ExploitGen: Automated Smart Contract Vulnerability Verification via Taxonomy-Specialized Multi-Agent Detection and Proof-of-Concept Exploit Generation

by

Aysheh AMINNEZHAD

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, SEPTEMBER 07, 2026

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

M. Kaiwen Zhang, Thesis supervisor
Department of Software Engineering and IT, ÉTS

Mme Camille Coti, Thesis Co-Supervisor
Department of Software Engineering and IT, ÉTS

M. Ulrich Aïvodji, Chair, Board of Examiners
Department of Software Engineering and IT, ÉTS

Mme Valérie Hayot-Sasson, Member of the Jury
Department of Software Engineering and IT, ÉTS

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON JUL 23 2026

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

FOREWORD

My interest in blockchain security developed during my work in protocol development. While deploying smart contracts on the Ethereum mainnet, I observed a recurring problem in decentralized finance: because deployed contracts are largely immutable, even minor flaws can lead to severe and irreversible financial losses. This observation redirected my focus from building decentralized applications toward securing them.

As I moved into security auditing, I spent extended periods manually reviewing protocols and came to recognize a fundamental asymmetry between attacker and defender. Traditional static analysis tools, while valuable, rely on syntactic pattern matching and struggle with the contextual nuances of Solidity semantics. Reviewing their output, I encountered the "detection–verification gap" directly: a high volume of false positives from tools that flag potential issues without establishing whether those issues are genuinely exploitable.

This problem motivated my master's research at the École de technologie supérieure (ÉTS). My initial approach sought to address the gap through detection alone, using a multi-agent system of taxonomy-specialized language-model agents to improve coverage over conventional analyzers. This design achieved high recall, but empirical evaluation revealed that the improvement came at the cost of an excessive false-positive rate: a detector operating on syntactic and semantic resemblance cannot, in principle, determine whether a flagged pattern is exploitable under a contract's runtime semantics. Recognizing this limitation as structural rather than incidental redirected the research from detection toward verification. The resulting system, ExploitGen, treats a flagged candidate not as a finding but as a hypothesis to be tested, confirming exploitability only through an automatically generated proof-of-concept that succeeds on the vulnerable contract and fails on a patched counterpart.

In concluding my Master of Software Engineering, I regard ExploitGen as a contribution toward more reliable decentralized security and a practical foundation for a safer, more resilient ecosystem.

ACKNOWLEDGEMENTS

I thank my supervisor, Professor Kaiwen Zhang, for his guidance, feedback, and support throughout this master's research, which gave me the freedom to explore new directions in automated vulnerability detection.

I also thank my co-supervisor, Professor Camille Coti, for her guidance and feedback throughout this research.

Finally, my appreciation goes to my parents and my brothers. Their encouragement, patience, and support have been foundational to my academic journey and the successful completion of this thesis.

ExploitGen : Vérification Automatisée de Vulnérabilités dans les Contrats Intelligents par Détection Multi-Agents Spécialisée par Taxonomie et Génération Automatique d'Exploits

Aysheh AMINNEZHAD

RÉSUMÉ

Les contrats intelligents constituent l'infrastructure des protocoles de finance décentralisée (DeFi), où les vulnérabilités logicielles entraînent fréquemment des pertes financières irréversibles. Les approches d'audit automatisées existantes souffrent d'un écart critique entre la détection et la vérification : les systèmes multi-agents fondés sur de grands modèles de langage (LLM) atteignent un rappel élevé, mais génèrent des taux de faux positifs prohibitifs qui submergent les auditeurs et annulent les bénéfices attendus de l'automatisation.

Cette thèse présente ExploitGen, un pipeline multi-agents orchestré par LangGraph qui comble cet écart en intégrant une détection spécialisée par taxonomie à une vérification automatisée. Le système déploie d'abord huit agents de détection alignés sur la classification OpenSCV, chacun limité à une seule catégorie afin de maximiser la couverture. Chaque candidat est ensuite soumis à une séquence de vérification en plusieurs étapes : extraction de graphes statiques, validation par falsification, caractérisation sémantique, planification d'attaque structurée (SCoT), synthèse d'une preuve de concept (PoC) exécutable et confirmation contrefactuelle. Une vulnérabilité n'est confirmée que si le test PoC réussit sur le contrat vulnérable et échoue sur une version corrigée préservant la même interface.

Évalué sur le banc d'essai SB Curated, ExploitGen vérifie 86 exploits confirmés par contre-exemple sur 139 vulnérabilités de référence, avec un taux de confirmation de 0,741 et un score F1 de 0,675. Par rapport à une configuration à tentative unique du même pipeline, les candidats non confirmés diminuent de 23,1 %. Par rapport à la détection seule, les signalements non confirmés passent de 342 à 30. Une étude d'ablation montre que la sélection des composants doit être adaptée à chaque catégorie de vulnérabilité plutôt qu'appliquée uniformément.

Mots-clés: contrats intelligents, détection de vulnérabilités, systèmes multi-agents, vérification automatisée, sécurité blockchain

ExploitGen: Automated Smart Contract Vulnerability Verification via Taxonomy-Specialized Multi-Agent Detection and Proof-of-Concept Exploit Generation

Aysheh AMINNEZHAD

ABSTRACT

Smart contracts underpin decentralized finance (DeFi), where software vulnerabilities cause irreversible financial losses. Existing automated auditing approaches face a *detection–verification gap*: static analyzers and single-agent large language model (LLM) systems exhibit low recall, whereas unverified multi-agent architectures achieve high recall at the cost of prohibitive false-positive rates that overwhelm human auditors and negate the efficiency benefits of automation.

This thesis presents ExploitGen, a LangGraph-orchestrated multi-agent pipeline that closes this gap by coupling taxonomy-specialized detection with executable verification. Eight detection agents, each scoped to a single OpenSCV category, first maximize recall. Each candidate then passes through a verification sequence: static graph extraction, falsification-first validation, semantic characterization, Structured Chain-of-Thought (SCoT) attack planning, proof-of-concept (PoC) synthesis, and counterfactual confirmation. A vulnerability is confirmed only when its generated Foundry PoC passes on the vulnerable contract and fails on a signature-preserving patched counterpart, establishing exploitability causally rather than by syntactic resemblance.

Evaluated on the SB Curated benchmark against 139 ground-truth vulnerabilities, ExploitGen verifies 86 counterfactually confirmed exploits at a confirmation yield of 0.741 and an F1-score of 0.675. Against a single-attempt configuration of the same pipeline, unconfirmed candidates fall by 23.1% while confirmed exploits rise from 73 to 86. Against the detection-only paradigm, which uses unscoped parallel detection, unconfirmed findings fall from 342 to 30. A controlled ablation shows that SCoT planning improves reentrancy F1 by 0.200 but degrades access-control performance, establishing that component selection must be adapted to the vulnerability class. The augmented dataset of 139 patched contracts is released to support reproducibility.

Keywords: smart contracts, vulnerability detection, multi-agent systems, automated exploit generation, counterfactual verification

TABLE OF CONTENTS

Page

INTRODUCTION 1

CHAPTER 1 BACKGROUND 9
1.1 Blockchain Technology and Smart Contracts 9
1.2 The DeFi Ecosystem and Security Stakes 10
1.3 Smart Contract Vulnerability Landscape 11
1.3.1 Reentrancy 11
1.3.2 Arithmetic Issues 11
1.3.3 Access Control 11
1.3.4 Unchecked Low-Level Calls 12
1.3.5 Additional Categories 12
1.4 The OpenSCV Vulnerability Taxonomy 12
1.5 Limitations of Traditional Analysis Methods 14
1.5.1 Static Analysis 14
1.5.2 Dynamic Analysis and Fuzzing 14
1.5.3 Formal Verification 15
1.5.4 Confirming Exploitability Requires Executable PoCs 15
1.6 Large Language Models for Program Analysis 15
1.7 Multi-Agent Systems for Code Analysis 16
1.8 Foundry and EVM Testing Environments 17
1.9 Chapter Summary 17

CHAPTER 2 RELATED WORK 19
2.1 Traditional Smart Contract Auditing Tools 19
2.2 Automated Exploit Generation for Smart Contracts 20
2.3 AI-Driven Smart Contract Security 22
2.3.1 Neural Pattern-Based Detection 22
2.3.2 Multi-Agent LLM Detection 22
2.3.3 GPT-Augmented Static Analysis 23
2.4 Invariant Inference and Repair 23
2.5 Differential Testing and Counterfactual Verification 23
2.6 Positioning ExploitGen 24
2.6.1 Patch-Differential Proof-of-Concept Generation 25
2.7 Chapter Summary 26

CHAPTER 3 PRELIMINARY STUDY: EXPLOITGEN-DETECTION AND THE DETECTION–VERIFICATION GAP 27
3.1 Design of ExploitGen-Detection 27
3.2 Evaluation on SB Curated 27
3.3 Analysis of the Detection–Verification Gap 29

3.4 Why AutoGPT is Insufficient for Verification 30
3.5 Requirements for a Verification Pipeline 31
3.6 Chapter Summary 32

CHAPTER 4 METHODOLOGY 33
4.1 Pipeline Overview 33
4.2 Stage 1: Taxonomy-Specialized Vulnerability Detection 35
4.3 Stage 2: Static Graph Extraction 36
4.4 Stage 3: Falsification-First Validation 37
4.5 Stage 4: Semantic Vulnerability Characterization 38
4.6 Stage 5: SCoT-Based Attack Planning 39
4.7 Stage 6: Exploit Generation and Iterative Repair 40
4.8 Stage 7: Counterfactual Confirmation and Verdict Assignment 41
4.9 Formal Definitions and Correctness Properties 42
4.10 Chapter Summary 43

CHAPTER 5 IMPLEMENTATION 45
5.1 System Architecture 45
5.2 Language Model Configuration 45
5.3 Workspace and Environment Management 47
5.4 Exploit Synthesis, Repair, and Confirmation 48
5.5 Dataset Augmentation: Patched Contract Generation 48
5.6 Data and Code Availability 49
5.7 Chapter Summary 50

