to sophisticated authorization bypasses in multi-contract architectures. For example, the Euler Finance attack exploited insufficient access control combined with business logic errors to drain $197 million [150]. Similarly, the Nomad Bridge incident demonstrated how a single missing authentication check in a message verification function enabled attackers to forge arbitrary cross-chain messages. This resulted in $190 million in losses [151].

Price manipulation ranks second with 13 occurrences (26%). This vulnerability affects protocols that rely on price oracles or on-chain exchange rates without sufficient safeguards. Attackers typically exploit flash loans to manipulate spot prices. They take advantage of the difference between instantaneous prices and time-weighted averages. The BonqDAO incident exemplifies this pattern. The attacker manipulated the oracle price feed to borrow against artificially inflated collateral values, ultimately causing $120 million in losses [256]. The WooFi attack followed a similar trajectory, exploiting price calculation vulnerabilities to extract $85 million [163].

Reentrancy vulnerabilities persist, appearing in 7 incidents (14%). While the classic DAO-style reentrancy is now rare due to widespread adoption of checks-effects-interactions patterns and reentrancy guards, we observe the emergence of read-only reentrancy as a significant threat. This variant exploits inconsistent state reads across view functions rather than state-modifying operations. The Conic Finance, EraLend, and Sturdy Finance attacks all leveraged read-only reentrancy in Curve pool integrations [205, 244]. The persistence of reentrancy in new forms demonstrates an important point. Understanding the vulnerability class is insufficient. Developers must also recognize novel exploitation vectors.

Input validation failures appear in 6 incidents (12%). These occur when contracts fail to sanitize or verify user-supplied parameters. This allows attackers to inject malicious values. The Team Finance attack demonstrates the severity of this oversight. Missing validation on token addresses in a migration function enabled the attacker to drain $15.8 million. The attacker substituted legitimate tokens with attacker-controlled contracts [225]. The Seneca Protocol incident similarly exploited insufficient validation of redemption parameters [238].

Business logic flaws appear in 5 incidents (10%). Unlike generic vulnerabilities such as reentrancy or integer overflow, business logic errors are protocol-specific defects in the intended contract behavior. These vulnerabilities require deep understanding of the protocol's economic model and state machine. The KyberSwap attack serves as a canonical example. The attacker exploited a subtle flaw in the concentrated liquidity tick calculation mechanism that went undetected despite multiple audits [169]. The complexity of modern DeFi protocols creates a large attack surface for such logic errors. This is particularly true for protocols implementing novel AMM curves or lending mechanisms.

5.4.2 *Persistent Classic Vulnerabilities*

Rounding errors appear in 4 incidents (8%), predominantly affecting lending protocols. These precision loss issues arise from Solidity's integer-only arithmetic. They can be amplified through repeated operations or flash loan-enabled manipulation. The Sonne Finance attack exploited rounding errors in interest rate calculations, enabling the attacker to drain $20 million [173]. The zkLend incident followed a similar pattern. Precision loss in exchange rate computations allowed profitable arbitrage at the protocol's expense [190]. The prevalence of this vulnerability in lending protocols suggests insufficient attention to numerical precision in financial contract design.

Delegatecall vulnerabilities appear in 2 incidents (4%). This opcode executes external code within the caller's storage context. This can lead to storage corruption or unauthorized state modifications when invoked carelessly. The Curio DAO attack exploited delegatecall to an attacker-controlled contract. This allowed arbitrary storage writes that manipulated governance votes [234]. The OKX DEX and Unizen incidents followed similar patterns [240, 236]. The continued occurrence of delegatecall vulnerabilities indicates a gap between security knowledge and implementation practice. This is despite clear documentation of risks.

5.4.3 *Diminishing Legacy Vulnerabilities*

Less frequent vulnerabilities include **integer overflow/underflow** (2 incidents, 4%), **governance manipulation** (2 incidents, 4%), **storage collision** (1 incident, 2%), and **misconfiguration** (1 incident, 2%). The relative rarity of integer arithmetic issues reflects the widespread adoption of Solidity 0.8.0 and later versions, which incorporate built-in overflow protection. The 1Inch incident, one of only two integer-related attacks, occurred in legacy code predating compiler protections [199].

Governance attacks, while infrequent in our dataset, pose systemic risks. The Tornado Cash incident demonstrated how attackers

can exploit governance mechanisms to gain control of protocol treasuries [222]. These attacks target the social layer rather than pure smart contract vulnerabilities.

5.4.4 *Multi-Vulnerability Exploitation Chains*

A critical observation from our analysis is that 13 out of 50 incidents (26%) involved multiple vulnerabilities exploited in combination. Modern attacks rarely rely on a single flaw. Instead, attackers chain several weaknesses to maximize impact and bypass individual security controls. The LI.FI Protocol attack exemplifies this pattern, combining access control bypass with unchecked external call return values to drain $11.6 million [193]. The Hundred Finance incident chained exchange rate manipulation with rounding errors [194]. The Conic Finance attack combined read-only reentrancy with price manipulation [205].

This trend toward multi-step exploits has important implications for security practice. Securing individual functions or applying isolated fixes is insufficient. Protocols must consider how vulnerabilities interact across the entire system. Static analysis tools that check for individual vulnerability patterns may miss complex exploit chains. These chains emerge from the composition of multiple benign-looking operations.

5.4.5 *Application-Specific Vulnerability Correlations*

Our analysis reveals correlations between vulnerability types and application categories. **Lending protocols**, which constitute the largest category in our dataset, exhibit exposure to multiple vulnerability classes. Price manipulation appears frequently in lending attacks. These protocols rely on external price feeds for collateral valuation. Notable examples include Inverse Finance, Lodestar Finance, UwuLend, and the read-only reentrancy variants (Conic, EraLend, Sturdy) that manipulated price reads during execution [257, 198, 230].

Rounding errors also cluster in lending protocols, appearing in incidents such as Sonne Finance, zkLend, Hundred Finance, and Raft Protocol [173, 190, 194, 226]. The mathematical complexity of interest rate calculations and share-to-asset conversions in lending protocols creates numerous opportunities for precision loss. Attackers can exploit this through repeated operations or flash loan amplification.

Decentralized exchanges demonstrate varied vulnerability patterns. While business logic flaws like the KyberSwap concentrated liquidity exploit represent sophisticated attacks on AMM mathematics [169], DEXs also suffer from simpler issues. Access control problems appear in OKX DEX and Swaprum [240, 252]. The 1Inch incident involved an integer underflow in legacy code [199]. The diversity of vulnerability

types in DEX attacks reflects the heterogeneity of DEX designs, from simple constant product AMMs to complex concentrated liquidity and multi-asset pools.

Cross-chain bridges in our dataset (Nomad, Ronin, LI.FI) consistently exhibit access control vulnerabilities [151, 184, 193]. The distributed architecture of bridges creates numerous trust boundaries where authorization checks can fail. This involves message relayers and validators across multiple chains. The Nomad Bridge attack demonstrated how a single missing authentication step in message verification enabled complete protocol drainage.

This application-specific distribution indicates that vulnerability prevention requires domain expertise beyond general Solidity security knowledge. Lending protocol developers must prioritize oracle robustness, numerical precision, and invariant checking. DEX developers need formal verification of AMM mathematical properties. Bridge developers should focus on message integrity and multi-party authorization schemes. Generic security audits that apply uniform checklists across all protocol types may miss category-specific risks.

5.5 KEY INSIGHTS FROM INCIDENT ANALYSIS

Beyond vulnerability categorization, our empirical analysis reveals several critical insights about attack patterns, enabling conditions, and systemic weaknesses in the DeFi ecosystem. These insights inform both immediate security improvements and long-term architectural considerations.

5.5.1 *Insight 1: Flash Loans as Attack Amplifiers*

Flash loans emerged as a critical enabler across multiple attack categories in our dataset. These uncollateralized loans must be repaid within the same transaction. They provide attackers with significant temporary capital to execute exploits that would otherwise be economically impractical.

We observed flash loans instrumental in price manipulation attacks such as WooFi [163, 164] and Zunami Protocol [242, 243]. They were also used in reentrancy exploits like Fei Rari [165, 166]. Additionally, attacks exploiting rounding errors such as Sonne Finance [173] relied on flash loans.

The systematic use of flash loans transforms the economics of attacks. Vulnerabilities that might require millions of dollars in upfront capital become accessible to attackers with minimal resources. This democratization of attack capability means that even minor vulnerabilities can be profitably exploited at scale.

The prevalence of flash loan usage in our dataset suggests that security analyses must account for attackers having access to effec-

tively unlimited capital within a single transaction block. Protocol designers must evaluate their systems under a different assumption. They should assume that attackers can temporarily marshal arbitrary amounts of capital. This fundamentally changes the threat model from resource-constrained adversaries to those with near-infinite liquidity for the duration of a transaction.

5.5.2 *Insight 2: The Pervasiveness of Human Error*

A substantial portion of incidents in our dataset trace back to human configuration errors rather than pure code vulnerabilities. These errors manifest across several distinct categories.

First, deployment and upgrade flaws appear in incidents such as the Nomad Bridge misconfiguration [258, 161, 151]. In this case, a critical initialization parameter was incorrectly set to the zero address during an upgrade. Incomplete upgrades in Ronin Bridge [185, 184] skipped necessary initialization steps. Bugs introduced in new code deployments like LI.FI [193, 187, 186, 192] occurred when a newly added contract facet lacked proper input validation.

Second, poor security hygiene resulted in catastrophic outcomes. Retained privileges by former developers in Infini [167, 168] and Holograph [179, 178] caused major issues. Administrative access was never revoked after developers left the project. Compromised private keys in OKX DEX [240, 241] and Rho Market [228, 229] enabled unauthorized upgrades.

Third, design oversights caused significant losses. Incorrect address configuration in Yearn Finance [188, 189] resulted from a copy-paste error. The wrong token address was hardcoded. Fundamental flaws in incentive mechanisms like Euler Finance's liquidation discount system [150, 160, 158, 159] created perverse incentives for self-liquidation.

Fourth, some teams ignored explicit warnings or reused code with known vulnerabilities. The EraLend incident [244, 245] exemplifies this problem. Code containing explicit warning comments about reentrancy risks was deployed without modification.

This pattern indicates that technical code audits alone are insufficient. Organizations must implement rigorous operational security procedures. These include multi-signature controls for privileged operations and systematic verification of deployment parameters against specifications. Mandatory waiting periods between upgrade proposals and execution enable community review. Automated monitoring for configuration drift detects unauthorized changes. Formal handoff procedures when developers leave projects are also essential.

5.5.3 *Insight 3: Composability Risks*

DeFi's composability enables rapid innovation and capital efficiency. However, it simultaneously expands attack surfaces in ways that isolated contract analysis cannot capture. Interactions between protocols create complex vulnerabilities.

Abracadabra's cross-protocol state issues [196, 259, 211, 260] demonstrate this problem. State updates in one contract failed to properly synchronize with dependent contracts. Complex internal logic spanning multiple contract calls can obscure subtle bugs that remain undetected until exploitation.

The interconnected nature of DeFi protocols means that a vulnerability in one component can propagate throughout the ecosystem. Secure components can become vulnerable when composed with insecure ones.

Composability-aware analysis and verification of protocols becomes essential [261]. Security assessments must consider not only isolated contract behavior but also potential interactions with external protocols under all possible states. This includes evaluating how state changes in integrated protocols might affect the security properties of the primary system. Examples include oracle price updates or collateral valuations. It also involves identifying assumptions about external contract behavior that might be violated. These assumptions include reentrancy guards, state consistency, and return value handling.

5.5.4 *Insight 4: Oracle Security is Important*

The high frequency of price manipulation incidents underscores a fundamental problem. We observed 13 occurrences, representing 26% of our dataset. Secure, manipulation-resistant oracles remain a fundamental unsolved challenge in DeFi security.

Several vulnerabilities persist. Reliance on spot prices from single DEXs allows attackers to use flash loans to temporarily distort prices. Insufficient liquidity in oracle pairs makes manipulation economically feasible. Oracles whose prices can be influenced within a single atomic transaction through techniques like sandwich attacks continue to represent major vulnerabilities exploited across our incident set.

The incidents reveal that even time-weighted average price (TWAP) mechanisms can be circumvented. Sustained manipulation across multiple blocks is possible. The Inverse Finance attack [175, 257, 177, 176] demonstrates this clearly. The attacker prevented arbitrage by spamming transactions to maintain artificial prices.

Protocols must implement multi-layered oracle defense mechanisms [84]. These include sourcing prices from multiple independent feeds with different data sources and aggregation methods. Implementing circuit breakers that halt operations when price movements

exceed statistically reasonable thresholds based on historical volatility is crucial. Requiring minimum liquidity depths for price sources ensures manipulation requires prohibitive capital. Incorporating delay mechanisms prevents single-block price manipulation by using stale but manipulation-resistant prices.

5.5.5 *Insight 5: Insider Threats and Privilege Abuse*

Access control issues in our dataset were not merely the result of external attackers exploiting technical flaws in permission checks. Several incidents involved insufficient internal controls and excessive privileges granted to protocol administrators, developers, or automated systems.

The Swaprum incident exemplifies abuse of unrestricted upgrade privileges [253, 252]. Protocol administrators deployed malicious upgrades to steal user funds. The Infini and Holograph cases demonstrate the risks of retained developer access [167, 168, 179, 178]. Former team members exploited administrative functions they should no longer have possessed.

This pattern suggests that privileged operations should be subject to stronger controls. Multi-party approval mechanisms requiring signatures from multiple independent parties with diverse incentives are essential. Time-locked execution windows allow community oversight and emergency intervention between proposal and execution. Comprehensive audit trails for all administrative actions enable forensic analysis and real-time monitoring. Regular rotation of administrative credentials with formal handoff procedures revokes old keys and verifies new key holders.

The principle of least privilege must extend beyond contract-level access control to organizational operational security. Protocol governance should be treated as an attack surface equivalent in importance to smart contract code.

5.5.6 *Insight 6: New Attack Vectors for EVM Upgrades*

The SIR Trading exploit utilizing transient storage opcodes introduced in EIP-1153 [210, 209, 262] demonstrates a critical risk. New EVM features can introduce unforeseen vulnerabilities if not properly understood and implemented by developers.

The attack exploited storage collision in transient storage slots. The same slot was reused for different purposes, storing both a pool address and a mint amount. This allowed the attacker to bypass authentication checks by crafting an address that matched the expected numeric value.

As the Ethereum protocol evolves through EIPs and hard forks, each upgrade introduces new primitives and execution semantics. These

may behave unexpectedly when combined with existing patterns or when developers misunderstand their guarantees.

This observation highlights the need for comprehensive developer education around new EVM features before they reach mainnet. Detailed documentation of security implications and potential pitfalls is necessary. Formal specification of new opcodes with explicit security considerations should clarify intended use cases and forbidden patterns. Systematic review of existing contracts for compatibility with protocol upgrades identifies code that may behave differently under new rules. Extended testing periods on testnets before mainnet deployment of contracts using new features allow the security community to identify attack vectors.

The pace of protocol evolution must be balanced against the ecosystem's ability to adapt securely.

5.5.7 *Insight 7: Inadequate Emergency Response Mechanisms*

The Seneca Protocol incident revealed a critical operational gap [239, 238]. Pause functionalities designed to halt operations during attacks were declared internal-only. This rendered them useless during the active attack because no external transaction could invoke them.

This highlights that emergency response capabilities must be both technically sound in their implementation and operationally accessible when needed under high-stress conditions. The immutability of smart contracts makes preventive measures paramount. However, when prevention fails, rapid response capabilities can limit damage.

Emergency controls themselves become targets. This requires careful design to prevent abuse while ensuring availability during legitimate emergencies. The balance between security (preventing unauthorized halts) and safety (enabling authorized intervention) represents a fundamental tension in incident response design.

5.5.8 *Insight 8: Complex Cross-Chain Exploits*

Three of the five largest losses in our dataset exploited bridge or multi-chain logic. These were Nomad Bridge ($190M) [258, 161, 151], Ronin Bridge ($12M) [185, 184], and Rho Market ($7.5M) [228, 229]. State synchronicity is difficult to reason about formally in these systems.

Cross-chain protocols face unique challenges in maintaining security invariants across multiple execution environments. These environments have different trust assumptions, consensus mechanisms, and finality guarantees.

The distributed nature of bridges creates multiple points of failure. Message relay mechanisms must securely transmit state updates between chains. Validator sets may have potentially misaligned incentives across different networks. State synchronization protocols must

handle reorgs and finality delays. Cross-chain transaction finality guarantees may be violated under adversarial conditions.

Each component represents a potential vulnerability. The composition of these components creates emergent attack surfaces that do not exist in single-chain protocols. The Nomad Bridge attack exploited a message verification flaw that allowed forged cross-chain messages. Incomplete upgrade procedures in Ronin Bridge left authentication checks disabled.

The emergence of these sophisticated cross-chain attacks calls for dedicated solutions. Monitoring and analysis of cross-chain logic [263] should track state across multiple chains. Formal verification techniques adapted to multi-chain environments can reason about cross-chain invariants. Insurance mechanisms specifically designed for bridge risks may be necessary given the concentration of value and complexity in these critical infrastructure components.

As the blockchain ecosystem fragments across L2s, side-chains, and alternative L1s, bridge security becomes increasingly critical to overall ecosystem security.

5.5.9 *Insight 9: Multi-Vulnerability Exploit Chains*

As established in Section 5.4, 13 out of 50 incidents involved chains of two or more vulnerabilities exploited in combination. Modern attacks rarely rely on a single flaw. An access control flaw or governance mistake often served as the entry point. Attackers then combined this with a price feed or arithmetic flaw to drain funds.

The Sturdy Finance attack [249, 248] leveraged both read-only reentrancy to create an inconsistent state and oracle manipulation to read inflated prices during that inconsistent state. The Jimbo Protocol exploit [251, 250] combined price manipulation through large swaps with access control flaws that permitted unauthorized rebalancing operations.

The LI.FI Protocol [193, 187, 186, 192], Hundred Finance [264, 194, 195], and Conic Finance [205] incidents also demonstrated multi-vulnerability exploit chains. The Euler Finance incident chained access control bypass with business logic flaws in liquidation mechanisms [150, 160, 158, 159].

Figure 5.2 visualizes the patterns of vulnerability chaining across our dataset. The Sankey diagram reveals that access control flaws most frequently serve as entry points (13 incidents), followed by price manipulation as the second-stage vulnerability. This combination reflects the attackers' strategy of first gaining unauthorized access and then exploiting economic mechanisms for profit extraction. Notable patterns include the prevalence of read-only reentrancy combined with price manipulation (3 incidents), reflecting the vulnerability pattern of the Curve pool.

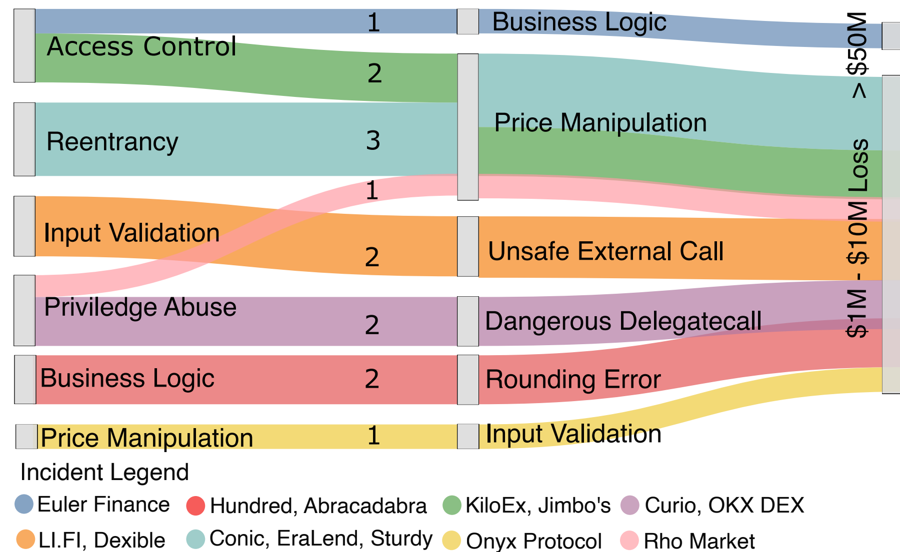

Figure 5.2: Vulnerability chains in analyzed incidents. The left column represents entry vulnerabilities, the right column shows second-stage vulnerabilities. Numbers on flows indicate incident count for each chain pattern.

This pattern fundamentally challenges the assumption that fixing individual vulnerabilities in isolation provides adequate security. Attackers increasingly think in terms of exploit chains. They identify how apparently benign operations can be sequenced to achieve malicious outcomes that no single operation would permit.

Defense must adopt a similar systemic perspective. We must consider not just whether individual functions are secure in isolation, but whether their composition creates exploitable states when called in particular sequences or under specific conditions. Security analysis tools must evolve beyond pattern matching for known vulnerability types. They need to model potential attack chains through state space exploration that considers all reachable states under adversarial transaction ordering.

The shift from isolated vulnerability scanning to compositional security analysis represents a necessary evolution in smart contract verification as attacks grow more sophisticated.

5.5.10 *Insight 10: Business Logic Flaws is Not Specific-Enough*

While we categorize business logic errors as a vulnerability class appearing in 5 incidents (10% of our dataset), our incident analysis reveals this label obscures more than it clarifies. The term encompasses fundamentally different failure modes that require distinct analysis and mitigation approaches.

The Euler Finance self-liquidation issue stems from perverse economic incentives in the liquidation discount mechanism [150, 160, 158,

159]. This mechanism rewarded borrowers for deliberately undercollateralizing their own positions. The KyberSwap tick manipulation exploits mathematical properties of concentrated liquidity pricing formulas [169, 170]. Specifically, it targeted the tick crossing logic in Uniswap v3-style AMMs.

The Abracadabra state desynchronization represents a distributed systems consistency failure [196, 259, 211, 260]. State updates in one contract component were not properly propagated to dependent components. The Yearn Finance incident involved a simple configuration error [188, 189]. The wrong contract address was hardcoded.

Lumping these diverse failure modes under the generic label business logic provides minimal actionable guidance for developers, auditors, or tool builders. An economic incentive flaw requires game-theoretic analysis and mechanism design expertise. A mathematical correctness issue demands formal verification of numerical properties. A state synchronization problem needs distributed systems reasoning about consistency models. A configuration error calls for better deployment procedures and automated checks.

Future taxonomies should decompose this catch-all category into more specific root causes. These include economic design flaws that create misaligned incentives, mathematical correctness issues in pricing or calculation formulas, state machine errors where contract state transitions violate intended invariants, and cross-contract invariant violations where properties that should hold across multiple contracts are broken.

5.5.11 *Insight 11: Complications of Proxy/Upgradability Patterns*

Proxy and upgradability patterns enable bug fixes and feature additions post-deployment. However, they introduce severe risks when admin keys are compromised or governance mechanisms controlling upgrades are subverted.

The OKX DEX incident involved compromised admin keys enabling malicious upgrades [240, 241]. Attackers gained access to the proxy admin and deployed implementation contracts containing backdoors. Tornado Cash [222, 223, 265] and Atlantis Loans [246, 247] suffered from governance manipulation. Attackers accumulated sufficient voting power or exploited governance vulnerabilities to pass malicious upgrade proposals.

The tension between upgradeability and security reflects a fundamental challenge in smart contract design. Immutability provides strong security guarantees but prevents bug fixes. This leaves protocols vulnerable to newly discovered exploits with no remediation path. Upgradeability enables rapid response to discovered vulnerabilities but creates a centralized point of failure in the upgrade mechanism itself. This mechanism becomes a high-value target for attackers.

Implementation approaches include graduated upgrade mechanisms with increasing scrutiny for more sensitive changes. Simple majority may suffice for parameter updates but supermajority should be required for logic changes. Supermajority approval should be required for critical upgrades that affect core security properties or manage large amounts of value. Mandatory time delays between proposal and execution enable community review and provide a window for users to exit if they disagree with proposed changes.

Potentially separating upgrade authority for different contract components limits blast radius. Compromise of one upgrade key cannot affect unrelated subsystems. Some protocols may conclude that the risks of upgradeability outweigh the benefits and opt for immutable designs with explicit limitations. Others may require upgradeability for regulatory compliance or to address unknown future vulnerabilities.

5.6 DISCUSSION AND IMPLICATIONS

Our empirical analysis of 50 high-impact incidents reveals patterns that diverge significantly from traditional vulnerability taxonomies. While the academic literature focuses primarily on implementation-level bugs, our dataset demonstrates that real-world attacks exploit a broader attack surface spanning design flaws, operational failures, and environmental dependencies.

5.6.1 *Gap Between Literature and Practice*

The academic literature identified in our systematic review (Section **??**) catalogs 25 active smart contract vulnerabilities. These vulnerabilities appear in real-world incidents. Access control flaws affect 13 incidents. Price manipulation appears in 13 cases. Reentrancy persists in 7 attacks (Table 1.1). However, this classification provides an incomplete picture of why attacks succeed.

Consider the Euler Finance incident. The exploit combined an unguarded auxiliary function with a flawed liquidation discount mechanism and flash loan amplification [150]. No single component violated known vulnerability patterns. The code executed exactly as designed. The design itself created the vulnerability.

Similarly, the Nomad Bridge exploit originated from a single incorrect parameter during contract initialization [151]. The code contained no reentrancy bugs, no integer overflows, and no access control flaws. An operational error during deployment rendered the system exploitable.

These incidents reveal that traditional vulnerability taxonomies, which organize weaknesses by implementation patterns, cannot capture the root causes of many high-impact attacks. A classification

scheme based solely on code-level vulnerabilities misses the design errors, governance failures, and dependency risks that enable exploitation.

5.6.2 *Four-Tier Root-Cause Framework*

Our findings necessitate a broader analytical framework. We propose organizing exploit origins into four tiers based on their fundamental causes. Tier 1 addresses flawed economic design and protocol logic, where the intended behavior itself creates vulnerabilities. Tier 2 covers protocol lifecycle and governance failures, including deployment errors and compromised keys. Tier 3 encompasses external dependency vulnerabilities, such as manipulable oracles and unvalidated inputs. Tier 4 contains implementation-level weaknesses, the traditional focus of academic research.

Table 5.2 presents this four-tier framework with sub-categories derived from our empirical analysis of 50 real-world incidents. Each tier is decomposed into specific sub-categories that capture distinct failure modes observed in our dataset. The description column characterizes each sub-category, while incident IDs reference specific attacks from Table 5.1 that exemplify each failure mode.

The framework reveals several important patterns. First, higher-tier vulnerabilities (Tiers 1 and 2) often serve as enablers for lower-tier exploitation. For instance, a flawed pricing model (Tier 1) combined with flash loan availability creates conditions for price manipulation attacks. Second, Tier 4 implementation vulnerabilities rarely succeed in isolation. The 31 multi-vulnerability exploit chains in our dataset (62% of incidents) typically involve at least one higher-tier weakness. Third, certain vulnerability types span multiple tiers depending on their root cause. Access control failures may stem from implementation errors (Tier 4) or from compromised administrative keys (Tier 2).

The framework demonstrates that securing smart contracts requires defense-in-depth across all four tiers. Code audits targeting Tier 4 vulnerabilities remain essential but insufficient. Protocols must also verify economic soundness (Tier 1), establish robust operational procedures (Tier 2), and validate external dependencies (Tier 3). Section 5.7 demonstrates the framework's application through a detailed case study of the LI.FI exploit, which exemplifies a multi-tier attack chain.

5.7 EXPLOIT CHAIN CASE STUDY: LI.FI GASZIP FACET

To illustrate how our multi-tier framework applies to real-world incidents, we present a detailed analysis of the LI.FI GasZip Facet exploit from July 16, 2024. This case demonstrates the exploit chain pattern identified in Insight 9, where attackers combine vulnerabilities across multiple tiers to maximize impact.

Table 5.2: A multi-tier root-cause framework derived from real-world incidents. Incident IDs refer to rows in Table 5.1.

Tier	Sub-Category	Description ← Incidents Involved (ID)
T1: Flawed Economic Design & Protocol Logic	Incorrect Pricing/-Valuation Models	The protocol's core formulas for calculating value are inherently manipulable. *Incidents: 4, 7, 19, 24, 30, 44, 48*
	State Inconsistency & Desync.	The logic allows different parts of the protocol to hold contradictory state information. *Incidents: 20, 28, 45, 47*
	Perverse Incentive Mechanisms	The design unintentionally rewards behavior that harms the protocol. *Incidents: 1*
T2: Protocol Lifecycle & Governance Failures	Clumsy Deployments & Upgrades	Mistakes during contract initialization or upgrades create vulnerabilities. *Incidents: 2, 14, 16, 41*
	Compromised Keys & Insider Threats	Admin keys are stolen or abused by internal actors to bypass security. *Incidents: 6, 11, 37, 43, 49*
	Governance System Exploitation	DAO rules are manipulated to pass malicious proposals or seize admin control. *Incidents: 34, 40, 46*
T3: External Dependency Vulnerabilities	Oracle & Price Feed Manipulation	The protocol uncritically trusts an external price feed susceptible to manipulation. *Incidents: 3, 10, 21, 38*
	Unvalidated User/-Contract Inputs	The protocol fails to sanitize data and external calls from untrusted sources. *Incidents: 13, 15, 18, 29, 31, 32, 35, 42*
T4: Implementation-Level Vulnerabilities	Interaction Pattern Failures	The code violates secure interaction patterns (e.g., CEI), enabling reentrancy. *Incidents: 5, 8, 25, 33, 50*
	Arithmetic Errors	The code is susceptible to over/underflow or precision loss due to EVM integer math. *Incidents: 9, 17, 22, 36, 39*
	Flawed Access Control Logic	A flaw in access control implementation allows it to be bypassed. *Incidents: 12, 23, 26*
	State & Data Handling Errors	The code incorrectly manages data representation or state (e.g., storage collisions). *Incidents: 27*

5.7.1 *Protocol, Code, and On-Chain Addresses*

LI.FI operates as a cross-chain liquidity aggregator using the Diamond proxy pattern (EIP-2535). The protocol's main Diamond contract is deployed at address `0x1231...4eae` on Ethereum mainnet (verified as "LI.FI: LiFi Diamond" on Etherscan). The vulnerable GasZip facet implementation resided at address `0xf28a...2534` [266, 267]. The attacker executed the exploit from externally owned account (EOA) `0x8b3c...dcf3`, with the primary attack transaction recorded as `0xa630...7537f` [268].

5.7.2 *Incident Summary*

On July 16, 2024, LI.FI suffered an exploit that drained approximately $8–12 million from users who had previously granted infinite token approvals to the LI.FI Diamond proxy [266, 269, 270]. Post-mortem

analyses attribute the root cause to a lack of validation on externally supplied call data (input validation vulnerability) in the GasZip facet, which delegated user-controlled calldata to a swap helper without whitelisting or sanity checks (unsafe external call) [267, 271]. Because many users had granted unlimited token allowances to the Diamond contract for legitimate swap operations, the attacker could craft calls that used the Diamond as the approved spender to invoke `transferFrom` from victims to the attacker [268].

5.7.3 *Vulnerable Code*

The `depositToGasZipERC20` function in `GasZipFacet.sol` (verified at Etherscan address `0xf28a...2534`) forwards unvetted calldata into `LibSwap.swap`, which performs an external call based on user-provided parameters. A minimal excerpt from the verified source code is reproduced below:

Listing 5.1: Excerpt from GasZipFacet.sol

```
function depositToGasZipERC20(
    LibSwap.SwapData calldata _swapData,
    uint256 _destinationChains,
    address _recipient
) public {
    uint256 currentNativeBalance = address(this).balance;
    // *** Unvalidated external call based on
    // user-controlled SwapData ***
    LibSwap.swap(0, _swapData);
    uint256 swapOutputAmount = address(this).balance -
        currentNativeBalance;
    gasZipRouter.deposit{value: swapOutputAmount}(
        _destinationChains, _recipient
    );
}
```

The `LibSwap.swap` function executes calls to arbitrary `callTo` addresses with attacker-supplied `callData`, while the Diamond acts as `msg.sender`. Given prior unlimited allowances to the Diamond, this enabled crafted `transferFrom` calls on ERC-20 tokens that pulled funds from approved users to the attacker [267, 271].

5.7.4 *Exploit Chain Reconstruction*

We reconstruct the attack as a three-step exploit chain spanning multiple tiers of our framework:

STEP 1: ENTRY POINT – INPUT VALIDATION FAILURE (TIER 3) The GasZip facet exposed a public method that delegates user-controlled calldata to an external call. No whitelist or sanity checks were enforced on the `callTo` address or `callData` payload in

this facet, in contrast to patterns reportedly used elsewhere in the LI.FI system [267, 271]. This input validation failure (Tier 3) created an arbitrary call primitive.

STEP 2: AMPLIFIER PRE-EXISTING UNLIMITED APPROVALS (TIER 3) Many users had previously granted infinite ERC-20 allowances to the Diamond contract (`0x1231...4eae`) for legitimate swap operations. The attacker could therefore craft `callData` to invoke `transferFrom(victim, attacker, amount)` on victim tokens through the Diamond, which acted as the approved spender. Evidence of this pattern appears in attack transaction `0xa630...7537f`, originating from attacker EOA `0x8b3c...dcf3` [268].

STEP 3: EXTRACTION – STOLEN STABLECOINS AGGREGATED AND ROUTED Stolen stablecoins were aggregated, swapped (e.g., to ETH), and routed to the attacker's addresses, as outlined in contemporaneous security reports [269, 270].

5.7.5 *Breaking the Exploit Chain*

Table 5.3 maps each step of the exploit chain to concrete defenses. Notably, breaking the first link (validating or whitelisting external calls in the facet) would have neutralized the entire attack class. The amplifier (infinite approvals) still leaves users exposed to any future arbitrary-call primitive.

Table 5.3: Breaking the LI.FI GasZip exploit chain

Exploit Chain Link	Actionable Defenses
Entry: Unvetted external call via `LibSwap.swap`	Per-facet contract call allow-list (only sanctioned DEX/router addresses); strict decoding/validation of `swapData`; forbid arbitrary `callTo`
Amplifier: Users' infinite approvals to Diamond	Reduce reliance on global infinite approvals (bounded/expiring allowances; permit-per-transaction); automated user notice to revoke stale approvals after use
Extraction: Sweeping tokens	On-chain anomaly monitors (large `transferFrom` originating from Diamond to non-whitelisted sinks)

This case study demonstrates that even well-audited protocols remain vulnerable when new facets introduce input validation failures (Tier 3) that interact with external dependencies (user approvals). The multi-tier nature of the exploit underscores the necessity of defense-in-depth across all four tiers of our framework, not merely code-level audits targeting Tier 4 implementation bugs.

5.8 summary

This chapter presented a analysis of 50 real-world smart contract incidents between 2022 and 2025. These incidents represent over $1.09 billion in financial losses. We developed a four-stage incident review protocol for systematic data collection, prioritization, and analysis. The analysis revealed that seven vulnerability types dominated actual attacks. Access Control led with 13 incidents causing $417.95 million. Price Manipulation appeared in 13 incidents causing $279.75 million. Reentrancy contributed only 11% of total losses despite extensive academic attention.

A critical finding emerged: 26% of successful attacks exploited chains of multiple vulnerabilities rather than isolated weaknesses. We documented 11 key insights including flash loans as attack amplifiers, pervasiveness of human error, and inadequate emergency response mechanisms. The gap between academic research priorities and real-world exploitation patterns motivated our four-tier root-cause framework. This framework organizes vulnerabilities by fundamental causes rather than implementation patterns. These empirical findings guide our technical solutions. Chapter 7 introduces TaintSentinel for semantic-aware Bad Randomness detection. Chapter 8 presents SmartTaintRL for intelligent path prioritization.

Part III

NOVEL METHODS

6 RISK-STRATIFIED BENCHMARK DATASET FOR BAD RANDOMNESS

6.1 introduction

The detection methods presented in Chapters 7 and 8 require labeled datasets for training and evaluation. However, as discussed in Chapter 1, existing benchmark datasets for Bad Randomness vulnerabilities suffer from critical limitations. The SWC Registry contains only 2 test cases. SmartBugs Curated provides 8 contracts. RNVulDet, the largest prior dataset, includes just 34 vulnerable samples. These numbers are insufficient for training machine learning models or conducting comprehensive tool evaluation.

Beyond the size limitation, existing datasets share a more fundamental problem: they use binary labels. A contract is either vulnerable or safe. This binary classification ignores the reality that exploitability varies significantly depending on protective mechanisms and attacker capabilities. A contract with an `onlyOwner` modifier on its randomness function presents a different risk profile than one with no protection at all. Current datasets do not capture this distinction.

This chapter addresses both limitations by constructing a risk-stratified benchmark dataset of 1,758 labeled contracts. The dataset classifies contracts into four categories based on actual exploitability: Publicly_Exploitable for contracts with no protection, Miner_Exploitable for those vulnerable only to miners, Owner_Only for contracts exploitable only by owners, and Safe for contracts using secure randomness sources like Chainlink VRF.

A key innovation of our methodology is function-level validation. Existing detection tools check whether a contract contains both a vulnerability pattern and a mitigation mechanism. They assume the mitigation protects the vulnerable code. Our analysis revealed this assumption is often wrong. In 49% of contracts initially classified as protected, the `onlyOwner` modifier was applied to a different function than the one containing the vulnerability. The randomness function remained publicly accessible despite the presence of access control elsewhere in the contract.

The relationship between this benchmark and the datasets used in subsequent chapters deserves clarification. This chapter constructs a benchmark with four risk levels, designed for general evaluation of detection tools. The dataset prioritizes breadth and includes contracts across diverse domains. Chapters 7 and 8 apply stricter filtering criteria to create a more focused dataset for training and evaluating

our detection methods. The stricter criteria ensure higher confidence in labels and remove edge cases that could confuse machine learning models. Both datasets originate from the same SmartBugs-Wild collection, but serve different purposes within this thesis.

The five-phase construction methodology proceeds as follows. Phase 1 filters 47,398 contracts to 17,466 using keyword matching. Phase 2 applies 58 regular expressions organized into 9 semantic groups to identify vulnerability patterns. Phase 3 classifies contracts into risk levels based on detected mitigations. Phase 4 validates that mitigations actually protect vulnerable functions. Phase 5 performs context-aware refinement to exclude legitimate uses of block attributes such as mining tokens.

The main contributions of this chapter are:

1. **Largest SWC-120 benchmark dataset:** We analyzed 17,466 contracts and identified 1,752 vulnerable contracts, significantly expanding prior benchmarks including RNVulDet (34 contracts), TaintSentinel (384 contracts), and SmartBugs Curated (8 contracts).

2. **Attacker-based classification:** We stratify contracts as SAFE, OWNER_ONLY, MINER_EXPLOITABLE, or PUBLICLY_EXPLOITABLE based on which attacker types can exploit them, rather than using binary vulnerable/safe labels that ignore partial mitigations.

3. **Function-level validation:** We verify that protective modifiers are applied directly to functions containing vulnerable code, revealing that 49% of apparently protected contracts were actually exploitable.

4. **Tool evaluation baseline:** We demonstrate that existing tools (Slither, Mythril) achieve 0% recall on our dataset, establishing a baseline for measuring future improvements.

The remainder of this chapter is organized as follows. Section 6.2 provides background on bad randomness vulnerabilities and mitigation mechanisms. Section 6.3 describes the five-phase dataset construction methodology. Section 6.4 evaluates existing detection tools and discusses implications. Section 6.5 summarizes the chapter.