CHAPTER 6 EVALUATION 51
6.1 Experimental Setup 51
6.1.1 Benchmark 51
6.1.2 Evaluation Metrics 51
6.1.3 Ablation Configurations 52
6.1.4 Baselines 53
6.2 RQ1: End-to-End Pipeline Effectiveness 53
6.3 RQ2: Component Contribution via Ablation Study 55
6.4 RQ3: Computational Cost and Runtime Profile 57
6.5 Threats to Validity 60
6.5.1 Baseline Comparability 60
6.5.2 Dataset Scale 60
6.5.3 Single-Run Ablation and Model Non-determinism 60
6.5.4 Detection vs. Generation Count Discrepancy 61
6.5.5 Execution Model Constraints 61
6.5.6 Cross-Generation Sensitivity and Generalizability 61
6.5.7 Metric Definition 62
6.5.8 Adversarial Inputs and Prompt Injection 62
6.5.9 Data Contamination and Benchmark Memorization 63

6.5.10 Oracle Category Scoping 63
6.6 Chapter Summary 64

CHAPTER 7 DISCUSSION 65
7.1 Interpreting End-to-End Results 65
7.2 Component Interaction Effects 66
7.3 The Detection–Verification Gap: Quantified Impact 66
7.4 Limitations of the Current Approach 67
7.4.1 Single-Transaction Execution Model 67
7.4.2 Benchmark Scale and Distribution 67
7.4.3 Component Calibration 67
7.4.4 Dependence on Patch Correctness 68
7.4.5 Contract Complexity and Dependency Scope 68
7.5 Implications for Practical Deployment 69
7.6 Chapter Summary 69

CONCLUSION AND RECOMMENDATIONS 71

LIST OF REFERENCES 75

LIST OF TABLES

Page

Table 1.1 Mapping between DASP evaluation categories and OpenSCV root causes 13

Table 2.1 Comparison of related work across key dimensions 25

Table 3.1 Detection performance of ExploitGen-Detection on SB Curated 29

Table 3.2 Per-category detections of ExploitGen-Detection 30

Table 5.1 Model bound to each tier per configuration 46

Table 6.1 Full pipeline performance per vulnerability category 53

Table 6.2 Ablation results per configuration and vulnerability category 56

Table 6.3 Category-adaptive component recommendations from ablation 58

Table 6.4 Pipeline wall-clock time and estimated API cost 58

Table 6.5 Cross-generation ablation 62

LIST OF FIGURES

Page

Figure 3.1 Architecture of ExploitGen-Detection 28

Figure 4.1 Architecture of the ExploitGen pipeline 35

LIST OF ALGORITHMS

Page

Algorithm 4.1 ExploitGen Top-Level Pipeline 34

Algorithm 4.2 SynthesizeAndConfirm: Stages 6–7 41

LIST OF ABBREVIATIONS

AC	Access Control (vulnerability category)
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CoT	Chain-of-Thought
DASP	Decentralized Application Security Project
DeFi	Decentralized Finance
DFG	Data Flow Graph (Data Dependency Graph)
DoS	Denial of Service (vulnerability category)
EVM	Ethereum Virtual Machine
F1	F1-Score (harmonic mean of precision and recall)
FN	False Negative
FP	False Positive
JSON	JavaScript Object Notation
LLM	Large Language Model
MAS	Multi-Agent System
OpenSCV	Open Smart Contract Vulnerability (taxonomy)
PoC	Proof-of-Concept
RE	Reentrancy (vulnerability category)
ReAct	Reasoning and Acting (LLM paradigm)

SB	SmartBugs (benchmark suite)
SCoT	Structured Chain-of-Thought
SMT	Satisfiability Modulo Theories
SWC	Smart Contract Weakness Classification
TN	True Negative
TP	True Positive
UC	Unchecked Low-Level Calls (vulnerability category)

LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

P	Precision: $\|\text{TP}\|/(\|\text{TP}\| + \|\text{FP}\|)$
R	Recall: $\|\text{TP}\|/(\|\text{TP}\| + \|\text{FN}\|)$
F_1	F1-Score: $2PR/(P + R)$
C	A smart contract under analysis
C^*	Counterfactually patched version of C
$\mathcal{V}$	Set of vulnerability candidates produced by Stage 1
v_i	Individual vulnerability candidate $i \in \mathcal{V}$
$\mathcal{T}$	Generated proof-of-concept Foundry test
γ	Verdict function: $\{exploit\_proven, inconclusive, false\_positive\}$
k	Per-vulnerability retry budget (set to 5)
$\mathcal{G}_i$	CFG for contract function i
$\mathcal{D}_i$	Data-dependency graph for contract i

INTRODUCTION

Context and Motivation

Blockchain-based smart contracts have evolved from experimental financial instruments into the primary infrastructure of a multi-trillion-dollar decentralized finance (DeFi) ecosystem. Unlike traditional software, smart contracts are deployed to a public, immutable ledger where they autonomously execute financial logic, manage digital assets, and enforce protocol rules without human intermediaries. This combination of immutability, public accessibility, and financial stake creates a uniquely hostile adversarial environment: a single exploited vulnerability can result in permanent, irreversible loss of user funds at a scale that has no parallel in conventional software systems.

The severity of this threat has been demonstrated repeatedly. Cumulative losses attributable to smart contract vulnerabilities exceed ten billion dollars DefiLlama (2025), with recurring attack patterns including reentrancy, arithmetic overflow, and access control failures Vidal, Ivaki & Laranjeiro (2024). The 2016 DAO attack drained 3.6 million Ether by exploiting a reentrancy vulnerability and precipitated a contentious hard fork of the Ethereum blockchain. More recent incidents confirm that the attack surface continues to expand in step with DeFi protocol complexity.

The central challenge in smart contract security is achieving accurate, efficient auditing at scale. Because deployed contracts cannot be patched, vulnerabilities must be identified prior to deployment or, at a minimum, prior to significant user adoption. Manual auditing by security experts remains the current gold standard but is prohibitively expensive and non-scalable given the volume of contracts deployed daily on major blockchain networks.

Automated tools address the scalability problem but introduce a fundamental accuracy trade-off. Static and symbolic-execution analyzers such as Slither Feist, Grieco & Groce (2019) and

Mythril Mueller (2018) identify patterns associated with known vulnerability classes but cannot evaluate runtime state invariants, inheritance hierarchies, or dynamic guards, resulting in high false-positive and false-negative rates that limit practical utility Hamiaz & Driss (2025).

Large language model (LLM) based approaches have demonstrated superior semantic reasoning, but single-agent systems lack the task specialization required to cover all vulnerability categories systematically Chen *et al.* (2025); Tamberg & Bahsi (2025). Recent multi-agent architectures that decompose the auditing task across specialized agents have shown substantially improved detection recall Wei *et al.* (2025); Jin *et al.* (2025), but this recall improvement is purchased at the cost of catastrophic precision loss.

This thesis demonstrates empirically that an eight-agent detection system aligned to the OpenSCV vulnerability taxonomy achieves 93% recall while generating 342 false positives alongside 306 true positives, for a precision of only 0.47. OpenSCV (Open Smart Contract Vulnerability) is an open, hierarchical taxonomy that organizes known smart contract weaknesses into well-defined categories, among them reentrancy, arithmetic faults, access control failures, and unchecked low-level calls. It was constructed by consolidating the established but now largely outdated classification schemes, notably the SWC Registry SmartContractSecurity (2020) and the DASP Top 10 NCC Group (2018), and validated through expert review, providing a current and community-maintained reference for vulnerability research Vidal *et al.* (2024). The benchmark used in this thesis is annotated at DASP granularity, which is finer than OpenSCV's root-cause level; the eight detection agents are therefore aligned to the eight DASP categories present in SB Curated and grouped under OpenSCV root causes for analysis, as set out in Table 1.1. Scoping each agent to a single category gives it a narrow, well-defined hypothesis space, which is what enables the high recall reported above; because two pairs of DASP categories share a root cause, the agent scopes are not strictly disjoint, and the resulting duplicate reports are discussed in Section 3.3. The root cause of the accompanying precision failure is the *detection–*

verification gap: LLM-based detection identifies code patterns that *resemble* vulnerabilities but fundamentally cannot determine whether a flagged pattern is *actually exploitable* under the contract's runtime semantics.

Problem Statement

The detection–verification gap has four structural dimensions:

1. Detection systems identify code patterns that resemble vulnerabilities. They cannot determine whether a flagged instance is genuinely exploitable given its complete execution context, including inherited modifiers, Solidity version semantics, and protocol-level invariants.
2. Verification of exploitability requires an executable PoC demonstrating causation: the exploit must cause a specific state change, and a patched version of the contract must prevent that change.
3. Automated PoC generation for smart contracts is non-trivial due to environmental complexity (contract dependency graphs), the logical causality gap between test assertions and exploit success, and LLM syntactic hallucination in code synthesis.
4. Existing multi-agent auditing systems that omit executable verification produce false-positive rates that overwhelm human auditors, negating the efficiency benefits of automation and eroding practitioner trust in AI-assisted tools.

Proposed Approach: ExploitGen

This thesis introduces ExploitGen, a system that closes the detection–verification gap by reframing smart contract analysis around *demonstrated exploitability* rather than pattern detection alone. ExploitGen is a LangGraph-orchestrated multi-agent pipeline that couples taxonomy-specialized detection with automated exploit verification. Detection is performed by eight category-specialized agents, each aligned with a single OpenSCV vulnerability class, which

together surface candidate vulnerabilities at high recall. Every candidate is then treated not as a finding but as a hypothesis to be tested: it is passed through a verification sequence comprising static graph extraction, falsification-first validation, semantic characterization, Structured Chain-of-Thought (SCoT) attack planning, proof-of-concept (PoC) synthesis, and counterfactual confirmation.

The defining criterion is counterfactual: a candidate is reported as a vulnerability only when an automatically generated Foundry PoC succeeds against the vulnerable contract C and fails against a signature-preserving patched counterpart C^*. This establishes exploitability *causally*, the exploit provably depends on the flaw, not merely on code that resembles one, which is precisely the determination that detection-only systems cannot make. In effect, verification converts the detector's large set of plausible-looking candidates into a small set of demonstrated, auditor-actionable exploits, directly targeting the false-positive burden identified above.

ExploitGen is therefore central to this thesis: it is the artifact through which each research objective is pursued and against which each research question is answered. The remainder of this chapter states those objectives and questions, summarizes the contributions the system enables, and outlines the organization of the thesis.

Research Objectives

This thesis pursues four primary research objectives:

RO1. Design and implement a multi-agent pipeline integrating taxonomy-specialized vulnerability detection with automated, execution-backed PoC verification, improve precision by an order of magnitude at a bounded, measured cost in recall, and characterise that cost.

RO2. Develop and evaluate a causal reasoning framework for LLM-based exploit synthesis that forces explicit articulation of exploit preconditions before code generation, and assess its category-dependent performance characteristics.

RO3. Establish the first broad-taxonomy evaluation of counterfactual differential verification across multiple OpenSCV vulnerability categories, quantifying the precision-recall trade-off introduced by each pipeline component.

RO4. Release an augmented evaluation dataset of counterfactually patched smart contracts to support reproducibility and future evaluation of verification-backed auditing systems.

Research Questions

Three research questions guide the empirical evaluation:

RQ1 (Effectiveness). To what extent does ExploitGen's integrated detection-and-verification pipeline improve precision and F1-score relative to detection-only baselines, with static-analysis performance provided as external context.

RQ2 (Component Analysis). What is the individual contribution of each pipeline component, SCoT-based attack planning, falsification-first validation, CFG/DFG graph grounding, and the Critic repair loop, to the overall F1-score, and how do these contributions vary by vulnerability category?

RQ3 (Cost Profile). What is the computational cost and runtime profile of the full pipeline relative to ablated configurations, and how does the cost-effectiveness ratio vary by vulnerability category?

Contributions

This thesis makes four primary contributions to automated smart contract security:

1. **A Unified Taxonomy-Specialized Architecture.** The design and implementation of ExploitGen, a LangGraph-orchestrated multi-agent pipeline that scales automated exploit verification across eight DASP vulnerability categories spanning six OpenSCV root causes, using deterministic state-managed verification graphs to overcome the unreliability of loop-based autonomous agents in strict auditing environments.
2. **Broad-Taxonomy Counterfactual Verification.** The first application of differential exploit verification, requiring each PoC to pass on the vulnerable contract and fail on a patched counterpart, across a broad smart-contract vulnerability taxonomy, covering six categories for which a signature-preserving counterfactual is constructible, and reducing unconfirmed synthesis-stage candidates by 23.1% over a single-attempt configuration of the same pipeline.
3. **Empirical Analysis of SCoT Reasoning Trade-offs.** A rigorous ablation study revealing that while SCoT attack planning improves F1 for multi-step exploits (reentrancy: +0.200), it actively degrades performance for interface-constrained vulnerabilities (access control: −0.121), establishing that LLM prompt scaffolding must be category-dependent rather than uniformly applied.
4. **Augmented Evaluation Dataset.** An extension of the SB Curated benchmark Durieux, Ferreira, Abreu & Cruz (2020); SmartBugs Project (2024) with 139 counterfactually patched contracts produced using a minimal-patch principle, released to support reproducibility and future evaluation of verification-backed auditing systems.

Thesis Organization

Chapter 1 provides background on smart contracts, the Ethereum ecosystem, the DeFi security landscape, vulnerability taxonomies, static and dynamic analysis primitives, LLMs for program

analysis, and the Foundry testing framework. Chapter 2 surveys related work. Chapter 3 presents ExploitGen-Detection, the detection-only preliminary system that empirically establishes the cost of operating without exploit verification. Chapters 4 and 5 describe the ExploitGen pipeline design and implementation. Chapter 6 presents the experimental evaluation. Chapter 7 interprets results and discusses limitations. The conclusion summarizes contributions and directions for future research.

CHAPTER 1

BACKGROUND

This chapter provides the technical foundations required to understand the problem addressed and the solution presented in this thesis. Sections are organized from the general (blockchain and smart contracts) to the specific (LangGraph orchestration and Foundry exploit testing).

1.1 Blockchain Technology and Smart Contracts

A blockchain is a distributed ledger maintained by a peer-to-peer network in which transactions are grouped into cryptographically linked blocks. Consensus among nodes ensures that the ledger's history is append-only and tamper-evident without the need for a trusted central authority. Bitcoin (2008) established blockchain as a viable technology for decentralized value transfer. Ethereum (2015) generalized this model by introducing a Turing-complete virtual machine, the Ethereum Virtual Machine (EVM), capable of executing arbitrary programs stored on the ledger Wood (2014).

Smart contracts are programs deployed to the EVM at a specific address. Once deployed, a contract's bytecode is immutable: it cannot be updated, deleted, or patched. Contracts are activated by transactions sent to their address, execute deterministically across all nodes, and can hold and transfer Ether (the native cryptocurrency) or ERC-20 tokens. Solidity, a statically typed language that compiles to EVM bytecode, is the predominant language for Ethereum smart contract development Ethereum Foundation (2024).

Three properties of the EVM execution model have direct security implications:

Immutability. Deployed code cannot be patched. A vulnerability that is exploited post-deployment cannot be remedied without deploying a new contract and migrating all state, a process that is costly, disruptive, and often infeasible for production protocols.

Determinism. All nodes must arrive at identical execution outcomes. This constrains the randomness model and makes the global state machine an attractive analysis target for formal methods.

Financial Criticality. Contracts routinely control significant assets. The DeFi ecosystem had a total value locked (TVL) of over $100B at its peak, with individual protocol contracts managing hundreds of millions of dollars.

1.2 The DeFi Ecosystem and Security Stakes

Decentralized finance (DeFi) refers to financial services, lending, borrowing, trading, and yield generation — implemented as smart contract / protocols on public blockchains. DeFi protocols interact through composable interfaces: a lending protocol may integrate a decentralized exchange, which integrates a price oracle, creating dependency graphs that dramatically expand the attack surface beyond a single contract's source code.

The financial consequences of smart contract vulnerabilities are well-documented. Reentrancy attacks alone have caused losses exceeding $3.6B historically (DAO, 2016; Cream Finance, 2021; Euler Finance, 2023). Access control failures enabled the Ronin Bridge hack ($625M, 2022) and the Wormhole exploit ($320M, 2022). Arithmetic vulnerabilities underlie numerous token manipulation attacks. These losses are irreversible by design: the blockchain ledger records them permanently, and recovery requires out-of-band negotiation with attackers (occasionally successful in high-profile cases) or insurance mechanisms.

The asymmetry between attacker and defender is extreme. An attacker needs to find a single exploitable vulnerability in a publicly available contract. A defender must identify and remediate all vulnerabilities before deployment or attract sufficient auditing resources to close the gap. This asymmetry motivates investment in automated auditing tools.

1.3 Smart Contract Vulnerability Landscape

Smart contract vulnerabilities arise from the unique characteristics of blockchain execution. Unlike conventional software vulnerabilities (buffer overflows, injection attacks), smart contract vulnerabilities typically exploit the contract's own logic operating within the EVM's semantics. The following categories are most prevalent and financially consequential.

1.3.1 Reentrancy

Reentrancy occurs when an external call to an untrusted contract allows that contract to re-enter the calling function before the initial execution completes and state is updated. The classic pattern involves a withdrawal function that sends Ether before updating the caller's balance. An attacker contract implements a fallback function that recursively calls the withdrawal, draining the victim's balance beyond the intended amount. The DAO attack (2016) is the canonical example.

1.3.2 Arithmetic Issues

Prior to Solidity 0.8.0, integer arithmetic in smart contracts was unchecked by default, allowing overflow (exceeding the maximum value of a type) and underflow (going below zero for unsigned types) without runtime errors. An attacker can exploit a subtraction underflow to wrap an unsigned integer from zero to $2^{256} - 1$, effectively granting themselves an arbitrarily large token balance. Solidity 0.8+ introduced built-in overflow/underflow checks that revert on violation, substantially mitigating this category for newly deployed contracts.

1.3.3 Access Control

Access control vulnerabilities arise when a function that should be restricted to authorized callers (e.g., contract owner, protocol governance) is callable by arbitrary addresses. Common patterns include missing `onlyOwner` modifiers on administrative functions, unprotected initializer

functions callable post-deployment, and `tx.origin`-based authentication that can be bypassed by a phishing contract.

1.3.4 Unchecked Low-Level Calls

The EVM provides low-level call instructions (`call`, `delegatecall`, `send`) that return a boolean indicating success rather than reverting on failure. Contracts that do not check these return values may silently proceed after a failed external call, leading to incorrect state or fund loss. This pattern is particularly common in contracts predating the `revert` keyword.

1.3.5 Additional Categories

The OpenSCV taxonomy also identifies denial of service (DoS), front-running (transaction ordering manipulation), bad randomness (predictable pseudorandom values from miner-controlled fields), and time manipulation (dependence on `block.timestamp` for critical logic) as vulnerability categories Vidal *et al.* (2024). These categories present distinct exploitability characteristics and, as shown in this thesis, distinct challenges for automated PoC generation.

1.4 The OpenSCV Vulnerability Taxonomy

SB Curated is annotated using the DASP scheme rather than OpenSCV, so the operational category set used throughout this thesis is DASP, mapped onto OpenSCV root causes as shown in Table 1.1. The mapping is not one-to-one in two places: reentrancy and unchecked low-level calls both derive from unsafe external calls, and front-running and time manipulation both derive from incorrect control flow. Agents are scoped at DASP granularity to match the benchmark's annotations and to keep per-category measurement well defined; the two collisions are the source of the overlapping detections reported in Section 3.3.

Multiple vulnerability classification systems exist for smart contracts, including SWC (Smart Contract Weakness Classification) SmartContractSecurity (2020) and DASP (Decentralized Application Security Project) NCC Group (2018). This thesis adopts the OpenSCV taxonomy

Table 1.1 Mapping between the DASP categories used in the evaluation and the OpenSCV root causes they derive from

DASP category (agent scope)	OpenSCV root cause
Reentrancy	(1) Unsafe external calls
Unchecked low-level calls	(1) Unsafe external calls
Denial of service	(2) Denial of service
Bad randomness	(5) Bad practices and weak points
Front-running	(6) Incorrect control flow
Time manipulation	(6) Incorrect control flow
Arithmetic	(7) Arithmetic issues
Access control	(8) Improper access control

Vidal *et al.* (2024), which organizes 94 vulnerability types into eight root-cause categories based on underlying semantic mechanisms rather than surface-level consequences.