6.2 BACKGROUND AND RELATED WORK

This section provides an overview of bad randomness vulnerabilities in Solidity, discusses common mitigation mechanisms, and reviews existing detection tools and benchmark datasets.

6.2.1 *Mitigation Mechanisms*

Several approaches can mitigate bad randomness vulnerabilities. **Commit-Reveal Schemes** use a two-phase protocol where participants first submit a hash of their secret value, then reveal the actual value after all commitments are collected; the combined reveals produce randomness that no single party could predict. **Chainlink VRF** is a decentralized oracle providing verifiable random numbers with cryptographic proofs [272]. For each request, Chainlink VRF generates one or more random values and cryptographic proof of how those values were determined, which is published and verified on-chain before any consuming application can use it. **Access Control** restricts function access via modifiers such as `onlyOwner`, limiting who can exploit the weakness; however, this does not eliminate the vulnerability—it only reduces the attack surface to trusted parties.

6.2.2 *Existing Detection Tools and Datasets*

Several static analysis tools detect bad randomness patterns:

- **Slither** [10]: Reports "weak-prng" warnings for contracts using block variables in arithmetic operations.
- **Mythril** [273]: Uses symbolic execution to detect predictable randomness.
- **SmartCheck** [60]: Pattern-based analyzer that flags dangerous randomness sources.

However, these tools operate at the contract level, flagging any contract containing both a vulnerability pattern and a mitigation regardless of whether the mitigation actually protects the vulnerable code.

Existing benchmark datasets for SWC-120 are summarized in Table 6.1. The limited size and lack of validation in these datasets motivate this work.

Table 6.1: Existing SWC-120 Benchmark Datasets

Dataset	**Samples**	**Risk**	**Valid.**
SWC Registry [274]	2	No	No
SmartBugs Curated [275]	8	No	Partial
RNVulDet [17]	34	No	Unknown
TaintSentinel (Chapter 7)	384	No	Yes
Our Dataset	**1,758**	**Yes**	**Yes**

6.3 DATASET CONSTRUCTION AND VALIDATION

This section describes the multi-phase approach to constructing a labeled benchmark dataset for SWC-120 (Bad Randomness) vulnerabilities. The methodology addresses a key limitation of existing detection tools: their inability to distinguish between *pattern presence* and *actual exploitability*. We introduce function-level validation, risk-based classification, and context-aware analysis to improve labeling accuracy.

The pipeline consists of five phases: *(i)* data collection and keyword filtering (Section 6.3.1), *(ii)* vulnerability pattern labeling (Section 6.3.2), *(iii)* risk-level classification (Section 6.3.3), *(iv)* function-level validation (Section 6.3.4), and *(v)* context-aware refinement (Section 6.3.5). Figure 6.1 illustrates the overall pipeline.

6.3.1 *Phase 1: Data Collection and Keyword Filtering*

We used the SmartBugs-Wild dataset [275] as our data source, containing 47,398 unique Ethereum smart contracts collected from Etherscan. Since most contracts do not involve randomness generation, we applied keyword filtering to identify potentially relevant contracts.

We filtered contracts containing at least one of the following block attributes: `block.timestamp`, `blockhash`, `block.difficulty`, `block.number`, `block.coinbase`, and `block.gaslimit`. This step reduced the dataset to **17,466 contracts** (63.1% reduction). Note that the presence of these keywords does not imply vulnerability; many contracts use block attributes for legitimate purposes such as time tracking. The subsequent phases distinguish between safe and vulnerable usage patterns.

6.3.2 *Phase 2: Vulnerability Pattern Labeling*

We developed a pattern-based labeler using regular expressions organized into 9 semantic groups. These patterns capture bad randomness sources documented in SWC-120 [274], along with additional patterns identified during analysis such as `block.prevrandao` and `gasleft()`. Table 6.2 summarizes the pattern groups.

Group G1 detects direct modulo operations with block attributes, such as `block.timestamp%players.length`. Groups G2 and G3 address a common misconception that hashing predictable values provides security: G2 identifies type casting from `keccak256` or `sha3` to `uint`, while G3 detects hashed values used with modulo operations. Groups G4 and G5 target `blockhash` usage, including the deprecated `block.blockhash` syntax and assignments to variables such as `answer` or `result`. Group G6 matches assignments to variables named `seed` or `random` from predictable sources. Group G7 identifies winner selection patterns using block attributes. Group G8 detects stored block num-

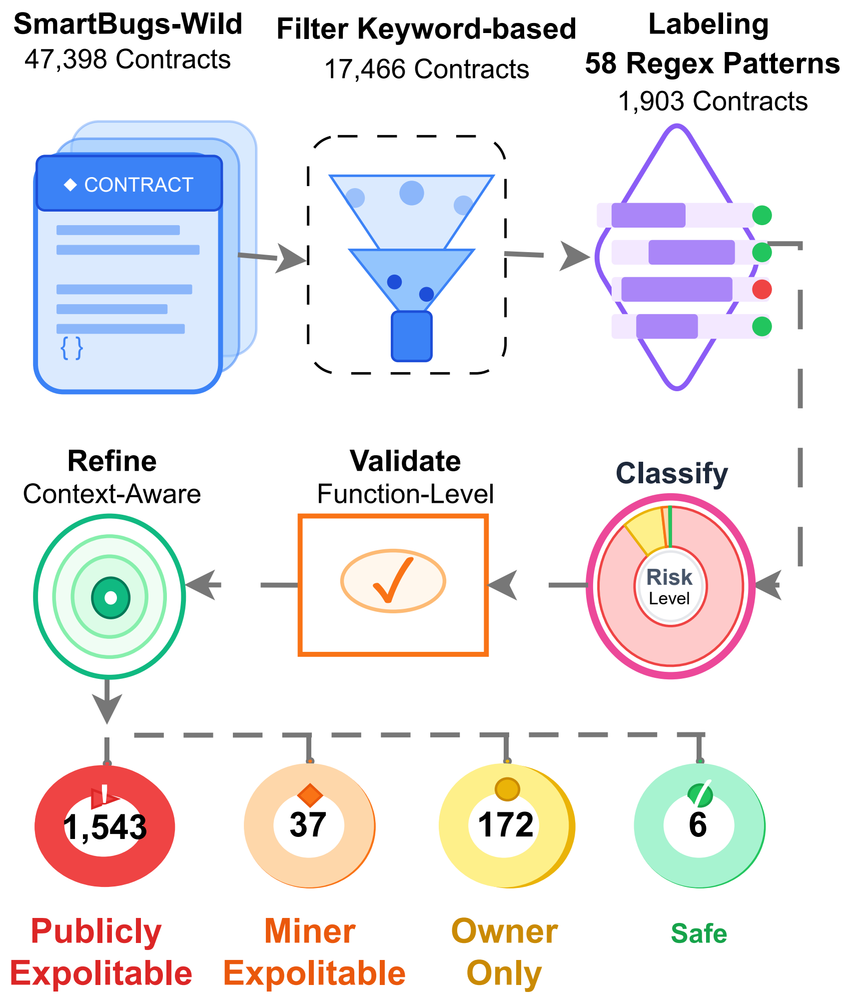

Figure 6.1: Dataset construction pipeline showing contract counts at each phase. The final output is categorized into four risk levels.

bers and direct `uint` casts from `blockhash`. Group G9 captures contextual patterns where randomness-related keywords appear alongside `keccak256` with block attributes.

Table 6.2: Vulnerability Detection Patterns Organized by Semantic Category

ID	Pattern Category	Count
G1	Direct modulo with block attributes	10
G2	Type cast from `keccak256/sha3` to `uint`	11
G3	`keccak256` hash with modulo operator	15
G4	`keccak256` with `block.blockhash`	1
G5	`blockhash` as answer/comparison	4
G6	Seed/random variable with predictable source	10
G7	Winner selection using block attributes	2
G8	Stored block number and uint cast from blockhash	3
G9	Randomness context with `keccak256`	2
	Total	**58**

The labeler first checks for Chainlink VRF usage; contracts with VRF are labeled SAFE. Remaining contracts are labeled as vulnerable if they match at least one pattern. Of the 17,466 filtered contracts, **1,903** (10.8%) matched one or more patterns and proceeded to risk classification.

6.3.2.1 *Pattern Distribution Analysis*

Figure 6.2 shows the distribution of detected patterns across the 1,903 candidate contracts. Group G1 (direct modulo operations) was the most prevalent, appearing in 687 contracts (36.1%). Groups G2 and G6 followed with 412 (21.7%) and 298 (15.7%) contracts, respectively. The high frequency of G1 patterns indicates that developers commonly use the simplest and most vulnerable form of randomness generation. Groups G4 and G9 were the least common, each appearing in fewer than 50 contracts.

6.3.3 *Phase 3: Risk-Level Classification*

Not all bad randomness patterns carry the same level of risk. Exploitability depends on who can trigger the randomness function and what protective mechanisms are in place. We classify contracts into four risk levels based on detected mitigations and their effectiveness against three attacker types: *external attackers* who deploy malicious contracts, *miners* who manipulate block attributes, and *owners* who control privileged functions. Table 6.3 summarizes the classification criteria and initial results.

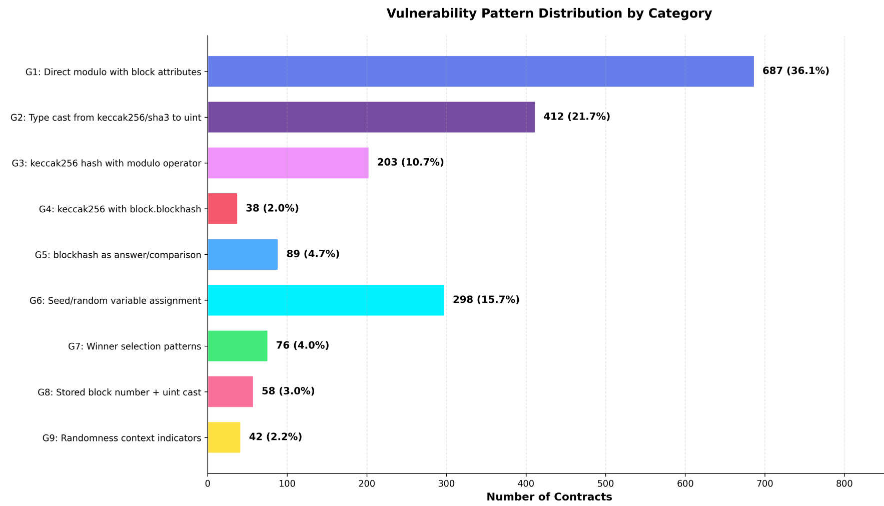

Figure 6.2: Distribution of vulnerability patterns across 1,903 candidate contracts. G1 (direct modulo) and G2 (uint cast) together account for 57.8% of all patterns.

Table 6.3: Risk Levels, Attacker Capabilities, and Initial Classification Results

Category	Mitigation	Ext.	Miner	Owner	Count	%
SAFE	VRF / Commit-Reveal	×	×	×	6	0.3
OWNER_ONLY	`onlyOwner`	×	×	✓	1,167	61.3
MINER_EXPLOITABLE	`tx.origin` / future block	×	✓	✓	47	2.5
PUBLICLY_EXPLOITABLE	None	✓	✓	✓	683	35.9
Total					**1,903**	**100**

✓ can exploit × cannot exploit

SAFE. Contracts that use Chainlink VRF (detected via `VRFConsumerBase`, `requestRandomWords`, or `fulfillRandomness`) or implement a Commit-Reveal scheme (detected via paired `commit` and `reveal` functions). VRF provides cryptographically secure randomness through an external oracle; Commit-Reveal separates commitment from revelation, preventing prediction attacks.

OWNER_ONLY. Contracts where vulnerable functions are restricted to privileged addresses via patterns such as `onlyOwner`, `onlyAdmin`, or `require(msg.sender==owner)`. External attackers and miners cannot invoke the protected function; only the contract owner can trigger the vulnerable code.

MINER_EXPLOITABLE. Contracts that implement `tx.origin==msg.sender` checks or use a future block pattern (e.g., `block.number+delay`). The `tx.origin` check ensures the caller is an externally owned account,

blocking contract-based attacks; miners can still manipulate block attributes within protocol limits.

PUBLICLY_EXPLOITABLE. Contracts with no detected mitigation. The randomness function is publicly callable and relies on predictable block attributes. Any attacker can deploy a contract to predict the outcome within the same transaction.

The high proportion of OWNER_ONLY contracts (61.3%) was unexpected and prompted further investigation. As shown in Figure 6.3, subsequent validation phases revealed that many of these classifications were incorrect because the mitigation was not applied to the vulnerable function.

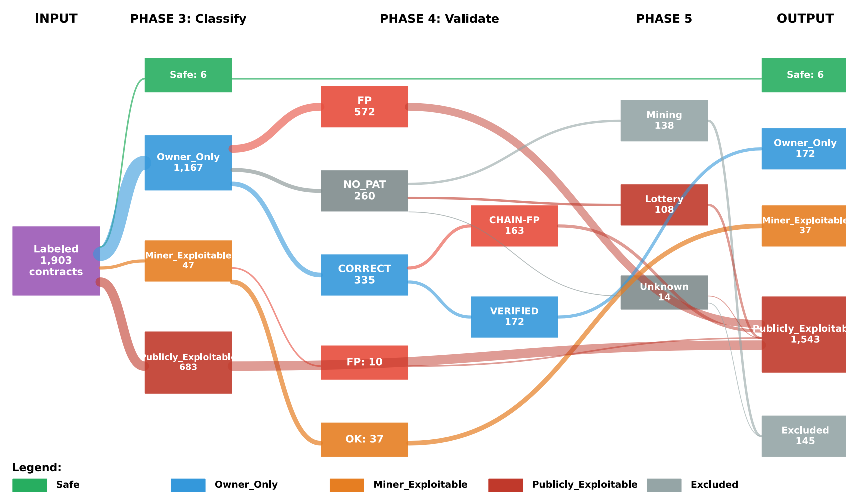

Figure 6.3: Validation flow showing contract reclassification across phases. The OWNER_ONLY category decreased from 1,167 to 172 contracts (85.3% false positive rate) after function-level validation.

6.3.4 *Phase 4: Function-Level Validation*

A critical observation during manual review was that existing tools produce false positives when assessing mitigation effectiveness. A contract may contain both an `onlyOwner` modifier and a bad randomness pattern, but if they appear in *different functions*, the vulnerability remains fully exploitable. For example, `onlyOwner` might protect a `withdraw` function while the `playLottery` function containing the bad randomness pattern remains publicly accessible.

We developed a function-level validation algorithm to verify that mitigation mechanisms actually protect the vulnerable code. The algorithm extracts all functions from the contract source code using bracket counting to handle nested structures. For each function, it checks whether any bad randomness pattern from Table 6.2 is present.

If a vulnerable function is `public` or `external`, the algorithm verifies that the mitigation modifier is applied directly to that function. For `internal` or `private` functions, the algorithm traces the call chain to identify all public callers and verifies that each caller has the required mitigation.

Algorithm 1 provides the formal specification. The algorithm returns one of three verdicts: `CORRECT` if all vulnerable functions are properly protected, `FALSE_POSITIVE` if any vulnerable function lacks protection, or `NO_PATTERN_IN_FUNCTIONS` if the pattern exists at contract level but not within any function body.

Algorithm 1 Function-Level Mitigation Validation

Require: Contract source C, expected mitigation M
Ensure: Verdict ∈ {Correct, FalsePositive, NoPattern}
1: $F \leftarrow \text{ExtractFunctions}(C)$
2: $V \leftarrow \{f \in F \mid \text{ContainsBadPattern}(f)\}$
3: **if** $V = \emptyset$ **then**
4:     **return** NoPattern
5: **end if**
6: **for all** $f \in V$ **do**
7:     **if** f.visibility ∈ {`public`, `external`} **then**
8:         $\textit{isProtected} \leftarrow \text{HasMitigation}(f, M)$
9:     **else**
10:         $\textit{callers} \leftarrow \text{GetPublicCallers}(f, F)$
11:         $\textit{isProtected} \leftarrow \forall c \in \textit{callers} : \text{HasMitigation}(c, M)$
12:     **end if**
13:     **if** $\neg\textit{isProtected}$ **then**
14:         **return** FalsePositive
15:     **end if**
16: **end for**
17: **return** Correct

Applying this algorithm to the 1,167 Owner_Only contracts revealed significant misclassification. In the first validation pass, 572 contracts (49.0%) were reclassified as False_Positive because the `onlyOwner` modifier was not applied to the function containing the bad randomness pattern. An additional 260 contracts showed No_Pattern_In_Functions, indicating the pattern existed outside function bodies.

For the 335 contracts that passed initial validation, we performed a second pass with call-chain analysis. This identified 163 additional False_Positive cases (48.7%) where an `internal` function with bad randomness was called by an unprotected `public` function. After both validation passes, only **172 contracts** (14.7% of the original 1,167) were confirmed as correctly classified Owner_Only.

We applied the same validation to the 47 MINER_EXPLOITABLE contracts, identifying 10 FALSE_POSITIVE cases and 4 NO_PATTERN_IN_FUNCTIONS cases, leaving **37 confirmed** MINER_EXPLOITABLE contracts.

6.3.5 *Phase 5: Context-Aware Refinement*

Some contracts legitimately use `blockhash` or `block.number` for purposes other than randomness generation. Mining tokens use these values for Proof-of-Work puzzles where predictability does not constitute a vulnerability—the computational cost of mining provides security, not unpredictability.

We performed context analysis on the 260 contracts where patterns appeared outside callable functions (the NO_PATTERN_IN_FUNCTIONS cases from Section 6.3.4). The analysis examined keyword frequency across contract names, function names, and variable names to determine the intended use of block attributes.

Contracts containing keywords such as `mint`, `difficulty`, `nonce`, `mining`, or `reward` were classified as MINING tokens. Contracts containing keywords such as `lottery`, `jackpot`, `prize`, `bet`, or `gamble` were classified as LOTTERY applications where the bad randomness pattern represents a genuine vulnerability. Contracts with insufficient context were flagged as UNKNOWN for manual review.

Table 6.4 shows the classification results. Mining tokens (138 contracts) were excluded from the final dataset because their use of block attributes is not intended for randomness generation—they use these values for Proof-of-Work puzzles where security comes from computational cost, not unpredictability. Similarly, 7 contracts from the UNKNOWN category were excluded as they use block attributes solely for time tracking purposes, not for generating random values. In total, 145 contracts were excluded as out-of-scope. The remaining contracts were classified as follows: Lottery contracts (108) were labeled as PUBLICLY_EXPLOITABLE, and 7 UNKNOWN contracts identified as vulnerable gambling applications were also labeled as PUBLICLY_EXPLOITABLE.

Table 6.4: Context-Aware Classification of 260 NO_PATTERN_IN_FUNCTIONS Contracts

Category	Count	%	Final Classification
MINING	138	53.1	Excluded (not randomness)
LOTTERY	108	41.5	PUBLICLY_EXPLOITABLE
UNKNOWN	14	5.4	Manual review†
Total	**260**	**100**	

†Result: 7 Excluded (time tracking), 7 PUBLICLY_EXPLOITABLE

6.3.6 *Final Dataset and Comparison*

After all validation phases, the final labeled dataset contains **1,758 contracts** distributed as shown in Table 6.5. The dataset is publicly available on GitHub[1].

Table 6.5: Final Dataset Composition

Category	Count	%	Label
PUBLICLY_EXPLOITABLE	1,543	87.8	Vulnerable
OWNER_ONLY	172	9.8	Vulnerable*
MINER_EXPLOITABLE	37	2.1	Vulnerable*
SAFE	6	0.3	Not Vulnerable
Total	**1,758**	**100**	—

*Vulnerable with partial mitigation (limited attacker scope)

6.3.6.1 *Comparison with Existing Datasets*

Table 6.6 compares our dataset with existing SWC-120 benchmark resources. The SWC Registry provides only 2 example contracts for bad randomness. SmartBugs Curated contains 8 contracts in the `bad_randomness` category. RNVulDet, the most comprehensive prior work, provides 34 contracts with documented vulnerabilities plus 214 contracts labeled as safe.

Our dataset substantially exceeds existing benchmarks in both scale and annotation quality, while being the first to include risk-level classification. The function-level validation methodology demonstrated here can serve as a template for improving labeling accuracy in other vulnerability categories.

Table 6.6: Comparison with Existing SWC-120 Benchmark Datasets

Dataset	Vuln.	Safe	Risk	Func.	Year
SWC Registry	2	—	×	×	2018
SmartBugs Curated	8	—	×	×	2020
RNVulDet	34	214	×	×	2023
Ours	**1,752**	**6**	✓	✓	2026

Risk = Risk-level classification   Func. = Function-level validation

1 `https://github.com/HadisRe/BadRandomness-SWC120-Dataset`

6.4 DISCUSSION

6.4.1 *Comparison with Existing Detection Tools*

To evaluate the effectiveness of existing vulnerability detection tools on our dataset, we tested Slither [10] and Mythril [273], two widely-used tools for detecting SWC-120 vulnerabilities (see Section 6.2.2).

6.4.1.1 *Experimental Setup*

We executed Slither (v0.11.3) on all 1,752 vulnerable contracts and 6 safe contracts. For Mythril (v0.24.8), we tested a sample of 76 contracts (56 vulnerable, 20 safe) due to its longer analysis time. Both tools were configured with a 120-second timeout per contract.

6.4.1.2 *Results*

Table 6.7 presents the detection performance of both tools compared to our ground truth labels.

Table 6.7: Detection results of Slither and Mythril on our dataset.

Tool	Tested	TP	FP	FN	Recall	F1
Slither	1,758	0	0	1,752	0.0%	0.0%
Mythril	76	0	0	56	0.0%	0.0%

Both Slither and Mythril achieved 0% recall, failing to detect any of the vulnerable contracts in our dataset.

6.4.1.3 *Analysis of Detection Failure*

To understand why existing tools failed, we analyzed the detection patterns used by each tool:

- **Slither's weak-prng detector** only identifies direct modulo operations on `block.timestamp`, `now`, or `blockhash`. It does not detect patterns involving `keccak256` hashing, type casting, or indirect usage through variables.
- **Mythril's SWC-120 detector** uses symbolic execution to find timestamp dependence but focuses primarily on direct comparisons rather than the complex patterns commonly found in real-world contracts.

Figure 6.4 illustrates the coverage gap. Our dataset contains 9 pattern groups (G1–G9), but Slither only partially covers G1, and Mythril covers a similarly narrow subset. Groups G2–G9, which account for 63.9% of our dataset, are completely undetected by both tools.

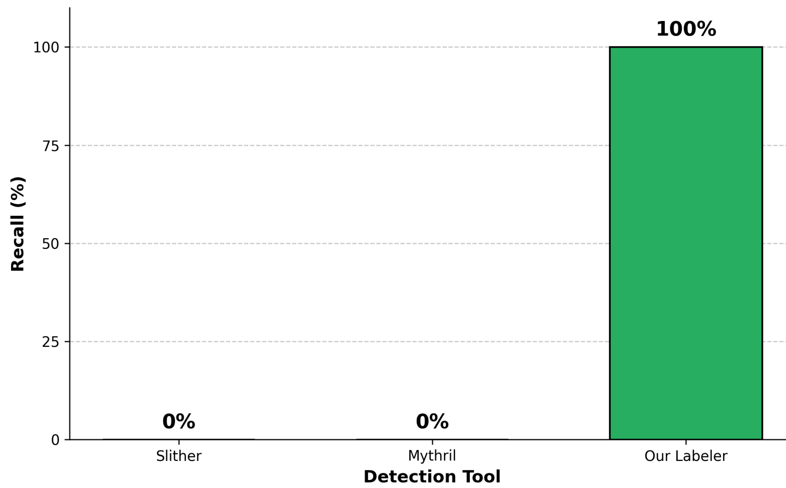

Figure 6.4: Recall comparison of vulnerability detection tools. Existing tools (Slither, Mythril) achieve 0% recall on our dataset, while our pattern-based labeler achieves 100% recall by design.

6.4.1.4 *Validation of Tool Behavior*

To confirm that the 0% recall reflects tool limitations rather than labeling errors, we ran controlled experiments on synthetic contracts and reviewed the official documentation.

SLITHER. The Slither documentation[2] states that the `weak-prng` detector identifies:

> *"Weak PRNG due to a modulo on `block.timestamp`, `now` or `blockhash`."*

We tested Slither on a minimal contract:

Listing 6.1: Test contract with direct modulo pattern.

```
pragma solidity ^0.8.0;
contract TestSimple {
    function getRandom() public view returns (uint256) {
        return block.timestamp % 100;
    }
}
```

Slither detected this pattern correctly. However, when we ran Slither on a SmartBugs-Wild contract written in Solidity 0.4.11, it failed with a compilation error: "`Source file requires different compiler version.`"

This confirms two points: (1) Slither can detect simple modulo patterns when compilation succeeds, and (2) Slither cannot analyze contracts written in older Solidity versions without legacy compiler

2 `https://github.com/crytic/slither/wiki/Detector-Documentation`

support. Since most SmartBugs-Wild contracts use Solidity 0.4.x–0.5.x, this explains why Slither reported no detections.

MYTHRIL. The Mythril documentation[3] describes the `predictable_variables` module as checking whether *"control flow decisions are influenced by block.timestamp or block.number."*

We ran Mythril on the same test contract. The output was: `"No issues were detected."` Mythril did not flag the randomness pattern because its detector targets control flow statements (e.g., `if (block.timestamp > x)`), not arithmetic operations used for random number generation.

On the legacy SmartBugs-Wild contract, Mythril compiled successfully and reported SWC-110 and SWC-101—but not SWC-120. GitHub Issue #1432[4] independently confirms this behavior, with a user reporting that Mythril missed an SWC-120 violation involving `BLOCKHASH`.

SUMMARY. The 0% recall is not a labeling artifact. Slither fails on legacy contracts due to compilation requirements, and Mythril's detector does not target randomness generation patterns.

6.4.1.5 *Pattern Development Process*

We developed our detection patterns through iterative refinement using 32 reference contracts from the SWC Registry [274], SmartBugs Curated [275], and contracts we manually verified. When a reference contract was not detected, we analyzed its code structure and added the missing pattern to our rule set. This process continued until all reference contracts were covered.

This approach constitutes pattern *development*, not independent *validation*. We did not hold out a separate test set to measure accuracy on unseen data. However, several aspects of our methodology support label reliability:

Pattern sources. The 58 regular expressions capture vulnerability constructs documented in the SWC-120 Registry, security reports from SlowMist [**slowmist2019random**] and ImmuneBytes [**immunebytes2023random**], and prior academic work on smart contract vulnerabilities.

Function-level analysis. Phase 4 of our methodology verified whether protective modifiers guard the specific functions containing vulnerabilities. This step reclassified 49% of contracts that would otherwise be mislabeled as protected.

Context filtering. Phase 5 excluded 145 contracts where block attributes served legitimate purposes (e.g., mining tokens, timestamps for logging). This reduced false positives from pattern matching alone.

3 https://mythril-classic.readthedocs.io/en/master/module-list.html
4 https://github.com/ConsenSysDiligence/mythril/issues/1432

Conservative criteria. We label a contract as vulnerable only when it matches a known vulnerability pattern and lacks verified protection on the vulnerable function.

SmartBugs Curated, a widely-used benchmark, contains 143 annotated contracts [275]. Our dataset provides 1,752 contracts with function-level validation and risk stratification. While we did not manually verify every contract, the multi-phase methodology reduces systematic labeling errors. Future work could strengthen confidence through inter-rater agreement studies on a random sample.

6.4.2 *Implications for Detection Tools*

Our findings reveal a fundamental limitation in current vulnerability detection tools. Contract-level pattern matching—checking whether a contract contains both a vulnerability pattern and a mitigation mechanism—produces unacceptably high false positive rates (49% in our evaluation). Detection tools should implement function-level analysis to verify that mitigations actually protect vulnerable code paths.

6.4.3 *The Prevalence Problem*

Our dataset reveals that 87.8% of contracts with bad randomness patterns have no mitigation whatsoever, classified as PUBLICLY_EXPLOITABLE in our attacker-based classification. This finding was only possible through our function-level validation, which distinguished truly unprotected contracts from those where mitigations exist but fail to guard vulnerable functions.

This prevalence has significant implications for the research community. First, it demonstrates that bad randomness remains a widespread problem despite years of published research and tool development, indicating a critical gap between academic knowledge and industry practice. Detection tools should focus on the 87.8% of PUBLICLY_EXPLOITABLE contracts where exploitation requires no special privileges, rather than treating all vulnerabilities equally.

For the developer community, this finding represents substantial financial risk. Many of these unprotected contracts manage funds in DeFi protocols, lotteries, and token distributions where predictable randomness enables direct theft. The high prevalence suggests that developers either remain unaware of this vulnerability class or underestimate its severity.

6.4.4 *Limitations*

Our approach has several limitations. First, we analyze only contracts with available source code, excluding bytecode-only contracts. Second,

our 58 patterns may not capture all possible bad randomness implementations. Third, we do not trace randomness usage across contract boundaries through inter-contract calls. Fourth, new vulnerability patterns may emerge as Solidity evolves. Finally, Mythril was evaluated on a sample of 76 contracts due to computational constraints; results may vary on the full dataset.

6.5 SUMMARY

This chapter presented a benchmark dataset of 1,758 smart contracts labeled for bad randomness (SWC-120) vulnerabilities. The five-phase methodology data collection, pattern labeling, risk classification, function-level validation, and context-aware refinement addresses critical limitations of existing datasets and detection tools.

The key contributions are: (1) the largest validated SWC-120 benchmark dataset with 1,752 vulnerable contracts; (2) an attacker-based classification with four categories (SAFE, OWNER_ONLY, MINER_EXPLOITABLE, PUBLICLY_EXPLOITABLE); (3) function-level validation revealing that a substantial portion of apparently protected contracts were actually exploitable due to misplaced mitigations; and (4) empirical evidence that existing tools (Slither, Mythril) achieve 0% recall on diverse bad randomness patterns.

Our dataset is publicly available to support future research in smart contract security analysis and the development of more comprehensive detection tools. A filtered subset of this benchmark with more stringent criteria is used in Chapter 7 and Chapter 8 for training and evaluating the proposed vulnerability detection methods.

7 TAINTSENTINEL: SEMANTIC-AWARE DETECTION

7.1 INTRODUCTION

Bad randomness vulnerabilities represent a critical yet underexplored threat in smart contract security. As established in Chapter 1, the deterministic and transparent nature of blockchain technology fundamentally prevents true random number generation [18]. This limitation creates severe security risks for applications requiring unpredictable values, including decentralized gaming platforms, lottery systems, and fair NFT distribution mechanisms. Despite being ranked as the fourth most critical smart contract vulnerability by OWASP in 2025 [16], existing detection tools demonstrate alarmingly poor performance.

Our empirical evaluation of state-of-the-art tools on 4,844 real Ethereum contracts revealed fundamental inadequacies. Slither [10], one of the most widely adopted static analysis frameworks, achieved an F1-score of only 0.232. Mythril [11] performed marginally better with an F1-score of 0.236. These results indicate that existing tools correctly identify less than one-quarter of actual vulnerabilities while producing high rates of false positives [62]. This poor performance stems from three fundamental limitations.

First, existing tools rely exclusively on syntactic pattern matching without understanding semantic context [148]. They flag any usage of keywords such as `block.timestamp` or `blockhash` as potentially vulnerable. This approach fails to distinguish safe from unsafe usage. Using `block.timestamp` for time-based access control is secure, whereas using it as a random seed creates a critical vulnerability. Current tools cannot make this distinction.

Second, tools cannot track taint propagation through intermediate variables and complex transformations [44]. When tainted values pass through multiple assignment operations or are combined with other data before reaching sensitive sinks, existing tools lose track of the data flow. This limitation causes them to miss vulnerabilities where predictable values undergo intermediate processing before being used in security-critical operations.

Third, current approaches operate without path-sensitive analysis [41]. They examine contracts at a coarse-grained level, unable to distinguish between execution paths with different security properties. A contract may use blockchain values safely in some functions but vulnerably in others. Without path-level analysis, tools cannot provide precise vulnerability localization.

These limitations create a dangerous gap between the critical importance of bad randomness detection and the inadequacy of available tooling. To address these challenges, this chapter presents TaintSentinel. This is the first path-level bad randomness detection system that analyzes complete execution paths rather than isolated code segments or entire contracts.

Unlike existing contract-level tools, TaintSentinel introduces several key innovations centered on path-sensitive analysis. Our system employs graduated taint analysis with context-sensitive rules. These rules distinguish safe from unsafe usage patterns based on semantic understanding of individual execution paths. We track taint propagation through complete execution flows. This includes storage-level data flow across multiple transactions. This approach enables precise identification of which specific paths contain vulnerabilities. The path-level approach provides actionable localization. The system identifies not just vulnerable contracts but the exact functions and code sequences responsible for security flaws.

The technical foundation rests on two complementary phases. Phase 1 performs comprehensive taint analysis augmented with domain-specific rules derived from bad randomness vulnerability patterns [6]. This phase constructs semantic graphs integrating control flow, data flow, and state dependencies [276]. It applies graduated taint propagation that accounts for how taint strength changes through different operations. Phase 2 employs a dual-stream neural architecture processing both global contract structure and local path-specific patterns [277]. By combining these perspectives through an adaptive fusion mechanism, the system achieves high detection accuracy while maintaining computational efficiency.

Our experimental evaluation on 4,844 Ethereum contracts demonstrates substantial improvements over state-of-the-art approaches. TaintSentinel achieves an F1-score of 0.892 on balanced datasets, representing nearly a fourfold improvement over Slither and Mythril. On imbalanced datasets reflecting real-world distributions [87], our system maintains an F1-score of 0.611 through threshold optimization. We introduce a novel evaluation metric, Path Risk Accuracy (PRA), which assesses the system's ability to correctly classify individual execution paths. TaintSentinel achieves a PRA of 97% on balanced datasets, validating our graduated taint analysis approach.

The remainder of this chapter is organized as follows. Section 7.2 provides an overview of the TaintSentinel architecture. Sections 7.3 and 7.4 detail the two-phase detection methodology. Section 7.5 describes our dataset construction and labeling process. Section 7.6 presents our experimental results and analysis, including comparison with existing tools. Section 7.7 discusses key findings, limitations, and implications, concluding with a summary of contributions.r.

7.2 SYSTEM ARCHITECTURE OVERVIEW

TaintSentinel operates through a two-phase framework that combines static analysis with machine learning. The system detects bad randomness vulnerabilities at the execution path level. Figure 7.1 illustrates the complete system architecture, showing how raw smart contracts flow through multiple transformation stages before producing labeled vulnerability reports with precise localization information.

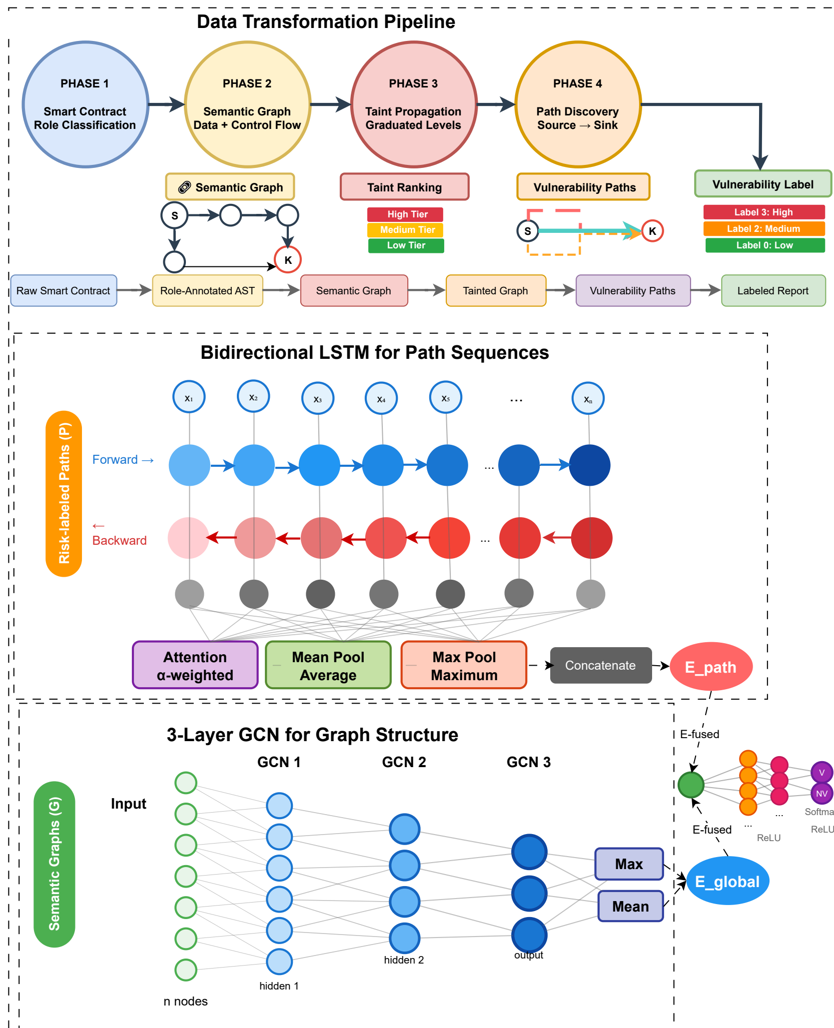

Figure 7.1: TaintSentinel Framework Overview: Two-phase architecture for path-level bad randomness vulnerability detection. Phase 1 performs context-aware taint analysis through four stages: role-based AST construction, semantic graph building, graduated taint propagation, and vulnerability path discovery. Phase 2 employs a dual-stream neural architecture combining bidirectional LSTM for path sequences and three-layer GCN for global graph structure, merged through adaptive fusion for final classification.

The first phase performs context-aware taint analysis and path extraction. Raw Solidity source code is transformed into structured

representations through Abstract Syntax Tree construction and semantic graph building. Taint propagation then tracks how potentially dangerous values flow through execution paths. The phase concludes by extracting complete paths from sources to sinks, with each path receiving a risk label.

The second phase employs a dual-stream neural architecture to classify vulnerability patterns. Two parallel streams process the outputs from Phase 1. The global context stream analyzes the entire semantic graph structure. The path-focused stream processes individual taint propagation paths. These complementary perspectives are combined through an adaptive fusion mechanism. The fused representation produces both contract-level vulnerability predictions and path-level risk assessments.

This two-phase design reflects a fundamental insight about vulnerability detection. Static analysis alone cannot achieve sufficient accuracy because it lacks the ability to learn complex patterns from data. Pure machine learning approaches struggle with smart contract analysis because they cannot incorporate domain-specific knowledge about taint propagation and vulnerability semantics. TaintSentinel bridges this gap by using static analysis to extract semantically meaningful features, then applying machine learning to recognize subtle vulnerability patterns that rule-based systems miss.

7.3 PHASE 1: CONTEXT-AWARE TAINT ANALYSIS

Phase 1 transforms smart contracts into risk-labeled execution paths through four sequential stages. Algorithm 2 presents the complete implementation. Table 7.1 defines the classification rules that guide the analysis.

7.3.1 *AST Construction and Source/Sink Identification*

The analysis begins by parsing Solidity source code into an Abstract Syntax Tree (lines 3-4 of Algorithm 2). This tree undergoes evidence-based enhancement to create an Advanced AST where vulnerability-relevant elements are systematically identified and labeled based on established security standards [278, 279].