The eight OpenSCV root-cause categories are: (1) unsafe external calls (reentrancy, unchecked returns); (2) denial of service; (3) unsafe credit management; (4) improper cryptographic verification; (5) bad practices and weak points (poor randomness); (6) incorrect control flow (front-running, time manipulation); (7) arithmetic issues (overflow, underflow); and (8) improper access control.

OpenSCV is adopted over SWC and DASP for three reasons. First, its root-cause organization provides clean semantic category boundaries that enable category-specific prompt scoping for detection agents. Second, its hierarchical structure allows precise identification of the vulnerability mechanism rather than the observed consequence. Third, its current maintenance and comprehensive coverage of the vulnerability landscape make it the most appropriate foundation for a research system intended to generalize beyond historically catalogued attacks.

1.5 Limitations of Traditional Analysis Methods

1.5.1 Static Analysis

Static analyzers operate on abstract syntax trees (ASTs) or control flow graphs (CFGs) to identify syntactic patterns associated with known vulnerability classes. Tools in this class include Slither Feist *et al.* (2019) and Securify Tsankov *et al.* (2018). A related family of symbolic-execution tools, including Mythril Mueller (2018) and Oyente, instead explores execution paths to reason about reachable states. These tools are computationally efficient and have broad coverage of known patterns, but suffer from two fundamental limitations.

False positives arise because syntactic resemblance to a vulnerability pattern does not imply exploitability. A reentrancy-shaped call sequence protected by a `ReentrancyGuard` modifier is indistinguishable from a genuine reentrancy at the AST level. An arithmetic operation that will never overflow due to domain-specific invariants appears identical to one that might. This limitation is structural, not a calibration issue: static analyzers cannot evaluate runtime state invariants.

False negatives arise because novel vulnerability patterns or complex inter-contract interactions that are not captured by existing rules are invisible to pattern-matching tools. As DeFi protocol complexity increases, this limitation becomes more severe.

1.5.2 Dynamic Analysis and Fuzzing

Dynamic analysis executes the contract under test with concrete inputs and observes behavior. Fuzzing tools such as Echidna Grieco, Song, Cygan, Feist & Groce (2020) generate random or coverage-guided inputs to trigger violations of user-specified invariants. Dynamic analysis provides execution context that static tools lack but has two limitations: (1) high computational cost for complex state-dependent logic requiring long initialization sequences; and (2) the difficulty of synthesizing the specific input sequences that trigger vulnerabilities in realistic contract environments.

1.5.3 Formal Verification

Formal verification tools such as VeriSol and K-EVM provide the strongest correctness guarantees by mathematically proving that a contract satisfies a formal specification. However, these approaches require exhaustive manual specification writing, rendering them cost-prohibitive for the rapid auditing cycles required in practice Kezadri Hamiaz & Driss (2025). The specification burden scales with contract complexity and requires domain expertise beyond most development teams.

1.5.4 Confirming Exploitability Requires Executable PoCs

These limitations motivate a common conclusion: confirming that a flagged pattern is genuinely exploitable under a specific contract's runtime semantics requires an executable proof-of-concept test that demonstrates exploitation in a realistic EVM environment. The survey of verification tools by Kezadri Hamiaz and Driss Kezadri Hamiaz & Driss (2025) documents the underlying tool limitations on which this conclusion rests. The generation of such tests is the central technical problem addressed in this thesis.

1.6 Large Language Models for Program Analysis

Large language models (LLMs) are neural language models trained on large corpora of text and code, capable of performing zero-shot and few-shot reasoning about program behavior Chen *et al.* (2025). For smart contract analysis, LLMs offer three capabilities that static analysis tools lack: semantic understanding of function purpose, ability to reason about multi-step execution sequences, and generalization to novel vulnerability patterns not captured by existing rules.

The Structured Chain-of-Thought (SCoT) prompting paradigm Li, Li, Li & Jin (2025) augments LLM reasoning by requiring the model to produce structured intermediate reasoning steps before generating a final output. In the context of exploit synthesis, SCoT forces the model to explicitly specify attack preconditions, execution sequences, and verification assertions before writing Solidity code, reducing the performance degradation that arises when planning and code

generation are conflated. SCoT has been shown to outperform standard CoT by up to 13.79% in Pass@1 on code generation benchmarks Li *et al.* (2025).

The ReAct (Reasoning and Acting) paradigm Yao *et al.* (2023) enables LLM agents to interleave reasoning traces with actions (tool calls, code execution) and to update their reasoning based on environment feedback. In ExploitGen, the Critic agent implements ReAct by treating forge execution output as environment feedback and producing revised attack blueprints as actions.

1.7 Multi-Agent Systems for Code Analysis

Multi-agent systems (MAS) decompose a complex task across multiple specialized agents, each operating on a narrowly scoped subtask. For smart contract auditing, the motivation for MAS is clear: no single agent can simultaneously maintain high precision and recall across eight semantically distinct vulnerability categories, because prompts optimized for one category (e.g., reentrancy, requiring callback analysis) interfere with detection of another (e.g., access control, requiring modifier inspection). Category-specialized agents allow each prompt to concentrate analytical attention on a well-defined hypothesis space Wei *et al.* (2025); Jin *et al.* (2025). We use an LLM agent rather than a static or dynamic detector at this stage for two reasons. First, detection here serves recall, not precision: its role is to surface every plausible candidate for the verification stage to adjudicate, and LLM agents achieve higher recall than pattern-based static tools on this benchmark (Chapter 3), while their characteristic weakness,false positives, is absorbed by the downstream counterfactual filter rather than propagated to the auditor. Second, a uniform LLM interface lets the same structured context (source, CFG, data-dependency graphs) and the same typed hand-off feed all eight categories, which a heterogeneous mix of category-specific static and dynamic tools could not provide without bespoke integration per category. This does not preclude hybridization: for categories with mature, high-precision static signatures, notably Solidity 0.8+ arithmetic overflow, which the compiler itself now guards, a static pre-filter could cheaply discharge candidates before the LLM stage, and the ablation of Section 6.3, which shows graph grounding helps non-local categories but not localized

arithmetic faults, points to exactly where such substitution would be most beneficial. We leave category-adaptive detection back-ends as future work.

LangGraph LangChain (2024) is a framework for building stateful, graph-structured multi-agent workflows as directed computation graphs. Unlike autonomous loop-based agents (e.g., AutoGPT Fezari & Ali-Al-Dahoud (2023)), LangGraph exposes explicit state management, typed inter-node message passing, conditional routing via edge functions, and native support for back-edges enabling iterative repair loops. These features are essential for the structured verification pipeline described in Chapter 4.

1.8 Foundry and EVM Testing Environments

Foundry is a Solidity-native smart contract development and testing framework that provides: `forge` (build and test runner), `anvil` (local EVM node), and `cast` (command-line EVM interface). Foundry test cases are written in Solidity and executed by `forge test`, which spawns an `anvil` instance providing accurate EVM simulation including gas costs, transaction boundaries, and state semantics that mirror mainnet behavior.

Foundry is adopted as the exploit execution environment in ExploitGen for three reasons. First, Solidity-native tests eliminate the impedance mismatch between the LLM-synthesized attack contracts and the test harness. Second, `anvil` provides deterministic EVM semantics that enable counterfactual testing against patched contracts. Third, Foundry's `forge` output is structured and classifiable, enabling the Critic agent to generate root-cause diagnoses from typed outcome labels.

1.9 Chapter Summary

This chapter has established the technical foundations for the thesis: the immutable, financial-critical nature of smart contracts; the principal vulnerability categories and their exploitability characteristics; the limitations of static, dynamic, and formal analysis methods; the capabilities and limitations of LLM-based program analysis; and the orchestration and execution tools used

in the ExploitGen pipeline. The following chapter surveys how existing research has addressed the smart contract auditing problem.

CHAPTER 2

RELATED WORK

This chapter surveys the literature on automated smart contract auditing across four paradigms: traditional static and dynamic tools, AI-driven detection, automated exploit and repair systems, and differential testing approaches. Each section concludes with a critical assessment of the paradigm's limitations relative to the problem addressed in this thesis.

2.1 Traditional Smart Contract Auditing Tools

Traditional non-AI methods inspect code without execution or use concrete inputs to trigger failures. Slither Feist *et al.* (2019) applies taint analysis and control flow inspection to identify reentrancy, unchecked calls, and other patterns, offering fast, lightweight detection suitable for CI integration. Mythril Mueller (2018) uses symbolic execution to explore execution paths and detect assertion violations. Securify Tsankov *et al.* (2018) encodes compliance and violation patterns as Datalog rules evaluated over a semantic representation of the contract. Echidna Grieco *et al.* (2020) employs property-based fuzzing with coverage guidance to trigger user-defined invariant violations.

These tools share a structural limitation: they operate on fixed rule sets encoding known vulnerability patterns. They cannot reason about the compositional semantics of novel DeFi protocol interactions, fail to leverage contextual information about caller behavior, and cannot confirm that a flagged pattern is actually exploitable under the contract's runtime semantics Kezadri Hamiaz & Driss (2025). On the SB Curated benchmark, Slither attains a detection recall of 0.611 at a precision of 0.061 when all severity levels are counted as detections, for an F1 of 0.111 (Chapter 3). It is used in this thesis as an indicative external reference rather than as a controlled comparison.

2.2 Automated Exploit Generation for Smart Contracts

A distinct line of work generates executable attacks rather than warnings. Krupp and Rossow Krupp & Rossow (2018) propose teEther, which operates on EVM bytecode alone, without source code or user-supplied specifications. It recovers a control-flow graph, locates *critical paths* reaching one of four value-transferring or code-injecting instructions (`CALL`, `SELFDESTRUCT`, `CALLCODE`, `DELEGATECALL`) with an attacker-controllable argument, and prepends *state-changing paths* containing an `SSTORE` where a vulnerable state must first be reached. Symbolic execution over the combined path sequence yields constraints that a solver turns into a concrete transaction sequence transferring Ether to an attacker-controlled address. Nikolić et al. Nikolić, Kolluri, Sergey, Saxena & Hobor (2018) propose MAIAN, which characterises three classes of *trace* vulnerability, prodigal, suicidal and greedy, as properties of a sequence of contract invocations rather than of a single call, and detects them by inter-procedural symbolic analysis of bytecode. Prodigal and suicidal candidates are confirmed concretely by replaying the generated transactions on a private fork of the Ethereum chain; greedy is a liveness property and cannot be confirmed that way, so it is validated by a separate bytecode check instead. Concrete validation is applied to a sample rather than to every flagged contract: MAIAN flags 34,200 contracts and validates 3,759 of them, at an 89% true-positive rate.

Later work moves from symbolic search to fuzzing. ETHPLOIT Zhang, Wang, Li & Ma (2020) takes Solidity source, applies static taint analysis built on Slither to constrain which transaction sequences are worth generating, and fuzzes them against an instrumented EVM that can set block properties and force external calls to throw, addressing precisely the constraints, such as hash checks, that defeat solver-based tools. SMARTIAN Choi *et al.* (2021) returns to bytecode and uses static data-flow analysis to seed promising transaction orders, then dynamic def-use coverage as fuzzing feedback, on the grounds that branch coverage alone cannot distinguish transaction sequences that change persistent state in a meaningful way.

These systems establish that executable confirmation is achievable for smart contracts and, like ExploitGen, decline to report a finding without a demonstration. Two things separate them from the present work.

The first is what defines success. Each encodes its success condition as a fixed, tool-defined oracle: for teEther and MAIAN, reaching a value-transferring or self-destructing instruction under attacker control; for the fuzzers, a catalogue of built-in detectors, of which SMARTIAN implements thirteen, including mishandled exceptions, block-state dependency and reentrancy Choi *et al.* (2021). Coverage is therefore broad but closed: extending it to a vulnerability class the tool does not already model requires implementing a new oracle inside the tool. In ExploitGen the success condition is derived per candidate from the detected vulnerability and expressed as assertions in the generated test. This trades the completeness and termination guarantees of solver-based search for extensibility across a taxonomy, at the cost of a probabilistic synthesiser.

The second is what confirmation establishes. In all of these systems the generated input is executed against the vulnerable contract alone, and a positive result means the built-in oracle fired; none compares behaviour against a patched counterpart. Execution against a single contract does not by itself eliminate false positives: teEther reports that 5.71% of its `CALL`-based and 9.69% of its `SELFDESTRUCT`-based exploits fail once re-derived against the contract's actual storage Krupp & Rossow (2018), MAIAN reports an 89% true-positive rate over the sample it validated concretely Nikolić *et al.* (2018), and SMARTIAN reports 8 false positives among 219 alarms in its large-scale study Choi *et al.* (2021); ETHPLOIT reports none on its dataset Zhang *et al.* (2020). The counterfactual criterion adopted here is strictly stronger than single-contract execution: a test that fires on the vulnerable contract but also fires on a minimally patched counterpart is rejected, whichever oracle produced it.

2.3 AI-Driven Smart Contract Security

2.3.1 Neural Pattern-Based Detection

Early AI-driven approaches apply learned representations to detect vulnerability patterns in contract code. Krichen Krichen (2023) applies graph neural networks to contract ASTs. Multimodal deep learning pipelines Deng *et al.* (2023) exploit both structural and semantic representations to uncover complex vulnerability patterns. These approaches demonstrate superior recall over rule-based tools but inherit the same structural limitation: they identify patterns that resemble vulnerabilities without determining exploitability.

2.3.2 Multi-Agent LLM Detection

Multi-agent orchestration extends LLM-based detection by decomposing the auditing task across specialized agents. LLM-BSCVM Jin *et al.* (2025) coordinates specialized agents across vulnerability categories and achieves high detection F1 on standard benchmarks. Wei et al. Wei *et al.* (2025) propose LLM-SmartAudit, a conversational multi-agent framework using a buffer-of-thought mechanism that achieves 98% accuracy on a common-vulnerability benchmark and identifies 12 of 13 CVEs. SmartGuard Ding, Liu, Piao, Song & Ji (2025) combines in-context learning with self-generated chains of thought, achieving 95.06% recall on the SolidiFI benchmark. Recently, Alam et al. Alam, Paswan, Halder & Maiti (2026) introduced AutoSCAR, an agentic framework that explicitly separates vulnerability detection from validation. By employing a multi-agent adjudication process (comprising a Debator, Skeptic, and Judge) constrained by extracted semantic context such as control-flow graphs and state invariants, AutoSCAR significantly improves precision over raw static analysis and guides automated repair.

Despite these results, the majority of multi-agent detection systems share a structural limitation: they identify code that *resembles* a vulnerability without determining whether the flagged instance is *actually exploitable*. While recent systems like AutoSCAR attempt to bridge this gap

through semantic constraint adjudication, they still rely on static graph approximations rather than runtime validation. This thesis's preliminary study (Chapter 3) provides direct empirical evidence of the precision failure inherent in non-executable detection, motivating a shift toward counterfactual runtime verification.

2.3.3 GPT-Augmented Static Analysis

Sun et al. Sun *et al.* (2024) propose GPTScan, a hybrid system that matches functions against per-vulnerability code-level scenarios and reduces false positives via static confirmation modules. GPTScan achieves precision above 90% on token contracts. Like all detection systems, however, GPTScan identifies likely vulnerable functions without producing an executable proof-of-concept or confirming exploitability at runtime.

2.4 Invariant Inference and Repair

Wang et al. Wang, Pei & Yang (2024) propose SmartInv, which fine-tunes a foundation model on labeled smart contracts using a Tier-of-Thought prompting strategy to generate bug-preventive invariants subsequently verified by a bounded model checker. SmartInv is effective at detecting functional bugs but generates no executable exploit tests and cannot confirm exploitability at runtime.

Karanjai et al. Karanjai, Xu & Shi (2025) propose Smartify, a five-agent framework that patches vulnerabilities in Solidity and Move, achieving state-of-the-art repair on Not-So-Smart-Contracts. Smartify operates downstream of detection: it assumes a known vulnerability and produces a patch without generating a proof-of-concept or confirming exploitability.

2.5 Differential Testing and Counterfactual Verification

Differential testing compares the behavior of two program variants — typically a modified and an original version, to identify behavioral divergence introduced by the modification. In the context of smart contract security, differential testing has been applied to detect regression vulnerabilities

introduced by contract upgrades and to validate automated repair patches. However, applying differential testing to *exploit confirmation*, requiring that an attack succeeds on a vulnerable contract and fails on a patched counterpart, represents a distinct application that has not been systematically evaluated across multiple vulnerability categories before this thesis.

The closest prior work also generates proof-of-concept exploits for smart contracts and validates them differentially against a patched version of the target. Andersson et al. Andersson, Bobadilla, Hobbelhagen & Monperrus (2025) propose PoCo, an agentic system that synthesizes proof-of-concept exploits for known smart contract vulnerabilities and validates each candidate by re-executing it against the project's own security patch, retaining only those that succeed before the patch and fail after it. PoCo additionally defines an *inconclusive* outcome for candidates that neither confirm nor refute, and identifies *coincidental invalidation*, a candidate failing on the patched version for reasons unrelated to the fix, as a principal threat to the method. Sun et al. Sun *et al.* (2025) propose PoCShift, which migrates existing real-world proof-of-concept exploits to related vulnerabilities and filters candidates that cannot be generalized.

2.6 Positioning ExploitGen

Table 2.1 summarizes the positioning of ExploitGen relative to representative prior work across four dimensions: detection paradigm, exploit generation, counterfactual confirmation, and taxonomy breadth. In the table, "Det." denotes detection-only, "Exec." denotes executable exploit generation, "CF" denotes counterfactual confirmation, and "Cats." indicates the number of OpenSCV categories covered.

ExploitGen departs from all prior paradigms by requiring that every reported vulnerability be demonstrated via an executable test that passes on the vulnerable contract and fails on a patched counterpart. Patch-differential validation has been applied to proof-of-concept generation for smart contracts (Section 2.6.1) Andersson *et al.* (2025); Sun *et al.* (2025), but in the reproduction setting, where the target is an already-reported vulnerability and the counterfactual is an existing fix. This thesis applies the criterion where no fix exists, constructing the counterfactual under a

Table 2.1 Comparison of related work across key dimensions Note: The values in the F1 column are not comparable across rows, as they represent different benchmarks and tasks, and are reproduced as reported in their respective studies.

* Slither's 0.61 figure is recall; its F1 under all-severity counting is 0.111 (Table 3.1).

System	Approach	Exec. PoC	CF Confirm.	Cats.	F1
Slither Feist *et al.* (2019)	Static rules	✗	✗	5–6	0.11*
Echidna Grieco *et al.* (2020)	Fuzzing	✓	✗	User-def.	—
GPTScan Sun *et al.* (2024)	LLM + static	✗	✗	4	—
LLM-SmartAudit Wei *et al.* (2025)	Multi-agent det.	✗	✗	6+	—
SmartGuard Ding *et al.* (2025)	LLM CoT	✗	✗	4	0.95
SmartInv Wang *et al.* (2024)	Invariant inference	✗	✗	3	—
Smartify Karanjai *et al.* (2025)	Repair	✗	✗	4	—
ExploitGen (this thesis)	**Multi-agent**	✓	✓	**8**	**0.675**

minimal-patch discipline, and integrates it with taxonomy-scoped detection so that verification is applied across a vulnerability taxonomy rather than to individually reported incidents. To the author's knowledge this is the first evaluation of counterfactual exploit confirmation stratified across a broad category taxonomy, which is what makes the per-category component analysis of Chapter 6 possible.

2.6.1 Patch-Differential Proof-of-Concept Generation

Both operate in the vulnerability *reproduction* setting: the target is a specific, already-reported vulnerability in a deployed protocol, and the counterfactual is a security patch that already exists, either in the project's history or as a known fix. ExploitGen shares the differential criterion but is positioned differently in three respects. First, no patch exists for the targets evaluated here: SB Curated supplies vulnerable contracts without fixes, so the counterfactual must be *constructed* rather than retrieved. This is the principal difference, and it cuts both ways: it extends the method to targets with no repair history, and it introduces a dependence on patch correctness that the reproduction setting does not have, analysed in Section 7.4. Second, the pipeline begins at detection rather than at a known vulnerability report, so exploit synthesis is applied across a broad taxonomy of candidate classes rather than to individual reported instances, which requires a per-category success oracle rather than a single fixed one. Third,

the evaluation is deliberately category-stratified, allowing per-category measurement of which verification components help which vulnerability classes, which incident-driven evaluations cannot provide. Conversely, PoCo and PoCShift are evaluated on real-world, multi-contract production targets, whereas the contracts evaluated here are small and self-contained, so the synthesis task is correspondingly easier, and no claim of parity in difficulty is made.