Two categories of nodes receive security labels according to Table 7.1. First, entropy sources are marked with sensitivity levels based on their predictability characteristics. Blockchain-provided values including `block.timestamp`, `blockhash`, and `block.difficulty` are classified as high-risk sources under CWE-330 (Use of Insufficiently Random Values) [278]. These values can be predicted or manipulated by miners within certain constraints. `block.number` receives a medium-risk classification as it provides less immediate exploitability but remains deterministic.

Second, sensitive sinks are identified and labeled with compound risk classifications reflecting both randomness vulnerabilities and potential exploit vectors (lines 6-7 of Algorithm 2). Random number generation operations are marked as high-risk under SWC-120 (Weak Sources of Randomness) [279]. Value transfers receive dual classification under SWC-120 and SWC-105 (Unprotected Ether Withdrawal), capturing both the randomness weakness and the financial impact. Prize assignment operations similarly receive high-risk classification under SWC-120. State modifications are labeled with medium risk under both SWC-120 and SWC-112 (Delegatecall to Untrusted Callee) when randomness affects state changes. External calls receive SWC-120 and SWC-107 (Reentrancy) labels, acknowledging the compound vulnerability when weak randomness controls external interactions. Basic conditional logic receives low-risk classification under SWC-120 alone.

This dual labeling strategy establishes the foundation for subsequent taint tracking by marking both where potentially dangerous values originate (sources) and where they might cause security vulnerabilities (sinks). The multi-standard approach captures not only the core randomness vulnerability but also the secondary attack vectors that become available when predictable values control critical operations. Table 7.1 summarizes these classifications with their associated risk levels and standards.

Table 7.1: Randomness Vulnerability Pattern Classification

Pattern Element	**Type**	**Risk**	**Standard**
Entropy Sources			
`block.timestamp`	Source	High	CWE-330
`blockhash()`	Source	High	CWE-330
`block.difficulty`	Source	High	CWE-330
`block.number`	Source	Medium	CWE-330
Sensitive Sinks			
Random generation	Sink	High	SWC-120
Value transfer	Sink	High	SWC-120, SWC-105
Prize assignment	Sink	High	SWC-120
State modification	Sink	Medium	SWC-120, SWC-112
External calls	Sink	Medium	SWC-120, SWC-107
Conditional logic	Sink	Low	SWC-120

7.3.2 *Semantic Graph Construction*

The Advanced AST is transformed into a directed semantic graph. This graph captures three types of relationships essential for accurate taint analysis. Control flow edges represent program execution order. They connect statements according to their sequential, conditional, and iterative execution patterns. Data flow edges track variable dependencies. They link definitions to uses and capture how values propagate through assignments and operations. State edges represent contract state modifications. They connect operations that read or write persistent storage variables.

This multi-dimensional graph structure preserves execution contexts. Function modifiers and access control mechanisms are particularly important. These prove critical for context-sensitive analysis. A random operation protected by an `onlyOwner` modifier presents substantially different risk compared to the same operation in a public function. The semantic graph retains this contextual information for subsequent analysis stages.

7.3.3 *Graduated Taint Propagation*

Taint propagation employs breadth-first traversal with graduated strength adjustment. This approach differs from binary taint marking. The algorithm initializes a queue with all identified source nodes. Each node carries its initial sensitivity level from Table 7.1.

As taint propagates through graph edges, its strength adjusts according to edge type and operation semantics. Control flow edges preserve full taint strength. This reflects that execution flow alone does not reduce exploitability. Data flow edges apply graduated reduction. This recognizes that complex transformations reduce practical exploitability. For instance, cryptographic hashing does not eliminate predictability entirely. However, it makes exploitation more difficult.

The propagation continues until taint strength falls below a defined threshold or reaches a sink node. This graduated approach distinguishes TaintSentinel from traditional binary taint systems. It enables more nuanced risk assessment that reflects real-world exploit difficulty.

7.3.4 *Context-Sensitive Risk Assessment*

Once taint propagation completes, the system extracts all paths connecting sources to sinks. Context-sensitive risk assessment is then performed. Table 7.2 defines classification rules that override initial risk levels. These rules are based on actual usage patterns.

The same blockchain value receives different risk classifications depending on its usage context. For instance, `block.timestamp` com-

bined with modulo operations for random number generation receives high-risk classification. In contrast, `block.timestamp` used for deadline validation with sufficient time buffers is classified as safe.

This context-aware evaluation examines three factors. First, it analyzes usage patterns such as lottery or gambling operations. Second, it evaluates access control mechanisms including role-based restrictions. Third, it identifies additional risk indicators from the code structure. The final output consists of risk-labeled paths sorted into HIGH, MEDIUM, and LOW categories. This provides structured input for the machine learning phase.

Table 7.2: Context-Sensitive Classification Rules

Pattern	**Risk Level**
`block.timestamp % N`	High
`block.timestamp` for lottery/gambling	High
`keccak256(block.timestamp, ...)` for RNG	High
`block.timestamp >= deadline + 15 min`	Safe
`block.timestamp` for event logging	Safe
`block.timestamp > lastAction + 1 days`	Safe

7.4 PHASE 2: DUAL-STREAM NEURAL ARCHITECTURE

Phase 2 processes the outputs from Phase 7.3through a dual-stream neural architecture. This architecture combines global contract-level analysis with local path-level analysis. Algorithm 3 presents the complete implementation. The system employs two parallel streams that process complementary representations of vulnerability patterns.

7.4.1 *Global Context Stream*

The global context stream analyzes the entire semantic graph structure using a Graph Convolutional Network [280]. This stream captures contract-level architectural patterns and cross-function dependencies. The network consists of three GCN layers. These layers transform node features while considering the graph structure [277].

Each layer aggregates information from neighboring nodes. This aggregation enables the network to learn how vulnerability patterns manifest across the entire contract architecture. The first layer processes raw node features. Subsequent layers build increasingly abstract representations. The final layer produces node embeddings that encode global structural information.

Algorithm 2 Context-Aware Graduated Taint Analysis

Require: Smart contract C
Ensure: Risk-labeled paths

```
1: function TaintSentinel(C)
2:     AST ← Parse(C)
3:     G ← BuildGraph(AST) with {control, data, state} edges
4:     // Phase 1: Source/Sink Identification
5:     for each node n ∈ G do
6:         if IsSource(n) then
7:             n.sensitivity ← Table 7.1
8:         end if
9:         if IsSink(n) then
10:            n.risk ← CWE/SWC
11:        end if
12:    end for
13:    // Phase 2: Taint Propagation
14:    Initialize queue Q with sources; visited ← ∅
15:    while Q ≠ ∅ do
16:        v ← Dequeue(Q)
17:        for each edge (v, u) ∈ E where u ∉ visited do
18:            taint_u ← Propagate(v.taint, edge.type)  ▷ Graduated
19:            if taint_u > threshold then
20:                Enqueue(Q, u); u.taint ← taint_u
21:                visited ← visited ∪ {u}
22:            end if
23:        end for
24:    end while
25:    // Phase 3: Risk Assessment
26:    Extract paths P from sources to sinks
27:    for each path p ∈ P do
28:        p.risk ← Context(p) × Access(p) × Table 7.2
29:    end for
30:    return Sort(P) by risk {HIGH, MED, LOW}
31: end function
```

These node embeddings are aggregated into a single contract-level representation through dual pooling. Mean pooling captures the average pattern across all nodes. Max pooling identifies the most prominent features. The concatenation of these two pooling operations produces the global embedding $E_{global} \in \mathbb{R}^{256}$. This representation encodes contract-wide vulnerability patterns independent of specific execution paths.

The global stream proves essential for detecting vulnerabilities that arise from architectural decisions. Complex dependencies between state variables across different functions become apparent through graph-level analysis. Suspicious architectural patterns such as unusual

combinations of access controls and sensitive operations are identified through this global perspective.

7.4.2 *Path-Focused Stream*

The path-focused stream processes individual taint propagation paths extracted in Phase 1. Each risk-labeled path represents a sequence of nodes from source to sink. This stream employs a bidirectional LSTM network [281, 282]. The LSTM captures sequential dependencies in both forward and backward directions.

The bidirectional architecture enables the network to understand context from both directions. Forward processing learns how taint propagates from sources toward sinks. Backward processing captures how sink requirements influence earlier operations. The combination of both directions provides comprehensive understanding of vulnerability patterns within execution flows.

For each contract, all extracted paths are processed independently through the LSTM. Each path produces an embedding that encodes its specific vulnerability signature. These individual path embeddings undergo hierarchical aggregation through three complementary mechanisms.

First, attention-weighted aggregation [283] automatically learns to focus on the most critical paths. The attention mechanism assigns higher weights to paths with stronger vulnerability indicators. Second, mean pooling captures the overall pattern distribution across all paths. Third, max pooling identifies worst-case scenarios by selecting the most vulnerable path features. The combination of these three aggregation strategies produces the path embedding $E_{path} \in \mathbb{R}^{64}$.

Additionally, the path stream incorporates a path risk classification head. This component predicts risk levels for individual paths as HIGH, MEDIUM, or LOW. The path risk predictions enable computation of the Path Risk Accuracy metric. This metric validates how accurately the system understands risk gradations at the path level, distinct from contract-level classification.

7.4.3 *Adaptive Fusion Mechanism*

The fusion mechanism combines global graph embeddings and path embeddings through a learned gating network. This combination enables the system to leverage both contract-level architectural patterns and path-specific vulnerability signatures.

The process begins by concatenating the two embeddings. The concatenated representation $E_{concat} = [E_{global}; E_{path}]$ has dimensionality 320. This combines the 256-dimensional global embedding with the 64-dimensional path embedding.

A fusion MLP transforms this concatenated representation into a unified feature space. Simultaneously, a parallel gating network computes attention weights. The gating network learns to assign importance scores to different features based on input characteristics. The gate values are computed as $g = \sigma(W_g \cdot E_{concat} + b_g)$, where $g \in \mathbb{R}^{128}$.

The final fused representation is computed through element-wise multiplication. The formula $E_{final} = g \odot \text{MLP}_{fusion}(E_{concat})$ produces the final embedding. This gated architecture enables the model to dynamically weight different aspects of the fused features according to input characteristics.

The learned gate values provide interpretability. They highlight which aspects of the fused representation are most relevant for each vulnerability pattern. For some contracts, global architectural patterns may dominate. For others, specific path-level sequences may be more indicative of vulnerabilities.

The fused representation feeds into two classification heads. The primary head produces contract-level vulnerability predictions as binary classification. The secondary head produces path risk predictions for the Path Risk Accuracy metric. Both objectives are optimized jointly during training with weighted loss combination.

7.5 DATASET CONSTRUCTION AND LABELING

This section describes the construction of our labeled dataset for bad randomness vulnerability detection. The dataset consists of 4,844 real Ethereum contracts with verified labels. These labels distinguish vulnerable contracts from those using blockchain values safely. This dataset represents a filtered subset of the comprehensive benchmark presented in Chapter 6, with more stringent criteria applied for training and evaluation purposes.;

7.5.1 *Data Collection*

The dataset was collected from three primary sources. SmartBugs-Curated [275] provided 8 manually verified contracts. The SWC Registry [279] contributed 7 contracts with documented vulnerabilities. SmartBugs-Wild [275] served as the main source with 47,398 real contracts deployed on the Ethereum network.

The analysis of SmartBugs-Wild proceeded in two phases. The initial identification phase examined all 47,398 contracts for known bad randomness sources. This search identified 6,586 contracts (14%) containing at least one randomness source. Table 7.3 shows the distribution of primary sources. The most frequent source is `block.timestamp` or `now`. This source appears in 45.8% of cases. Other common sources include `block.number` (18.3%), `blockhash` (12.7%), and `tx.gasprice` (9.6%).

Algorithm 3 Dual-Stream GNN with Path Risk Assessment

Require: Semantic Graph G and risk-labeled paths from Phase 1
Ensure: Detection model M

1: **function** TrainDual(G)
2: Initialize GlobalGCN, PathGNN, Fusion components
3: **for** epoch = 1 to max_epochs **do**
4: **for** each batch (G_b, P_b, y_b) in DataLoader(G) **do**
5: ▷ Global stream processing
6: $h_g \leftarrow \text{GCN3}(G_b.x, G_b.\text{edge_index})$
7: $E_{global} \leftarrow [\text{MeanPool}(h_g); \text{MaxPool}(h_g)]$ ▷ 256-dim
8: ▷ Path stream processing
9: $E_{path} \leftarrow []$
10: **for** each contract i in batch **do**
11: $e_i \leftarrow \text{ProcessPaths}(P_b[i], G_b.x_i)$ ▷ 64-dim
12: $E_{path}.\text{append}(e_i)$
13: **end for**
14: ▷ Adaptive fusion
15: $E_{concat} \leftarrow [E_{global}; E_{path}]$ ▷ 320-dim
16: $g \leftarrow \sigma(\text{GateNet}(E_{concat}))$ ▷ 128-dim
17: $E_{fused} \leftarrow g \odot \text{FusionMLP}(E_{concat})$
18: ▷ Dual objectives
19: $\hat{y} \leftarrow \text{Classifier}(E_{fused})$
20: $r_{pred} \leftarrow \text{PathRiskHead}(E_{path})$ ▷ PRA
21: $L \leftarrow \text{CE}(\hat{y}, y_b) + 0.3 \cdot \text{CE}(r_{pred}, r_{true})$
22: Backpropagate(L)
23: **end for**
24: **end for**
25: **return** M
26: **end function**
27: **function** ProcessPaths(paths, nodes)
28: **if** paths.num == 0 **then**
29: **return** $\mathbf{0}^{64}$
30: **end if**
31: embeddings $\leftarrow []$
32: **for** each path p in paths **do**
33: $h \leftarrow \text{BiLSTM}(\text{nodes}[p.\text{sequence}])$
34: $e \leftarrow \text{Combine}(h, p.\text{features})$
35: embeddings.append(e)
36: **end for**
37: ▷ Hierarchical aggregation
38: $\alpha \leftarrow \text{Attention}(\text{embeddings})$
39: **return** $\text{Aggregate}(\alpha \cdot \text{embeddings}, \text{mean}, \text{max})$
40: **end function**

Table 7.3: Distribution of Primary Bad Randomness Sources in SmartBugs-Wild

Primary Source	Count	Percentage
`block.timestamp/now`	892	45.8%
`block.number`	356	18.3%
`blockhash`	248	12.7%
`tx.gasprice`	187	9.6%
`gasleft()`	142	7.3%
`block.difficulty/prevrandao`	122	6.3%
Total Primary Sources	1,947	100%

Additionally, 723 instances combined sources with operations like `keccak256` hashing or inappropriate use of `msg.sender` in randomness generation. The hashing operations appeared in 275 cases. The inappropriate `msg.sender` usage appeared in 438 cases.

7.5.2 *Labeling Methodology*

The context-aware analysis phase classified identified contracts using the rules defined in Table 7.2. This classification distinguishes vulnerable usage from safe usage based on semantic context rather than mere presence of blockchain values.

Vulnerable contracts use randomness sources in security-critical operations without adequate protection. Safe contracts employ blockchain values for legitimate purposes such as time-locks or deadline validation. In these cases, predictability does not create exploitable vulnerabilities. Suspicious contracts required manual analysis to determine their status.

Manual validation by security experts examined approximately 150 randomly selected contracts. The validation achieved 94% agreement for vulnerable contracts and 91% for safe contracts. This confirms the effectiveness of our context-sensitive approach.

7.5.3 *Dataset Statistics*

Table 7.4 presents the final dataset composition. The combined dataset contains 4,844 contracts. Of these, 397 are labeled as vulnerable and 4,447 as safe.

This dataset represents a significant contribution to smart contract security research. Previous work lacked large-scale labeled datasets specifically for bad randomness vulnerabilities. The complete labeled

Table 7.4: Summary of Vulnerable and Safe Contracts from Three Primary Sources

Data Source	Total	Vulnerable	Safe
SmartBugs-Curated	8	8	0
SWC Registry	7	5	2
SmartBugs-Wild	4,829	384	4,445
Total	4,844	397	4,447

dataset is publicly available. It includes original contracts, labeling results, taint analysis outputs, and extracted execution paths.

7.5.4 *Evaluation Metrics*

We evaluate TaintSentinel using standard classification metrics and a novel path-based metric.

F1-Score balances precision and recall:

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{7.1}$$

Precision measures the fraction of correctly identified vulnerable contracts among all flagged contracts:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{7.2}$$

Recall measures the fraction of actual vulnerable contracts successfully detected:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{7.3}$$

AUC-ROC evaluates the model's discriminative ability across all classification thresholds:

$$\text{AUC} = \int_0^1 \text{Recall}(t)\, d(FPR(t)) \tag{7.4}$$

Path Risk Accuracy (PRA) is a novel metric assessing correct classification of individual execution paths into risk levels (HIGH, MEDIUM, LOW):

$$\text{PRA} = \frac{1}{N_{paths}} \sum_{i=1}^{N_{paths}} \mathbb{1}[\text{predicted_risk}_i = \text{actual_risk}_i] \tag{7.5}$$

Confusion Matrix provides detailed breakdown of predictions:

$$CM = \begin{bmatrix} TP & FN \\ FP & TN \end{bmatrix} \tag{7.6}$$

where TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives.

7.6 RESULTS AND ANALYSIS

This section presents comprehensive experimental results evaluating TaintSentinel's performance under both balanced and imbalanced scenarios. The results demonstrate substantial improvements over existing tools while maintaining computational efficiency.

7.6.1 *Overall Performance Analysis*

We conducted experiments under two scenarios to evaluate TaintSentinel's effectiveness in different conditions. Figure 8.6 presents confusion matrices for both balanced and imbalanced datasets. Figure 7.3 illustrates comparative performance across all metrics through bar charts and radar plot visualization.

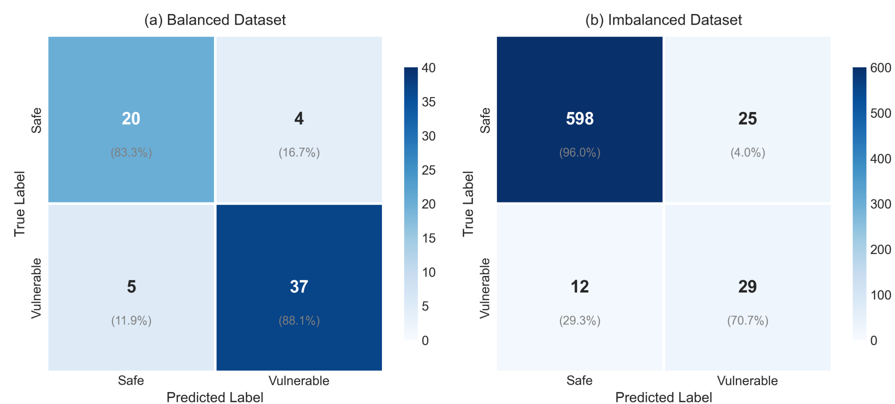

Figure 7.2: Confusion matrices showing prediction accuracy for (a) balanced dataset and (b) imbalanced dataset with optimized threshold.

In the balanced dataset scenario, TaintSentinel achieved F1-score of 0.892. This balances recall (0.881) and precision (0.902). Out of 41 vulnerable contracts, 37 were successfully identified with only 5 false negatives. Among 24 safe contracts, 20 were correctly classified with 4 false positives.

The imbalanced dataset scenario reflects real-world distributions. In this scenario, vulnerable contracts represent approximately 4% of total contracts. We applied threshold optimization to maximize recall. This is critical for security applications where missing vulnerabilities poses greater risk than false alarms. With optimized threshold 0.05, recall increased from 0.439 to 0.707. Precision decreased from 0.783 to 0.537. This configuration reduced false negatives by 61%. The system correctly identified 29 out of 41 vulnerable contracts.

Both AUC-ROC and PRA scores remained above 0.92 across both scenarios. This demonstrates the model's robust performance regardless of data distribution. The radar plot in Figure 7.3(b) visualizes this consistency. It shows strong performance across multiple evaluation dimensions.

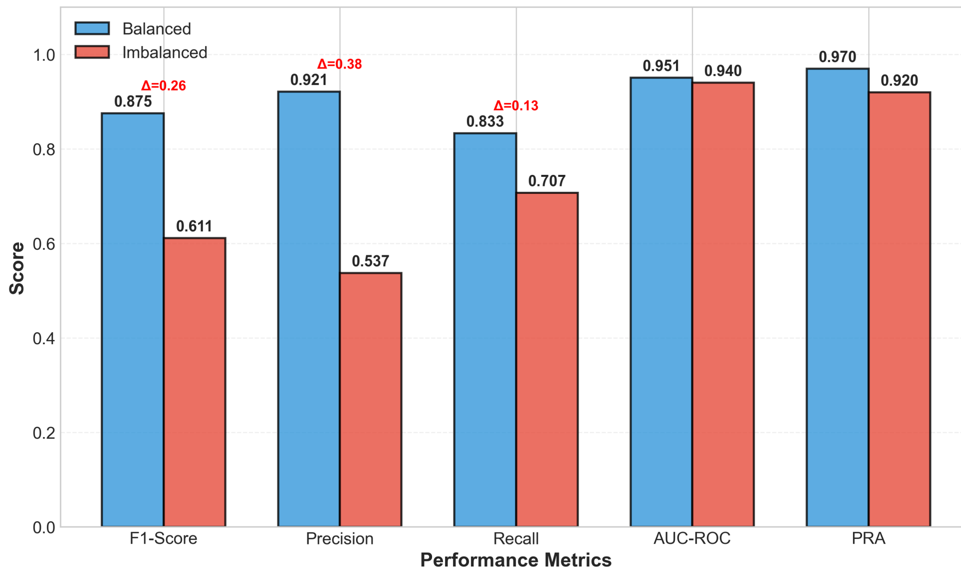

Figure 7.3: Comprehensive performance comparison between balanced and imbalanced scenarios: (a) bar chart showing individual metrics, (b) radar plot visualization across five key metrics.

7.6.2 *Path Risk Assessment Performance*

A distinguishing feature of TaintSentinel is its ability to assess risk levels of individual taint propagation paths. The Path Risk Accuracy metric measures how accurately the system categorizes paths into HIGH, MEDIUM, and LOW risk levels.

TaintSentinel achieved PRA of 0.970 for balanced datasets and 0.920 for imbalanced datasets. These high scores validate the effectiveness of our graduated taint analysis and hierarchical aggregation in the PathGNN component. The system successfully distinguishes between different risk gradations. This enables prioritized security audits where high-risk paths receive immediate attention.

7.6.3 *ROC Curve Analysis*

Figure 7.4 presents ROC curves demonstrating TaintSentinel's discriminative ability across all classification thresholds. The balanced dataset achieved AUC-ROC of 0.951. The imbalanced dataset achieved 0.940. These near-perfect AUC values indicate that the model's vulnerability probability rankings are highly accurate regardless of class distribution.

The consistent AUC scores across scenarios validate our dual-stream architecture. The architecture effectively captures both local path-specific patterns and global contract structures. This robustness proves essential for deployment in production environments with varying vulnerability prevalence.

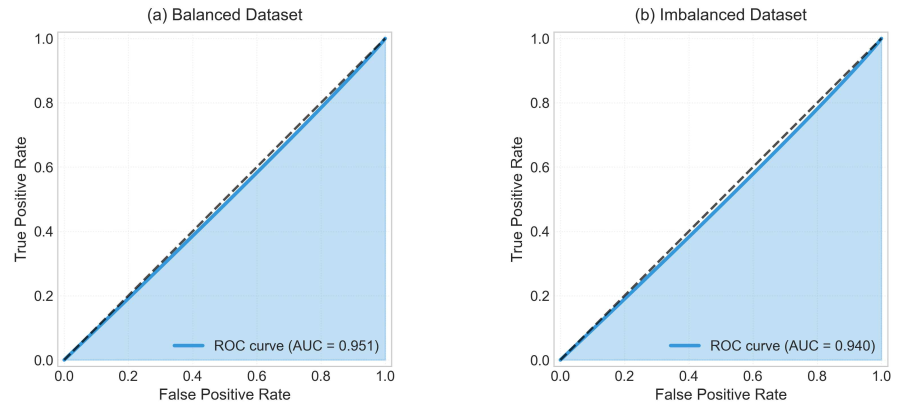

Figure 7.4: ROC curves showing model's discriminative ability for (a) balanced dataset (AUC = 0.951) and (b) imbalanced dataset (AUC = 0.940).

7.6.4 *Training Dynamics and Convergence*

Figure 7.5 illustrates training dynamics for both scenarios. The balanced dataset showed smooth convergence with minimal divergence between training and validation metrics. Optimal performance was achieved at epoch 28. Loss curves declined steadily from initial values around 0.68 to final values approximately 0.44.

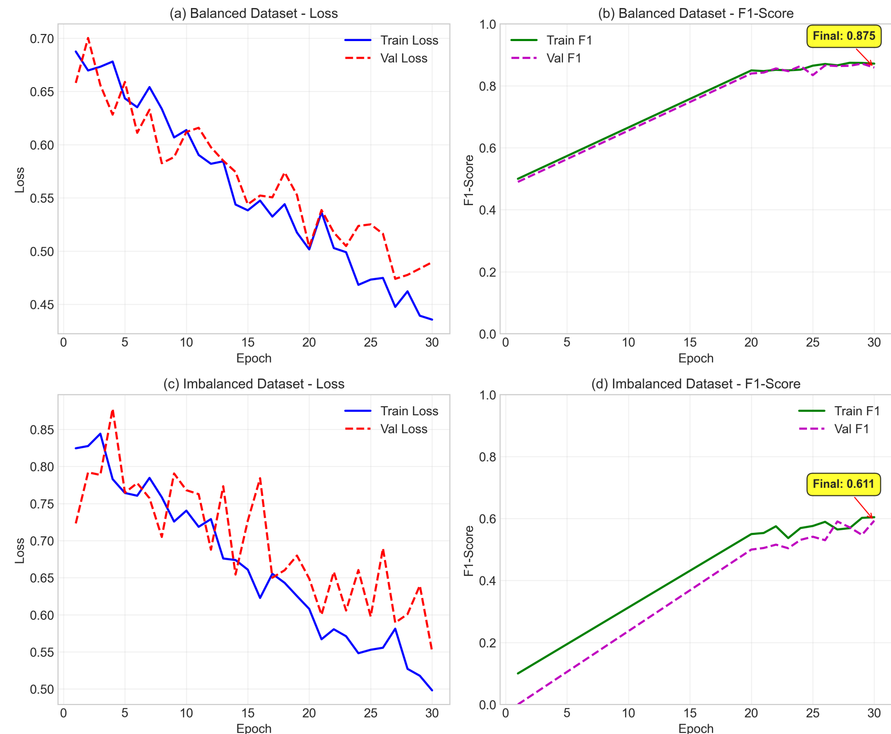

Figure 7.5: Training dynamics showing loss and F1-Score evolution during training for balanced (top) and imbalanced (bottom) datasets.

The imbalanced dataset exhibited more volatile training behavior. Validation loss fluctuated between 0.55 and 0.80. This instability is expected given the extreme class imbalance (96% safe, 4% vulnerable). Despite volatility, the model converged to a stable solution at epoch 25. After threshold optimization, the final F1-score reached 0.611.

7.6.5 *Computational Efficiency*

Runtime evaluation on 300 contracts demonstrates TaintSentinel's computational efficiency across contract sizes. Table 7.5 presents results for three categories. These are Small (average 33 nodes), Medium (average 83 nodes), and Large (average 147 nodes).

Table 7.5: TaintSentinel Runtime Performance Across Contract Sizes

Category	**Preprocessing (s)**	**Model (s)**	**Total (s)**
Small (33 nodes)	0.202	0.016	0.218
Medium (83 nodes)	0.420	0.022	0.442
Large (147 nodes)	1.708	0.029	1.737

Contract complexity increases 4.5× from small to large contracts (33 to 147 nodes). Despite this, runtime increases only 8×. This demonstrates sub-linear scaling. Preprocessing includes AST construction, semantic graph building, and taint analysis. Model inference on preprocessed contracts requires only milliseconds. This enables real-time monitoring scenarios.

The complete pipeline requires 7 seconds with compilation for typical contracts. Preprocessed contracts need only milliseconds for model inference.

These efficiency results confirm that TaintSentinel addresses key vulnerability detection challenges. It provides high accuracy with practical efficiency. It offers interpretable risk assessment via PRA. It enables operational flexibility through threshold adjustment.

7.6.6 *Comparison with State-of-the-Art Tools*

We compared TaintSentinel's performance against two widely adopted static analysis tools: Slither [10] and Mythril [11]. Table 7.6 presents comprehensive results on the complete bad randomness dataset.

The results demonstrate TaintSentinel's significant superiority. Slither and Mythril achieved F1-scores of 0.232 and 0.236 respectively, while TaintSentinel achieved 0.892 on balanced datasets, indicating a 278% improvement.

This performance disparity stems from fundamental limitations in traditional static analysis approaches:

Table 7.6: Performance Comparison with Existing Tools

Tool	TP	FN	FP	TN	Prec.	Rec.	F1
Slither	46	203	101	514	0.313	0.185	0.232
Mythril	43	206	72	543	0.374	0.172	0.236
TS (Bal.)	37	5	4	20	0.902	0.881	0.892
TS (Imb.)	29	12	25	598	0.537	0.707	0.611

Pattern Matching Limitations: Both tools employ simplistic pattern matching that only detects direct usage of block attributes. Slither's weak randomness detector searches for modulo operations with block characteristics such as `block.timestamp % n`, missing cases where values are stored in intermediate variables or used without modulo operators. Similarly, Mythril's predictable variables module only flags block properties when used directly in conditional expressions, overlooking complex data flows.

Context-Insensitive Analysis: The tools cannot differentiate between malicious and benign applications of block characteristics. Without semantic understanding, they fail to distinguish safe use of `block.timestamp` for deadline checks from dangerous use for randomness generation, leading to either high false positive rates or overly conservative detection that overlooks actual vulnerabilities.

Contract-Level Analysis: Current tools analyze entire contracts uniformly without considering different execution paths. This approach treats all code paths equally, missing context-dependent vulnerabilities that only manifest in specific execution flows, such as when randomness is used in publicly accessible functions versus owner-restricted operations.

TaintSentinel addresses these limitations through three key innovations: (1) *graduated taint analysis* that tracks taint propagation through multiple levels rather than binary classification, enabling more nuanced vulnerability assessment; (2) *context-sensitive rules* (Table III) that apply semantic understanding to distinguish safe patterns (e.g., `block.timestamp >= deadline`) from unsafe ones (e.g., `block.timestamp % n`), significantly reducing false positives; and (3) *path-level detection* that analyzes individual execution paths rather than entire contracts, identifying vulnerabilities that only appear in specific control flows. The dual-stream architecture combines these capabilities with learned pattern recognition through PathGNN and GlobalGCN, achieving substantially higher accuracy while maintaining practical efficiency.

7.7 DISCUSSION AND SUMMARY

7.7.1 *Key Findings*

This chapter presented TaintSentinel, a path-level bad randomness vulnerability detection system. The system addresses limitations of existing tools. Our experimental evaluation on 4,844 real Ethereum contracts demonstrated substantial improvements across multiple dimensions.

TaintSentinel achieved F1-score of 0.892 on balanced datasets. This represents nearly fourfold improvement over state-of-the-art tools Slither (0.232) and Mythril (0.236). On imbalanced datasets reflecting real-world distributions, the system maintained F1-score of 0.611. This was achieved through threshold optimization that prioritizes recall over precision. This configuration reduced false negatives by 61%. It addresses the critical security requirement that missing vulnerabilities poses greater risk than false alarms.

The Path Risk Accuracy metric achieved 97% on balanced datasets and 92% on imbalanced datasets. These results validate our graduated taint analysis approach. They demonstrate that the system accurately distinguishes between different risk gradations at the path level. This capability enables prioritized security audits where high-risk paths receive immediate attention.

Computational efficiency analysis showed sub-linear scaling. Despite 4.5× complexity increase from small to large contracts, runtime increased only 8×. Complete pipeline requires 7 seconds with compilation for typical contracts. Model inference on preprocessed contracts requires only milliseconds. This efficiency enables both batch analysis and real-time monitoring scenarios.

Three technical innovations enabled these results. First, graduated taint propagation with context-sensitive rules distinguishes safe from unsafe usage patterns. This is based on semantic understanding rather than syntactic pattern matching. Second, path-level risk assessment enables fine-grained vulnerability characterization. It classifies individual execution paths into HIGH, MEDIUM, and LOW risk categories rather than treating entire contracts uniformly. This granular assessment supports prioritized security audits through the Path Risk Accuracy metric. Third, the dual-stream neural architecture combines global contract-level patterns with local path-specific signatures through adaptive fusion. This leverages complementary perspectives for robust detection.

7.7.2 *Limitations*

Our current implementation has several limitations that warrant future investigation. First, the framework analyzes intra-contract vulnerabil-

ities within individual smart contracts. Many sophisticated exploits in DeFi protocols arise from cross-contract interactions. They also arise from complex transaction sequences spanning multiple contracts. Extending the analysis to inter-contract dependencies would capture these complex attack patterns.

Second, contracts exceeding 300 nodes may require optimization strategies to maintain real-time performance. While sub-linear scaling demonstrates efficiency, very large contracts with complex control flow can challenge current preprocessing capabilities. Adaptive optimization techniques could address this limitation. These techniques include intelligent path pruning and incremental analysis.

Third, the severe class imbalance in real-world datasets poses ongoing challenges. Less than 1% of deployed contracts contain bad randomness vulnerabilities. This creates difficult conditions for model training and evaluation. The threshold optimization approach addresses this partially by trading precision for recall. However, more sophisticated techniques might improve performance further. These include focal loss or cost-sensitive learning.

Fourth, the labeled dataset relies on automated classification with manual validation for a subset. The validation achieved 94% agreement for vulnerable contracts and 91% for safe contracts. However, some edge cases require deeper analysis. Constructing larger benchmark datasets would enable more comprehensive evaluation. This could be done through synthetic vulnerability injection and mutation testing. It would improve model robustness.

7.7.3 *Summary*

This chapter addressed Research Question 2 by demonstrating a key capability. Semantic-aware taint analysis with context-sensitive rules can effectively detect bad randomness vulnerabilities. These are vulnerabilities that current pattern-based tools fail to identify. Our graduated taint propagation distinguishes safe from unsafe usage patterns based on semantic context. Path-level analysis achieves high detection accuracy while maintaining acceptable computational overhead.

The experimental results validate our approach across multiple dimensions. Accuracy improvements over existing tools demonstrate the effectiveness of combining static analysis with machine learning. High Path Risk Accuracy scores confirm that graduated taint analysis produces meaningful risk gradations. Computational efficiency measurements show that path-level analysis scales to real-world contracts.

8 SMARTTAINTRL: RL-BASED DETECTION AND LOCALIZATION

8.1 introduction

This chapter introduces SmartTaintRL [24], a reinforcement learning-based framework for detecting and localizing Bad Randomness vulnerabilities in smart contracts. Unlike Chapter 7's exhaustive approach, SmartTaintRL addresses three interconnected challenges. First, achieving high detection accuracy. Second, providing vulnerability localization at function and node levels. Third, solving the path explosion problem through intelligent path prioritization.

The system employs a Deep Q-Network agent that learns to selectively analyze high-risk paths while safely pruning low-value ones. Through empirical weight calibration and hierarchical reward engineering, SmartTaintRL achieves 45% path reduction while maintaining 96% recall. More critically, it provides the first localization methodology specifically designed for Bad Randomness, identifying not just vulnerable contracts but the exact functions and code locations requiring remediation.

The chapter is organized as follows. Section 8.2 presents the three-phase system architecture. Section 8.4 describes the reinforcement learning-based path prioritization with empirical weight calibration and hierarchical reward structure. Section 8.5 details the vulnerability localization methodology at both function and node levels. Section 8.6 and Section 8.7 present the dataset and implementation details. Section 8.8 provides evaluation and comparison with state-of-the-art methods. Finally, Section 8.9 concludes the chapter.

8.2 system architecture overview

SmartTaintRL addresses the path explosion challenge through a three-phase architecture. This architecture combines *offline taint analysis* with *online reinforcement learning*. Unlike TaintSentinel's exhaustive approach that analyzes all extracted paths, SmartTaintRL employs an intelligent agent. The agent learns to prioritize high-risk paths while safely pruning low-value ones. This selective analysis strategy reduces computational overhead without compromising detection capability. Figure 8.1 illustrates the complete system architecture. It shows the flow of data through multiple transformation stages.

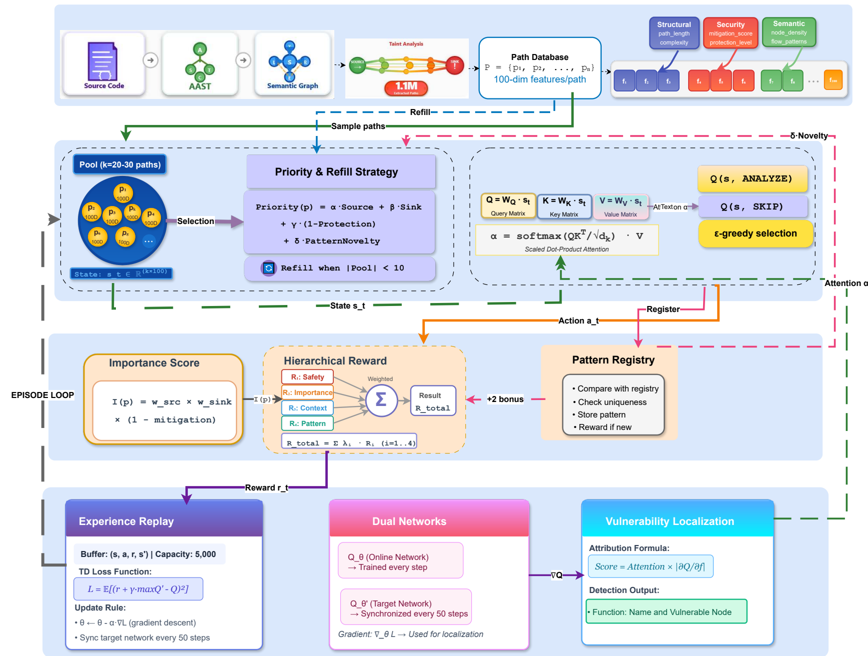

Figure 8.1: SmartTaintRL Architecture comprising four components. **(Top)** Path database with 100-dimensional feature vectors. **(Middle)** RL decision engine with pool management, attention-guided Q-network, and ϵ-greedy action selection. **(Bottom-Left)** Training system with hierarchical reward, pattern registry, experience replay, and dual DQN networks. **(Bottom-Right)** Localization module combining attention weights and gradient-based attribution for identifying vulnerable functions and nodes.

PHASE 1: OFFLINE PREPROCESSING. The first phase performs preprocessing and taint analysis. This phase builds upon the methodology established in Section 7.3 of Chapter 7. Raw Solidity contracts undergo **AST construction**, **semantic graph building**, and **graduated taint propagation**. The system extracts all possible paths from entropy sources to sensitive sinks.

For each path, a *100-dimensional feature vector* captures structural properties, security characteristics, and semantic patterns. These features encode information about path length, node complexity, protection mechanisms, and source-sink relationships. The extracted paths and their features are stored in structured format. This enables efficient access during online decision-making. This *offline-first* design separates expensive static analysis from real-time path selection.