2.7 Chapter Summary

This chapter has surveyed the smart contract security literature across four paradigms and identified the detection–verification gap as the common structural limitation shared by all existing approaches. The following chapter provides empirical evidence for this gap through the evaluation of ExploitGen-Detection, a detection-only system that directly motivates the full pipeline's design.

CHAPTER 3

PRELIMINARY STUDY: EXPLOITGEN-DETECTION AND THE DETECTION–VERIFICATION GAP

This chapter presents ExploitGen-Detection, a detection-only system developed to empirically characterize the cost of operating without exploit verification. The system serves both as a direct motivating experiment and as the detection foundation upon which the full ExploitGen pipeline is built.

3.1 Design of ExploitGen-Detection

ExploitGen-Detection deploys eight independent, category-specialized LLM agents, each receiving a prompt encoding only the heuristics, structural indicators, and Solidity idioms relevant to its assigned OpenSCV category. For example, the reentrancy agent's prompt encodes callback mechanics, the check-effects-interactions pattern, and EVM call stack semantics. The access control agent's prompt encodes modifier inspection, `tx.origin` vs. `msg.sender` semantics, and initializer patterns.

All eight agents run in parallel under an AutoGPT-based orchestration framework Fezari & Ali-Al-Dahoud (2023). Agent outputs are merged through a deduplication procedure that resolves overlapping detections by location and vulnerability type, producing a final candidate set. No downstream verification is performed; candidates are reported directly.

Figure 3.1 illustrates the system architecture. Eight category-specialized LLM agents run in parallel under AutoGPT orchestration. Outputs are aggregated by a merge-and-deduplicate node and reported directly, with no downstream verification stage.

3.2 Evaluation on SB Curated

Evaluated on the SB Curated benchmark Durieux *et al.* (2020), ExploitGen-Detection identifies 306 true positives against 342 false positives, for an overall precision of 0.47 at a detection recall of 93% (Table 3.1). This recall substantially exceeds that of single-agent LLM

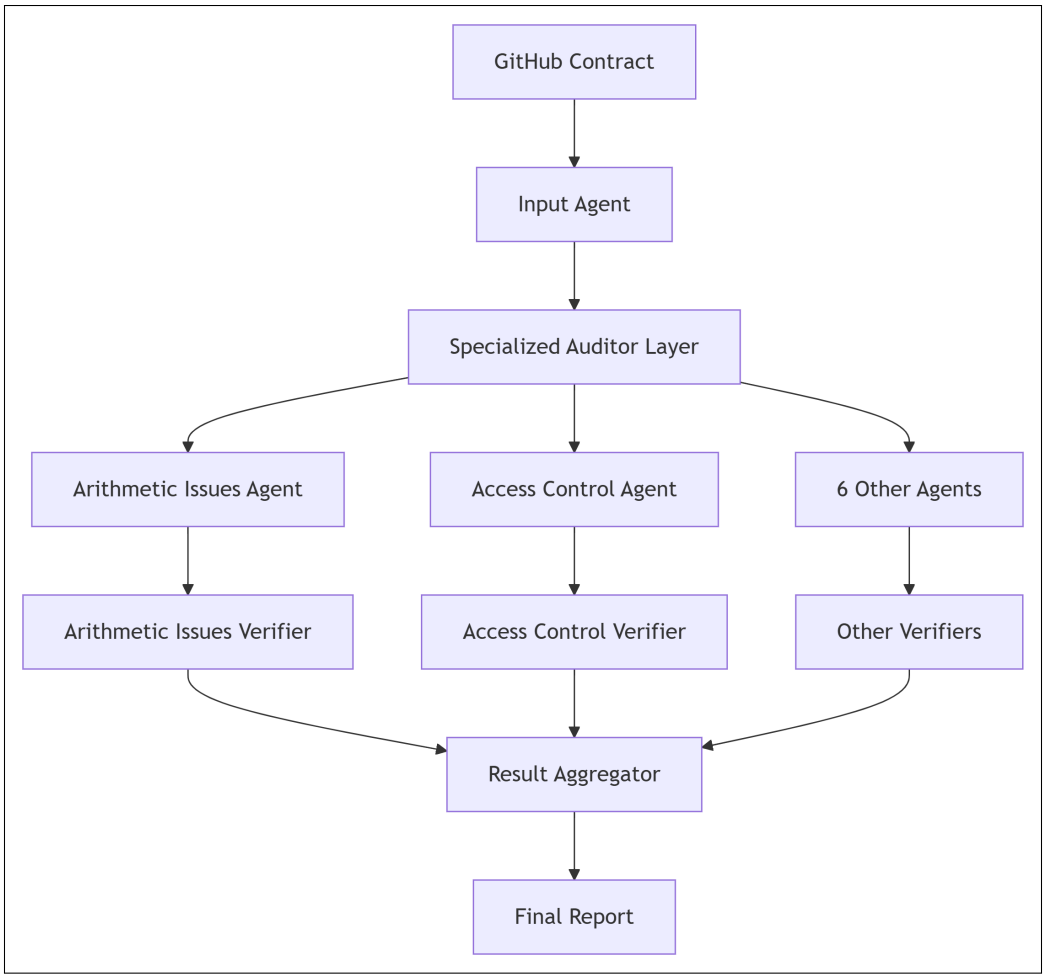

Figure 3.1 Architecture of ExploitGen-Detection

approaches (61%) and of static analyzers such as Slither (61%), consistent with the expected benefit of deploying category-specialized agents in parallel for detection coverage. The static analyzer's recall is accompanied by a false-positive volume an order of magnitude larger than either LLM-based system (1,866 detections at precision 0.06, with all severity levels counted as positive detections), illustrating in the starkest terms the detection–verification gap that motivates this thesis.

Precision and recall here are computed over different bases, a distinction essential to interpreting the result. *Recall* is the fraction of annotated (marked) benchmark vulnerabilities the system detects, giving 0.87. *Precision* is computed over all reported detections: the true-positive

count of 306 comprises the marked-vulnerability detections together with additional genuine vulnerabilities flagged beyond the benchmark annotations, each manually reviewed and confirmed to represent a genuine vulnerability in the target contract. Under the most conservative accounting, which credits only the marked detections, precision falls to $153/(153 + 342) = 0.31$. The two accountings lead to the same conclusion: operating without exploit verification produces a false-positive volume that overwhelms the detector's otherwise high recall.

Table 3.1 Detection performance of ExploitGen-Detection on SB Curated

System	TP	FP	Recall	Precision	F1
Single-agent LLM	119	74	0.61	0.617	0.616
Slither (static)	121	1866	0.61	0.061	0.111
ExploitGen-Detection	**306**	**342**	**0.93**	**0.47**	**0.62**

Table 3.2 decomposes the false positives by category. Precision is calculated as $TP/(TP + FP)$. The distribution is strongly non-uniform: unchecked low-level calls (180) and reentrancy (88) together produce 268 of the 342 false positives (78%), identifying these two categories as the dominant source of detection noise and motivating the verification stages developed in Chapter 4. Per-category recall and F1 are not reported, because the detector flags multiple candidates per marked instance and a per-category recall denominator is therefore undefined; recall is meaningful only in aggregate (Table 3.1). The *short addresses* category is listed for completeness but lies outside the eight OpenSCV-aligned categories carried into the verification pipeline and is not evaluated in subsequent chapters.

3.3 Analysis of the Detection–Verification Gap

The 342 false positives produced by ExploitGen-Detection are not an incidental calibration failure; they are a structural consequence of the detection paradigm. Three failure patterns were identified:

1. **Mitigated patterns.** A reentrancy-shaped call sequence protected by a `ReentrancyGuard` modifier, an arithmetic operation whose overflow is caught by Solidity 0.8+ built-in

Table 3.2 Per-category detections of ExploitGen-Detection Slither's performance is reported with all severity levels counted as positive detections.

Category	TP	FP	Precision
Reentrancy	85	88	0.491
Arithmetic	21	9	0.700
Access Control	56	15	0.789
Unchecked Low-Level	81	180	0.310
Denial of Service	8	4	0.667
Front-Running	8	5	0.615
Bad Randomness	34	25	0.576
Time Manipulation	12	15	0.444
Short Addresses	1	1	0.500
Overall	**306**	**342**	**0.472**

semantics, and a permission-gated external call are each indistinguishable from genuine vulnerabilities at the detection stage. The agent flags the pattern; it cannot evaluate the guard.

2. **Benign structural patterns.** Some contracts implement patterns that superficially resemble vulnerabilities as intentional design choices (e.g., a function calling an external contract before updating state for gas efficiency when the external contract is trusted). Without execution context, the detection agent cannot distinguish this from a reentrancy risk.
3. **Category boundary bleed.** With eight agents running in parallel on the same contract, overlapping patterns (e.g., a function vulnerable to both reentrancy and unchecked calls) produce duplicate reports that the deduplication procedure may not fully resolve.

3.4 Why AutoGPT is Insufficient for Verification

Closing the detection–verification gap within the AutoGPT architecture proved infeasible for two concrete reasons.

First, AutoGPT's loop-based orchestration does not expose the graph-level state management required to pass typed, structured output from a detection agent to a downstream verification

agent. The detection stage produces unstructured natural-language descriptions of candidates; the verification stage requires structured characterizations with precise field boundaries (entry points, sink operations, preconditions) that a free-form loop cannot enforce.

Second, conditional routing, selectively forwarding high-confidence candidates to exploit generation while discarding low-confidence flags, cannot be reliably enforced through prompt engineering alone in an autonomous loop. These are not tuning deficiencies; they reflect a fundamental mismatch between autonomous agent orchestration and structured multi-stage verification.

3.5 Requirements for a Verification Pipeline

The preliminary study establishes five requirements for a verification pipeline to close the detection–verification gap:

R1 (State Management). The pipeline must maintain typed, structured state across detection and verification stages.

R2 (Conditional Routing). The pipeline must support deterministic conditional routing to filter unexploitable candidates before expensive synthesis resources are consumed.

R3 (Causal Confirmation). Verification must require causal demonstration of exploitability, the PoC must fail when the vulnerability is absent, not merely assertion satisfaction.

R4 (Iterative Repair). Synthesis failures must trigger structured, environment-informed repair rather than unguided retries.

R5 (Adversarial Realism). Synthesized exploits must be constrained to realistic attacker capabilities, prohibiting direct state manipulation via cheatcodes.

These requirements directly motivate the LangGraph-based design of ExploitGen described in the following chapter.

3.6 Chapter Summary

This chapter has empirically established the detection–verification gap: an eight-agent detection system achieves 93% recall at a precision of 0.47, generating 342 false positives. The structural analysis identified three failure patterns and established five requirements for a verification pipeline. Chapter 4 describes how ExploitGen addresses each requirement.

CHAPTER 4

METHODOLOGY

This chapter describes the design of the ExploitGen pipeline, addressing the five requirements established in Chapter 3. Section 4.1 provides an architectural overview. Sections 4.2–4.8 describe each stage in detail. Section 4.9 provides formal definitions and correctness properties.

4.1 Pipeline Overview

ExploitGen addresses the detection–verification gap through a two-stage pipeline comprising seven ordered stages. The first stage (Stage 1) performs taxonomy-specialized vulnerability detection, producing a deduplicated list of candidates $\mathcal{V}$. The second stage (Stages 2–7) subjects each candidate $v_i \in \mathcal{V}$ to semantic analysis, structural grounding, adversarial validation, SCoT-based attack planning, exploit synthesis with iterative repair, and counterfactual confirmation.

A candidate is assigned verdict `exploit_proven`, the sole verdict under which it is reported as a verified vulnerability, only if: (1) an executable PoC test $\mathcal{T}$ passes on the original contract C; and (2) the same test fails on a semantically patched counterpart C^*.

Algorithm 4.1 presents the top-level pipeline logic.

Algorithm 4.1 ExploitGen Top-Level Pipeline

Input: Smart contract C, patched contract C^*, retry budget $k = 5$

Output: Verified exploit report $\mathcal{R}$

$\mathcal{G}, \mathcal{D} \leftarrow \textsc{ExtractGraphs}(C)$ /* Stage 2: CFG + DFG */

$\mathcal{V} \leftarrow \textsc{TaxonomyDetect}(C)$ /* Stage 1: parallel detection */

$\mathcal{R} \leftarrow \emptyset$

for *each candidate* $v_i \in \mathcal{V}$ **do**

  $m \leftarrow \textsc{Falsify}(v_i, C, \mathcal{G})$ /* Stage 3: find mitigations */

  if $m \neq \varnothing$ **then**

    $\gamma(v_i) \leftarrow$ *false_positive*; **continue**

  end if

  $\chi_i \leftarrow \textsc{Characterize}(v_i, C, \mathcal{G}, \mathcal{D})$ /* Stage 4 */

  $\beta_i \leftarrow \textsc{PlanAttack}(\chi_i, C)$ /* Stage 5: SCoT blueprint */

  $\gamma(v_i) \leftarrow \textsc{SynthesizeAndConfirm}(\beta_i, C, C^*, k)$ /* Stages 6-7 */

  if $\gamma(v_i) =$ *exploit_proven* **then**

    $\mathcal{R} \leftarrow \mathcal{R} \cup \{v_i\}$

  end if

end for

return $\mathcal{R}$

Figure 4.1 illustrates the complete pipeline architecture. Detected vulnerability candidates enter parallel processing branches covering semantic characterization, falsification-first validation, and SCoT-based attack planning, which feed a test generation loop governed by an iterative critic-repair cycle. Confirmed exploits are then validated against a patched contract before admission to the final report.

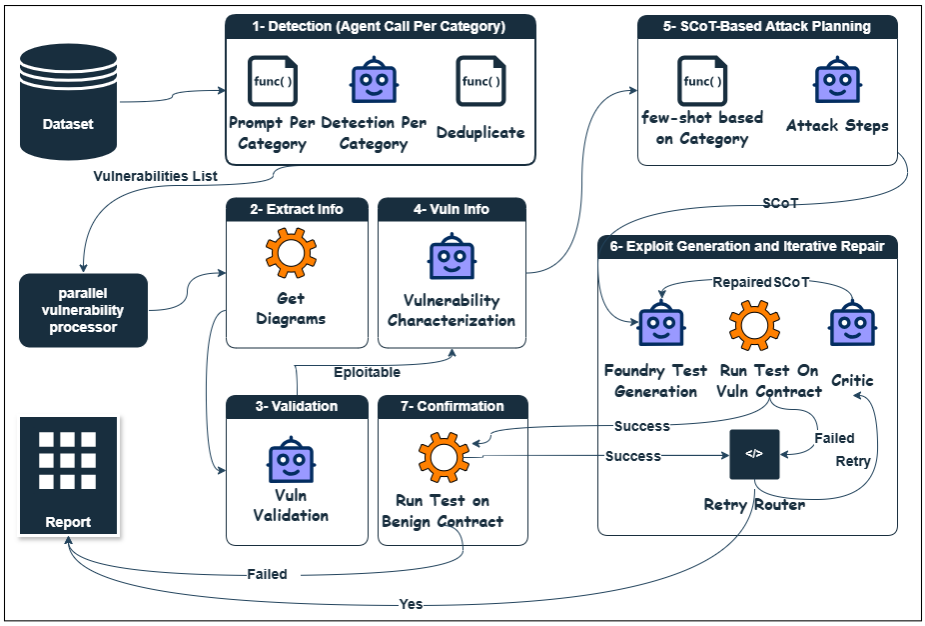

Figure 4.1 Architecture of the ExploitGen pipeline

4.2 Stage 1: Taxonomy-Specialized Vulnerability Detection

Detection applies the same taxonomy-specialized design established in Chapter 3, reimplemented within the LangGraph pipeline: the eight category-specialized detection prompts are reused, but the AutoGPT orchestration is replaced by graph-structured invocation that routes only the relevant category detector and feeds candidates to the downstream verification stages rather than reporting them directly. As established there, each agent is scoped to a single OpenSCV category, concentrating analytical attention on a well-defined hypothesis space to maximize per-category recall and enable clean per-category precision and recall measurement.

In production, all eight agents run in parallel. For the SB Curated evaluation, the known category structure of the benchmark allows each run to be scoped to the relevant agent, eliminating cross-category confounding. This scoping is an evaluation-time oracle: it isolates the verification pipeline from detection noise, but it is not available in deployment, and the numbers it affects are identified in Section 6.5. Agent outputs are merged through a deduplication procedure that

resolves overlapping detections by source location and vulnerability type before forwarding candidates to Stage 2.

Design rationale.

The category-specialized design is motivated by the observation that prompts optimized for reentrancy detection (requiring callback and call-stack reasoning) interfere with access control detection (requiring modifier and initializer inspection). A single general-purpose prompt forces the model to maintain conflicting analytical frames simultaneously, degrading performance on both, a category-dependent effect that the ablation study confirms quantitatively (Section 6.3).

4.3 Stage 2: Static Graph Extraction

Before per-vulnerability analysis begins, Slither is invoked in printer-only mode to extract two structural artifacts for each target contract C:

- **Per-function Control Flow Graphs** ($\mathcal{G}_i$): extracted via `-print cfg`. Each CFG encodes basic blocks, conditional branches, loop edges, and external call sites.
- **Inter-variable Data Dependencies** ($\mathcal{D}_i$): extracted via `-print data-dependency`. The dependency graph encodes write-after-read relationships between state variables and function parameters.

Both artifacts are stored once per file and shared across all downstream vulnerability branches (R1: State Management). Slither's vulnerability checker is *not* invoked; its rule-based detection suffers from the same pattern-matching limitations the LLM stage is designed to overcome Kezadri Hamiaz & Driss (2025). The resulting graphs ground subsequent LLM reasoning in the contract's actual execution semantics rather than raw source text.

Design rationale.

The ablation study (Section 6.3) demonstrates that CFG and DFG inputs are beneficial for categories requiring non-local reasoning (reentrancy, unchecked calls) and harmful for categories

involving localized single-expression faults (arithmetic). This motivates a category-adaptive graph inclusion policy as a direction for future work.

4.4 Stage 3: Falsification-First Validation

Before any code is generated, each candidate is subjected to an adversarial reasoning pass whose purpose is to reject structurally plausible but unexploitable candidates (R2: Conditional Routing). An LLM agent searches for mitigations that the detection stage may have missed:

- `ReentrancyGuard` or equivalent modifier protecting the flagged external call site;
- Solidity 0.8+ overflow protection making arithmetic candidates non-exploitable;
- Access control restrictions protecting the flagged entry point;
- State initialization conditions that prevent the preconditions for exploitation from being met.

Candidates for which a viable mitigation is identified are classified as `false_positive` without consuming synthesis resources. Only candidates for which no mitigation is found proceed to attack planning.

This stage is precisely where benign patterns are flagged and set aside rather than carried forward. A code fragment that resembles a vulnerability but is in fact safe, a reentrancy-shaped call sequence guarded by a `ReentrancyGuard`, an arithmetic operation whose overflow is caught by Solidity 0.8+ semantics, or a privileged operation behind an intact access-control check, is identified here as mitigated and classified `false_positive`, and takes no further part in the pipeline. Two later stages provide additional backstops for benign candidates that survive this pass: falsification-first validation errs toward proceeding when uncertain, but the counterfactual criterion (Stage 7) rejects any candidate whose generated test passes on *both* the original and the patched contract, the signature of a test not actually tied to a vulnerability, and such candidates re-enter repair rather than being reported. A genuinely benign candidate therefore has three independent opportunities to be excluded: mitigation detection at Stage 3, failure to produce a discriminating exploit during synthesis, and the pass-on-both check at confirmation. This

layered filtering is the direct mechanism behind the precision gain reported in Chapter 6: benign patterns are actively removed, not merely down-weighted.

Design rationale.

The ablation study reveals that this validation stage is miscalibrated for access control and arithmetic categories (Section 6.3), suggesting that category-aware falsification heuristics are required. The current implementation uses category-agnostic mitigation prompts; category-specific versions represent a high-priority future improvement.