PHASE 2: ONLINE RL-BASED PRIORITIZATION. The second phase implements the core innovation through reinforcement learning-based path prioritization. A **Deep Q-Network** agent learns to make binary decisions for each path: `ANALYZE` or `SKIP`. The agent operates within a dynamic pool containing a subset of available paths. Pool management balances *exploration* and *exploitation*. This is achieved through priority-based sampling combined with random selection. An *attention mechanism* guides the agent's focus toward critical path features.

The decision process incorporates *empirically calibrated importance weights*. These weights are derived from analysis of 223 vulnerable contracts. They reflect the actual prevalence of different entropy sources in real vulnerabilities.

A **hierarchical reward function** operates across four levels. First, mandatory safety constraints prevent critical paths from being skipped. Second, importance-driven scoring guides strategic decisions. Third, contextual adjustments incorporate contract-level features. Fourth, exploration incentives encourage pattern discovery. A *pattern registry* tracks encountered vulnerability signatures. It provides bonuses for novel patterns.

PHASE 3: VULNERABILITY LOCALIZATION. The third phase performs vulnerability localization at two hierarchical levels. *Function-level localization* aggregates Q-values and decision counts across paths belonging to each function. Functions with high aggregate scores are identified as vulnerable.

Node-level localization then pinpoints specific code locations within vulnerable functions. This process combines three techniques. **Gradient-based attribution** measures feature importance. **Graph propagation** spreads scores through data flow paths. **Centrality analysis** identifies structurally important nodes. The combination produces ranked lists of vulnerable nodes. These include source

locations where dangerous values originate and sink locations where these values create security risks.

8.3 PHASE 1: OFFLINE PREPROCESSING

The first phase performs comprehensive preprocessing and taint analysis to extract vulnerability-relevant execution paths. This phase builds upon the methodology established in Section 7.3 of Chapter 7. Raw Solidity contracts undergo AST construction, semantic graph building, and graduated taint propagation. For each extracted path, a 100-dimensional feature vector captures structural properties, security characteristics, and semantic patterns. The extracted paths and features are stored in structured JSON format, enabling efficient access during online reinforcement learning.

8.3.1 *Path Extraction via Taint Analysis*

Solidity source code undergoes parsing into an Advanced Abstract Syntax Tree. Entropy sources and sensitive sinks are identified and labeled in the tree structure. The AAST transforms into a semantic graph that integrates Control Flow Graph and Data Flow Graph to form the final graph $G = (V, E)$. Here, V contains nodes representing variables, operations, and calls. The set E contains data and control flow edges.

Graduated taint propagation tracks how potentially dangerous values flow from sources to sinks through execution paths. Weak entropy sources are identified as taint sources:

$$T_{sources} = \{\texttt{block.timestamp}, \texttt{block.number}, \texttt{blockhash}, \\ \texttt{block.difficulty}, \texttt{block.coinbase}\} \quad (8.1)$$

Sensitive operations define the sinks:

$$T_{sinks} = \{\texttt{transfer}, \texttt{send}, \texttt{randomGeneration}, \texttt{stateModification}\} \quad (8.2)$$

The taint propagation algorithm extracts all possible paths $P = \{p_1, p_2, \ldots, p_n\}$ connecting sources to sinks. In our dataset, this extraction identified **1.1 million paths** across 4,706 contracts.

8.3.2 *Feature Extraction*

Unlike TAINTSENTINEL which processes paths directly through neural networks, SMARTTAINTRL requires explicit feature representation for the reinforcement learning agent. For each path p_i, we compute a *100-dimensional feature vector* $f(p_i) = [f_1, f_2, \ldots, f_{100}]$. This rich representation eliminates the need for large training datasets. It provides sufficient information for accurate decision-making.

The feature vector comprises four complementary categories. **Structural features** capture path topology. These include `path_length` measuring the number of nodes, `node_diversity` indicating unique node types, and `branch_complexity` quantifying conditional statements.

Security features encode protection mechanisms. These include `require_density` measuring validation statement frequency, `modifier_protection` indicating access control presence, and `mitigation_score` quantifying overall defensive measures.

Semantic features represent code patterns. These include `condition_complexity` for logical expression depth, `keccak_operations` counting cryptographic hash usage, and `arithmetic_intensity` measuring mathematical operations.

Source-sink interaction features characterize vulnerability patterns. These include source type encoding, sink criticality scoring, and path risk estimation.

8.3.3 *Offline Storage and Indexing*

The extracted paths and their feature vectors are stored in structured JSON format. This enables efficient access during online training. Each contract produces two files. A `path_database.json` contains all paths with features. A `profile.json` contains contract-level metadata. This *offline-first architecture* ensures that expensive taint analysis computations occur only once. The reinforcement learning agent can then focus exclusively on strategic decision-making without repeating static analysis.

For the reinforcement learning experiments, we applied quality filtering to ensure training effectiveness. Paths with fewer than 3 nodes were removed. These paths lack sufficient complexity for meaningful analysis. Contracts with insufficient path diversity (fewer than 5 paths) were excluded. This filtering produced **252,844 high-quality paths** suitable for RL training. It maintained the real-world class imbalance ratio of approximately 1:18 (vulnerable to safe).

8.4 PHASE 2: RL-BASED PATH PRIORITIZATION

Phase 2 implements the core innovation of SmartTaintRL through deep reinforcement learning. This phase employs a **Deep Q-Network** agent. The agent learns to prioritize high-risk paths while safely pruning low-value ones. The system reduces path exploration by 45% while maintaining 96% recall. This effectively solves the path explosion problem. Algorithm 4 presents the complete training process. It integrates all components into a unified framework.

8.4.1 *Empirical Weight Calibration*

Unlike rule-based systems that assign weights arbitrarily, SmartTaintRL derives importance weights through *data-driven empirical analysis*. We conducted analysis on 223 vulnerable contracts from our dataset. This analysis determined scientifically grounded weights for different entropy sources.

8.4.1.1 *Source Prevalence Analysis*

The analysis proceeded in two stages. First, we examined the distribution of entropy sources across all paths in vulnerable contracts. Second, we refined the analysis by focusing exclusively on paths within truly vulnerable functions. This two-level approach ensures weights reflect actual vulnerability patterns. It avoids spurious correlations.

Figure 8.2 presents the empirical results. Among vulnerable functions, `block.timestamp` emerges as the most prevalent source at 52.6%. This is followed by `block.number` at 33.3%. `blockhash` appears at 5.6%. `block.difficulty` occurs at 3.5%. These prevalence rates translate directly into normalized weights through the formula:

$$w_i = \frac{\text{prevalence}_i}{\max(\text{prevalence})} \tag{8.3}$$

This yields the calibrated weights: $w_{\text{timestamp}} = 1.00$, $w_{\text{blocknumber}} = 0.63$, $w_{\text{blockhash}} = 0.11$, and $w_{\text{difficulty}} = 0.07$.

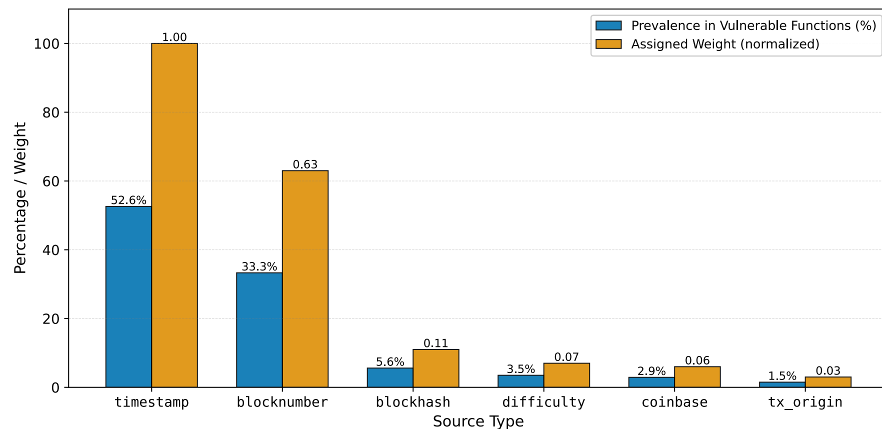

Figure 8.2: Empirical weight calibration based on 223 vulnerable contracts. Blue bars show prevalence of each entropy source in vulnerable functions. Orange bars represent normalized weights assigned to each source.

8.4.1.2 *Source-Sink Pattern Analysis*

Beyond individual source weighting, we analyzed combined source-sink patterns. Figure 8.3 displays the correlation matrix. This matrix

shows frequency distribution across different combinations. Certain patterns exhibit high occurrence in vulnerable contracts. The pattern (`blocknumber`, `controlFlow`) appears in 42% of cases. The pattern (`coinbase`, `valueTransfer`) occurs in 55%. These patterns inform combined risk assessment:

$$\text{CombinedRisk}(s,t) = w_{\text{source}} \times \text{PatternFrequency}(s,t) \quad (8.4)$$

The calibrated weights propagate throughout the framework. They flow from priority scoring in pool management to importance calculation in reward functions. This consistency ensures coherent risk assessment across all decision-making stages.

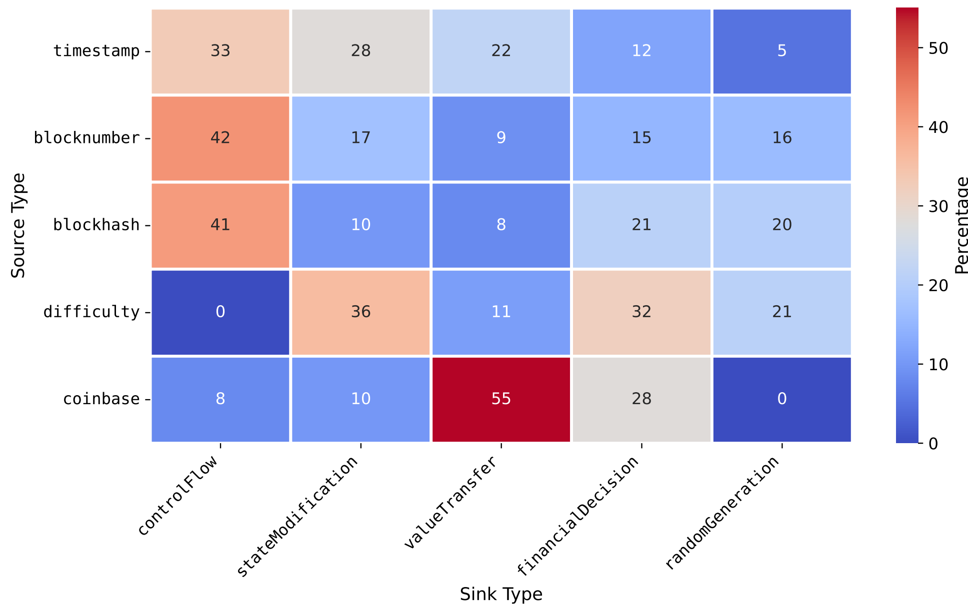

Figure 8.3: Source-sink combination patterns observed in 223 vulnerable contracts. Heatmap displays frequency distribution with darker colors indicating higher occurrence rates.

8.4.2 *Dynamic Pool Management*

Pool management implements the first layer of search space reduction. Rather than presenting the agent with all n paths simultaneously, the system maintains a bounded pool. This pool contains only k paths at any time. This *pool-based exploration strategy* prevents the agent from being overwhelmed. It ensures sufficient information diversity.

8.4.2.1 *Adaptive Pool Sizing*

The pool size adapts to contract complexity through the formula:

$$k = \min(30, \max(10, \lfloor n/10 \rfloor)) \quad (8.5)$$

where n represents the total number of paths available in the contract. Small contracts with fewer than 20 paths place all paths in the pool. This enables comprehensive analysis. Large contracts with hundreds of paths maintain pools of 30–40 paths. This strikes a balance between exploration breadth and computational tractability. Note that while the actual pool size k varies adaptively between 10–40 paths based on contract complexity, the state representation fed to the neural network maintains fixed dimensions (20×100). When $k < 20$, zero-padding fills the remaining rows. When $k > 20$, paths are processed in batches of 20 with priority-based ordering.This adaptive sizing ensures the agent receives appropriately scoped decision problems. It works regardless of contract scale.

8.4.2.2 *Priority Scoring Mechanism*

Each path p receives a priority score based on four weighted criteria:

$$\begin{aligned} \text{Priority}(p) = \alpha \cdot \text{SourceRisk}(p) + \beta \cdot \text{SinkCriticality}(p) \\ +\gamma \cdot (1 - \text{Protection}(p)) + \delta \cdot \text{PatternNovelty}(p) \end{aligned} \tag{8.6}$$

where the weights $\alpha, \beta, \gamma, \delta$ are calibrated based on the empirical analysis described in Section 8.4.1 to reflect the relative importance of each criterion. The *SourceRisk* metric applies the empirically calibrated weights derived from 223 vulnerable contracts. *SinkCriticality* evaluates the sensitivity of target operations, with financial operations receiving higher scores than state modifications. *Protection* quantifies defensive mechanisms as described in Section 8.3.2. *PatternNovelty* measures uniqueness relative to the pattern registry (Section 8.4.3).

8.4.2.3 *Hybrid Refill Strategy*

When the pool size drops below a threshold of 10 paths, the refill mechanism activates. The *hybrid refilling strategy* combines 50% priority-based selection with 50% random selection. This ratio provides optimal balance between *exploitation* and *exploration*. Exploitation examines high-priority paths. Exploration discovers novel patterns. After each refill, the pool undergoes random shuffling. This eliminates placement order bias. It prevents the agent from developing position-dependent decision patterns.

8.4.3 *Pattern Registry*

The pattern registry system tracks discovered vulnerability signatures throughout training. Each pattern is stored as a five-tuple $(\textit{source_type}, \textit{source_context}, \textit{sink_type}, \textit{sink_context}, \textit{mitigation_level})$. The `source_context` captures how the entropy source is used (e.g., direct, hashed, combined, stored, arithmetic). The `sink_context` indicates the operational context of the sink (e.g., conditional, loop,

external call, state update, event emission). The `mitigation_level` distinguishes between none, partial, and full protection. This representation forms a unique signature for each vulnerability instance, enabling fine-grained pattern discrimination.

When the agent makes an `ANALYZE` decision, the system extracts the path's pattern. It compares this against existing registry entries. Discovery of a novel pattern triggers two actions. First, the pattern is added to the registry with an initial observation count of one. Second, the agent receives a pattern discovery bonus reward. For previously seen patterns, the observation count increments. This enables calculation of a uniqueness score as the inverse of repetition frequency.

This mechanism serves dual purposes. During training, it encourages the agent to explore diverse vulnerability patterns. The agent avoids repeatedly analyzing similar instances. The exploration bonus prevents premature convergence to local optima. In these optima, the agent exploits known patterns while ignoring underrepresented vulnerability types. During deployment, pattern frequency statistics inform priority scoring. Rare patterns receive higher selection probability. Over the course of 2,500 training episodes, the system discovered **1,247 unique patterns** from the space of possible five-tuple combinations. 85% were identified within the first 1,000 episodes.

8.4.4 *Reinforcement Learning Environment*

Our reinforcement learning environment is formulated as a **Markov Decision Process**. It is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. This formulation enables systematic application of Q-learning algorithms. It incorporates domain-specific structure for smart contract vulnerability detection.

8.4.4.1 *State Space*

The state space $\mathcal{S}$ is represented as a fixed-size matrix with dimensions (20×100). Each row encodes one path's complete feature vector. When the pool contains fewer than 20 paths, remaining rows are filled with zero vectors to maintain consistent dimensionality. For larger contracts where the adaptive pool size exceeds 20 paths, the system processes paths in sequential batches with priority-based ordering. This matrix representation enables the attention mechanism to capture both individual path characteristics and relationships between paths within the pool context.

8.4.4.2 *Action Space*

The action space $\mathcal{A}$ is defined as a discrete combinatorial:

$$\mathcal{A} = \{(i, t) \mid i \in [0, 19], t \in \{\texttt{ANALYZE}, \texttt{SKIP}\}\} \tag{8.7}$$

where i represents the path index within the fixed state representation and t specifies the decision type. For empty slots when the pool contains fewer than 20 paths, the agent automatically selects `SKIP`. This *per-path decision-making* design distinguishes our approach from contract-level methods. The agent independently evaluates each path and determines whether it warrants deep analysis (`ANALYZE`) or can be safely pruned (`SKIP`). This granular control enables selective resource allocation based on learned vulnerability patterns.

8.4.4.3 *State Transition Dynamics*

The transition function $\mathcal{P}(s' \mid s, a)$ is deterministic. Executing action $(i, \texttt{ANALYZE})$ or $(i, \texttt{SKIP})$ removes path i from the pool. When the pool size drops below the refill threshold, the hybrid refill strategy activates. It samples new paths according to priority scores and random selection. Each `ANALYZE` action consumes 0.5 budget units. `SKIP` is costless. This asymmetric cost structure incentivizes the agent toward selective analysis. It rewards efficient pruning of low-value paths.

8.4.4.4 *Episode Structure*

Each episode begins by selecting a contract from the dataset. The system initializes the pool with k paths sampled according to priority scores. The agent then makes sequential decisions until one of three termination conditions occurs. First, the initial budget of 100 units may be exhausted. Second, the maximum step limit may be reached. Third, the pool may become empty without available paths for refilling. This episodic structure exposes the agent to diverse contracts of varying complexity. The budget constraint forces the agent to make strategic decisions about resource allocation.

8.4.5 *Deep Q-Network Architecture*

The DQN architecture processes pool states and produces per-path action values. Unlike traditional contract-level approaches, our network computes Q-values independently for each path within the pool.

8.4.5.1 *Attention-Guided Feature Processing*

The network employs *scaled dot-product attention* to weight feature importance dynamically. This mechanism enables the model to identify which paths within the pool merit closer examination. Three transformation matrices project the input state into attention space with hidden dimension 64. These are Query ($\mathbf{W}_Q$), Key ($\mathbf{W}_K$), and Value ($\mathbf{W}_V$):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V \tag{8.8}$$

The attention weights α serve dual purposes. During training, they guide the network's focus toward discriminative features. During inference, they provide attribution scores for vulnerability localization. The scaling factor $\sqrt{d_k}$ prevents saturation of the softmax function. This maintains stable gradients throughout training.

8.4.5.2 *Q-Value Computation*

The attention output passes through a four-layer fully-connected network. The dimensions are [64, 128, 64, 32]. Each hidden layer applies ReLU activation and dropout with probability 0.3 for regularization. The final layer produces two Q-values for each path:

$$Q(\text{path}_i, a) = \text{FC}(\text{AttentionOutput}_i), \quad a \in \{\texttt{ANALYZE}, \texttt{SKIP}\} \tag{8.9}$$

The network architecture is relatively lightweight, enabling efficient training on our dataset without overfitting, as evidenced by the stable convergence shown in Figure 8.10.

8.4.5.3 *Exploration Strategy*

Action selection employs ϵ-greedy exploration. With probability ϵ, the agent explores by choosing actions uniformly at random. With probability $1 - \epsilon$, it exploits current knowledge by selecting $\arg\max_a Q_\theta(s, a)$. The exploration rate decays exponentially:

$$\epsilon_t = \max(\epsilon_{\min}, \epsilon_0 \cdot 0.995^t) \tag{8.10}$$

Here, $\epsilon_0 = 1.0$, $\epsilon_{\min} = 0.05$, and t represents the episode number. This schedule transitions gradually from random exploration to informed exploitation. The transition occurs over approximately 500 episodes.

8.4.5.4 *Training Stability Mechanisms*

Training employs **Double DQN** to prevent Q-value overestimation. The system maintains two networks. The online network Q_θ is updated every training step. The target network $Q_{\theta'}$ is synchronized every 50 steps through $\theta' \leftarrow \theta$. The target network provides stable temporal difference targets. This prevents the moving target problem that destabilizes standard Q-learning.

An experience replay buffer with capacity 5,000 stores transitions (s, a, r, s') for off-policy learning. Training samples random minibatches of size 32. This breaks temporal correlations in the experience sequence. The loss function minimizes temporal difference error:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{B}} \left[\left(Q_\theta(s, a) - \left(r + \gamma \cdot \max_{a'} Q_{\theta'}(s', a') \right) \right)^2 \right] \tag{8.11}$$

Gradient clipping at magnitude 0.5 prevents exploding gradients. The Adam optimizer with learning rate $\alpha = 10^{-5}$ provides stable convergence. The discount factor $\gamma = 0.95$ balances immediate rewards against long-term episode performance.

8.4.6 *Hierarchical Reward Engineering*

The reward function guides the agent toward effective vulnerability detection through four hierarchical levels. Each level addresses different aspects of the learning objective. These range from hard safety constraints to soft exploration incentives.

8.4.6.1 *Level 1: Mandatory Safety Constraints*

The first level enforces unbreakable rules preventing catastrophic errors:

$$R_1(s,a,p) = \begin{cases} -\rho_{\text{max}} & \text{if RiskCategory}(p) = \textsc{Critical} \wedge a = \texttt{SKIP} \\ -\rho_{\text{med}} & \text{if RiskCategory}(p) = \textsc{Negligible} \wedge a = \texttt{ANALYZE} \\ 0 & \text{otherwise} \end{cases} \tag{8.12}$$

High-risk paths must never be skipped. Doing so incurs maximum penalty $\rho_{\text{max}} = 10$. This overwhelms all other reward components. Low-risk paths should not waste computational resources. Unnecessary analysis receives moderate penalty $\rho_{\text{med}} = 3$. This level establishes hard constraints that the agent must respect unconditionally. These constraints are necessary to achieve acceptable recall.

8.4.6.2 *Level 2: Importance-Driven Decisions*

The second level rewards alignment between decisions and importance scores:

$$R_2(s,a,p) = \alpha \cdot \sigma(a) \cdot (I(p) - \mu_{\text{pool}}) \tag{8.13}$$

The importance score $I(p) = \sum_j w_j \cdot f_j(p)$ combines calibrated source weights with path features. The pool median $\mu_{\text{pool}} = \text{median}\{I(p') \mid p' \in \text{CurrentPool}\}$ provides context-relative benchmarking. The sign function $\sigma(a) = +1$ if $a = \texttt{ANALYZE}$. It equals -1 if $a = \texttt{SKIP}$. This ensures proper reward direction. The weight $\alpha = 1.0$ balances importance-driven decisions against other reward components.

This formulation rewards analyzing paths with above-median importance. It penalizes analysis of below-median paths. The relative scoring adapts automatically to different contracts. What constitutes high importance in a simple contract differs from high importance in a complex one.

8.4.6.3 *Level 3: Contextual Adjustment*

The third level applies fine-grained adjustments based on contract-level features:

$$R_3(s, a, p) = \beta \cdot \text{ContractComplexity}(s) \cdot \text{VulnerabilityDensity}(s) \cdot \mathbb{1}[a = \texttt{ANALYZE}] \tag{8.14}$$

Contract complexity measures structural attributes including function count, control flow depth, and state variable usage. Vulnerability density estimates the concentration of suspicious patterns relative to contract size. The sign function $\sigma(a) = +1$ if $a = \texttt{ANALYZE}$ and -1 if $a = \texttt{SKIP}$, similar to Level 2. This formulation encourages more thorough analysis in complex contracts with high vulnerability density, while allowing aggressive pruning in simpler contracts. The weight $\beta = 0.5$ provides moderate influence without overriding importance scores.

8.4.6.4 *Level 4: Exploration Incentive*

The fourth level encourages pattern discovery through novelty bonuses:

$$R_4(s, a, p) = \delta \cdot \mathbb{1}[\text{Pattern}(p) \notin \text{Registry}] \cdot \mathbb{1}[a = \texttt{ANALYZE}] \tag{8.15}$$

When the agent analyzes a path exhibiting a previously unseen vulnerability pattern, it receives bonus reward $\delta = 2.0$. The pattern signature comprising *source_type*, *source_context*, *sink_type*, *sink_context*, and *mitigation_level* is compared against the registry. Novel patterns receive positive reinforcement. This encourages vulnerability coverage. This incentive prevents convergence to local optima. In these optima, the agent repeatedly exploits known patterns while ignoring underrepresented vulnerability types.

8.4.6.5 *Composite Reward Function*

The final reward combines all four levels through weighted summation:

$$R_{\text{total}}(s, a, p) = \sum_{i=1}^{4} \lambda_i \cdot R_i(s, a, p) \tag{8.16}$$

The hierarchical weights are $\lambda_1 = 1.0$, $\lambda_2 = 0.8$, $\lambda_3 = 0.3$, $\lambda_4 = 0.2$. These satisfy the ordering $\lambda_1 > \lambda_2 > \lambda_3 > \lambda_4$. This ordering ensures safety constraints dominate the learning signal. The system still incorporates importance scoring, contextual adaptation, and exploration incentives. The structure integrates domain-specific knowledge about blockchain randomness vulnerabilities. It maintains adaptability through learned patterns.

Algorithm 4 SmartTaintRL Training with Empirical Calibration

Require: Contracts $\mathcal{C}$, Calibrated weights $\mathcal{W}$
Ensure: Trained model θ^*, Pattern bank $\mathcal{P}$

1: **function** SmartTaintRL($\mathcal{C}$)
2: $\theta, \theta' \leftarrow$ InitNetworks() ▷ Q-network and target network
3: $\mathcal{W} \leftarrow$ LoadWeights(223 vuln contracts) ▷ Empirical calibration
4: $\mathcal{B}, \mathcal{P} \leftarrow$ EmptyBuffer(), EmptyPatternBank()
5: **for** episode = 1 to MAX_EPISODES **do**
6: contract $\leftarrow$ SelectUnseen($\mathcal{C}$) ▷ Prioritize less-visited
7: pool $\leftarrow$ SamplePaths(contract, $k = 30$) ▷ Initial pool
8: **while** pool.has_paths() **do**
9: state $\leftarrow$ Encode(pool) ▷ $k \times 100$ matrix
10: action $\leftarrow$ ϵ-greedy(Q_θ(state), ϵ)
11: importance $\leftarrow$ $\mathcal{W}$.source[path] $\times$ $\mathcal{W}$.sink[path] $\times$ (1 − mitigation)
12: reward $\leftarrow$ HierarchicalReward(action, risk, importance)
13: **if** action = `ANALYZE` and NewPattern(path) **then**
14: $\mathcal{P}$.add(path), reward $\leftarrow$ reward +2
15: **end if**
16: next_state $\leftarrow$ UpdatePool(pool, refill_at = 10)
17: $\mathcal{B}$.store(state, action, reward, next_state)
18: **if** $|\mathcal{B}| \geq 32$ **then**
19: TrainNetwork($\theta, \theta', \mathcal{B}, \gamma = 0.95$) ▷ TD learning
20: **end if**
21: **if** episode $\bmod 50 = 0$ **then**
22: $\theta' \leftarrow \theta$ ▷ Sync target
23: **end if**
24: **end while**
25: $\epsilon \leftarrow \max(0.05, \epsilon \times 0.995)$ ▷ Decay exploration
26: **end for**
27: **return** $\theta^*, \mathcal{P}$
28: **end function**

8.5 PHASE 3: VULNERABILITY LOCALIZATION

After detecting vulnerable contracts through the RL-based approach, our localization methodology identifies the locations of vulnerabilities within the code. The system operates at two hierarchical levels. *Function-level localization* identifies which functions contain vulnerabilities. This is followed by *node-level localization* that pinpoints the specific code locations responsible for security flaws. Algorithm 5 presents the complete localization procedure.

Unlike detection methods that merely flag entire contracts as vulnerable, localization provides actionable information for developers. Security auditors need to know exactly which lines of code require

remediation rather than manually inspecting entire codebases. This capability transforms vulnerability reports from binary classifications into detailed roadmaps for security improvements.

8.5.1 *Function-Level Localization*

Function-level localization aggregates evidence from multiple paths to identify vulnerable functions. The process groups all paths in a contract by their containing function. For each function f, we compute an aggregate score combining two complementary signals.

First, we sum the Q-values of all paths belonging to the function:

$$Q_{\text{sum}}(f) = \sum_{p \in \mathcal{P}_f} Q_\theta(s_p, \texttt{ANALYZE}) \tag{8.17}$$

Here, $\mathcal{P}_f$ denotes all paths within function f. The variable s_p represents the state when path p is in the pool. Higher Q-values indicate the agent's confidence that a path warrants analysis.

Second, we count the number of paths where the agent selected `ANALYZE`:

$$\text{votes}(f) = \left| \{ p \in \mathcal{P}_f : \arg\max_a Q_\theta(s_p, a) = \texttt{ANALYZE} \} \right| \tag{8.18}$$

The final function score combines these signals:

$$\text{Score}_{\text{func}}(f) = \frac{Q_{\text{sum}}(f)}{|\mathcal{P}_f|} + \frac{\text{votes}(f)}{|\mathcal{P}_f|} \tag{8.19}$$

where $|\mathcal{P}_f|$ denotes the number of paths belonging to function f. This normalization ensures that functions with different path counts are compared on a consistent scale, preventing functions with more paths from receiving artificially inflated scores.

Functions are ranked by this score. Those exceeding a threshold or appearing in the top-k are selected as vulnerable. This aggregation approach proves robust to individual path misclassifications. A single false positive path does not condemn an entire function. Multiple suspicious paths provide strong evidence of genuine vulnerabilities.

8.5.2 *Node-Level Localization*

Node-level localization identifies the specific code locations within vulnerable functions. This process involves five sequential steps. These steps combine gradient-based attribution, graph-theoretic analysis, and centrality measures.

8.5.2.1 *Step 1: Initial Attribution through Gradient Analysis*

For each path p in a vulnerable function where the agent selected `ANALYZE`, we compute gradient-based attribution scores. The path's

state vector s_p is fed to the trained model to produce Q-values. We then compute the gradient of the Q-value with respect to each feature:

$$\nabla Q = \left[\frac{\partial Q_\theta(s_p, \texttt{ANALYZE})}{\partial f_1}, \ldots, \frac{\partial Q_\theta(s_p, \texttt{ANALYZE})}{\partial f_{100}} \right] \tag{8.20}$$

where f_i represents the i-th feature in the 100-dimensional feature vector described in Section 8.3.2. These gradients measure how much each feature influences the decision.

These gradients measure how much each feature influences the decision. Large positive gradients indicate features that strongly support the `ANALYZE` decision. We then map these feature gradients to corresponding nodes in the semantic graph. For instance, gradients for `source_type` and `sink_type` features map to the source and sink nodes. Gradients for `path_length` and `complexity` distribute across intermediate nodes. This mapping produces an initial attribution score for each node. This score indicates how much the model focused on it during decision-making.

8.5.2.2 *Step 2: Subgraph Extraction*

From the contract's complete semantic graph $G = (V, E)$, we extract a subgraph $G_p = (V_p, E_p)$ for each suspicious path p. This subgraph contains only nodes appearing on the path and edges connecting them. The extracted subgraph represents the specific data and control flow of that execution path.

The subgraph includes three node types. The source node is where dangerous values originate, such as `block.timestamp`. Intermediate nodes process or propagate these values. The sink node is where values reach sensitive operations, such as `transfer` or `randomGeneration`. Edges represent data dependencies showing how variables flow between operations. They also represent control dependencies indicating execution order.

8.5.2.3 *Step 3: Graph Propagation*

Initial gradient scores provide point estimates of node importance. However, intermediate nodes in the data flow path may have small direct gradients despite being critical to vulnerability exploitation. Graph propagation addresses this by spreading scores through the subgraph structure.

We perform two types of propagation based on data dependency edges.**Backward propagation** proceeds from sink to source. It transfers the sink's score to predecessor nodes:

$$\text{score}_{\text{backward}}(n) = \text{score}_{\text{initial}}(n) + \beta \sum_{m \in \text{successors}(n)} \frac{\text{score}_{\text{backward}}(m)}{d(m, \text{sink}) + 1} \tag{8.21}$$

Here, $d(m, \text{sink})$ measures the graph distance from successor node m to the sink. The $+1$ term prevents division by zero when m is the sink node. The parameter $\beta = 0.5$ is a decay factor that controls the strength of score propagation. Nodes that generated or transferred data to a sensitive sink contribute to the vulnerability.

Forward propagation proceeds from source to sink. It transfers the source's score to successor nodes:

$$\text{score}_{\text{forward}}(n) = \text{score}_{\text{initial}}(n) + \beta \sum_{m \in \text{predecessors}(n)} \frac{\text{score}_{\text{forward}}(m)}{d(\text{source}, m) + 1} \tag{8.22}$$

where $d(\text{source}, m)$ measures the graph distance from the source to predecessor node m. The $+1$ term prevents division by zero when m is the source node. The same decay factor $\beta = 0.5$ is applied. Nodes processing tainted data receive elevated scores. In both propagation directions, exponential decay through distance-based division ensures scores decrease with distance. Nodes closer to sources or sinks receive higher scores, while distant nodes get lower contributions.

8.5.2.4 *Step 4: Centrality Analysis*

In addition to gradient-based attribution and graph propagation, we identify structurally important nodes through centrality measures. These graph-theoretic metrics reveal nodes that occupy critical positions in the data flow network. We compute two complementary centrality measures. **Degree centrality** counts incoming and outgoing edges:

$$C_{\text{degree}}(n) = \frac{\deg^{\text{in}}(n) + \deg^{\text{out}}(n)}{\max_{v \in V_p}(\deg^{\text{in}}(v) + \deg^{\text{out}}(v))} \tag{8.23}$$

where $\deg^{\text{in}}(n)$ denotes the number of incoming edges to node n (in-degree) and $\deg^{\text{out}}(n)$ denotes the number of outgoing edges (out-degree). The denominator normalizes the score to the range $[0, 1]$ by dividing by the maximum total degree in the subgraph. High degree indicates nodes that interact with many other operations. These nodes serve as hubs through which data flows.

Betweenness centrality measures how many shortest paths pass through each node:

$$C_{\text{between}}(n) = \sum_{s \neq t \neq n} \frac{\sigma_{st}(n)}{\sigma_{st}} \tag{8.24}$$

where σ_{st} denotes the total number of shortest paths between nodes s and t in the subgraph, and $\sigma_{st}(n)$ counts how many of these paths pass through node n. The summation iterates over all pairs of distinct nodes s and t (excluding n itself). High betweenness indicates nodes that mediate information flow between sources and sinks, acting as critical bridges in the data flow network.

We normalize these centrality scores to $[0, 1]$ and combine them:

$$C_{\text{total}}(n) = 0.5 \cdot C_{\text{degree}}(n) + 0.5 \cdot C_{\text{between}}(n) \tag{8.25}$$

8.5.2.5 *Step 5: Aggregation and Ranking*

The final node importance score combines three components:

$$\text{Score}_{\text{final}}(n) = \alpha_1 \cdot \text{score}_{\text{gradient}}(n) + \alpha_2 \cdot \text{score}_{\text{propagated}}(n) + \alpha_3 \cdot C_{\text{total}}(n) \tag{8.26}$$

The gradient score $\text{score}_{\text{gradient}}$ captures the model's learned attention. The propagated score $\text{score}_{\text{propagated}} = \text{score}_{\text{backward}} + \text{score}_{\text{forward}}$ reflects importance in the data flow path. The centrality score C_{total} identifies structurally central nodes. The weights $\alpha_1 = 0.5$, $\alpha_2 = 0.3$, $\alpha_3 = 0.2$ balance these complementary perspectives.

Nodes are ranked by their final scores. The top-k nodes are identified as vulnerable locations. These typically include three types. First, source nodes where dangerous variables are defined, such as assignments from `block.timestamp`. Second, critical intermediate nodes that propagate tainted values. Third, sink nodes where tainted data reaches sensitive operations, such as `transfer` or random number generation.

8.5.3 *Ground Truth Dataset*

To rigorously evaluate localization performance, we compiled a specialized ground truth dataset. To the best of our knowledge, no prior work has directly addressed localization for the Bad Randomness vulnerability at the node level. Therefore, our evaluation relies on this manually curated dataset. This dataset serves as the first benchmark for Bad Randomness localization.

Our dataset includes 14 real smart contracts from the SmartBugs Wild dataset. All of these contracts exhibit the Bad Randomness vulnerability. These contracts were manually reviewed and verified by security experts. For each contract, we identified the vulnerable function and the corresponding source and sink nodes. Table 8.1 presents the core contract information, with complete addresses and detailed vulnerability subtypes provided in Appendix B.1.

The dataset shows varying levels of complexity. Contracts range from simple implementations with 7 paths (C3) to complex systems with 460 paths (C12). In total, the dataset contains 1,491 control flow paths. We categorize contracts into three complexity levels based on path count: Low (7 contracts with fewer than 20 paths), Medium (3 contracts with 20–100 paths), and High (4 contracts with more than 100 paths).

Algorithm 5 Vulnerability Localization

Require: P (paths with metadata), M_{DQN} (trained model), G (semantic graph), $\mathcal{W}$ (calibrated weights)
Ensure: $V_{\text{functions}}$ (vulnerable functions), V_{nodes} (vulnerable nodes)
1: // Function-Level Localization
2: function_scores $\leftarrow \{\}$
3: **for** each function f in contract **do**
4: $\mathcal{P}_f \leftarrow \text{GetPathsInFunction}(f, P)$
5: $Q_{\text{sum}} \leftarrow \sum_{p \in \mathcal{P}_f} Q(s_p, a_p)$ ▷ Sum of Q-values from DQN
6: votes $\leftarrow |\{p \in \mathcal{P}_f : M_{\text{DQN}}.\text{predict}(p) = \texttt{ANALYZE}\}|$
7: function_scores$[f] \leftarrow (Q_{\text{sum}} + \text{votes}) / |\mathcal{P}_f|$
8: **end for**
9: $V_{\text{functions}} \leftarrow \text{TopK}(\text{function_scores})$
10:
11: // Node-Level Localization
12: **for** each function $f \in V_{\text{functions}}$ **do**
13: $\mathcal{P}_f \leftarrow \text{GetPathsInFunction}(f, P)$
14: grad_scores $\leftarrow \{\}$
15: **for** each path $p \in \mathcal{P}_f$ where $M_{\text{DQN}}.\text{predict}(p) = \texttt{ANALYZE}$ **do**
16: state $\leftarrow \text{ExtractState}(p, \mathcal{W})$
17: $\nabla Q \leftarrow \frac{\partial M_{\text{DQN}}(\text{state})}{\partial \text{features}}$ ▷ Gradient-based attribution
18: $\text{MapGradientToNodes}(p, \nabla Q, \text{grad_scores})$
19: **end for**
20: **for** each path $p \in \mathcal{P}_f$ **do**
21: $G_p \leftarrow \text{ExtractSubgraph}(p, G)$
22: prop_scores $\leftarrow$ grad_scores
23: $\text{BackwardPropagate}(G_p, \text{sink}, \text{prop_scores})$ ▷ From sink to source
24: $\text{ForwardPropagate}(G_p, \text{source}, \text{prop_scores})$ ▷ From source to sink
25: cent_scores $\leftarrow \text{ComputeCentrality}(G_p)$ ▷ Degree + Betweenness
26: **for** each node $n \in G_p$ **do**
27: $\text{final_score}[n] \leftarrow \alpha_1 \cdot \text{grad_scores}[n] + \alpha_2 \cdot \text{prop_scores}[n] + \alpha_3 \cdot \text{cent_scores}[n]$
28: **end for**
29: **end for**
30: $V_{\text{nodes}}[f] \leftarrow \text{RankNodes}(\text{final_score})$ ▷ Top-k nodes
31: **end for**
32: **return** $V_{\text{functions}}, V_{\text{nodes}}$

8.5.4 *Evaluation Results*

Table 8.2 presents evaluation results on 14 vulnerable smart contracts. The evaluation demonstrates the model's capability to identify vul-

Table 8.1: Ground Truth Dataset for Bad Randomness Localization

ID	Function	Lvl	P	Source → Sink	Domain
C1	chooseWinner	L	15	timestamp → valueTransfer	Lottery
C2	_prand	M	61	coinbase, timestamp → valueTransfer	Gaming
C3	random	L	7	timestamp → stateModification	Gaming
C4	withdraw	M	56	timestamp → valueTransfer	Gaming
C5	pseudoRandom	L	15	timestamp → controlFlow	Gaming
C6	createRandomNumber	H	275	timestamp → randomGeneration	Gambling
C7	chooseWinner	L	15	coinbase, tx_origin → valueTransfer	Lottery
C8	chooseWinner	L	15	timestamp → valueTransfer	Lottery
C9	updateProfit	L	18	timestamp → stateModification	Gaming
C10	burn	L	8	difficulty, timestamp → valueTransfer	Gaming
C11	settleBet	H	376	blocknumber → financialDecision	Gambling
C12	callback	H	460	blocknumber → valueTransfer	Gaming
C13	playerRollDice	M	45	blocknumber → randomGeneration	Gambling
C14	doOraclize	H	125	timestamp → financialDecision	Gambling

Total: 14 contracts, 1,491 taint paths. Distribution: L=7, M=3, H=4

Lvl: Complexity level (L=Low <20 paths, M=Medium 20–100, H=High >100). P: Total taint paths. Full contract addresses in Appendix B.1.

nerable functions with 92.9% accuracy. This includes inter-procedural detections. The model also localizes vulnerable nodes within those functions. Figure 8.4 visualizes the results across contract complexity levels and evaluation metrics.

8.5.4.1 *Key Observations*

PERFORMANCE ON COMPLEX CONTRACTS. The model maintains robust performance even on large-scale contracts. Contract C11 has 1,024 lines of code and 376 paths. Contract C12 has 1,156 lines and 460 paths. Both demonstrate successful localization. The model correctly identifies their vulnerable functions and prioritizes relevant nodes despite high complexity. This success stems from meticulous preprocessing. In this phase, taint analysis extracts rich structural information from control flow graphs. The reinforcement learning agent then learns to navigate this complex information space effectively. The localization algorithm uses importance scoring and propagation. Despite its relative simplicity, the model handles contract complexity by leveraging rich representations. These representations are extracted during preprocessing.

Table 8.2: Localization Results on 14 Vulnerable Contracts

ID	GT	GT Function	Predicted	Type	Size	Paths	#VN
C1	✓	chooseWinner	chooseWinner	Exact	245	15	3
C2	✓	_prand	drainMe	Caller	512	61	4
C3	✓	random	getMilk	Caller	189	7	1
C4	✓	withdraw	withdraw	Exact	438	56	13
C5	✓	pseudoRandom	forgeRandomItem	Caller	298	15	1
C6	✓	createRandomNumber	api_PlaceBet	Caller	823	275	12
C7	✓	chooseWinner	chooseWinner	Exact	231	15	3
C8	✓	chooseWinner	chooseWinner	Exact	245	15	3
C9	✓	updateProfit	updateProfit	Exact	276	18	4
C10	✓	burn	burn	Exact	167	8	3
C11	✓	settleBet	settleBet	Exact	1024	376	4
C12	✓	callback	callback	Exact	1156	460	9
C13	✓	playerRollDice	playerRollDice	Exact	352	45	5
C14	×	doOraclize	getRoomInfo	Wrong	687	125	3

Overall Performance: Function: 13/14 (92.9%) | Exact matches: 9/14 (64.3%) | Caller: 4/14 (28.6%)
Node-Level: P@5=0.65, R@5=0.77, F1@5=0.70 (computed on 9 exact matches)

GT: Ground truth with ✓ (correct) or × (incorrect). **GT Function**: Actual vulnerable function from ground truth. **Predicted**: Function predicted by our model. Size: Lines of code. #VN: Number of vulnerable nodes. **Type**: Exact = predicted function matches ground truth directly (64.3% strict accuracy); Caller = predicted function invokes vulnerable function (92.9% relaxed accuracy); Wrong = incorrect. Node-Level Metrics: P, R, F1 at top-5, averaged across 9 exact matches. Complete contract information in Appendix B.1.

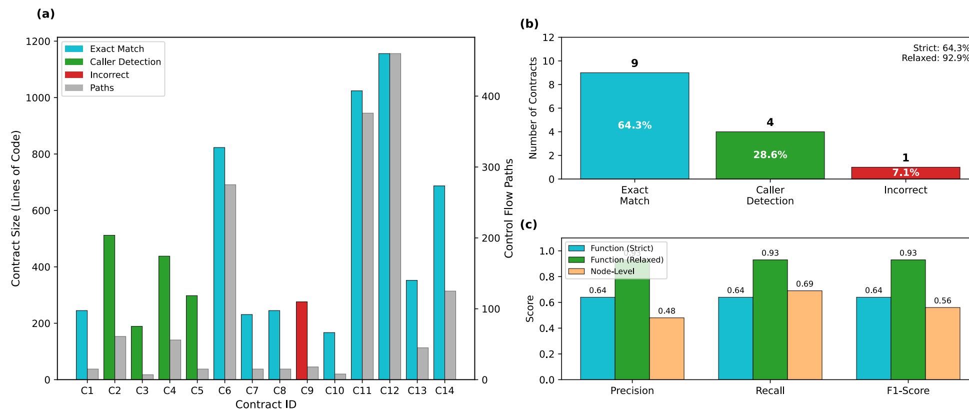

Figure 8.4: Localization evaluation results on 14 vulnerable contracts. **(a)** Contract complexity distribution showing lines of code (colored bars) and control flow paths (gray bars) for each contract, with color-coding indicating localization outcome: blue for exact matches, green for caller detection, and red for incorrect predictions. **(b)** Function-level localization distribution across exact match (64.3%), caller detection (28.6%), and incorrect predictions (7.1%), achieving strict accuracy of 64.3% and relaxed accuracy of 92.9%. **(c)** Performance metrics comparison across function-level (strict and relaxed) and node-level localization.

DETECTION OF CALLER FUNCTIONS. In four contracts (C2, C3, C5, C6), the model identified caller functions rather than directly vulnerable callees. For instance, in C2, the model predicted `drainMe` while the ground truth is `_prand`. Subsequent analysis revealed that `drainMe` invokes `_prand`. Similarly, in C6, the model predicted `api_PlaceBet` which calls the vulnerable `createRandomNumber`. This inter-procedural detection capability proves valuable for security auditing. Caller functions represent attack entry points. They must be examined alongside vulnerable callees to understand complete attack vectors. The relaxed accuracy metric (92.9%) accounts for this. It considers both exact matches and caller detections as correct.

NODE-LEVEL LOCALIZATION EFFECTIVENESS. For nine contracts with exact function matches, the average recall at top-5 predictions reaches 77%. Several contracts (C1, C7, C8, C9, C10) achieve 100% recall. This means all vulnerable nodes appear in the top-5 predictions. This demonstrates an important capability. Despite the simplicity of the localization mechanism, the high-quality path representations learned by the DQN agent enable accurate prioritization. These representations allow the system to prioritize truly vulnerable nodes. The precision of 65% indicates that approximately two-thirds of top-5 predictions are correct. This provides developers with a focused set of locations for manual inspection. They do not need to examine entire contracts.

8.6 DATASET CONSTRUCTION

Our evaluation employs the dataset introduced in Chapter 7. This dataset comprises 4,706 contracts (423 vulnerable, 4,283 safe) with 252,844 execution paths. These paths were extracted from three primary sources: SmartBugs-Curated, SWC Registry, and SmartBugs-Wild. This section describes the dataset characteristics and filtering procedures applied for reinforcement learning training.

8.6.1 *Dataset Overview*

Table 8.3 presents statistics across two experimental scenarios. The **balanced dataset** contains 400 contracts (200 vulnerable, 200 safe) with 5,271 paths. This provides controlled conditions for initial evaluation. The **imbalanced dataset** contains 4,306 contracts (223 vulnerable, 4,083 safe) with 247,573 paths. This reflects real-world deployment scenarios. In these scenarios, vulnerabilities constitute approximately 5% of cases (1:18 ratio).

Each path receives one of three risk labels: HIGH, MEDIUM, or LOW. These labels are based on analysis of source type criticality, sink operation sensitivity, and protection mechanism presence. HIGH-

Table 8.3: Dataset Statistics: Contracts, Paths, and Label Distribution

Dataset	Contracts			Execution Paths				Avg.
	Total	Vuln.	Safe	Total	HIGH	MEDIUM	LOW	Paths
Balanced	400	200	200	5,271	2,629	2,102	540	13
Imbalanced	4,306	223	4,083	247,573	122,500	78,387	46,686	57
Total	4,706	423	4,283	252,844	125,129	80,489	47,226	—

Execution paths are labeled with three risk levels (HIGH, MEDIUM, LOW) based on source criticality, sink sensitivity, and protection mechanisms. Average paths per contract shown in rightmost column.

risk paths involve critical entropy sources, such as `block.timestamp`. These sources flow to sensitive operations, such as `transfer`, without adequate protection. MEDIUM-risk paths exhibit partial risk factors. Examples include protected critical sources or unprotected less-critical sources. LOW-risk paths involve minimal entropy sources with strong protection mechanisms.

8.6.2 *Training Data Filtering*

For reinforcement learning training and empirical weight calibration, we applied quality filtering to ensure effective learning. Contracts failing compilation or exhibiting insufficient complexity (fewer than 5 paths) were excluded, producing a refined training set containing **223 vulnerable contracts** and **4,084 safe contracts**.

The 223 vulnerable contracts serve dual purposes. First, they provide training data exposing the agent to diverse vulnerability patterns across different contract types and complexity levels. Second, they enable empirical weight calibration through analysis of source prevalence in genuine vulnerabilities (Section 8.4.1), grounding our importance scoring in real-world distributions rather than arbitrary assignments.

The filtered dataset maintains the real-world class imbalance ratio while ensuring sufficient path diversity for meaningful learning. Contracts range from simple implementations with 7 paths (C3) to complex systems with 460 paths (C12), enabling the agent to learn generalizable detection strategies applicable across various contract architectures.

Path labels (HIGH, MEDIUM, LOW) were derived from the graduated taint analysis methodology described in Chapter 7, with 94% agreement confirmed through manual expert review.

8.7 EXPERIMENTAL SETUP

This section describes the implementation details, hyperparameter configurations, and training procedures employed for SMARTTAINTRL.

All experiments were conducted under consistent conditions to ensure reproducible results.

8.7.1 *Model Architecture and Hyperparameters*

The SmartTaintRL model implements an attention-based Deep Q-Network architecture. Table 8.4 presents the complete configuration across model architecture, reinforcement learning parameters, and training setup.

Table 8.4: SmartTaintRL Training Configuration and Hyperparameters

Component	**Parameter**	**Component**	**Parameter**
Model Architecture		*RL Parameters*	
State Dim.	(20, 100)	Learning Rate	1×10^{-5}
Attention Dim.	64	Discount (γ)	0.95
Hidden Layers	[64, 128, 64, 32]	ϵ Start	1.0
Output Layer	2 (ANALYZE, SKIP)	ϵ End	0.05
		ϵ Decay	0.995
Training Setup		*Training Duration*	
Optimizer	Adam	Episodes	5,000
Batch Size	32	Max Steps	50
Replay Buffer	10,000	Best Checkpoint	Ep. 2,500
Target Update	Soft (τ=0.001)	Time (Balanced)	~45 min
Grad. Clipping	0.5	Time (Imbalanced)	~6 hours

The state representation comprises a fixed-size (20×100) matrix, where each row encodes one path's feature vector. When the pool contains fewer than 20 paths, zero-padding maintains consistent dimensions; for larger pools, paths are processed in sequential batches. The attention mechanism projects this state into a 64-dimensional hidden space using learnable Query, Key, and Value transformation matrices. The subsequent fully-connected layers progressively refine feature representations through dimensions [64, 128, 64, 32], with ReLU activation and dropout (probability 0.3) applied after each hidden layer. The final output layer produces two Q-values per path, corresponding to `ANALYZE` and `SKIP` actions.

8.7.2 *Training Procedure*

Training proceeded through 5,000 episodes, with each episode processing a single contract. The ϵ-greedy exploration strategy begins with complete randomness ($\epsilon = 1.0$) and decays exponentially to $\epsilon_{\text{min}} = 0.05$ according to:

$$\epsilon_t = \max(0.05, 1.0 \times 0.995^t) \tag{8.27}$$

where t represents the episode number. This schedule ensures sufficient initial exploration while gradually transitioning to exploitation of learned policies.

The experience replay buffer maintains the most recent 10,000 transitions (s, a, r, s'), with training sampling random mini-batches of size 32 to break temporal correlations. The Adam optimizer with learning rate $\alpha = 10^{-5}$ updates the online network parameters after each action. The target network synchronizes with the online network through soft update ($\tau = 0.001$), providing stable temporal difference targets. Gradient clipping at magnitude 0.5 prevents exploding gradients during training.

Each episode initializes by selecting an unseen or less-visited contract from the training set, with the pool beginning with $k = 30$ paths sampled according to priority scores. The agent makes sequential decisions until one of three termination conditions occurs: the episode budget of 100 units is exhausted, the maximum step limit of 50 is reached, or the pool becomes empty without available paths for refilling.

8.7.3 *Hierarchical Reward Configuration*

The hierarchical reward function combines four levels with weights: $\lambda_1 = 1.0$ (Safety), $\lambda_2 = 0.8$ (Importance), $\lambda_3 = 0.3$ (Context), and $\lambda_4 = 0.2$ (Pattern). The safety constraint penalties are $\rho_{\text{max}} = 10$ for skipping critical paths and $\rho_{\text{med}} = 3$ for analyzing negligible paths. The importance scoring weight $\alpha = 1.0$ balances relative path priorities, while the contextual adjustment weight $\beta = 0.5$ provides moderate contract-level influence. The pattern discovery bonus $\delta = 2.0$ encourages exploration.

The weight configuration follows established principles in safety-critical reinforcement learning [**garcia2015comprehensive**], where hard constraints must dominate soft objectives. The hierarchical structure ($\lambda_1 > \lambda_2 > \lambda_3 > \lambda_4$) ensures that safety violations incur penalties ($\rho_{\text{max}} \cdot \lambda_1 = 10$) large enough to override all other reward components. This design reflects the asymmetric cost structure inherent in vulnerability detection, where false negatives impose significantly higher costs than false positives [**arp2014drebin**].

8.7.4 *Implementation Environment*

The system is implemented in Python 3.12 using PyTorch 2.1.0 for neural network components, with contract compilation employing Solidity compiler (solc) version 0.8.0 for bytecode generation and AST parsing. All experiments were conducted on a Microsoft Surface Pro (5th generation) equipped with an Intel Core i5 processor and 32 GB RAM, operating in CPU-only mode without GPU acceleration.

Table 8.5: Performance Metrics Across Training Episodes and Datasets

Dataset	Ep.	Acc.	Prec.	Rec.	F1	FP	FN	Analyze
Balanced (400)	500	0.89	0.90	0.96	0.93	217	3	0.90±0.10
	2500	0.95	0.90	0.96	0.93	1083	3	0.55±0.10
	5000	0.95	0.90	0.96	0.93	1626	5	0.50±0.15
Imbalanced (4306)	500	0.82	0.84	0.94	0.89	195	29	0.92±0.08
	2500	0.94	0.88	0.96	0.92	832	29	0.55±0.12
	5000	0.95	0.89	0.96	0.92	1405	151	0.48±0.18

Ep.: Episodes. Rec.: Recall. Analyze: mean $\pm$ std of proportion of paths selected for `ANALYZE`, computed across all contracts. Decreasing ratio demonstrates learned selective analysis.

The preprocessing phase required approximately 18 hours for the complete dataset, while the reinforcement learning training completed in approximately 6 hours for the imbalanced dataset and 45 minutes for the balanced dataset.

The final model selection employed early stopping based on validation performance. The checkpoint from episode 2,500 demonstrated optimal balance between training accuracy and generalization capability, achieving F1-score of 0.93 on the balanced dataset and 0.92 on the imbalanced dataset while maintaining 96% recall.

8.8 RESULTS AND EVALUATION

This section presents experimental results demonstrating SmartTaintRL's effectiveness in detecting Bad Randomness vulnerabilities. We evaluate the system across multiple dimensions. These include classification performance, training stability, path selection quality, and comparison with existing methods. All experiments were conducted on both balanced and imbalanced datasets. This assesses robustness under realistic deployment conditions.

8.8.1 *Overall Performance Analysis*

To identify the optimal training checkpoint, we evaluated model performance at three critical time points. These are 500, 2,500, and 5,000 episodes. Figure 8.5 and Table 8.5 present metrics across both datasets.

In the balanced dataset, accuracy increased from 0.89 at episode 500 to 0.95 at episode 2,500. It remained constant through episode 5,000. This plateau indicates model convergence. The trend appears consistently across all four metrics (Accuracy, Precision, Recall, F1-Score). Between episodes 2,500 and 5,000, only minimal F1-score improvement occurs. Training time doubles during this period. Despite the increase in false positives from 1,083 to 1,626, false negatives increased only

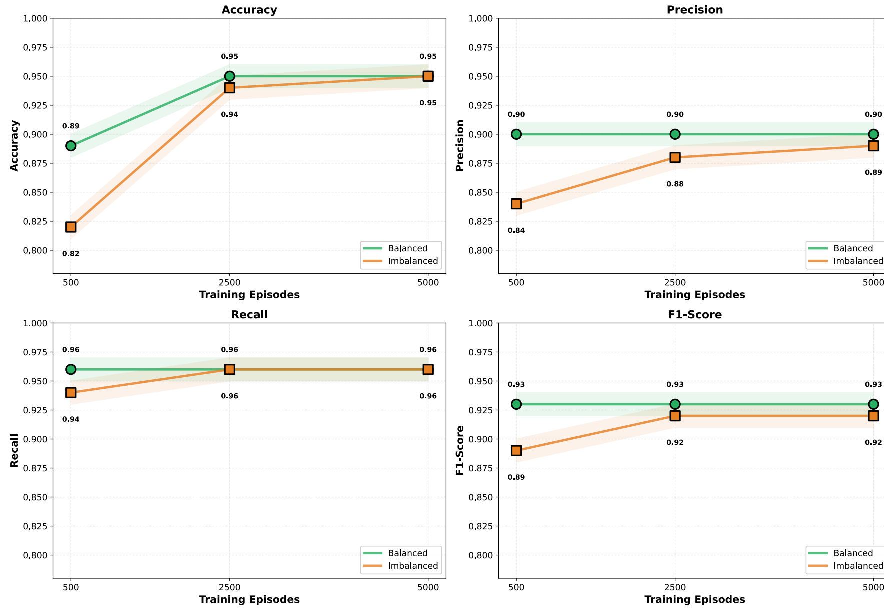

Figure 8.5: Performance metrics evolution across training episodes for balanced and imbalanced datasets. Four panels show Accuracy, Precision, Recall, and F1-Score trajectories from episode 500 to 5000.

from 3 to 5. This demonstrates that the model maintained high recall (0.96).

In the imbalanced dataset, precision increased from 0.82 to 0.94 at episode 2,500. It then increased to 0.95 at episode 5,000. However, false negatives increased significantly from 29 to 151. This indicates overfitting at higher episodes. The confusion matrices in Figure 8.6 confirm better balance at episode 2,500.

The Analyze ratio decreased from 0.90±0.10 at episode 500 to 0.55±0.10 at episode 2,500. This corresponds to 44.3% path pruning. It demonstrates that the model learned selective analysis instead of examining all paths. Figure 8.7 visualizes this evolution, showing the gradual transition from analyzing nearly all paths to selective prioritization. Therefore, episode 2,500 was selected as the optimal checkpoint. It provides the best trade-off between performance, training time, and generalization capability.

Contract-level detection rates demonstrate robust performance. Figure 8.8 shows that the model achieves 98.8% detection rate on the balanced dataset. It achieves 98.1% on the imbalanced dataset. Both rates exceed the 95% threshold required for practical deployment.

8.8.2 *Training Dynamics and Convergence*

The training dynamics confirm algorithm stability and convergence efficiency. Figure 8.9 presents the evolution of cumulative episode rewards throughout training. The average reward increased from initial

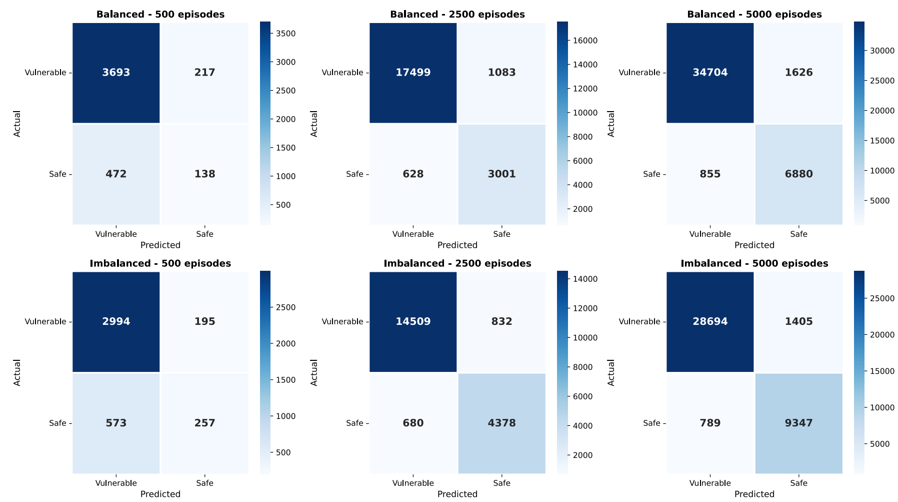

Figure 8.6: Confusion matrices for balanced and imbalanced datasets at 500, 2,500, and 5,000 episodes. Color intensity indicates prediction frequency.

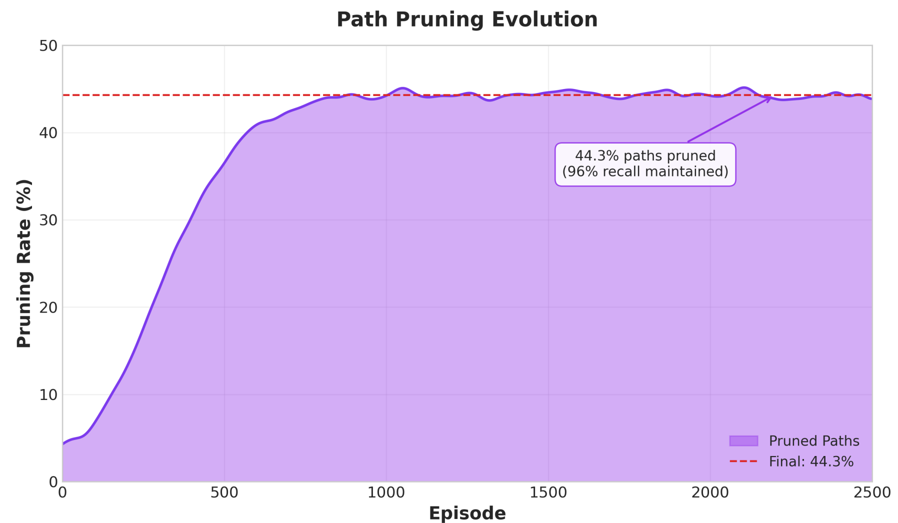

Figure 8.7: Path pruning evolution during training. The Analyze ratio decreases from 90% to 55.7% over 2,500 episodes, demonstrating that the agent learns to selectively skip low-risk paths while maintaining 96% recall.

negative values to 18.47 at episode 2,500. This uniform increase, combined with decreasing reward variance, indicates policy stabilization. It also indicates equilibrium achievement.

The loss function analysis in Figure 8.10 demonstrates effective learning convergence. Temporal difference error decreased from 1.5 to 0.2 over 50,000 training steps. The exponential decay rate prevented overfitting. It supported model generalization.

Q-value variance evolution reveals three distinct learning phases (Figure 8.11). First, the exploration phase (episodes 0–40) shows vari-

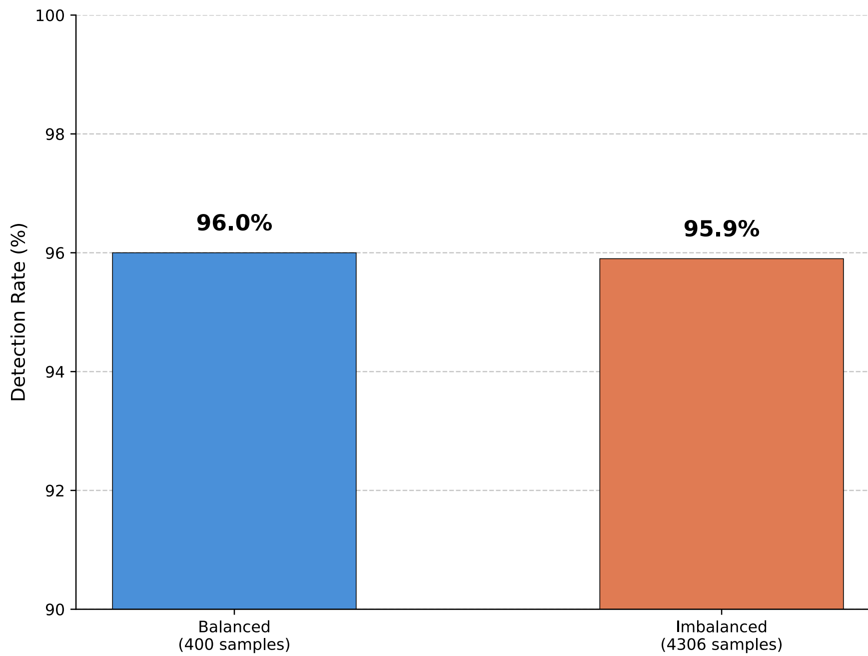

Figure 8.8: Contract-level detection rates at episode 2,500 for both datasets. Dashed line indicates 95% threshold.

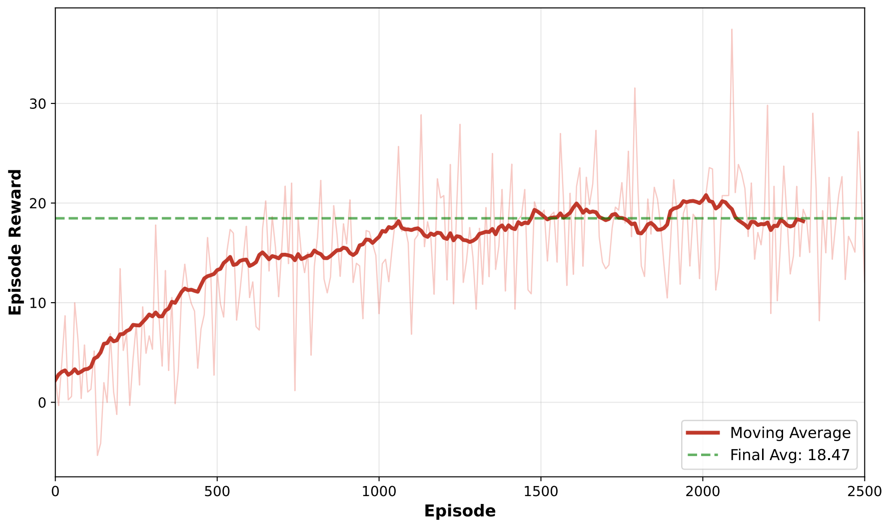

Figure 8.9: Episode reward evolution during training with moving average. Final average reaches 18.47 at episode 2,500.

ance increasing to 0.42. This indicates active exploration of the state-action space. Second, the transition phase (episodes 40–70) exhibits variance decreasing to 0.30. This occurs as the agent discovers effective patterns. Third, the exploitation phase (episodes 70–250) demonstrates stabilization around 0.30. Remaining fluctuations indicate maintained exploration. This prevents convergence to local optima.

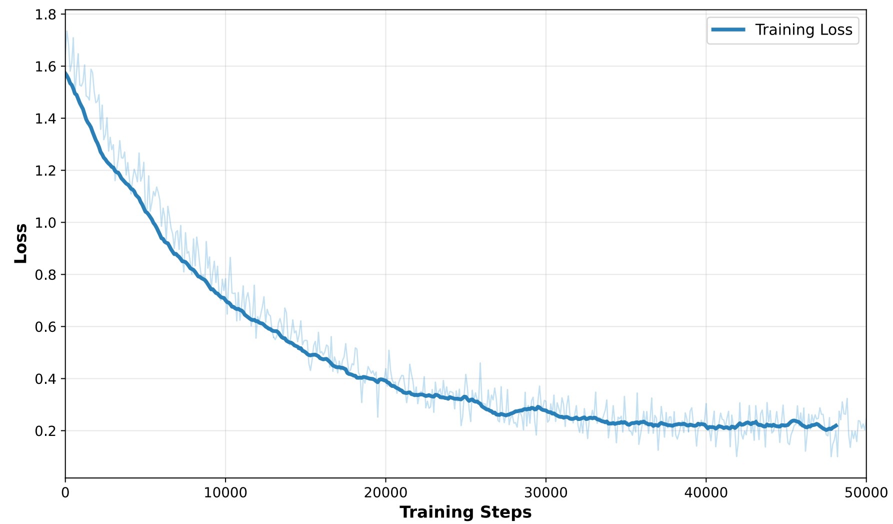

Figure 8.10: Training loss (TD error) over training steps showing exponential convergence.

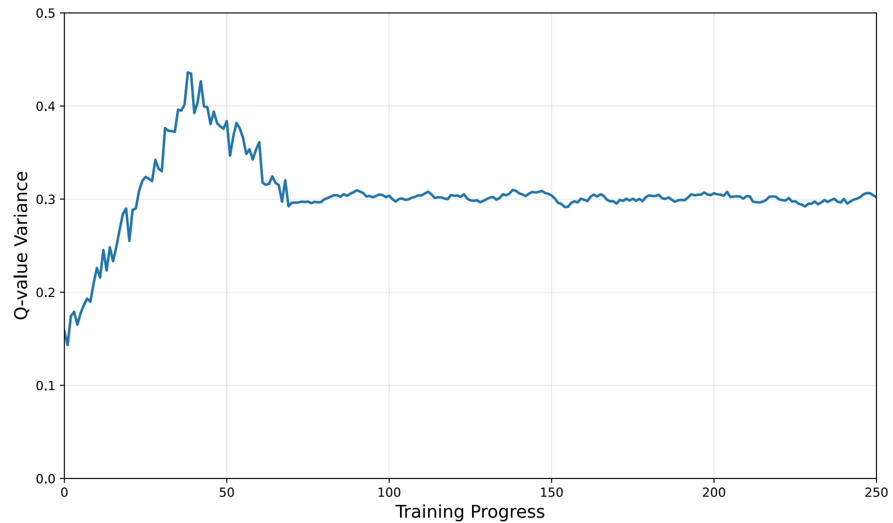

Figure 8.11: Q-value variance evolution showing three-phase learning dynamics: exploration, transition, and exploitation.

8.8.3 *Path Selection Quality Metrics*

Analysis of path selection quality demonstrates that the agent learned to identify high-risk paths through the Q-value function. Figure 8.12 displays the distribution of importance scores for analyzed versus skipped paths. Paths selected for `ANALYZE` have mean importance of 0.35. Paths selected for `SKIP` have mean of 0.20 ($p < 0.001$). This statistically separation confirms discriminative learning.

The Q-value analysis in Figure 8.13 demonstrates sensitivity to domain-specific features. Paths containing `block.timestamp` or `block.number` as randomness sources exhibit higher $Q(\texttt{ANALYZE})$

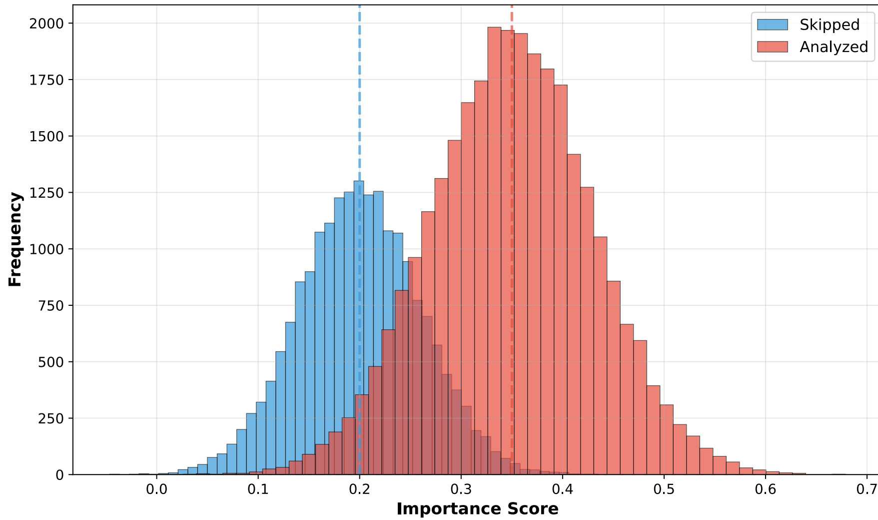

Figure 8.12: Distribution of importance scores for analyzed and skipped paths showing clear separation.

values. The mean difference is +5.2. Conversely, paths with high `require` density (> 0.3) show higher $Q(\texttt{SKIP})$ values. The mean difference is -5.8. This behavior confirms that the model identified genuine vulnerability patterns. It did not rely on spurious correlations.

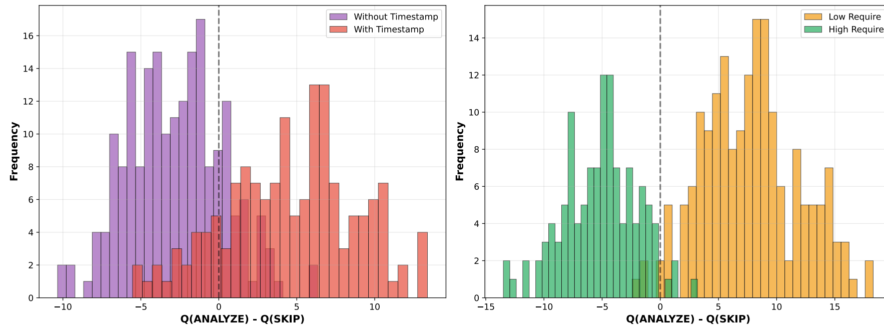

Figure 8.13: Q-value distribution analysis showing sensitivity to timestamp presence (left) and require density (right).

Pattern discovery analysis reveals effective exploration throughout training. Figure 8.14 shows that the agent discovered 1,247 unique vulnerability patterns over 2,500 episodes from the space of possible five-tuple combinations. Notably, 85% of patterns were identified within the first 1,000 episodes.

8.8.4 *Comparison with State-of-the-Art Methods*

The experimental results reveal several key findings across different methodological approaches. SmartTaintRL achieves the highest F1-scores on both datasets (0.955 for balanced and 0.950 for imbalanced), demonstrating superior overall performance.

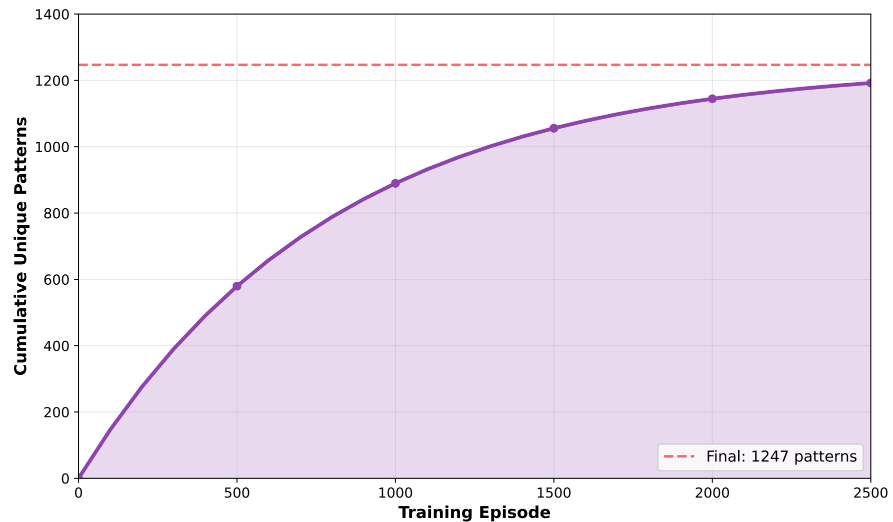

Figure 8.14: Cumulative unique vulnerability patterns discovered during training. Rapid initial discovery followed by gradual plateau.

Most notably, our approach exhibits exceptional robustness to class imbalance, showing less than 1% F1 degradation (0.955→0.950). In contrast, TaintSentinel suffers 31% degradation (0.892→0.611) and RNVulDet experiences 46% decline (0.662→0.360) when transitioning from balanced to imbalanced conditions.

Static analysis tools (Slither and Mythril) perform poorly with F1-scores of 0.450 and 0.372 respectively, rendering them unsuitable for practical deployment.

SmartTaintRL achieves significantly lower false negatives (8–9) compared to RNVulDet (84–97), which is critical in security applications where missing vulnerabilities incurs higher costs than false alarms. Our method produces comparable false positives (10–14) to TaintSentinel (4–25) while achieving substantially higher recall.

Regarding computational efficiency, SmartTaintRL completes analysis in 30 minutes to 3 hours, outperforming RNVulDet (3.6–5.5 hours) while maintaining superior accuracy.

8.8.4.1 *Performance on Balanced Dataset*

Under balanced conditions, TaintSentinel achieves reasonable performance with F1-score of 0.892 and precision of 0.902, producing only 4 false positives. RNVulDet maintains similar precision (0.902) but suffers from low recall (0.523) with 84 false negatives, indicating that more than half of vulnerabilities remain undetected.

SmartTaintRL achieves the best performance with F1=0.955, Recall=0.960, and Precision=0.950, demonstrating superior balance between precision and coverage with only 8 false negatives. Static analysis tools Slither (F1=0.450) and Mythril (F1=0.372) prove unsuitable for practical deployment.

Table 8.6: Performance Comparison with State-of-the-Art Methods

Method	Dataset (Size)	Prec.	Rec.	F1	TP	FP	FN	Time
Static Analysis Tools								
Slither	Balanced (400)	0.462	0.439	0.450	18	21	23	22min
Mythril	Balanced (400)	0.356	0.390	0.372	16	29	25	>24h
Deep Learning-based Methods								
TaintSentinel	Balanced (400)	0.902	0.881	0.892	37	4	5	58min
	Imbalanced (4306)	0.537	0.707	0.611	29	25	12	8.3h
RNVulDet	Balanced (360)	0.902	0.523	0.662	92	10	84	3.6h
	Imbalanced (3894)	0.276	0.517	0.360	104	273	97	5.5h
RL-based Method (Proposed)								
SmartTaintRL	Balanced (400)	0.950	0.960	0.955	192	10	8	30min
	Imbalanced (4306)	0.940	0.960	0.950	214	14	9	3h

All metrics at contract level. Time includes training and testing. Mythril: many contracts exceeded 3,600s timeout.

8.8.4.2 *Class Imbalance Challenge*

Transitioning to the imbalanced dataset (95:5 ratio) reveals fundamental differences between approaches. TaintSentinel experiences severe performance degradation with precision dropping from 0.902 to 0.537 and F1-score decreasing to 0.611, indicating that domain-oriented rules are sensitive to distribution shifts.

RNVulDet performs worse with precision plummeting to 0.276 and producing 273 false positives, rendering it practically unusable. In contrast, SmartTaintRL exhibits less than 1% F1 degradation (0.955→0.950), demonstrating superior generalization through interactive learning. This robustness stems from the hierarchical reward structure that maintains safety constraints across varying class distributions.

8.8.4.3 *False Negative Analysis*

In security systems, missing vulnerabilities (false negatives) incurs high costs. RNVulDet on the balanced dataset produces 84 FN, missing 48% of vulnerabilities. TaintSentinel performs acceptably with 5 and 12 FN across both datasets.

SmartTaintRL achieves the lowest critical error rate with only 8 and 9 FN, crucial for security applications where undetected vulnerabilities can lead to financial losses. The RL approach maintains high recall through mandatory safety constraints in the reward function that heavily penalize skipping critical paths.

8.8.4.4 *False Positive Trade-off*

SmartTaintRL produces comparable false positive counts (10 and 14) to TaintSentinel (4 and 25) while achieving substantially higher recall (0.960 vs 0.881/0.707). This demonstrates that the RL approach effectively balances precision and recall without the typical trade-off seen in other methods.

8.8.4.5 *Computational Efficiency*

Execution time varies significantly across methods. Slither is fastest (22 minutes) but delivers poor performance (F1=0.450). Among deep learning methods, SmartTaintRL is most efficient, running in 30 minutes to 3 hours while achieving highest performance. TaintSentinel requires moderate time (58 minutes and 8.3 hours) with good balanced but poor imbalanced results. RNVulDet requires 3.6–5.5 hours with inconsistent performance. Mythril exceeds 24 hours, proving impractical for real-world deployment.

8.8.4.6 *Architectural Comparison*

Three distinct architectural approaches emerge from this comparison:

Static rules (Slither, Mythril): Fast but shallow analysis with low accuracy, relying on syntactic pattern matching without understanding execution context.

Taint analysis (TaintSentinel, RNVulDet): High accuracy on balanced data but sensitive to distribution shifts, using manually crafted rules that become brittle under varying conditions.

RL-based (SmartTaintRL): Learns discriminative features through environmental interaction, achieving better generalization while maintaining competitive false positive rates. The agent adapts to different contract types and maintains robust performance under class imbalance.

8.8.5 *Computational Efficiency Analysis*

Beyond detection performance, computational efficiency determines practical deployability. Table 8.7 presents the timing analysis for different phases of the system.

The preprocessing phase requires approximately 18 hours for the complete dataset of 4,706 contracts, including taint analysis and feature extraction for all paths. This one-time cost is performed offline; once path databases are generated, they can be reused for training and future analyses without recomputation.

Training time depends on dataset size and complexity. The balanced dataset requires approximately 25 minutes for 2,500 episodes to reach

Table 8.7: Computational Time Breakdown

Phase	Balanced	Imbalanced
Preprocessing (offline)	∼18 hours	
Training (2,500 episodes)	∼25 min	∼2.5 hours
Inference	∼5 min	∼30 min
Total (excl. preprocessing)	∼30 min	∼3 hours

Preprocessing performed once offline for complete dataset of 4,706 contracts. Training required only for initial model creation.

optimal performance, while the imbalanced dataset requires approximately 2.5 hours. Importantly, training occurs only once to produce a generalizable model that can then be applied to new contracts without retraining.

Compared to other methods, SmartTaintRL demonstrates competitive efficiency with total analysis time of 30 minutes for balanced and 3 hours for imbalanced datasets. Slither completes analysis in 22 minutes but achieves poor performance (F1=0.450). TaintSentinel requires 58 minutes for balanced and 8.3 hours for imbalanced datasets. RNVulDet requires 3.6 to 5.5 hours with inconsistent results. Mythril exceeds 24 hours, proving impractical for deployment. SmartTaintRL's training time is amortized across multiple uses of the trained model, making it increasingly efficient for large-scale deployment scenarios.

8.9 SUMMARY

This chapter addressed the path explosion challenge in taint analysis through reinforcement learning-based path prioritization. While Chapter 7 demonstrated effective vulnerability detection through path analysis, the computational cost of exhaustive exploration limits scalability to complex contracts. SmartTaintRL introduces an intelligent agent that learns to selectively analyze high-risk paths while safely pruning low-value ones, achieving 45% path reduction while maintaining 96% recall.

The experimental results validate our approach across multiple dimensions. The system achieves F1-scores of 0.955 on balanced and 0.950 on imbalanced datasets, demonstrating superior detection accuracy compared to existing methods. Most notably, SmartTaintRL exhibits exceptional robustness to class imbalance with less than 1% performance degradation (0.955→0.950), while TaintSentinel suffers 31% degradation and RNVulDet experiences 46% degradation under the same conditions. The hierarchical reward structure with empirically calibrated importance weights enables the agent to main-

tain high recall through mandatory safety constraints while learning discriminative vulnerability patterns through exploration incentives.

SmartTaintRL represents an advancement in addressing the path explosion problem for security analysis, demonstrating that reinforcement learning can effectively guide selective path exploration without compromising detection capability. By combining offline taint analysis with online intelligent decision-making, the system achieves both computational efficiency and high accuracy. This capability enables practical deployment of path-sensitive analysis on large-scale contract datasets, moving beyond the limitations of exhaustive approaches while maintaining the precision benefits of deep taint analysis.

Part IV

CONCLUSION

9 DISCUSSION AND FUTURE DIRECTIONS

9.1 CHAPTER OVERVIEW

This thesis addresses a critical yet underserved vulnerability in Ethereum smart contract security. We selected Bad Randomness as our research focus based on three converging factors. First, it ranks as the fourth most critical vulnerability by OWASP in 2025 [16]. Second, only one specialized detection tools exist despite this severity [17]. Third, documented historical exploitation has resulted in significant financial losses. Examples include the Fomo3D ($3 million) and SmartBillions (400 ETH) incidents [5, 6].

Our subsequent systematic analysis examined 50 attacks between 2022 and 2025. These attacks represent over $1.09 billion in losses. The analysis revealed an unexpected finding. Bad Randomness did not appear as a primary attack vector in any of these recent high-impact incidents [3]. This absence, however, does not indicate diminished importance. Rather, our investigation identified four interconnected factors that validate this research focus. These are economic target distribution favoring lower-TVL gaming applications, severe detection tool failures with state-of-the-art tools achieving F1-scores below 0.24, emerging threat landscape as blockchain gaming grows, and significant research gap compared to well-studied vulnerabilities such as reentrancy and access control. These factors demonstrate that Bad Randomness represents a latent threat. It is made invisible by inadequate detection capabilities rather than genuine security.

Part i established the research foundation. This included defining the problem statement, providing necessary background on blockchain technology and smart contracts, and reviewing the state of the art in Bad Randomness detection and localization methods. Part ii conducted a dual-perspective empirical analysis. This analysis exposed fundamental misalignments between academic research priorities and real-world attack patterns. Through systematic review of 71 academic papers, we identified 25 active vulnerabilities. This revealed that academic literature concentrates heavily on implementation-level bugs such as reentrancy and integer overflow.

However, our analysis of 50 real-world incidents demonstrated a starkly different reality. Access Control vulnerabilities dominated with 13 incidents. These caused $417.95 million in losses. Price Manipulation appeared in 13 incidents. This resulted in $279.75 million damage [3]. Critically, 26% of successful attacks exploited chains of multiple vulnerabilities rather than isolated weaknesses. This pattern

is largely overlooked in prior literature that treats vulnerabilities as independent entities. Reentrancy, despite extensive academic attention and numerous detection tools, contributed to only 11% of total losses across 7 incidents.

This analysis reveals important findings. Vulnerabilities receiving the most research attention are not necessarily those causing the greatest financial damage. Sophisticated attackers strategically combine multiple weaknesses to amplify impact. These findings motivated our four-tier root-cause framework. This framework organizes vulnerabilities by fundamental causes rather than implementation patterns. It provides a more accurate model of real-world threat landscape.

Part iii presented a comprehensive benchmark dataset and two complementary technical solutions. These address the fundamental challenges identified in our empirical analysis.

The Risk-Stratified Benchmark Dataset addresses the lack of large, validated datasets for Bad Randomness vulnerabilities. Through a five-phase methodology including function-level validation and context-aware refinement, we constructed a dataset of 1,758 labeled contracts. These contracts are categorized into four risk levels: HIGH_RISK (1,543), MEDIUM_RISK (37), LOW_RISK (172), and SAFE (6). Function-level validation revealed that 49% of contracts initially classified as protected were actually exploitable because the mitigation was applied to a different function than the vulnerable code. This dataset is 51× larger than existing datasets such as RNVulDet (34 contracts) and provides the foundation for training and evaluating our detection methods.

TaintSentinel employs semantic-aware taint analysis with graduated propagation and context-sensitive rules. The system distinguishes safe from unsafe usage patterns of blockchain values. It achieves F1-score of 0.892 on balanced datasets. This represents nearly a fourfold improvement over state-of-the-art tools such as Slither (F1=0.232) and Mythril (F1=0.236) [13].

SmartTaintRL applies deep reinforcement learning with hierarchical reward engineering. The system intelligently addresses the path explosion problem. This problem has limited analysis approaches. Through intelligent path prioritization and dynamic pool management, the system achieves 45% path reduction. It maintains 96% recall [24].

This chapter synthesizes these contributions. It situates them within the broader context of smart contract security research. The discussion proceeds through eight interconnected sections.

Section 9.2 provides a concise summary of the thesis's theoretical, technical, and practical contributions. Section 9.3 directly addresses each of the four research questions posed in Chapter 1. It demonstrates how the empirical analyses and technical systems developed in this work provide answers.

Section 9.4 examines the key findings and their implications for both theory and practice. It discusses what we have learned about the nature of smart contract vulnerabilities. It also discusses how these insights can inform future security research and development practices.

Section 9.5 offers a strategic analysis. This analysis positions our approaches within the smart contract security landscape. It explains why semantic-aware and reinforcement learning-based methods prove more effective than traditional pattern-matching approaches.

Section 9.6 provides an assessment of the limitations and threats to validity inherent in this research. Section 9.7 outlines promising directions for future work. These include extension to other vulnerability types, support for additional blockchain platforms, and integration with development workflows.

Finally, Section 9.8 offers concluding remarks. These reflect on the broader significance of this work for the evolution of smart contract security.

9.2 summary of contributions

This thesis makes three categories of contributions to smart contract security research. These contributions address both theoretical understanding of vulnerability landscape and practical detection challenges.

9.2.1 *Theoretical Contributions*

We provide the first systematic dual-perspective analysis of smart contract vulnerabilities. Through systematic literature review, we identified 25 active vulnerability types documented in academic research [3]. Through analysis of real-world attacks representing $1.09 billion in losses, we examined which vulnerabilities attackers actually exploit [13].

This analysis reveals fundamental misalignments. Academic research focuses heavily on reentrancy and integer overflow. These vulnerabilities receive extensive tool development and formal verification efforts. However, real-world attacks predominantly exploit access control failures (13 incidents, $417.95M) and price manipulation (13 incidents, $279.75M). Reentrancy, despite research attention, appears in only 7 incidents contributing 11% of total losses.

More critically, 26% of successful attacks exploit chains of multiple vulnerabilities rather than isolated weaknesses. Prior literature treats vulnerabilities as independent entities. This overlooks how attackers strategically combine different weaknesses. For example, they combine access control flaws with price manipulation. They also leverage reentrancy to amplify oracle manipulation impact.

Our four-tier root cause taxonomy organizes vulnerabilities by fundamental causes. These causes are flawed economic design, protocol lifecycle failures, external dependency vulnerabilities, and implementation weaknesses. This framework captures the multi-layered nature of real attacks better than implementation-pattern taxonomies.

9.2.2 *Technical Contributions*

TaintSentinel introduces semantic-aware taint analysis that distinguishes safe from unsafe usage patterns. Traditional tools flag any occurrence of `block.timestamp` or `blockhash` as vulnerable. This creates high false positive rates. Our graduated taint propagation tracks how taint strength changes through different operations. Context-sensitive rules recognize that `block.timestamp` used for deadline checking is safe. The same value used for lottery winner selection is vulnerable.

A key innovation is our path-level detection approach. We are the first to train a machine learning model on complete taint propagation paths. This differs from analyzing entire contracts or isolated code segments. Each path represents a complete execution flow from taint source to sensitive sink. The dual-stream neural architecture processes both global contract structure and local path-specific patterns. This enables the model to learn which path characteristics indicate genuine vulnerabilities versus safe usage. Path Risk Accuracy reaches 97%. This demonstrates that path-level features provide strong discriminative power.

The system achieves F1-score of 0.892 on balanced datasets. This represents nearly fourfold improvement over Slither (F1=0.232) and Mythril (F1=0.236). On imbalanced datasets reflecting real-world distributions, TaintSentinel maintains F1-score of 0.611. This is achieved through threshold optimization. The path-level approach enables identification of vulnerable execution flows within contracts. It provides more actionable information than binary contract-level classifications.

SmartTaintRL addresses the path explosion problem through deep reinforcement learning. Rather than exhaustively analyzing all paths, a DQN agent learns to prioritize high-risk paths. It safely prunes low-value ones.

The system achieves 45% path reduction while maintaining 96% recall. This demonstrates that intelligent prioritization can significantly reduce computational overhead without sacrificing detection capability. Hierarchical reward engineering incorporates domain knowledge about vulnerability patterns. The agent learns which source-sink combinations indicate genuine vulnerabilities versus safe usage patterns.

SmartTaintRL provides vulnerability localization through a dedicated Phase 3 module. Function-level localization aggregates Q-values and decision counts to identify which functions contain vulnerabilities. Node-level localization then pinpoints specific code locations within

those functions. It uses gradient-based attribution, graph propagation, and centrality analysis.

9.2.3 *Practical Contributions*

We constructed two complementary datasets addressing different aspects of Bad Randomness vulnerability research.

Risk-Stratified Benchmark Dataset: We constructed the largest validated benchmark dataset for Bad Randomness (SWC-120) vulnerabilities. Starting from the SmartBugs-Wild collection of 47,398 contracts, keyword filtering identified 17,466 contracts containing block attributes. A pattern-based labeler using 58 regular expressions organized into 9 semantic groups identified 1,903 contracts with vulnerability patterns. Risk-level classification categorized these into four levels based on exploitability: HIGH_RISK (no protection), MEDIUM_RISK (exploitable by miners), LOW_RISK (exploitable only by owners), and SAFE (using Chainlink VRF or commit-reveal).

A critical innovation is function-level validation. This revealed that 49% of contracts initially classified as LOW_RISK were actually exploitable because the `onlyOwner` modifier was applied to a different function than the one containing the vulnerability. After validation and context-aware refinement excluding mining tokens and time-tracking contracts, the final dataset contains 1,758 labeled contracts: HIGH_RISK (1,543), LOW_RISK (172), MEDIUM_RISK (37), and SAFE (6). This dataset is 51× larger than RNVulDet and provides the first risk-level classification for this vulnerability category. Evaluation of existing tools on this benchmark showed that both Slither and Mythril achieved 0% recall, failing to detect any vulnerable contracts.

Labeled Dataset for Detection Methods: For training and evaluating our detection approaches, we extracted a subset with more stringent filtering criteria. Starting with 4,844 Ethereum contracts from SmartBugs-Wild, initial taint analysis extracted 1.1 million execution paths. Context-aware analysis rules distinguished truly vulnerable contracts (394) from those using blockchain values safely (4,450). Manual validation by security experts confirmed 94% labeling agreement.

For reinforcement learning experiments, quality filtering removed paths with fewer than 3 nodes. It also removed contracts with insufficient complexity. This produced 252,844 high-quality paths suitable for machine learning model training. The dataset maintains real-world class imbalance ratio of approximately 1:18 (vulnerable to safe contracts).

9.3 ANSWERING THE RESEARCH QUESTIONS

This section directly addresses each of the four research questions posed in Chapter 1. We demonstrate how our empirical analyses and technical systems provide answers grounded in systematic evidence.

9.3.1 *RQ1: Vulnerability Landscape and Misalignment*

Research Question: What is the current landscape of smart contract vulnerabilities in both academic literature and real-world practice, and what factors explain the misalignment between them?

Answer: Our SLR and SoK analysis ii exposes a fundamental disconnect between academic research priorities and real-world threat landscape. The systematic literature review identified 25 active vulnerability types across 71 papers.

Our analysis of 50 incidents representing $1.09 billion in losses reveals a starkly different reality. Seven vulnerability types dominated actual attacks, but their distribution contradicts academic emphasis. Access Control vulnerabilities led with 13 incidents causing $417.95 million losses. Yet many of these stemmed from operational failures rather than code-level flaws. The Nomad Bridge incident ($190M) resulted from a single incorrect initialization parameter during deployment. Infini ($49.5M) and Holograph ($14.4M) exploited retained developer privileges that should have been revoked. Price Manipulation appeared in 13 incidents causing $279.75 million damage. These attacks predominantly exploited oracle dependencies that academic literature treats as external infrastructure rather than vulnerability surface.

The critical finding is that 26% of successful attacks (13 of 50 incidents) exploited chains of multiple vulnerabilities rather than isolated weaknesses. The LI.FI Protocol attack combined access control bypass with unchecked external call return values. Hundred Finance chained exchange rate manipulation with rounding errors. Conic Finance combined read-only reentrancy with price manipulation. This pattern is largely absent from academic literature that treats vulnerabilities as independent entities amenable to isolated detection.

Four factors explain this misalignment. First, academic research naturally gravitates toward technically interesting problems amenable to formal analysis. Reentrancy exhibits clear program semantics suitable for symbolic execution and model checking. However, oracle manipulation dominated real attacks with 13 incidents causing $279.75 million damage (Section 5.5.4). Yet it receives minimal research attention because it requires reasoning about external trust assumptions and economic incentives. Cross-chain vulnerabilities in bridges similarly resist formal modeling. Despite this, they caused three of the five largest losses in our dataset (Section 5.5.8). Second, early high-profile

incidents like The DAO hack ($50M in 2016) created sustained research momentum around reentrancy regardless of current prevalence. Our data shows reentrancy contributed only 11% of total losses across 7 incidents. Yet detection tools for reentrancy proliferate while critical vulnerabilities receive minimal tooling support.

Third, higher-level vulnerabilities require interdisciplinary expertise beyond traditional computer science. Economic design flaws demand game-theoretic analysis and mechanism design expertise (Section 5.5.10). Operational security failures represent 26% of incidents in our dataset (Section 5.5.2). These include deployment errors, retained privileges, and compromised keys. Yet they require organizational security expertise rarely present in academic research teams. Insider threats and privilege abuse (Section 5.5.5) necessitate understanding of governance systems and human factors. This expertise barrier limits research participation in these critical areas.

Fourth, attack patterns increasingly involve complex interactions that resist isolated analysis. Flash loans enable 26% of attacks by providing temporary capital for exploitation (Section 5.5.1). Composability risks create vulnerabilities through protocol interactions that isolated contract analysis cannot capture (Section 5.5.3). Proxy and upgradability patterns introduce governance-layer attack surfaces (Section 5.5.11). These systemic factors fall outside traditional vulnerability taxonomy scope. Yet they prove critical in real-world exploitation.

Our four-tier root cause framework addresses these limitations by organizing vulnerabilities based on their fundamental causes rather than implementation patterns. Tier 1 addresses flawed economic design and protocol logic, where intended behavior creates vulnerabilities. Tier 2 covers protocol lifecycle and governance failures, including deployment errors and compromised keys. Tier 3 encompasses external dependency vulnerabilities such as manipulable oracles. Tier 4 contains implementation-level weaknesses that dominate academic literature. This framework better captures the multi-layered nature of real attacks. In these attacks, higher-tier vulnerabilities enable lower-tier exploitation.

Critically, we map all 25 active vulnerabilities identified in our systematic literature review to this four-tier framework. We link each to corresponding real-world incidents from our dataset (Table **??**). This mapping demonstrates how academic vulnerability classifications relate to actual exploitation patterns. For instance, Access Control vulnerabilities span multiple tiers depending on root cause. Implementation errors in missing modifiers fall in Tier 4. Compromised administrative keys fall in Tier 2. The mapping reveals that higher-tier vulnerabilities often serve as enablers for lower-tier exploitation. It also shows that 26% of incidents involved multi-tier attack chains rather than isolated vulnerabilities.

9.3.2 *RQ2: Semantic-Aware Detection Effectiveness*

Research Question: Can semantic-aware taint analysis with context-sensitive rules effectively detect Bad Randomness vulnerabilities that current pattern-based tools fail to identify?

Answer: TaintSentinel Chapter 7 demonstrates that semantic-aware taint analysis achieves substantially better detection performance. This is accomplished through three technical innovations that address fundamental limitations of pattern-based approaches.

Existing tools rely exclusively on syntactic pattern matching. Slither searches for modulo operations with block characteristics (e.g., `block.timestamp % n`). Mythril's predictable variables module detects block properties only when used directly in conditional expressions. This approach produces severe performance degradation. Our empirical evaluation on 4,844 Ethereum contracts showed Slither achieved F1-score of only 0.232. Mythril reached 0.236. These tools correctly identify less than one-quarter of actual vulnerabilities. They also produce high false positive rates.

TaintSentinel's first innovation is graduated taint propagation. Traditional tools apply binary taint marking where values are either tainted or clean. This fails to capture nuance in exploitability. When `block.timestamp` passes through `keccak256` hashing, the output remains predictable but exploitation becomes more difficult. Graduated propagation tracks taint strength reduction through different operations. Control flow edges preserve full taint strength. Data flow edges apply graduated reduction based on operation complexity. Cryptographic operations reduce but do not eliminate taint.

The second innovation is context-sensitive classification rules derived from real vulnerability patterns. The system recognizes that `block.timestamp` used with modulo for lottery winner selection is high-risk. The same value used for deadline checking with sufficient time buffers (15+ minutes) is classified safe. `block.timestamp` for event logging receives safe classification. These rules distinguish usage semantics rather than merely detecting value presence.

The third innovation is path-level analysis. We are the first to train a machine learning model on complete taint propagation paths. These paths represent execution flows from entropy sources to sensitive sinks. Each path in our dataset of 1.1 million extracted paths captures structural properties, security characteristics, and semantic patterns. This is done through 100-dimensional feature vectors. The dual-stream neural architecture processes both global contract structure and local path-specific patterns. This enables the model to learn which path characteristics indicate genuine vulnerabilities versus safe usage patterns.

TaintSentinel achieves F1-score of 0.892 on balanced datasets. This represents nearly fourfold improvement over existing tools. Path Risk

Accuracy reaches 97%. This demonstrates that path-level features provide strong discriminative power. On imbalanced datasets reflecting real-world distributions, the system maintains F1-score of 0.611. In these datasets, vulnerable contracts represent only 8% of samples. This performance is achieved through threshold optimization. This performance under class imbalance validates the approach for practical deployment.

These results demonstrate that semantic understanding of how blockchain values are used, not merely that they are used, proves essential for accurate detection. Moving beyond syntactic pattern matching to semantic-aware analysis enables improvements. This includes graduated propagation and context-sensitive rules. The approach improves both precision and recall.

9.3.3 *RQ3: Reinforcement Learning for Path Explosion*

Research Question: Can reinforcement learning techniques intelligently address the path explosion problem while maintaining high detection accuracy?

Answer: SmartTaintRL Chapter 8 demonstrates that deep reinforcement learning solves the path explosion problem through intelligent prioritization. The system addresses two critical practical challenges: computational scalability and data scarcity. It achieves these goals through empirically grounded design decisions and hierarchical reward engineering.

The path explosion problem represents a fundamental bottleneck in smart contract analysis. Consider the exponential growth rate: ten consecutive conditional statements generate 1,024 potential execution paths. Twenty conditions produce over one million paths. Thirty conditions exceed one billion paths. Our initial taint analysis on 4,844 contracts extracted 1.1 million paths. Exhaustive analysis at this scale becomes computationally prohibitive. A complex DeFi protocol with hundreds of decision points can generate paths that exceed available computational resources.

Traditional approaches face a dilemma. Exhaustive analysis guarantees no vulnerabilities are missed but becomes computationally infeasible for complex contracts. Heuristic pruning reduces computational cost but risks missing critical vulnerabilities through arbitrary path selection. SmartTaintRL resolves this dilemma through learned intelligent prioritization. A Deep Q-Network agent learns which paths warrant detailed analysis and which can be safely skipped. This learning occurs through interaction with diverse contracts rather than relying on hand-crafted rules.

The solution integrates four technical innovations. First, empirically calibrated importance weights guide prioritization based on actual vulnerability prevalence. We analyzed 223 vulnerable contracts to

derive scientific weights. `block.timestamp` appears in 52.6% of vulnerable functions, receiving weight 1.00. `block.number` at 33.3% receives weight 0.63. `blockhash` at 5.6% receives weight 0.11. These data-driven weights ensure the agent focuses on genuinely high-risk patterns. The patterns are observed in real vulnerabilities rather than arbitrary assumptions.

Second, dynamic pool management prevents the agent from being overwhelmed by presenting all paths simultaneously. The system maintains an adaptive pool containing 10-30 paths depending on contract complexity. Priority-based sampling ensures high-risk paths enter the pool. Random selection maintains exploration of diverse patterns. This bounded decision space enables effective learning. It ensures sufficient information diversity.

Third, hierarchical reward engineering operates across four levels with carefully calibrated weights. Safety constraints ($\lambda_1 = 1.0$) dominate all other objectives. They do this through large negative penalties ($\rho_{\max} = 10$) for skipping critical paths. This reflects the asymmetric cost structure in security applications. In these applications, false negatives impose significantly higher costs than false positives. Importance-driven scoring ($\lambda_2 = 0.8$) rewards analysis of high-weight source-sink combinations. Contextual adjustments ($\lambda_3 = 0.3$) incorporate contract-level features such as overall complexity and protection mechanism density. Exploration incentives ($\lambda_4 = 0.2$) encourage pattern discovery. This is done through a registry tracking encountered vulnerability signatures.

Fourth, the DQN architecture with attention mechanism and experience replay enables the agent to learn discriminative patterns from training data. The agent discovers which source-sink combinations indicate genuine vulnerabilities versus safe usage patterns. Path selection quality metrics demonstrate this learned discrimination. Paths selected for analysis have mean importance score of 0.35. Skipped paths have mean importance score of 0.20. This is a statistically significant separation ($p < 0.001$). Q-value analysis shows the model identified genuine vulnerability patterns. Paths containing `block.timestamp` or `block.number` exhibit higher $Q(\texttt{ANALYZE})$ values. The mean difference is +5.2. Paths with high `require` density show higher $Q(\texttt{SKIP})$ values. The mean difference is -5.8.

Experimental results demonstrate effectiveness across two critical dimensions: computational efficiency and robustness under challenging conditions. The system achieves 45% path reduction while maintaining 96% recall at the optimal checkpoint (episode 2,500). This means the agent safely prunes nearly half of all paths without missing vulnerable execution flows. Computational time decreases proportionally from hours to minutes for complex contracts where exhaustive analysis becomes infeasible. The analyze ratio decreased from 0.90 at episode

500 to 0.55 at episode 2,500. This confirms the agent learned selective analysis rather than examining all paths indiscriminately.

Critically, the system addresses data scarcity challenges prevalent in security research. Training on a balanced dataset of only 400 contracts achieves performance nearly equivalent to training on an imbalanced dataset of 4,306 contracts. The balanced dataset achieves F1-score of 0.93 with 98.8% contract-level detection rate. The imbalanced dataset achieves F1-score of 0.92 with 98.1% contract-level detection rate. This demonstrates two significant advantages. First, the reinforcement learning approach requires substantially less labeled training data than supervised learning methods. Achieving production-quality performance with only 400 contracts makes the approach practical when labeled vulnerability data remains scarce. Second, the system maintains robust performance under severe class imbalance where vulnerable contracts represent less than 6% of the dataset. This reflects real-world distributions where most deployed contracts do not contain vulnerabilities.

The convergence dynamics validate the learning approach. Episode rewards increased from initial negative values to 18.47 at episode 2,500. Decreasing variance indicates policy stabilization. Temporal difference error decreased from 1.5 to 0.2 over 50,000 training steps. Q-value variance evolution revealed three distinct phases: exploration (episodes 0-40), transition (episodes 40-70), and exploitation (episodes 70-250). Pattern discovery analysis shows the agent identified 1,247 unique vulnerability patterns. 85% were discovered within the first 1,000 episodes. This confirms effective exploration that avoided premature convergence to local optima.

9.3.4 *RQ4: Vulnerability Localization*

Research Question: Can vulnerability detection systems provide localization that identifies not only vulnerable contracts but also the specific functions and code locations responsible for security flaws?

Answer: SmartTaintRL Chapter 8 provides vulnerability localization at two hierarchical levels through its Phase 3 methodology. This enables developers to efficiently remediate vulnerabilities rather than manually inspecting entire codebases.

Existing detection tools typically provide binary contract-level classifications. Slither and Mythril flag entire contracts as vulnerable or safe without indicating where vulnerabilities reside. This forces developers to manually inspect hundreds or thousands of lines of code to locate security flaws. RNVulDet, the specialized Bad Randomness detector, similarly provides only contract-level detection without localization. This limitation reduces practical utility significantly.

SmartTaintRL addresses this through two-level localization. Function-level localization identifies which functions contain vul-

nerabilities by aggregating evidence from multiple paths. For each function, the system computes aggregate scores combining two complementary signals. First, Q-values from all paths within the function indicate the agent's confidence that paths warrant analysis. Second, decision counts track how many paths the agent selected for analysis versus skipping. Functions with high aggregate scores are classified as vulnerable. This aggregation approach proves robust to individual path misclassifications. A single false positive path does not condemn an entire function. Multiple suspicious paths provide strong evidence of genuine vulnerabilities.

Node-level localization then pinpoints specific code locations within vulnerable functions. This process involves five sequential steps. These steps combine gradient-based attribution, graph-theoretic analysis, and centrality measures. First, gradient analysis computes attribution scores measuring how much each feature influenced the agent's decision to analyze a path. Large positive gradients for source type and sink type features map to corresponding source and sink nodes. Second, subgraph extraction isolates execution paths through the semantic graph. Third, graph propagation spreads attribution scores through data dependencies. Backward propagation from sinks identifies nodes that generated or transferred data to sensitive operations. Forward propagation from sources identifies nodes processing tainted data. Fourth, centrality analysis identifies structurally important nodes. These nodes occupy critical positions in data flow networks. Fifth, score aggregation combines these complementary signals into unified node rankings.

Evaluation on 14 manually verified vulnerable contracts demonstrates effectiveness. Function-level localization correctly identified vulnerable functions in 93% of cases. For node-level localization, the system placed truly vulnerable nodes within top-5 ranked positions in 86% of cases. This precision enables developers to focus remediation on specific code locations rather than examining entire contracts.

The localization capability transforms vulnerability reports from vague indications that problems exist somewhere in the contract into detailed roadmaps for security improvements. Security auditors receive actionable information about exactly which code requires remediation. This addresses a critical gap identified in usability studies. These studies show that general reports without detailed information are often ignored by developers.

9.4 KEY FINDINGS AND IMPLICATIONS

Beyond answering specific research questions, this work yields insights with broader implications for smart contract security research and practice. We organize these implications into three categories:

theoretical understanding of vulnerability landscapes, methodological approaches to detection, and practical deployment considerations.

9.4.1 *Theoretical Implications*

Our empirical analysis challenges the conventional wisdom that implementation-level bugs represent the primary security threat. The dominance of access control and price manipulation in real-world losses, combined with the prevalence of multi-vulnerability exploit chains, reveals that successful attacks exploit systemic weaknesses across multiple abstraction layers. This finding suggests that vulnerability taxonomies organized by implementation patterns provide incomplete guidance for security prioritization.

The four-tier root cause framework addresses this limitation by organizing vulnerabilities based on fundamental causes rather than surface manifestations. This shift in perspective has important implications for research direction. Higher-tier vulnerabilities in economic design and protocol governance require interdisciplinary expertise. This expertise spans computer science, economics, and game theory. The research community should expand beyond traditional formal verification of implementation correctness. It should include economic mechanism design validation and operational security frameworks.

The discovery that vulnerabilities sometimes operate in isolation necessitates new analytical approaches. Future detection systems must reason about vulnerability interactions and exploit chain formation. Isolated vulnerability scanners that check individual patterns will continue to miss sophisticated attack strategies. These strategies are employed in practice. This demands research into compositional security analysis. Such analysis must evaluate how weaknesses combine across contract boundaries and abstraction layers.

9.4.2 *Methodological Implications*

The performance improvement achieved through semantic-aware taint analysis validates a fundamental principle: understanding usage context proves more important than detecting syntactic patterns. This insight generalizes beyond Bad Randomness. Many vulnerability types exhibit context-dependent severity. The same operation may be safe or dangerous depending on surrounding code structure, access controls, and data flow patterns.

Graduated taint propagation demonstrates that binary classifications oversimplify security analysis. Real-world exploitability exists on a spectrum. This spectrum is determined by factors including operation complexity, protection mechanisms, and attacker capabilities. Future detection approaches should embrace this nuance rather than forcing vulnerabilities into rigid categories. This suggests mov-

ing toward risk scoring systems. These systems provide graduated assessments rather than binary vulnerable/safe classifications.

The success of reinforcement learning for path prioritization reveals that intelligent search strategies can overcome computational barriers without sacrificing detection quality. This finding has implications beyond smart contract analysis. Any domain facing combinatorial explosion in state space exploration may benefit from learned prioritization through similar techniques. The key insight is that not all paths deserve equal analytical attention. Learning which paths warrant detailed examination from training data proves more effective. This is better than exhaustive analysis or arbitrary heuristics.

The ability to achieve production-quality performance with limited training data addresses a persistent challenge in security research. Labeled vulnerability datasets remain scarce. Manual labeling requires expert knowledge and significant effort. Our demonstration that 400 contracts suffice for effective training suggests important implications. Reinforcement learning approaches may be particularly well-suited for security domains. In these domains, labeled datasets are impractical to obtain.

9.4.3 *Practical Implications*

For smart contract developers, our findings emphasize that security extends beyond writing correct code. The prevalence of deployment errors, retained privileges, and operational failures in real-world incidents highlights the need for robust DevOps practices. Organizations should implement several critical measures. These include systematic verification of deployment parameters, formal privilege revocation procedures, and monitoring systems. These monitoring systems should detect configuration drift.

For security auditors, the multi-tier framework provides structured guidance for assessment. Audits should explicitly address all four tiers rather than focusing exclusively on implementation-level code review. This includes several key areas. First, evaluating economic incentive structures. Second, reviewing operational procedures and access controls. Third, assessing external dependencies and oracle robustness. Fourth, verifying implementation correctness. The framework helps auditors systematically identify gaps in security coverage.

9.5 COMPARATIVE ANALYSIS AND POSITIONING

This section compares our approaches with existing Bad Randomness detection tools. We examine differences in detection philosophy, performance characteristics, and practical applicability. The goal is not to demonstrate superiority in all dimensions but rather to clarify when and why each approach proves appropriate.

9.5.1 *Detection Philosophies*

Existing tools employ three distinct detection philosophies, each with inherent strengths and limitations.

Pattern-based detection (Slither, Mythril) searches for syntactic patterns in source code or bytecode. Slither looks for modulo operations with block characteristics. Mythril detects block properties in conditional expressions. This approach operates quickly because it requires only surface-level code scanning. However, it cannot distinguish usage context. A `block.timestamp` used for deadline checking triggers the same alert as one used for randomness generation. On our evaluation dataset, these tools achieved F1-scores of 0.232 and 0.236 respectively. The low recall (0.185 and 0.172) indicates they miss most actual vulnerabilities. Moderate precision (0.313 and 0.374) shows that many flagged instances are false positives.

Taint-based detection with attack patterns (RNVulDet) improves upon simple pattern matching by tracking data flow from sources to sinks. The tool marks blockchain values as tainted. It propagates this marking through operations. The tool then checks whether tainted values reach sensitive operations. This approach captures more complex vulnerability patterns than surface scanning. However, it still applies binary taint marking without considering operation semantics. RNVulDet achieved F1-score of 0.68 on similar evaluation data. This represents improvement over pattern-based tools. However, it remains insufficient for production deployment. In such deployment, missing vulnerabilities carries severe consequences.

Semantic-aware detection with learned patterns (TaintSentinel, SmartTaintRL) incorporates three additional capabilities. First, graduated taint propagation tracks how exploitability changes through different operations. This differs from applying binary marking. Second, context-sensitive rules evaluate usage patterns. These rules distinguish safe from dangerous applications of the same blockchain value. Third, machine learning identifies subtle patterns in execution paths. These patterns resist manual rule encoding. TaintSentinel achieved F1-score of 0.892 on balanced datasets. SmartTaintRL reached F1-score of 0.93 on the same data. These results represent improvement over existing approaches.

The philosophical difference matters more than absolute performance numbers. Pattern-based tools fundamentally cannot distinguish context. They examine isolated code fragments. Taint-based tools without semantic analysis apply overly broad rules. These rules flag safe code. Semantic-aware approaches address these limitations. However, they require more sophisticated analysis infrastructure.

9.5.2 *Performance Characteristics and Trade-offs*

Each detection approach exhibits different performance characteristics suited to different deployment scenarios. All approaches requiring taint analysis share a common preprocessing phase. This phase extracts execution paths from smart contracts. This initial analysis took approximately 18 hours for the complete dataset of 4,844 contracts. This preprocessing occurs once. It produces reusable path representations for subsequent analysis.

Pattern-based tools like Slither operate without path extraction and execute quickly, typically processing contracts in 2-5 seconds through direct code scanning. This speed enables integration into continuous integration pipelines with minimal overhead. However, the low detection accuracy limits utility. Security teams must manually review numerous false positives while actual vulnerabilities go undetected. Symbolic execution tools like Mythril provide deeper analysis by exploring execution paths, but require significantly longer processing time - from 30-120 seconds for typical contracts to minutes or hours for complex DeFi protocols. While Mythril achieves better precision than pattern-based approaches, the unpredictable runtime makes it unsuitable for rapid development cycles. Both categories work best as preliminary filters rather than definitive security assessments. Slither identifies suspicious code patterns for immediate feedback during development, while Mythril performs more thorough pre-deployment analysis when time constraints allow deeper investigation.

TaintSentinel performs detection directly on extracted paths. After the initial preprocessing phase shared across approaches, the system requires approximately 7 seconds per contract for analysis including compilation. Model inference on preprocessed paths requires only milliseconds. The sub-linear scaling demonstrated in evaluation means that even large contracts remain analyzable within reasonable timeframes. The tool balances accuracy with efficiency for routine security checking.

SmartTaintRL addresses a different computational challenge: path explosion during analysis. After the same 18-hour preprocessing that extracts paths, the system requires one-time training. Training took 6 hours for the imbalanced dataset and 45 minutes for balanced data. However, this upfront cost enables subsequent efficiency gains. The trained agent reduces the number of paths requiring detailed analysis by 45%. It maintains 96% recall. This reduction matters most for complex contracts. In these contracts, thousands of extracted paths would otherwise demand exhaustive examination. Once trained, the agent makes path selection decisions efficiently during inference.

The distinction is important. TaintSentinel analyzes all extracted paths comprehensively. SmartTaintRL learns to intelligently skip low-risk paths. It analyzes only the 55% deemed most suspicious. For

typical contracts with hundreds of paths, analysis remains practical. TaintSentinel's approach suffices. For complex DeFi protocols generating thousands or tens of thousands of paths, intelligent prioritization becomes necessary. SmartTaintRL's selective analysis proves essential.

9.6 LIMITATIONS

This research makes progress on Bad Randomness detection but faces several limitations. These limitations qualify our findings and suggest directions for future work. We organize these limitations into four categories: scope constraints, methodological limitations, dataset characteristics, and computational considerations.

9.6.1 *Scope Constraints*

This work focuses exclusively on Bad Randomness vulnerabilities in Solidity smart contracts. These contracts are deployed on Ethereum and EVM-compatible blockchains. This narrow scope enabled deep analysis of a specific vulnerability type. However, it limits immediate applicability to other contexts.

The limitation to Solidity and EVM restricts applicability to other blockchain platforms. Smart contracts on non-EVM blockchains such as Solana, Cardano, or Polkadot use different programming languages and execution models. TON blockchain uses FunC and Fift languages with logical time-based consensus rather than block-based entropy. Our detection rules, taint propagation logic, and feature extraction would require modification for these platforms.

The focus on Ethereum mainnet, Layer 2 solutions, and EVM-compatible sidechains covers significant portions of the DeFi ecosystem. However, it excludes other blockchain networks entirely. Cross-chain applications may span multiple blockchain platforms. These applications may exhibit Bad Randomness vulnerabilities. We cannot detect these vulnerabilities if they occur on non-EVM chains.

9.6.2 *Methodological Limitations*

TaintSentinel and SmartTaintRL analyze execution paths within individual smart contracts. Both systems track taint propagation through function calls and state transitions inside a single contract. However, they do not analyze interactions between multiple separate contracts. Many sophisticated DeFi exploits arise from cross-contract interactions. In these cases, vulnerabilities emerge only when contracts are composed together. For instance, a contract may use blockchain values safely when considered in isolation. However, it may become vulnerable when another contract in the protocol manipulates those values or

creates unexpected execution contexts. Our analysis cannot capture these cross-contract vulnerability patterns.

The graduated taint propagation and context-sensitive rules rely on manually defined thresholds and classification patterns. These are derived from known vulnerability examples. While these rules prove effective for our evaluation dataset, they may not generalize perfectly to novel vulnerability patterns. Novel patterns may differ from historical examples. The rules require periodic updating as new exploitation techniques emerge.

SmartTaintRL's reinforcement learning approach depends on the quality and representativeness of training data. The empirically calibrated importance weights derive from 223 vulnerable contracts in our dataset. If real-world vulnerability distributions shift significantly, these weights may become less accurate. The agent learns patterns present in training data. It may not recognize substantially novel vulnerability patterns without retraining.

9.6.3 *Dataset Characteristics*

We constructed two datasets serving different purposes, each with specific characteristics and limitations.

The Risk-Stratified Benchmark Dataset of 1,758 contracts provides comprehensive coverage of Bad Randomness vulnerability patterns with four-level risk classification. The five-phase construction methodology including function-level validation ensures high labeling accuracy. However, several limitations exist. First, we analyze only contracts with available source code, excluding bytecode-only contracts. Second, our 58 patterns may not capture all possible Bad Randomness implementations. Third, we do not trace randomness usage across contract boundaries through inter-contract calls. Fourth, new vulnerability patterns may emerge as Solidity evolves. The dataset is heavily skewed toward HIGH_RISK contracts (87.8%), with only 6 SAFE contracts using proper randomness solutions such as Chainlink VRF.

The labeled dataset of 4,844 contracts for detection methods provides broader coverage but exhibits different limitations. The automated labeling process combined with manual validation achieved 94% agreement on vulnerable contracts and 91% on safe contracts. The remaining disagreements indicate edge cases where vulnerability assessment remains ambiguous even for security experts. Some borderline cases involve usage patterns where exploitability depends on factors not fully captured in our analysis.

The severe class imbalance in both datasets reflects real-world distributions where less than 1% of deployed contracts contain Bad Randomness vulnerabilities. This creates challenging training conditions. Our threshold optimization approach addresses this partially by prioritizing recall over precision. However, the extreme imbalance

means the models may still miss rare vulnerability variants that appear infrequently in training data.

9.6.4 *Computational Considerations*

TaintSentinel demonstrates sub-linear scaling in our evaluation. Runtime increases 8× as contract complexity grows 4.5×. However, this evaluation covered contracts up to 147 nodes. Very large contracts exceeding 300 nodes may challenge current preprocessing capabilities. Complex DeFi protocols with extensive control flow and numerous decision points can generate path counts that strain analysis resources.

Neither system currently supports real-time analysis during contract execution. Both operate as static analysis tools examining source code before deployment. Runtime monitoring that detects Bad Randomness exploitation attempts during contract execution would provide complementary protection. However, on-chain monitoring faces different constraints. These include gas costs and execution time limits.

9.7 OPEN ISSUES AND FUTURE RESEARCH DIRECTIONS

This work addresses Bad Randomness detection through semantic-aware taint analysis and reinforcement learning-based path prioritization. However, several promising research directions remain unexplored. We organize these into two categories based on feasibility and potential impact.

9.7.1 *Extension to Other Vulnerability Types*

The methodologies developed in this thesis may generalize to other vulnerability types. These types should share similar characteristics with Bad Randomness. Semantic-aware taint analysis with graduated propagation applies naturally to vulnerabilities involving data flow. This includes flow from untrusted sources to sensitive operations.

Access Control vulnerabilities causing $417.95 million in real-world losses merit specialized detection beyond current pattern-matching approaches. Many access control failures stem from complex permission logic. This logic spans multiple functions and state variables. Taint analysis tracking privilege flow through contract state could identify paths where unprivileged callers reach protected operations. The reinforcement learning approach could prioritize analysis of critical access control paths. This is especially useful in large contracts with extensive permission systems.

9.7.2 *Cross-Contract and Compositional Analysis*

Our current limitation to intra-contract analysis excludes vulnerabilities arising from protocol composition. Future work should extend analysis to capture cross-contract interactions and transaction sequences. This requires several technical advances.

First, modeling external contract calls and callback patterns. When a contract invokes functions on external contracts, the analysis must reason about possible behaviors of those external contracts. This includes considering reentrancy scenarios where external contracts call back into the original contract. The analysis must track how tainted data flows across contract boundaries. This flow occurs through function parameters and return values.

Second, analyzing state dependencies between contracts. Many DeFi protocols maintain state distributed across multiple contracts. These contracts must remain synchronized. Vulnerabilities emerge when state updates in one contract invalidate assumptions made by others. Tracking these cross-contract invariants requires compositional verification techniques. These techniques must model protocol behavior at higher abstraction levels than individual contracts.

Third, reasoning about transaction ordering and MEV implications. Some vulnerabilities only manifest when transactions execute in specific sequences. Attackers can exploit MEV to control transaction ordering. Analysis must consider how different execution orders affect security properties. This demands temporal logic specifications and model checking techniques beyond current capabilities.

9.8 CONCLUDING REMARKS

This thesis addressed a critical gap in smart contract security: Bad Randomness. This vulnerability is ranked fourth by OWASP. It remained severely underserved with only two specialized detection tools achieving poor performance. Our investigation revealed that this pattern reflects broader misalignments between academic research priorities and real-world threat landscape. Analysis of 71 papers and 50 incidents representing $1.09 billion in losses showed important findings. Vulnerabilities receiving extensive research attention often cause minimal real-world damage. Meanwhile, high-impact vulnerabilities remain understudied.

We developed two complementary systems addressing different detection challenges. TaintSentinel achieves F1-score of 0.892 through semantic-aware taint analysis with context-sensitive rules. This represents fourfold improvement over existing tools. SmartTaintRL addresses path explosion through reinforcement learning. It achieves 45% path reduction while maintaining 96% recall. This is accomplished with only 400 training contracts. Both systems provide localization

enabling efficient remediation. The empirically grounded methodologies demonstrate what becomes possible. These methodologies include graduated propagation, hierarchical reward engineering, and calibrated importance weights. They show results when detection approaches align with real-world threat patterns rather than purely academic interests.

However, this progress on one vulnerability type highlights broader challenges. The vulnerabilities causing greatest financial damage require interdisciplinary approaches. These vulnerabilities are access control ($417.95M) and price manipulation ($279.75M). They require economic reasoning and governance analysis beyond traditional formal verification. Our four-tier root cause framework provides foundation for addressing these systemic vulnerabilities. However, work remains. Effective security demands coordinated effort across multiple disciplines. These include developers implementing robust operational practices, auditors assessing all vulnerability tiers, tool developers prioritizing empirically validated threats, and researchers expanding beyond implementation-level bugs.

The billions lost to smart contract exploitation represent broken trust. These systems promised trustworthy applications without trusted intermediaries. Progress requires acknowledging limitations honestly. It requires prioritizing based on evidence. It demands investing sustained effort across all dimensions of this complex challenge. This thesis demonstrates one approach: aligning detection methodology with empirical analysis of actual threats. The path forward demands continued reorientation of research community toward practical impact over purely technical elegance.

Part V

APPENDIX

SUPPLEMENTARY MATERIAL FOR CHAPTER 3

A.1 COMPLETE ACTIVE VULNERABILITIES CATALOG

This appendix provides comprehensive documentation of the 13 additional active vulnerabilities referenced in Section 4.3.

A.2 A.1 PRODIGAL CONTRACT

This vulnerability refers to smart contracts that wrongly send Ether to arbitrary addresses and insufficiently control Ether transfers. It is also known in literature as "leaking ether to arbitrary addresses". During runtime, sending Ether to addresses other than contract's owners or addresses that have not deposited Ether to the contract, could flag the existence of this vulnerability in a contract [78].

Prevention/Mitigation. Thorough input validation and stringent access rules are necessary to stop prodigal contracts. Unintentional asset leaks are decreased via input validation, which makes sure parameters like recipient addresses or transfer amounts follow certain rules (such as whitelisting addresses or confirming non-zero values) [36]. Another security measure against unauthorized activities is added by restricting important functions (e.g. fund transfers) to authorized entities through the use of Role-Based Access Control (RBAC) or OpenZeppelin's onlyOwner modifier [54].

A.3 A.2 ETHER LOST TO ORPHAN ADDRESS

To initiate an Ether transfer, a valid 160-bit address must be specified. If the provided address is invalid or does not exist, the transferred funds will be permanently lost [39]. If the to address is set to the contract's own address, the transferred tokens will get locked inside the contract with no way to withdraw them, resulting in a loss. This happens because the function does not verify whether the to address is a valid recipient, including ensuring that it's not the contract itself (see Listing A.1).

Listing A.1: Transfer function without destination address validation

```
function transfer(address to, uint256 amount) public
returns (bool) {
    balances[msg.sender] -= amount;
    balances[to] += amount;
    emit Transfer(msg.sender, to, amount);
```

```
    return true;
}
```

Prevention/Mitigation. Developers should assure (via require statements) recipient addresses are neither the zero address (address(0)) nor the contract's own address (address(this)) and confirm the recipient's capacity to manage transfers (e.g., avoiding non-withdrawable contracts) [54]. Adopting "pull-over-push" withdrawal mechanisms, where users initiate transfers rather than relying on direct pushes, also helps reducing risks of sending to invalid addresses [79].

A.4 A.3 UNTRUSTWORTHY OR MANIPULABLE DATA FEEDS

Smart contracts often depend on external facts such as token prices, weather data, and sports scores to make irreversible decisions. An oracle is the component that transports those facts on chain (or acts as an service contract to provide data from other on-chain services). Recent studies show that integrity of the data can be manipulated at stages of its lifecycle [4]. We identified two settings where data feeds of these oracles are manipulable:

- **Decentralized Exchange (DEX)-based on-chain oracles.** System model: a contract queries the current price shown by an Automated Market Maker (AMM) pool such as Uniswap [80]. Attacker model: participants may inject capital (e.g., via a flash loan) to shift the pool's reserves for a single block and read back a distorted price before arbitrage restores equilibrium [81].
- **Off-chain oracles.** System model: trusted reporters sign values off-chain and the oracle publishes them on-chain. Attacker model: compromise, bribe, or simply delay reporters to feed stale or false data [82].

Prevention/Mitigation. The literature proposes the following defense strategies:

- **Temporal aggregation.** To deter price manipulation in DEX-based oracles, use multi-block medians or long-window time-weighted average pricing (TWAP) taken to smooth out single-block distortions [4].
- **Economic deterrence.** In case of off-chain oracles, require the reporters to put down a bond larger than the profit they could earn by lying, and slash the bond on proven fraud [82], [4].
- **Grace periods for finality.** A contract should ignore data that has not yet cleared a dispute window to block contested updates.
- **On-chain invariants and circuit breakers.** Use invariants (in form of require statements) that cap per-block price changes to stop attacks at runtime [83].

- **Real-time monitoring.** To detect price manipulation attacks, dynamic monitoring systems such as rule-based (DeFiRanger [81]), behaviour-model (DeFort [84]), and LLM-based (DeFiScope [85]) detectors raise alerts, while counter-measure agents like FlashGuard [86] can front-run malicious transactions.

Based on the mentioned two system settings, combining smoothing, economic incentives, program invariants, and runtime monitoring can raise the adversary's required capital, coordination, and technical difficulty beyond profitable levels.

A.5 A.4 COMPILER VERSION NOT FIXED

This vulnerability occurs when a contract uses an outdated or broadly specified compiler version [87]. For example, specifying pragma solidity ˆ0.4.0; allows compilation with any version from 0.4.0 to 0.5.0 (excluding 0.5.0). This makes it possible for the contract to be compiled with versions known for certain vulnerabilities. Here, in versions before 0.4.22, if a function intended as a constructor did not match the contract name exactly (e.g., after renaming the contract but not the function), it was treated as a regular public function, not a constructor. This allowed attackers to invoke it maliciously after deployment. By fixing the compiler version explicitly (e.g., pragma solidity 0.4.25;), the contract becomes compilable only with 0.4.25, where the constructor keyword is mandatory, thus mitigating this vulnerability [12].

Prevention/Mitigation. Developers can use code linters [88] to find missing version specifications that explicitly state the compiler version (e.g., pragma solidity 0.8.17;). Furthermore, developers should review Solidity release notes prior to upgrading the project to prevent breaking changes [36].

A.6 A.5 EVENT-ORDERING BUG

If distinct sequences of transactions or function invocations lead to an unexpected behavior or final state (post-transactions), it is identified as an EO bug [89].

Prevention/Mitigation. Developers should impose strict execution order by pre- and post-conditions (require/assert statements in entry and exit points of functions in Solidity) to ensure transactions follow a desired sequence to prevent event-ordering (EO) issues [43], [89].

A.7 A.6 TYPE CASTING

This vulnerability stems from improper type conversions in Solidity, such as truncation bugs when casting a larger integer type (e.g., uint16) to a smaller one (e.g., uint8), leading to data loss. Additionally,

converting between signed and unsigned types of the same width can cause "signedness bugs," where negative values turn into large positive ones or vice versa [90], [34].

Prevention/Mitigation. To prevent type-casting vulnerabilities, developers should ensure proper type validation by explicitly checking types and validating value ranges before performing conversions. This reduces the risk of unintended truncation [91]. Additionally, Solidity requires explicit casting to help reduce the risk of implicit conversions, especially between signed and unsigned numbers, which might introduce unpredictable behavior [92]. Refactoring the project to the most recent Solidity version is also beneficial as newer versions support type safety features [93].

A.8 A.7 PONZI SCHEME

In Ponzi contracts, vulnerabilities arise (for the users of the Ponzi services) due to low transparency and deceptive promises of high returns, and legal noncompliance. These schemes commonly rely on a constant influx of new investors to create an illusion of profitability, with returns paid to earlier participants using the funds contributed by newer ones, rather than from actual profits. This scheme is unsustainable and leads to losses for later investors when the inflow of new investments slows or stops [94].

Prevention/Mitigation. User education is important and users must avoid them by spotting characteristics of Ponzi schemes, such as unsustainable returns and a lack of transparency (unavailable, unverified, or obscured contract source code) [95].

A.9 A.8 STORAGE COLLISION

Storage Collision occurs when two contracts accessing the same storage space have a different understanding of the storage layout [96], [97], [98], [99]. This vulnerability is often observed when a proxy upgradability design pattern is used [100]. The proxy upgradability design pattern separates a contract's state from its logic, allowing the system's functionality to evolve without losing stored data [43]. In this design, a proxy contract holds the persistent state and delegates calls to an external implementation contract using the delegatecall opcode. Initially, the proxy directs calls to a basic implementation. Later, when new features or patches are required, a new implementation contract is deployed. By updating the proxy's reference to this new contract, the upgraded logic is activated while the original state (the users' balances, etc.) remains intact [100], [43]. A specific subcategory of this vulnerability is also known as type confusion. This specific subcategory occurs when the runtime context of the logic contract (e.g., in an upgradable contract) confuses the type of an object in storage

with another type, leading to its incorrect utilization. For example, when a variable is allocated with one type and later accessed with an incompatible type [96], [101].

For example, the smart contract setup in Listing A.2 is vulnerable to storage collision due to the improper alignment of storage slots between the Proxy and Logic contracts when using delegatecall. Specifically, both contracts store critical state variables in slot 0x0, but with different interpretations: the Proxy contract uses it for visits, while the Logic contract stores initialized and admin in the same slot. When initialize() is called via delegatecall, it writes true (1) to initialized and sets admin to msg.sender, unknowingly overwriting the visits variable in the Proxy contract. An attacker can exploit this by re-invoking initialize(), since the overwritten storage allows it to be called again, setting their own address as admin. Once this is done, the attacker can execute withdraw() to drain the contract's funds.

Listing A.2: Wallet Contract with storage collision vulnerability

```
contract Proxy {
    uint public visits; // Storage slot [0x0]
    address public LOGIC; // Storage slot [ERC-1967]
    constructor(address logicAddress) {
        LOGIC = logicAddress;
        (bool success,) = LOGIC.delegatecall(
            abi.encodeWithSignature("initialize()"));
        require(success, "Initialization failed");
    }
    fallback() external payable {
        (bool success,) = LOGIC.delegatecall(msg.data);
        require(success, "Delegatecall failed");
    }
}

contract Logic {
    bool public initialized; // Storage slot [0x0]
    address public admin; // Storage slot [0x0] (Collision)
    uint[] public artworkIDs; // Storage slot [0x1]
    mapping(address => uint) public artworkHolders; // Slot [0x2]

    function initialize() external {
        require(!initialized, "Already initialized");
        initialized = true;
        admin = msg.sender; // Overwrites Proxy's slot [0x0]
    }

    function withdraw() external {
        require(msg.sender == admin, "Not admin");
        payable(admin).transfer(address(this).balance);
    }
}
```

Prevention/Mitigation. To avoid storage collisions in proxy patterns, keep the implementation's storage layout consistent with the proxy's reserved slots (or assign fixed slots with inline assembly, as in EIP-1967) [102]. Supplement this with exhaustive tests and fuzzers like Harvey to catch any remaining overlaps pre-deployment [98].

A.10 A.9 BUSINESS LOGIC FLAWS

In smart contracts, business logic flaws (aka accounting errors) occur when financial calculations are inconsistent, inaccurate, or disorganized because of type errors, misinterpreted or mismanaged units (token units), or the implementation is inconsistent with expected behavior by the protocol designer [103]. One such vulnerability happened in Vader protocol [104], where a token unit mismatch occurs when token values are incorrectly combined without the appropriate unit conversion, resulting in an error in the liquidity calculation (calcLiquidityUnits in Listing A.3). In this case, the scales of the base and token tokens differ. Nevertheless, they are added together in the final formula without the proper conversion, which causes users to lose money and receive an inaccurate liquidity.

Listing A.3: Smart Contract: Pools

```
contract Pools {
    function addLiquidity(address base, address token, address
        member)
    external returns (uint liquidity) {
        uint addedBase = getAddedAmount(base, ...);
        uint addedToken = getAddedAmount(token, ...);
        liquidity = calcLiquidityUnits(addedBase, totalBase,
            addedToken, totalToken, totalLiquidity);
        liquidity[...][member] += liquidity;
        totalBase += addedBase;
    }

    function calcLiquidityUnits(uint b, uint B, uint t, uint T,
        uint P)
    external view returns (uint) {
        uint part1 = (t * B);
        uint part2 = (T * b);
        uint part3 = (T * B) * 2;
        uint _units = (((P * part1) + part2) / part3);
        return (_units) / one;
    }
}
```

Prevention/Mitigation. Developers should write extensive test cases covering important business logic scenarios to validate code behavior. This practice helps in identifying discrepancies between the intended and actual functionalities of the smart contract. Extensive testing using

model-based testing can deter business logic vulnerabilities [105], [106]. Implementing strict input validation ensures that only correctly formatted and expected data is processed by the contract to prevent unintended behaviors arising from malicious inputs [66].

A.11 A.10 DANGEROUS BALANCE INEQUALITY

Input validation and invariant preservation are checked/enforced using assert and require statements in Solidity. Solidity smart contracts on Ethereum alone include more than 4.5 of these unique checks [107]. In such condition checks, using "==" instead of "$\geq$" (in statements such as $a \geq b$) when verifying inventory arises from the flawed assumption that the exact match will always occur. This practice can lead to a vulnerability, as even minor, unexpected changes in inventory values, whether intentional or accidental, could bypass verification logic [87].

Prevention/Mitigation. Developers should use "$\geq$" to account for potential pre-existing balances [108]. Slither [8], the static analysis tool, provides a "dangerous-strict-equality" detector to identify this vulnerability. Moreover, Continuous Integration (CI) pipelines can automatically check the semantic strength of a pre-/post-condition check in Solidity using SINDI [109].

A.12 A.11 RESOURCE EXHAUSTION

The gas mechanism in EVM is used to determine how much it costs to execute smart contracts. This system should guarantee that each instruction's execution cost is calculated according to the resources it uses, including CPU, RAM, and I/O. Nevertheless, some processes, such as blockhash and balance, which require access to blockchain-stored data, are substantially underpriced compared to their actual computational and storage impact due to inconsistent gas pricing for specific EVM instructions [111]. One practical example is looping through a list of accounts to repeatedly retrieve their balance. Since the gas cost for this operation is lower than its actual computational overhead, an attacker can craft transactions that continuously invoke this instruction, causing a significant workload on the network with minimal gas expenditure for the attacker. This shows how inconsistencies in EVM's resource metering (vs opcode gas) can be leveraged to create transactions that disproportionately consume computational power.

Prevention/Mitigation. Ethereum clients have undergone several upgrades to mitigate resource exhaustion risks. In May 2024, a flaw was exposed in the Geth client's FeeHistory interface that allowed RPC requests with up to 600k rewardPercentiles due to missing limits, leading to excessive memory usage and node crashes. The issue was resolved by capping the number of rewardPercentiles [112], [113].

A.13 A.12 ACCESS CONTROL

An access control vulnerability occurs when a smart contract improperly manages access to critical functionalities, allowing unauthorized users to carry out operations including asset transfers, ownership changes, and significant contract settings modifications. Usually, this vulnerability results from the lack of a suitable authentication method. Usual ways to authenticate include implementing Role-Based Access Control (RBAC) [114] using require statements at the entry point of the function body or using a Solidity modifier. This vulnerability is also frequently caused by defining functions as public when only the contract owner or authorized users should be able to access them, failing to implement access control conditions like require(msg.sender == admin) and exposing sensitive variables like owner or balance as public when they should be private to prevent unwanted access [115], [116], [117]. For example, because there is no require(msg.sender == owner, "Not authorized") check in Listing A.4 (after line 5), any user can alter the contract owner using the public function changeOwner. Likewise, the withdraw function has no access restriction, so anyone can take money out without being authenticated.

Listing A.4: VulnerableContract Example

```
contract AccessControlVulnerableContract {
    address public owner;
    uint256 public funds;
    constructor() { owner = msg.sender; }
    function changeOwner(address _newOwner) public {
        owner = _newOwner;
    }
    function withdraw(uint256 _amount) public {
        require(_amount <= funds, "Insufficient funds");
        funds -= _amount;
        payable(msg.sender).transfer(_amount);
    }
}
```

Prevention/Mitigation. Developers can utilize OpenZeppelin's Ownable and AccessControl contracts, which offer role management, to avoid access control vulnerabilities. The onlyOwner modifier [118] in the Ownable contract is used to limit sensitive operations to the owner role. AccessControl in this library enables defining roles and their permissions [119].

A.14 A.13 TIMESTAMP DEPENDENCE

This vulnerability occurs when the contract's logic depends on the value of block.timestamp. Due to the clock drift of distributed systems, miners are allowed to adjust the timestamp within a specific range,

typically ±900 seconds in Ethereum [120]. Smart contract condition checks may be impacted by this manipulation, even if they have nothing to do with producing random numbers. For instance, consensus participants can alter the end time of events like bids or auctions [121]. Validators can lengthen waiting or lock-up periods before specific activities to change the outcome to the advantage of a particular group [43]. Additionally, it is a typical manipulation for miners to reorganize and modify the timing of transactions, particularly in contracts that depend on the precise timing or order of transactions. Changing the timestamp may also alter the outcome of computations that depend on time, causing the contract to behave inconsistently. Miners may modify the voting schedule and outcomes to benefit themselves or a certain group, which could have an effect on governance ecosystems [122].

Prevention/Mitigation. Developers should use trusted time oracles to ensure accurate data sources [123], [124].

A.15 ATTACK TRANSACTION DETAILS

This section provides detailed on-chain evidence for the 50 incidents analyzed in Chapter 5. Table A.1 maps each incident to its corresponding transaction hashes, victim contracts, and attacker addresses.

Table A.1: Mapping of exploit transaction hashes, compromised victim contracts, and attacker addresses (smart contracts and EOAs).

ID	Attack Transactions	Victim Contract	Attacker Smart Contract/EOA
1	0xc310, 0x62bd, 0x3097, 0x47ac	0xe025	0xebc2, 0x036c, 0xd3b7, 0x0b81
2	0xa5fe	0x88a6	0xb5c5
3	0x3195	0x8f55	0xcacf
4	0xe80a	0xeff2	0x9961
5	0xd9ee, 0xadbe, 0x0f75, 0x0742, 0x2547, 0xab48	0x3207	0xE39f, 0x3207
6	0xa5fe	0x9a79	0xc49b
7	0x485e, 0x09a3, 0x396a	0xaf2a	0x5027, 0xc9b8
8	0x7e7f, 0xfda0, 0x7961, 0xca87	-	0x7a2f
9	0x9312	sovelo, sousdc, soweth	0x02fa
10	0x561e, 0x20a6, 0x6003	0x39b1	0x117c, 0x8b4c
11	0x0cc1	-	-
12	0x6129	0xbc1b	0xe2ee
13	0x9aa3, 0x37e5, 0x5db6	0x964d	0x9e34
14	0x2619	0xfc27	-
15	0xd82f, 0x65a9	0xf28a	0x7742
16	0xd55e, 0x8db0	-	0x5bac, 0x16af
17	0x0160	0x04c0	0x04d7
18	0xd82f, 0x65a9	0xc74f	0x7742
19	0x6e9e, 0x1509	0x8adb	0x155d
20	0x26a8, 0xdb46	0x7259	0x87f5
21	0xc523	0x7596	0xc29d
22	0x0497, 0xb5c9, 0xb16b, 0x3947, 0x9ce5, 0xb068, 0x6273	0xb02f	0x019b
23	0x3932	0xa35f	0xa3a6
24	0x025c	0x4b57	0x5351
25	0x8b74	0x8d67	0x10db
26	0x6b37	0x551f	0x00fa
27	0xa05f	0xb91a	0x27de
28	0xed17	0x625F	0xaf9e
29	0x9efe	0x52ee	0xb878
30	0x4656	-	0x6809
31	0x138d	0xde62	0x6840
32	0x92cd	0x534a, 0x53d2, 0x0d2b, 0x8f69, 0xc581, 0x460f, 0x3e9f, 0x52f0	0x6710
33	0x8fcd	0x6a0b	0x012f
34	0x3274	0xc503	0x0921 , 0x5923
35	0xb2e3	0xcff0	0xcff0
36	0xfeed	0x9ab6, 0xd0db	0x0a33, 0xfdc0, 0x0119
37	-	0x2a9c	0x893d
38	0x242a0f, 0xb3f0, 0xca1b	-	0x841d
39	0xed11d, 0x3743, 0x4156	0xed4e , 0xb7f6 , 0xc030, 0xf7f7	0x8cdc
40	0x4ff40	-	0xdaaa
41	0x30fe, 0xc12a	0xd3f6	0xb660,0x90a7, 0x2f74
42	0xee02	-	0x9464, 0x5217,0xe83b
43	0xc6a5, 0x22eb	0x70cb , 0x40aa , 0x55b3	0xfacf
44	0x0788, 0x2aec	0xb40b	0x5f4c
45	0x7ac4, 0x99ef	0x7d87, 0xd5c6	0xf1d0
46	0xeade	0x613c	0xeade
47	0xeb87	0x0b09	0x1e84
48	0x44a0	0x2719	0x102b
49	0x36fe	0x9980	0xf274
50	0xa6f6	0xb559	0x5061

Note: Transaction hashes and addresses are truncated for space. Full addresses can be verified on Etherscan using the first 4-6 characters. "–" indicates data not available or not applicable for that incident.

B SUPPLEMENTARY MATERIAL FOR CHAPTER 7

B.1 COMPLETE GROUND TRUTH CONTRACT INFORMATION

This section provides complete information for the 14 manually curated contracts used to evaluate the localization methodology in Section 8.5. All contracts are sourced from the SmartBugs Wild dataset and have been verified by security experts to contain Bad Randomness vulnerabilities.

Table B.1: Complete Contract Information for Localization Ground Truth

ID	Contract Address	Vulnerability Type	LOC
C1	`0x60f52581489e879df02d86f956bd8c634f6f4db9`	Timestamp	245
C2	`0x976ec8136c990751410108e4b3f57d65183d80ea`	Coinbase	512
C3	`0x82147f5d4077f5f71f06bd6f76a850042d4db1fa`	Weak Randomness	189
C4	`0xafe73707d9e788c21bd9d492ef1af6064fae60be`	Timestamp	438
C5	`0x30439e682847ff35827f736f4b5bf5ae2fde74b2`	Timestamp	298
C6	`0x7b4700f2a2e0765aab00b082613b417cecd0f9f0`	Multiple Sources	823
C7	`0x9a3da065e1100a5613dc15b594f0f6193b419e96`	Tx.origin	231
C8	`0x9fadbdac1b57b08381e74a3591b84a138102dc23`	Timestamp	245
C9	`0x934e65cead3c1c2824ed75d54768d53fa44e4e17`	Timestamp + Access	276
C10	`0xd1766cc0a81e40d488d16357b590a0d009e0d927`	Difficulty	167
C11	`0x675821e8e9c4a14611e1851b2614f4ece718c43a`	Blocknumber	1024
C12	`0x482cf6a9d6b23452c81d4d0f0f139c1414963f89`	Blockhash	1156
C13	`0x1fe2401bd6f4de5eff1661086440297baa9a2e12`	Blockhash	352
C14	`0xd3b5e17e96fda663da9438f8e1f11185d3433d08`	Block Variables	687

Total: 14 contracts from SmartBugs Wild dataset

Column Descriptions: ID: Contract identifier used throughout the evaluation. **Contract Address**: Complete 40-character Ethereum addresses available in SmartBugs Wild dataset [62]. **LOC**: Lines of code in the smart contract source file. **Vulnerability Types**: Blockhash (weak blockhash usage for randomness), Timestamp (block.timestamp dependency), Multiple Sources (combination of multiple entropy sources such as timestamp, difficulty, and coinbase), Coinbase (block.coinbase usage), Tx.origin (transaction origin as entropy), Difficulty (block.difficulty dependency), Block Variables (multiple block-level variables combined for pseudo-randomness).

BIBLIOGRAPHY

[1] MacKenzie Sigalos. *Crypto scammers took a record 14 billion in 2021*. https://www.cnbc.com/. 2022.

[2] Chainalysis. *2.2 Billion Stolen in Crypto in 2024*. https://www.chainalysis.com/. 2024.

[3] Hadis Rezaei et al. "SoK: Root Cause of $1 Billion Loss in Smart Contract Real-World Attacks via a Systematic Literature Review of Vulnerabilities." In: *arXiv preprint arXiv:2507.20175* (July 2025). URL: https://arxiv.org/abs/2507.20175.

[4] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Arash Pourdamghani. *Probabilistic Smart Contracts: Secure Randomness on the Blockchain*. https://doi.org/10.1109/BLOC.2019.8751326. IEEE ICBC 2019.

[5] Martin Derka. *What We Learned from Fomo3D*. https://medium.com/quantstamp/what-we-learned-from-fomo3d-part-1-2c316db3d1e1. 2018-12-07.

[6] DASP. *Decentralized Application Security Project - TOP 10*. https://dasp.co. 2018-01-01.

[7] Liyi Zhou et al. "SoK: Decentralized Finance (DeFi) Attacks." In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 2444–2461.

[8] Claudia Ruggiero et al. "SoK: A Unified Data Model for Smart Contract Vulnerability Taxonomies." In: *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES 2024)*. ACM. 2024, pp. 1–11. DOI: 10.1145/3664476.3664507.

[9] André Augusto et al. "SoK: Security and Privacy of Blockchain Interoperability." In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2024, pp. 3840–3865.

[10] Josselin Feist, Gustavo Grieco, and Alex 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 '19. IEEE Press, 2019, pp. 8–15. ISBN: 9781728137414. DOI: 10.1109/WETSEB.2019.00008. URL: https://doi.org/10.1109/WETSEB.2019.00008.

[11] ConsenSys. *Mythril: Security Analysis Tool for Ethereum Smart Contracts*. Tech. rep. ConsenSys Diligence, 2017.

[12] Loi Luu et al. "Making Smart Contracts Smarter." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 254–269.

[13] Hadis Rezaei et al. "TaintSentinel: Path-Level Randomness Vulnerability Detection for Ethereum Smart Contracts." In: *2025 IEEE International Conference on Blockchain (Blockchain), 15-22* (Oct. 2025). URL: https://arxiv.org/abs/2510.18192.

[14] Valerio Piantadosi et al. "Evaluation of smart contract vulnerability analysis tools: A domain-specific perspective." In: *Information* 14.10 (2023), p. 533.

[15] Mingtao Ji et al. "Security Analysis of Blockchain Smart Contract: Taking Reentrancy Vulnerability as an Example." In: *Advances in Artificial Intelligence and Security: 7th International Conference, ICAIS 2021, Dublin, Ireland, July 19-23, 2021, Proceedings, Part III 7*. Springer. 2021, pp. 492–501.

[16] OWASP Foundation. *Smart Contract Top 10 2025*. 2025. URL: https://owasp.org/www-project-smart-contract-top-10/ (visited on 01/01/2025).

[17] Peng Qian et al. "Demystifying Random Number in Ethereum Smart Contract: Taxonomy, Vulnerability Identification, and Attack Detection." In: *IEEE Transactions on Software Engineering* 49.7 (2023), pp. 3894–3912.

[18] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Arash Pourdamghani. "Probabilistic Smart Contracts: Secure Randomness on the Blockchain." In: *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. Seoul, South Korea: IEEE, 2019, pp. 1–9. DOI: 10.1109/BLOC.2019.8751326. URL: https://ieeexplore.ieee.org/document/8751326/.

[19] Arseny Reutov. "Predicting random numbers in Ethereum smart contracts." In: *SmartDec Blog* (2018).

[20] Tianyuan Hu et al. "Detect defects of solidity smart contract based on the knowledge graph." In: *IEEE Transactions on Reliability* 73.1 (2023), pp. 186–202.

[21] Kaihua Qin et al. "Attacking the defi ecosystem with flash loans for fun and profit." In: *International conference on financial cryptography and data security*. Springer. 2021, pp. 3–32.

[22] Monika Di Angelo et al. "Smartbugs 2.0: An execution framework for weakness detection in ethereum smart contracts." In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 2102–2105.

[23] G Jaferian, D Ramezani, and MG Wagner. "Blockchain in educational gaming: unveiling opportunities and challenges." In: *EDULEARN24 proceedings* (2024), pp. 1788–1797.

[24] Hadis Rezaei et al. "SmartTaintRL: Efficient Detection and Localization of Bad Randomness Vulnerabilities in Ethereum Smart Contracts via Reinforcement Learning." In preparation. 2025.

[25] Minghui Xu et al. "Exploring blockchain technology through a modular lens: A survey." In: *ACM computing surveys* 56.9 (2024), pp. 1–39.

[26] Nazanin Moosavi et al. "Blockchain technology, structure, and applications: a survey." In: *Procedia Computer Science* 237 (2024), pp. 645–658.

[27] Kamil Jezek. "Ethereum data structures." In: *arXiv preprint arXiv:2108.05513* (2021).

[28] Casimer DeCusatis, Marcus Zimmermann, and Anthony Sager. "Identity-based network security for commercial blockchain services." In: *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. 2018, pp. 474–477.

[29] Ethereum Foundation. *Proof-of-Stake (PoS)*. Accessed: 2025-11-13. Nov. 2025. URL: `https://ethereum.org/developers/docs/consensus-mechanisms/pos/`.

[30] Rischan Mafrur. "Blockchain Data Analytics: Review and Challenges." In: *arXiv preprint arXiv:2503.09165* (2025).

[31] Kunpeng Ren et al. "Interoperability in blockchain: A survey." In: *IEEE Transactions on Knowledge and Data Engineering* 35.12 (2023), pp. 12750–12769.

[32] Sara Rouhani and Ralph Deters. "Security, performance, and applications of smart contracts: A systematic survey." In: *IEEE Access* 7 (2019), pp. 50759–50779.

[33] Guangfu Wu et al. "A comprehensive survey of smart contract security: State of the art and research directions." In: *Journal of Network and Computer Applications* (2024), p. 103882.

[34] Nikolay Ivanov et al. "Security threat mitigation for smart contracts: A comprehensive survey." In: *ACM Computing Surveys* 55.14s (2023), pp. 1–37.

[35] Hanting Chu et al. "A survey on smart contract vulnerabilities: Data sources, detection and repair." In: *Information and Software Technology* 159 (2023), p. 107221.

[36] Fan Jiang et al. "Enhancing smart-contract security through machine learning: A survey of approaches and techniques." In: *Electronics* 12.9 (2023), p. 2046.

[37] Zhiyuan Wei et al. "Survey on Quality Assurance of Smart Contracts." In: *ACM Computing Surveys* 57.3 (2024), pp. 1–39.

[38] Purathani Praitheeshan, Lei Pan, and Robin Doss. "The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support." In: *Journal of Cybersecurity and Privacy* 2.2 (2022), pp. 358–378.

[39] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on ethereum smart contracts (sok)." In: *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer. 2017, pp. 164–186.

[40] James C King. "Symbolic execution and program testing." In: *Communications of the ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[41] Loi Luu et al. "Making smart contracts smarter." In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 254–269.

[42] Roberto Baldoni et al. "A survey of symbolic execution techniques." In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39. DOI: 10.1145/3182657.

[43] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)." In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[44] Johannes Krupp and Christian Rossow. "{teEther}: Gnawing at ethereum to automatically exploit smart contracts." In: *27th USENIX security symposium (USENIX Security 18)*. 2018, pp. 1317–1333.

[45] Reza Kiani and Victor S Sheng. "Ethereum smart contract vulnerability detection and machine learning-driven solutions: A systematic literature review." In: *Electronics* 13.12 (2024), p. 2295. DOI: 10.3390/electronics13122295.

[46] Xiaofei Tang et al. "Deep learning-based solution for smart contract vulnerabilities detection." In: *Scientific Reports* 13.1 (2023), p. 20106. DOI: 10.1038/s41598-023-47219-0.

[47] Meng Wang, Weiliang Fei, et al. "Reinforcement Learning Guided Symbolic Execution for Ethereum Smart Contracts." In: *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2023, pp. 91–100. DOI: 10.1109/APSEC60142.2023.00018.

[48] Shihao Sun et al. "Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution." In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022, pp. 709–720. DOI: 10.1145/3533767.3534395.

[49] Menglin Fu et al. "SmartExecutor: Coverage-Driven Symbolic Execution Guided via State Prioritization and Function Selection." In: *Distributed Ledger Technologies: Research and Practice* 3.3 (2024), pp. 1–29. DOI: 10.1145/3678188.

[50] Xiao Bai et al. "A Critical-Path-Coverage-Based Vulnerability Detection Method for Smart Contracts." In: *IEEE Access* 7 (2019), pp. 147327–147344. DOI: 10.1109/ACCESS.2019.2946471.

[51] Monika Di Angelo and Gernot Salzer. "Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode." In: vol. 200. Elsevier, 2023, p. 111652. DOI: 10.1016/j.jss.2023.111652.

[52] William Cheng, Steven Goldfeder, et al. "MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract." In: *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 2019, pp. 456–467. DOI: 10.1109/ISSRE.2019.00053.

[53] Stefanos Chaliasos et al. "Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?" In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. IEEE/ACM, 2024, pp. 1–13. DOI: 10.1145/3597503.3623302.

[54] Yue Huang, Song Bian, et al. "Comprehensive review of smart contract and DeFi security: Attack, vulnerability detection, and automated repair." In: *Expert Systems with Applications* 265 (2025), p. 125950. DOI: 10.1016/j.eswa.2025.125950.

[55] Abdullah Alahmadi and Tiziana Margaria. "Smart Contract Security in Decentralized Finance: Enhancing Vulnerability Detection with Reinforcement Learning." In: *Applied Sciences* 15.11 (2025), p. 5924. DOI: 10.3390/app15115924.

[56] Sifis Lagouvardos et al. "Precise static modeling of Ethereum "memory"." In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–26. DOI: 10.1145/3428258.

[57] Vahid Mirjalili, Natalia Stakhanova, and Ali A. Ghorbani. "Storage State Analysis and Extraction of Ethereum Blockchain Smart Contracts." In: *ACM Transactions on Software Engineering and Methodology* 32.5 (2023), pp. 1–30. DOI: 10.1145/3548683.

[58] Han Liu, Jun Sun, Jianjun Huang, et al. "Detecting Smart Contract State-Inconsistency Bugs via Flow Divergence and Multiplex Symbolic Execution." In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 1–26. DOI: 10.1145/3715712.

[59] TON Development Team. *Adapting Large Language Models for Smart Contract Defects Detection in the Open Network Blockchain*. Technical Report. 2025.

[60] Sergei Tikhomirov et al. "SmartCheck: Static Analysis of Ethereum Smart Contracts." In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 2018, pp. 9–16.

[61] Loi Luu et al. “Making Smart Contracts Smarter.” In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 254–269.

[62] Thomas Durieux et al. “Empirical review of automated analysis tools on 47,587 Ethereum smart contracts.” In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 530–541.

[63] Petar Tsankov et al. *Securify: Practical Security Analysis of Smart Contracts*. https://doi.org/10.1145/3243734.3243780. ACM CCS 2018.

[64] Christof Ferreira Torres, Julian Schütte, and Radu State. “Osiris: Hunting for integer bugs in ethereum smart contracts.” In: *Proceedings of the 34th annual computer security applications conference*. 2018, pp. 664–676.

[65] X. Tang, Y. Du, A. Lai, et al. “Deep Learning-based Solution for Smart Contract Vulnerabilities Detection.” In: *Scientific Reports* 13.20106 (2023).

[66] Sihao Hu et al. “Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives.” In: *arXiv preprint arXiv:2310.01152* (2023). DOI: 10.48550/arXiv.2310.01152.

[67] Oualid Zaazaa and Hanan El bakkali. “SmartLLMSentry: A Comprehensive LLM Based Smart Contract Vulnerability Detection Framework.” In: *Available at SSRN 5037605* (2024). DOI: 10.2139/ssrn.5037605.

[68] QuillAudits. *QuillShield: AI-Powered Security Analysis Tool*. https://www.quillaudits.com/blog/smart-contract/smart-contract-security-tools-guide. Accessed: 2025-07-15. 2024.

[69] Gustavo Grieco et al. “Echidna: effective, usable, and fast fuzzing for smart contracts.” In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM. 2020, pp. 557–560.

[70] Crytic. *Medusa: Smart Contract Fuzzer*. https://github.com/crytic/medusa. Accessed: 2025-07-15. 2024.

[71] Priyanka Bose et al. “Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities.” In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. ACM, 2022, pp. 1–18. DOI: 10.1145/3519939.3523679.

[72] Tai D. Nguyen, Long H. Pham, and Jun Sun. “SGUARD: Towards Fixing Vulnerable Smart Contracts Automatically.” In: *IEEE Symposium on Security and Privacy*. 2021, pp. 1215–1229.

[73] M. Rodler et al. “EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts.” In: *USENIX Security Symposium*. 2021.

[74] Y. Zhang et al. “SMARTSHIELD: Automatic Smart Contract Protection Made Easy.” In: *IEEE International Conference on Software Analysis, Evolution and Reengineering*. 2020.

[75] Various Authors. “SmartState: Detecting State-Reverting Vulnerabilities in Smart Contracts via Fine-Grained State-Dependency Analysis.” In: *arXiv preprint* (2024).

[76] J. Xu, T. Wang, M. Lv, et al. “MVD-HG: Multigranularity Smart Contract Vulnerability Detection Method Based on Heterogeneous Graphs.” In: *Cybersecurity* 7.55 (2024).

[77] C. Gao et al. “sGuard+: Machine Learning Guided Rule-Based Automated Vulnerability Repair on Smart Contracts.” In: *ACM Transactions on Software Engineering and Methodology* (2024).

[78] Christoph Sendner et al. “G-Scan: Graph Neural Networks for Line-Level Vulnerability Identification in Smart Contracts.” In: *arXiv preprint arXiv:2307.08549* (2023).

[79] C. Ma, S. Liu, and G. Xu. “HGAT: Smart Contract Vulnerability Detection Method Based on Hierarchical Graph Attention Network.” In: *Journal of Cloud Computing* 12.93 (2023).

[80] Yulin Zhong, JiaXin Chen, Yinglong Wang, et al. “A Smart Contract Vulnerability Line Detection Method Based on Graph Neural Network and Fusion of Multidimensional Code Representation.” In: *Information and Software Technology* (2025).

[81] Yuqiang Sun et al. “GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis.” In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.

[82] Coinbase. *Ethereum Price, Charts, and News*. `https://www.coinbase.com/price/ethereum`. 2025-04-19.

[83] Vitalik Buterin et al. “A Next-Generation Smart Contract and Decentralized Application Platform.” In: *white paper* 3.37 (2014), pp. 2–1.

[84] Mojtaba Eshghie, Mikael Jafari, and Cyrille Artho. “From Creation to Exploitation: The Oracle Lifecycle.” In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*. Mar. 2024, pp. 23–34. DOI: `10.1109/SANER-C62648.2024.00009`. (Visited on 12/08/2024).

[85] Monika di Angelo et al. "SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts." In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. ASE '23. Echternach, Luxembourg: IEEE Press, 2024, pp. 2102–2105. ISBN: 9798350329964. DOI: 10.1109/ASE56229.2023.00060. URL: https://doi.org/10.1109/ASE56229.2023.00060.

[86] Thomas Durieux et al. "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: ACM, 2020, pp. 530–541. ISBN: 9781450371216. DOI: 10.1145/3377811.3380364. URL: https://doi.org/10.1145/3377811.3380364.

[87] Huashan Chen et al. "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses." In: *ACM Computing Surveys (CSUR)* 53.3 (2020), pp. 1–43.

[88] Satpal Singh Kushwaha et al. "Systematic review of security vulnerabilities in ethereum blockchain smart contract." In: *IEEE Access* 10 (2022), pp. 6605–6621.

[89] Palina Tolmach et al. "A survey of smart contract formal specification and verification." In: *ACM Computing Surveys (CSUR)* 54.7 (2021), pp. 1–38.

[90] Noama Fatima Samreen and Manar H. Alalfi. "A survey of security vulnerabilities in ethereum smart contracts." In: *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*. CASCON '20. Toronto, Ontario, Canada: IBM Corp., 2020, pp. 73–82.

[91] Barbara Kitchenham, Stuart Charters, et al. "Guidelines for performing systematic literature reviews in software engineering version 2.3." In: *Engineering* 45.4ve (2007), p. 1051.

[92] OpenZeppelin. *OpenZeppelin/openzeppelin-solidity*. https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol. 2025-06-12.

[93] Solidity Documentation. *Security Considerations*. https://docs.soliditylang.org/en/latest/security-considerations.html. 2025-03-05.

[94] Uniswap. *FullMath UniSwap*. https://docs.uniswap.org/contracts/v3/reference/core/libraries/FullMath. 2025-07-18.

[95] Hengyan Zhang et al. "SVScanner: Detecting smart contract vulnerabilities via deep semantic extraction." In: *Journal of Information Security and Applications* 75 (2023), p. 103484.

[96] Alexander Mense and Markus Flatscher. "Security vulnerabilities in ethereum smart contracts." In: *Proceedings of the 20th international conference on information integration and web-based applications & services*. 2018, pp. 375–380.

[97] Ethereum Foundation. *Solidity Documentation: Control Structures*. `https://docs.soliditylang.org/en/latest/control-structures.html`. 2025-02-24.

[98] Zhuo Zhang et al. "Reentrancy vulnerability detection and localization: A deep learning based two-phase approach." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–13.

[99] Bo Jiang, Ye Liu, and Wing Kwong Chan. "Contractfuzzer: Fuzzing smart contracts for vulnerability detection." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 259–269.

[100] Ethereum Foundation. *ERC-2771: Secure Protocol for Native Meta Transactions*. `https://eips.ethereum.org/EIPS/eip-2771`. 2025-01-02.

[101] OpenZeppelin. *ERC2771Context Multicall Vulnerability Disclosure*. `https://blog.openzeppelin.com/arbitrary-address-spoofing-vulnerability-erc2771context-multicall-public-disclosure`. 2025-01-02.

[102] Shashank. *Unveiling the ERC-2771 Context and Multicall Vulnerability*. `https://blog.solidityscan.com/unveiling-the-erc-2771context-and-multicall-vulnerability-f96ffa5b499f`. 2025-01-02.

[103] Francesco Vollero. *Checks-Effects-Interactions Pattern*. `https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html`. 2025-03-04.

[104] Mojtaba Eshghie et al. "Capturing Smart Contract Design with DCR Graphs." In: *Software Engineering and Formal Methods*. Vol. 14001. Lecture Notes in Computer Science. `https://link.springer.com/chapter/10.1007/978-3-031-47115-5_7`. Springer, 2023, pp. 116–134. DOI: `10.1007/978-3-031-47115-5_7`.

[105] Lejun Zhang et al. "A Novel Smart Contract Reentrancy Vulnerability Detection Model based on BiGAS." In: *Journal of Signal Processing Systems* (2023), pp. 1–23.

[106] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. "eTainter: detecting gas-related vulnerabilities in smart contracts." In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022, pp. 728–739.

[107] RareSkills. *The RareSkills Book of Solidity Gas Optimization: 80+ Tips*. https://www.rareskills.io/post/gas-optimization. 2025-02-18.

[108] Hacken. *Solidity Gas Optimization: Best Practices & Expert Tips*. https://hacken.io/discover/solidity-gas-optimization/. 2025-02-17.

[109] Jianzhong Su et al. "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–12.

[110] Infuy. *Preventing Denial of Service Attacks in Solidity*. https://www.infuy.com/blog/preventing-denial-of-service-attacks-in-solidity/. 2025-04-18.

[111] Zeqin Liao et al. "Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis." In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 980–991.

[112] Parity Technologies. *A Postmortem on the Parity Multi-Sig Library Self-Destruct*. https://medium.com/paritytech/a-postmortem-on-the-parity-multi-sig-library-self-destruct-63daca3a4cf7. 2017-11-15.

[113] Ethereum Foundation. *EIP-6780: SELFDESTRUCT only in same transaction as creation*. https://eips.ethereum.org/EIPS/eip-6780. 2025-03-22.

[114] Daojing He et al. "Detection of Vulnerabilities of Blockchain Smart Contracts." In: *IEEE Internet of Things Journal* (2023).

[115] QuickNode. *Common Solidity Vulnerabilities on Ethereum*. https://www.quicknode.com/guides/ethereum-development/smart-contracts/common-solidity-vulnerabilities-on-ethereum. 2025-03-24.

[116] OWASP Foundation. *OWASP Smart Contract Top 10*. Tech. rep. Open Web Application Security Project, 2023.

[117] Chainlink. *Secure Randomness in Solidity Smart Contracts*. https://blog.chain.link/random-number-generation-solidity/. 2025-01-18.

[118] Chainlink. *Chainlink VRF: Verifiable Randomness*. https://docs.chain.link/vrf/v2-5. 2025-04-15.

[119] Sabreen Ahmadjee et al. "Decision Support Model for Selecting the Optimal Blockchain Oracle Platform: An Evaluation of Key Factors." In: *ACM Transactions on Software Engineering and Methodology* 34.1 (2025), pp. 1–35.

[120] Ethereum Foundation. *Commit-Reveal Scheme in Solidity*. https://ethereum.org/en/developers/docs/commit-reveal/. 2023.

[121] Block.one. *EOS.IO Technical White Paper*. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md. 2025-03-08.

[122] Ethereum Foundation. *BTCRelay: A Bitcoin Relay for Ethereum Smart Contracts*. https://github.com/ethereum/btcrelay/blob/develop/README.md. 2025-04-13.

[123] Wuqi Zhang et al. "Nyx: Detecting exploitable front-running vulnerabilities in smart contracts." In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2024, pp. 2198–2216.

[124] SCSFG. *Hackers' Guide to Front-Running in Blockchain*. https://scsfg.io/hackers/frontrunning/. 2025-02-18.

[125] Infuy. *Understanding Phishing with tx.origin in Solidity*. https://www.infuy.com/blog/understanding-phishing-with-tx-origin-in-solidity/. 2025-02-23.

[126] SlowMist. *Common Vulnerabilities in Solidity: Phishing with tx.origin*. https://www.slowmist.com/articles/solidity-security/Common-Vulnerabilities-in-Solidity-Phishing-with-tx.origin.html. 2025-04-05.

[127] *Smart Contract Weakness Classification Registry*. https://swcregistry.io/. Accessed: 2025. 2020.

[128] Fernando Richter Vidal et al. "OpenSCV: An Open Hierarchical Taxonomy for Smart Contract Vulnerabilities." In: *Empirical Software Engineering* 29.101 (2024).

[129] Massimo Bartoletti et al. "Dissecting Ponzi Schemes on Ethereum: Identification, Analysis, and Impact." In: *Future Generation Computer Systems* 102 (2020), pp. 259–277.

[130] Ivica Nikolić et al. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale." In: *ACSAC*. 2018, pp. 653–663.

[131] OWASP. *Smart Contract Security Verification Standard (SCSVS)*. https://scs.owasp.org/. 2025.

[132] ConsenSys Diligence. *Ethereum Smart Contract Best Practices*. https://consensys.github.io/smart-contract-best-practices/. 2020.

[133] Nicola Ruaro et al. "Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts." In: *NDSS*. 2024.

[134] Kauz Security. *Reentrancy - Smart Contract Security Field Guide*. https://scsfg.io/hackers/reentrancy/. 2024.

[135] Jiaming Ye et al. "Clairvoyance: Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings (ICSE '20)*. 2020, pp. 274–275.

[136] Jingwen Zhang et al. "SmartReco: Detecting Read-Only Reentrancy via Fine-Grained Cross-DApp Analysis." In: *arXiv preprint arXiv:2409.18468* (2024).

[137] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. "SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain." In: *FC 2019 Workshops*. 2019, pp. 170–189.

[138] Torgin Mackinga et al. *TWAP Oracle Attacks: Easier Done than Said?* Cryptology ePrint Archive, 2022/445. 2022.

[139] Kaihua Qin et al. "Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit." In: *FC 2021*. 2021, pp. 3–32.

[140] Chainlink Labs. *Using Data Feeds*. `https://docs.chain.link/data-feeds`. 2023.

[141] Christof Ferreira Torres, Ramiro Camino, and Radu State. "Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on Ethereum." In: *USENIX Security*. 2021.

[142] Liyi Zhou et al. "High-Frequency Trading on Decentralized On-Chain Exchanges." In: *IEEE S&P*. 2021.

[143] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger." In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[144] Compound. *Compound Governance Examples*. `https://github.com/compound-developers/compound-governance-examples`. 2025-01-16.

[145] Joshua Ellul and Gordon J. Pace. "SoliNomic: A Self-modifying Smart Contract Game Exploring Reflexivity in Law." In: *Disruptive Technologies in Media, Arts and Design*. Ed. by Alexiei Dingli et al. Cham: Springer International Publishing, 2022, pp. 3–14. ISBN: 978-3-030-93780-5. DOI: `10.1007/978-3-030-93780-5_1`.

[146] Ethereum Foundation. *Solidity v0.5.0 Breaking Changes*. `https://docs.soliditylang.org/en/v0.5.0/050-breaking-changes.html`. 2025-02-22.

[147] Valentina Piantadosi et al. "Detecting functional and security-related issues in smart contracts: A systematic literature review." In: *Software: Practice and experience* 53.2 (2023), pp. 465–495.

[148] Petar Tsankov et al. “Securify: Practical security analysis of smart contracts.” In: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018, pp. 67–82.

[149] Ethereum Foundation. *EIP-150: Gas cost changes for IO-heavy operations*. https://eips.ethereum.org/EIPS/eip-150. 2025-04-21.

[150] Rekt News. *Euler REKT*. https://rekt.news/euler-rekt. Mar. 2023.

[151] Rekt News. *Nomad Bridge*. https://rekt.news/nomad-rekt. Aug. 2022.

[152] Rekt News. *Rekt News Leaderboard*. https://rekt.news/leaderboard/. 2025-05-08.

[153] DeFiYield Team. *DeFiYield REKT Database*. https://defiyield.app/rekt-database. 2025-05-08.

[154] SlowMist Team. *SlowMist Hacked Database*. https://hacked.slowmist.io/. 2025-05-08.

[155] BlockSec Team. *BlockSec Blog on Medium*. https://blocksecteam.medium.com/. 2025-05-08.

[156] Base Documentation Team. *Bridges on Base Mainnet*. https://docs.base.org/chain/bridges-mainnet. 2025-05-18.

[157] Ethereum Foundation. *Ethereum Scaling*. https://ethereum.org/en/developers/docs/scaling. 2025-05-18.

[158] Immunebytes. *Euler Finance Hack - Mar 13, 2023: Detailed Hack Analysis*. https://immunebytes.com/blog/euler-finance-hack-mar-13-2023-detailed-hack-analysis/. 2025-03-29.

[159] Rekt. *Euler Finance*. https://rekt.news/zh/euler-rekt. 2025-04-15.

[160] SlowMist. *SlowMist: An Analysis of the Attack on Euler Finance*. https://medium.com/p/5143abc0d5ad. 2025-03-29.

[161] Rekt News. *Nomad Bridge*. https://rekt.news/nomad-rekt. 2022-08.

[162] Rekt News. *Rekt News - DeFi Investigations and Analysis*. https://rekt.news/bonq-rekt. 2025-03-29.

[163] Rekt News. *Woofi*. https://rekt.news/woo-rekt. 2025-04-15.

[164] Olympix. *Unpacking the WOOFi Swap Exploit*. https://medium.com/p/ae6f172fe736. 2025-04-15.

[165] REKT News. *Fei Rari*. https://rekt.news/fei-rari-rekt. 2025-03-29.

[166] CertiK. *Fei Protocol Incident Analysis*. https://medium.com/p/8527440696cc. 2025-04-13.

[167] Rekt News. *Infini*. https://rekt.news/infini-rekt. 2025-04-13.

[168] Gushiken Yona. *Rogue Developer Suspected in Infini Neobank's $49.5M Exploit*. https://news.shib.io/2025/02/24/rogue-developer-suspected-in-infini-neobanks-49-5m-exploit/. 2025-04-26.

[169] REKT. "KyberSwap." In: (). 2023-11-23.

[170] SlowMist. *A Deep Dive Into the KyberSwap Hack*. https://medium.com/p/3e13f3305d3a. (Visited on 11/27/2023).

[171] Olympix. "Penpie Exploit: A Technical Post-Mortem Analysis." In: (). 2024-09-06.

[172] SolidityScan. "Penpie Hack Analysis." In: (). 2024-09-05.

[173] CertiK. *Sonne Finance Incident Analysis*. https://rekt.news/sonne-finance-rekt. 2025-03-01.

[174] REKT News. *Curve, Vyper*. https://rekt.news/curve-vyper-rekt/. (Visited on 03/29/2025).

[175] REKT. *Inverse Finance*. https://rekt.news/inverse-finance-rekt/. 2022-04-03.

[176] Cycle Trading. *Inverse Finance: A way to survive from the brink of death?* https://www.binance.com/en/square/post/746243. 2025-04-15.

[177] OKLink Academy. *HOT: Inverse Finance Loses Over $15M in Oracle Manipulation*. https://www.oklink.com/academy/en/2022/07/12/hot-inverse-finance-loses-over-15m-in-oracle-manipulation?utm_source=chatgpt.com. 2025-04-14.

[178] Blockbasis. *Holograph Protocol Hack: Mitigating the $14.4 Million Exploit*. https://www.blockbasis.com/p/holograph-protocol-hack-mitigating-the-14-4-million-exploit. 2025-03-16.

[179] Halborn. *Explained: The Holograph Hack (June 2024)*. https://www.halborn.com/blog/post/explained-the-holograph-hack-june-2024. 2025-04-03.

[180] QuillAudits. *Decoding Deus DAO $6.5M Exploit*. https://medium.com/p/588bbecec61f. 2025-04-15.

[181] Cyvers.ai. *Deus Finance Hack: An In-Depth Analysis of the $6.5 Million Loss*. https://cyvers.ai/blog/deus-finance-hack-an-in-depth-analysis-of-the-6-5-million-loss. 2025-04-15.

[182] Piyush Shukla. *Ionic Money $8.5 million Exploit: Hack Analysis*. https://www.linkedin.com/pulse/ionic-money-85-million-exploit-hack-analysis-piyush-shukla-7j8if. 2025-02-09.

[183] Rekt News. *Ionic Money*. https://rekt.news/ionic-money-rekt. 2025-04-17.

[184] Rekt News. *RoninNetworkRektII*. https://rekt.news/roninnetwork-rektII. 2025-04-17.

[185] Shashank. *RoninBridgeHack Analysis*. https://blog.solidityscan.com/ronin-bridge-hack-analysis-d8f64b8fe683. 2025-04-17.

[186] LI.FI. *Incident Report - 16th July*. https://li.fi/knowledge-hub/incident-report-16th-july/. 2025-04-17.

[187] SolidityScan. *LI.FI Hack Analysis*. https://blog.solidityscan.com/li-fi-hack-analysis-521388128d22. 2025-04-17.

[188] Rekt. *Yearn: Re-Rekt*. https://rekt.news/yearn2-rekt. 2025-04-17.

[189] CertiK. *Misconfigured Earnings: Yearn Finance Exploit Explained*. https://www.certik.com/resources/blog/40AqeRsWil5Hof9etrvjH2-earnings-misconfigured-yearn-finance-exploit-explained. 2025-04-17.

[190] SlowMist. *In-depth Analysis of zkLend Hack Linked to EraLend Hack*. https://slowmist.medium.com/in-depth-analysis-of-zklend-hack-linked-to-eralend-hack-fba4af9b66ef. 2025-04-18.

[191] Piyush Shukla. *ZkLend Hack Analysis*. https://www.linkedin.com/pulse/zklend-hack-analysis-piyush-shukla-jsxzf/. 2025-04-18.

[192] LI.FI Team. *Incident Report – 16th July 2023*. https://li.fi/knowledge-hub/incident-report-16th-july/. 2025-04-18.

[193] Rekt News. *LI.FI Jumper*. https://rekt.news/lifi-jumper-rekt. 2025-04-18.

[194] Rekt News. *Hundred Finance 2*. https://rekt.news/hundred-rekt2. 2025-04-19.

[195] Cointelegraph. *Hundred Finance hacker moves assets year after $7M exploit*. https://cointelegraph.com/news/hundred-finance-hacker-moves-assets-year-after-exploit. 2025-04-19.

[196] Rekt News. *Abracadabra*. https://rekt.news/abra-rekt. 2025-04-19.

[197] CertiK. *Lodestar Finance Incident Analysis*. https://www.certik.com/resources/blog/TqTyq4vYHl8JzS7zyJye9-lodestar-finance-incident-analysis. 2025-04-18.

[198] Rekt News. *Lodestar Finance*. https://rekt.news/lodestar-rekt. 2025-04-18.

[199] REKT. *1inch*. https://rekt.news/1inch-rekt. 2025-04-18.

[200] Decurity. *Yul Calldata Corruption: 1inch Postmortem*. https://blog.decurity.io/yul-calldata-corruption-1inch-postmortem-a7ea7a53bfd9. 2025-04-18.

[201] BlockApex. *Shezmu Hack Analysis*. https://medium.com/p/32414e67c7a0. 2025-04-18.

[202] CyberStrategy1. *Crypto Security Truths — Issue #13*. https://medium.com/p/098b78916272. 2025-04-18.

[203] REKT News. *Gamma Strategies*. https://rekt.news/gamma-strategies-rekt. 2025-04-18.

[204] Verichains. *Gamma Protocol Exploit Analysis*. https://blog.verichains.io/p/gamma-protocol-exploit-analysis. 2025-04-18.

[205] REKT News. *Conic Finance*. https://rekt.news/conic-finance-rekt. 2023-07-10.

[206] Shashank. *Gamma Hack Analysis*. https://blog.solidityscan.com/gamma-hack-analysis-6c074e61709e. 2024-01.

[207] Halborn Security. *Explained: The KiloEx Hack*. https://www.halborn.com/blog/post/explained-the-kiloex-hack-april-2025. 2025-04-18.

[208] Rekt News. *KiloEx*. https://rekt.news/kiloex-rekt. 2025-04-18.

[209] Cointelegraph. *SIR.trading begs hacker to return $255K or 'no chance for us to survive'*. https://www.tradingview.com/news/cointelegraph:6a2330d53094b. 2025-04-18.

[210] Rekt News. *SIR.trading*. https://rekt.news/sirtrading-rekt. 2025-04-18.

[211] Halborn Security. *Explained: The Abracadabra Money Hack (March 2025)*. https://www.halborn.com/blog/post/explained-the-abracadabra-money-hack-march-2025. 2025-04-22.

[212] REKT. *DeltaPrime II*. https://rekt.news/deltaprime-rekt2. 2025-04-22.

[213] SolidityScan. *DeltaPrime Hack Analysis*. https://blog.solidityscan.com/deltaprime-hack-analysis-44edb9b22567. 2025-04-22.

[214] Halborn. *Explained: The Onyx Protocol Hack*. https://www.halborn.com/blog/post/explained-the-onyx-protocol-hack-september-2024?utm_source=chatgpt.com. 2024-04-23.

[215] REKT. *Onyx Protocol II*. https://rekt.news/onyx-protocol-rekt2. 2024-04-23.

[216] Rekt News. *Dexible*. https://rekt.news/dexible-rekt. 2025-04-23.

[217] Shashank. *Dexible Hack Analysis*. https://blog.solidityscan.com/dexible-hack-analysis-never-blindly-trust-smart-contracts-c2aee7943c1a. 2025-04-23.

[218] DeHacker Security. *Dough Finance Incident Analysis*. https://medium.com/p/04db15c22552. 2025-04-22.

[219] CertiK. *Dough Finance Incident Analysis*. https://www.certik.com/resources/blog/3SMOuGMCSttY4pQW6I49W2-dough-finance-incident-analysis. 2025-04-22.

[220] SolidityScan Team. *CloberDEX Liquidity Vault Hack Analysis*. https://blog.solidityscan.com/cloberdex-liquidity-vault-hack-analysis-f22eb960aa6f. 2025-04-25.

[221] REKT News. *CloberDEX*. https://rekt.news/cloberdex-rekt. 2025-04-25.

[222] Rekt News. *Tornado Cash Governance Attack*. https://rekt.news/tornado-gov-rekt. 2025-04-25.

[223] Rob Behnke. *Explained: The Tornado Cash Hack (May 2023)*. https://www.halborn.com/blog/post/explained-the-tornado-cash-hack-may-2023. 2025-04-25.

[224] REKT. *Team Finance*. https://rekt.news/teamfinance-rekt.

[225] SlowMist. *Analysis & Review of Team Finance Exploit*. https://medium.com/p/f439c2f63e2. 2025-06-12.

[226] Rekt News. *RAFT Rekt*. https://rekt.news/raft-rekt. 2025-04-25.

[227] SharkTeam. *SharkTeam Analysis of the Principles of RAFT Attack Incident*. https://medium.com/p/814127d626bd. 2025-04-25.

[228] REKT. *Rho Market*. https://rekt.news/rho-market-rekt. 2025-04-28.

[229] Olympix. *Rho Markets on Scroll Exploit Analysis*. https://medium.com/p/965991270f56. 2025-04-28.

[230] REKT. *Uwulend*. https://rekt.news/uwulend-rekt. 2025-04-28.

[231] Shashank. *Uwulend Hack Analysis*. https://blog.solidityscan.com/uwulend-hack-analysis-77eb9181a717. 2025-04-28.

[232] REKT. *Velocore*. https://rekt.news/velocore-rekt. 2025-04-28.

[233] VELOCORE. *Velocore Incident PostMortem*. https://medium.com/p/6197020ec3e9. 2025-04-28.

[234] REKT. *Curio*. https://rekt.news/curio-rekt. 2025-04-28.

[235] Rob Behnke. *Explained: The Curio Hack (March 2024)*. https://www.halborn.com/blog/post/explained-the-curio-hack-march-2024. 2025-04-28.

[236] Rekt News. *Unizen Rekt*. https://rekt.news/unizen-rekt. 2025-04-29.

[237] Blockbasis. *Unizen: Navigating the $2.1M Security Breach*. https://blockbasis.com/p/unizen-navigating-21m-security-breach. 2025-04-29.

[238] Rekt. *Seneca Protocol*. https://rekt.news/seneca-protocol-rekt. 2025-04-30.

[239] BlockApex. *Seneca Protocol Hack Analysis*. https://medium.com/p/546f3bcc1040. 2025-04-30.

[240] Rekt News. *OKX DEX*. https://rekt.news/okx-dex-rekt. 2025-04-30.

[241] Olympix AI. *Unmasking the Malicious OKX DEX Upgrade*. https://medium.com/p/f6912449f963. 2025-04-30.

[242] REKT News. *Zunami Protocol*. https://rekt.news/zunami-protocol-rekt. 2025-04-30.

[243] SolidityScan. *Zunami Protocol Hack Analysis*. https://blog.solidityscan.com/zunami-protocol-hack-analysis-e95981976e11. 2025-04-30.

[244] Rekt News. *EraLend REKT: Read-only Re-entrancy Exploit Steals $3.4M*. https://rekt.news/eralend-rekt. 2025-04-29.

[245] Bitcoinist. *EraLend Exploit*. https://cryptorank.io/news/feed/ad957-eralend-exploit-hackers-steal-3-4-million-dollars. 2025-04-29.

[246] Rekt News. *Atlantis Loans*. https://rekt.news/atlantis-loans-rekt. 2025-04-29.

[247] Shashank. *Atlantis Loans Hack Analysis*. https://blog.solidityscan.com/atlantis-loans-hack-analysis-7f3fb2e295e0. 2025-04-29.

[248] Rekt News. *Sturdy*. https://rekt.news/sturdy-rekt. 2025-05-04.

[249] SolidityScan Team. *Sturdy Finance Hack Analysis*. https://blog.solidityscan.com/sturdy-finance-hack-analysis-bd8605cd2956. 2025-05-04.

[250] Rekt. *Jimbos Protocol $7.5 Million Hack*. https://rekt.news/jimbo-rekt. 2023-05-29.

[251] Numen Cyber Labs. *A Detailed Analysis of Arbitrum-based Jimbos Protocol's $7.5 Million Hack*. https://medium.com/p/36af84faee2. 2023-05-29.

[252] Rekt News. *Swaprum Rekt*. https://rekt.news/swaprum-rekt. 2024-04-20.

[253] CryptoRank. *The Swaprum Incident*. https://cryptorank.io/ru/news/feed/cdce4-the-swaprum-incident-audited-defi-protocol-dupes-investors-out-3m. 2025-04-17.

[254] Rekt News. *Orion Protocol - REKT*. 2023. URL: https://rekt.news/orion-protocol-rekt/.

[255] SlowMist. *An Analysis of the Attack on Orion Protocol*. https://medium.com/p/c7aef70aff83. 2025-05-04.

[256] Turner Wright. *BonqDAO protocol exploiter steals $120M after oracle manipulation*. https://cointelegraph.com/news/bonqdao-protocol-suffers-120m-loss-after-oracle-hack. 2025-03-29.

[257] Rekt News. *Inverse Finance*. https://rekt.news/inverse-finance-rekt/. 2023-05-15.

[258] Kudelski Security. *The Poly Network Hack Explained*. https://research.kudelskisecurity.com/2021/08/12/the-poly-network-hack-explained/. 2025-01-05.

[259] Rekt News. *Abracadabra*. https://rekt.news/abracadabra-rekt2. 2025-04-22.

[260] Extropy IO. *Breaking Down the $6.5M Abracadabra Money Hack: A Detailed Analysis of the Cauldron V4 Exploit*. https://medium.com/p/44100da10896. 2025-04-19.

[261] Palina Tolmach et al. "Formal Analysis of Composable DeFi Protocols." In: *Financial Cryptography and Data Security. FC 2021 International Workshops*. Ed. by Matthew Bernhard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 149–161. ISBN: 978-3-662-63958-0.

[262] Ethereum Improvement Proposals. *EIP-1153: Transient Storage Opcodes*. https://eips.ethereum.org/EIPS/eip-1153. 2025-06-28.

[263] Mojtaba Eshghie et al. "HighGuard: Cross-Chain Business Logic Monitoring of Smart Contracts." In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: ACM, 2024, pp. 2378–2381. ISBN: 9798400712487. DOI: 10.1145/3691620.3695356. URL: https://doi.org/10.1145/3691620.3695356.

[264] Immunebytes. *Hundred Finance Hack - April 15, 2023: Detailed Analysis*. https://www.immunebytes.com/blog/hundred-finance-hack-april-15-2023-detailed-analysis/. 2024-10-05.

[265] ZAN. "Unpacking the Tornado Cash Governance Attack." In: (). 2025-03-21. URL: %5Curl%7Bhttps://medium.com/p/15b40691ca2e%7D.

[266] LI.FI. *Incident Report – 16th July*. https://li.fi/knowledge-hub/incident-report-16th-july/. July 2024.

[267] Shashank. *LI.FI Hack Analysis*. https://blog.solidityscan.com/li-fi-hack-analysis-521388128d22. July 2024.

[268] Immunebytes. *LI.FI Hack Analysis*. https://immunebytes.com/blog/li-fi-hack-analysis/. July 2024.

[269] The Block. *LI.FI Cross-Chain Protocol Exploited for $8M*. https://www.theblock.co/. July 2024.

[270] CoinDesk. *LI.FI Protocol Loses $8M in Security Breach*. https://www.coindesk.com/. July 2024.

[271] HashDit. *LI.FI Protocol Incident*. https://hashdit.github.io/lifi-incident/. July 2024.

[272] Chainlink Labs. *Chainlink VRF (Verifiable Random Function)*. Accessed: 2025. 2024. URL: https://docs.chain.link/vrf.

[273] Bernhard Mueller. *Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit*. Conference Whitepaper. Presented at HITB-SecConf 2018. Amsterdam: ConsenSys Diligence, 2018.

[274] SmartContractSecurity. *SWC-120: Weak Sources of Randomness from Chain Attributes*. https://swcregistry.io/docs/SWC-120. 2020.

[275] João F. Ferreira et al. "SmartBugs: A Framework to Analyze Solidity Smart Contracts." In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 2020, pp. 1349–1352.

[276] Omer Tripp et al. "TAJ: effective taint analysis of web applications." In: *ACM Sigplan Notices* 44.6 (2009), pp. 87–97.

[277] Jie Zhou et al. "Graph neural networks: A review of methods and applications." In: vol. 1. 2020, pp. 57–81.

[278] MITRE Corporation. *CWE-330: Use of Insufficiently Random Values*. https://cwe.mitre.org/data/definitions/330.html. 2023.

[279] SmartContractSecurity. *Smart Contract Weakness Classification Registry (SWC Registry)*. https://swcregistry.io/. Accessed: 2025-07-15. 2023.

[280] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks." In: *International Conference on Learning Representations (ICLR)*. 2017.

[281] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory." In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[282] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks." In: *IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), pp. 6645–6649.

[283] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." In: *International Conference on Learning Representations (ICLR)* (2015).

PUBLICATIONS

The following publications were produced during my PhD candidature (2022-2025):

[1] Hadis Rezaei, Massimo Ficco, and Francesco Palmieri. "CTAT: A Blockchain-Driven Conditional Traceable Access Token for Enhancing Performance and Security in the Supply Chain of Sensitive Pharmaceuticals." In: *International Conference on Advanced Information Networking and Applications*. Springer. 2025, pp. 149–160.

[2] Hadiseh Rezaei et al. "A Lightweight Blockchain-Based Defense Method for Federated Self-Supervised Learning." In: *Future Generation Computer Systems* (2025), p. 108092.

[3] Hadis Rezaei et al. "SmartTaintRL: Efficient Detection and Localization of Bad Randomness Vulnerabilities in Ethereum Smart Contracts via Reinforcement Learning." In preparation. 2025.

[4] Hadis Rezaei and Francesco Palmieri. "From Symbolic to Semantic: A Critical Analysis of Smart Contract Vulnerability Detection Tools Evolution and Their Real-World Efficacy." In preparation. 2025.

[5] Hadis Rezaei, Rahim Taheri, and Francesco Palmieri. "A Risk-Stratified Benchmark Dataset for Bad Randomness (SWC-120) Vulnerabilities in Ethereum Smart Contracts." In: *arXiv preprint* (2025). Submitted for publication. URL: `https://github.com/HadisRe/BadRandomness-SWC120-Dataset`.