4.5 Stage 4: Semantic Vulnerability Characterization

Each surviving candidate enters a semantic characterization stage. An LLM agent acting as a *vulnerability oracle* receives the candidate report, the CFG, the data-dependency graphs, and the full contract source. It produces a structured JSON characterization χ_i containing:

- **Entry points**: the externally callable functions through which an attacker can initiate exploitation;
- **Vulnerable sink**: the specific operation that produces the harmful outcome (e.g., the `.call{value}` before the balance update in reentrancy);
- **Missing guards**: control flow guards that would prevent exploitation if present;
- **Semantic preconditions**: conditions on contract state that must hold for the exploit to succeed.

This characterization replaces the vague natural-language detection description with a graph-grounded exploit specification and serves as the primary input to attack planning. It is important to note that while Slither is utilized in earlier setup stages for targeted context extraction, the raw Abstract Syntax Tree (AST) is strictly excluded as a direct input to the LLM in this stage. Excluding the AST prevents exceeding token context limits and avoids introducing dense syntactic noise, forcing the model to rely exclusively on its semantic understanding of the source code and the extracted control/data-flow graphs.

4.6 Stage 5: SCoT-Based Attack Planning

Attack planning applies SCoT prompting Li *et al.* (2025) to produce a structured `SCoTBlueprint` β_i before any Solidity code is synthesized. The model receives the vulnerability characterization, the static graphs, the full contract source, and category-specific few-shot examples, and produces a three-phase blueprint:

Phase 1: Setup. Specifies contract deployment parameters, initial Ether funding for all actors, and any prerequisite state initialization. Foundry `vm` cheatcodes are permitted in this phase (for deployment and funding only).

Phase 2: Exploit Execution. Specifies the precise call sequence through the contract's *public interface* required to trigger the vulnerability. Foundry cheatcodes are **prohibited** in this phase to prevent storage-manipulation bypasses (see R5: Adversarial Realism).

Phase 3: Verification. Specifies state-delta assertions that instantiate the category-specific success property, expressed relative to pre-exploit state. Assertions must compare post-exploit state against pre-exploit state or initial conditions, not against arbitrary thresholds.

By committing to a complete attack specification before any Solidity is written, the pipeline reduces Stage 6's synthesis task to syntactic translation of an already-resolved logical plan (R4: Iterative Repair). This separation prevents the performance degradation that arises when models simultaneously plan and generate code Li *et al.* (2025).

Cheatcode prohibition rationale.

Without the Phase 2 cheatcode prohibition, LLMs occasionally use `vm.store` to directly overwrite storage variables, bypassing the vulnerability entirely and producing a vacuously passing test. The counterfactual stage (Stage 7) provides a structural backstop against this failure mode, but the prohibition eliminates it at the planning level.

4.7 Stage 6: Exploit Generation and Iterative Repair

The SCoT blueprint β_i is translated into a complete Foundry Solidity test by a code synthesis LLM. The synthesizer operates under a *naming collision constraint*: all identifiers reserved by the forge-std `Test` inheritance chain are enumerated and mandated renamings are enforced to prevent Error 9097 (Identifier already declared) compilation failures.

Generated tests are executed against a locally spawned Anvil EVM node. Forge output is classified into typed outcome labels:

`exploit_proven`: The test passed; proceed to Stage 7 for counterfactual confirmation.

`compilation_failed`: Syntactic or import error; Critic diagnoses the specific error type.

`mitigated_reentrancy_guard`: Guard detected in runtime behavior; candidate reclassified.

`assertion_failed`: Test compiled and ran but assertions were not satisfied; Critic analyzes precondition failure.

`out_of_gas`: Gas limit exceeded; Critic adjusts gas estimates.

`revert`: Transaction reverted; Critic identifies the revert condition.

When a test fails, the pipeline enters a Critic-driven repair loop (R4: Iterative Repair) instantiating the ReAct paradigm Yao *et al.* (2023): the Critic agent receives the classified forge output as an observation, produces a structured root-cause diagnosis as a reasoning trace, and emits a revised `SCoTBlueprint` as an action routed back to synthesis. The trajectory of prior fix attempts is retained to prevent repeated failed strategies. The per-vulnerability retry budget is $k = 5$ attempts.

Algorithm 4.2 presents the synthesis-and-repair loop.

Algorithm 4.2 SynthesizeAndConfirm: Stages 6–7

Input: Blueprint β, contracts C, C^*, budget k

Output: Verdict $\gamma \in \{exploit\_proven, inconclusive\}$

attempts $\leftarrow 0$; history $\leftarrow \emptyset$
while *attempts* $< k$ **do**
  $\mathcal{T} \leftarrow$ Synthesize(β)
  $r \leftarrow$ ForgeRun($\mathcal{T}, C$)
  if $r =$ `exploit_proven` **then**
    $r^* \leftarrow$ ForgeRun($\mathcal{T}, C^*$) `/* Stage 7: counterfactual */`
    if $r^* =$ `test_failed` **then**
      return *exploit_proven*
    else
      $\beta \leftarrow$ Critic($\mathcal{T}, r, r^*$, "semantic FP: assert fails on patched", history)
    end if
  else
    $\beta \leftarrow$ Critic($\mathcal{T}, r, \varnothing, r$, history)
  end if
  history $\leftarrow$ history $\cup\{(\mathcal{T}, r)\}$
  attempts $\leftarrow$ attempts $+1$
end while
return *inconclusive*

4.8 Stage 7: Counterfactual Confirmation and Verdict Assignment

A passing test is not automatically a confirmed exploit. An LLM-generated assertion may be satisfied by the initial funding state regardless of exploit success (R3: Causal Confirmation). For example, a test asserting "attacker balance > 1 Ether" passes trivially if the attacker was funded with 10 Ether at setup, regardless of whether any Ether was drained.

To close this causality gap, every passing test is re-executed against C^*. A genuine exploit must fail on the patched version. Tests that pass on both contracts are classified as semantic false positives: the differential result is appended to the Critic's context and the branch re-enters the repair loop, requiring assertions that fail when the vulnerability is absent.

Three verdicts are possible:

1. `exploit_proven`: Assigned only when counterfactual confirmation is satisfied. This is the *sole* verdict under which a vulnerability is reported to the analyst.
2. `inconclusive`: Assigned when the retry budget k is exhausted without confirmation.
3. `false_positive`: Assigned to candidates rejected at Stage 3.

This three-way structure deliberately distinguishes confirmed safety (the vulnerability pattern does not yield a PoC) from confirmed exploitability (a PoC exists), two conclusions that are not equivalent.

4.9 Formal Definitions and Correctness Properties

Definition 4.1 (Exploit-Proven Vulnerability)**.** *A vulnerability candidate $v_i \in \mathcal{V}$ is* exploit-proven *with respect to contract C and patch C^* if and only if there exists a Foundry test $\mathcal{T}$ such that:*

$$\textsc{ForgeRun}(\mathcal{T}, C) = \texttt{pass} \quad \wedge \quad \textsc{ForgeRun}(\mathcal{T}, C^*) = \texttt{fail}$$

Definition 4.2 (Minimal Patch)**.** *A contract C^* is a* minimal patch *of C with respect to vulnerability v_i if: (1) C^* addresses only the structural cause of v_i; (2) C^* preserves all function signatures of C wherever possible; and (3) no broader refactoring is performed. Minimal patches ensure that counterfactual failures are causally attributable to the vulnerability's absence rather than interface changes.*

Property 1 (Sound Verification)**.** *Under the counterfactual confirmation criterion, ExploitGen never reports a vulnerability without an executable demonstration of exploitability in a deterministic EVM environment. This does not imply completeness: inconclusive verdicts may arise from synthesis limitations rather than genuine non-exploitability.*

4.10 Chapter Summary

This chapter has described the ExploitGen pipeline in detail, addressing all five requirements established in Chapter 3. The key design principle is that *reported vulnerabilities must be causally demonstrated, not merely pattern-matched*: every `exploit_proven` verdict is backed by an executable test that succeeds on the vulnerable contract and fails on its patched counterpart. Chapter 5 describes the system implementation.

CHAPTER 5

IMPLEMENTATION

This chapter describes the ExploitGen system implementation, covering the LangGraph orchestration architecture, language model configuration, workspace and environment management, and the dataset augmentation process.

5.1 System Architecture

ExploitGen is implemented as an asynchronous Python 3.10 application. Pipeline stages are implemented as LangGraph nodes sharing a Pydantic-typed `PipelineState` object that accumulates all artifacts produced by each stage: the raw contract source, extracted graphs, detection reports, characterization JSON, SCoT blueprints, synthesized test files, forge execution output, and final verdicts.

LangGraph was selected as the orchestration framework for three concrete reasons. First, the Critic-repair loop requires a native conditional back-edge routing failed tests back to synthesis; a DAG-only framework cannot express this without an external polling loop. Second, LangGraph's typed state management enforces the data contracts between stages (R1: State Management from Chapter 3). Third, conditional routing via edge functions enables deterministic candidate filtering at Stage 3 (R2: Conditional Routing).

5.2 Language Model Configuration

Two LLM tiers are instantiated at startup. Table 5.1 gives the model bound to each tier in the reported evaluation and in the cross-generation comparison of Section 6.5. Tier assignment is fixed in `PipelineConfig` and is not overridable from the command line.

Fast tier (`gpt-5.4-mini`): Handles category detection, initial vulnerability filtering, and lightweight classification stages.

Codegen tier (`gpt-5.3-codex`): Handles SCoT attack planning, exploit test generation, and Critic repair loops requiring deep syntactic synthesis. To prevent unbounded execution and control API costs, the Critic repair loop is strictly bounded by a maximum retry threshold ($N_{max} = 5$). If the generated exploit test fails to satisfy the counterfactual execution gate after N_{max} iterations, the pipeline flags the target as structurally unsupported by the current model reasoning capabilities and halts generation for that specific vulnerability.

Table 5.1 Model bound to each tier in the reported evaluation and in the cross-generation comparison of Section 6.5

Tier	Reported evaluation	Cross-generation comparison
Fast	`gpt-5.4-mini`	`gpt-4.1-mini-2025-04-14`
Codegen	`gpt-5.3-codex`	`gpt-4.1-2025-04-14`

Stage assignment is fixed in code: category detection, semantic characterization and falsification-first validation call the fast tier; SCoT attack planning, exploit synthesis and Critic repair call the codegen tier.

All LLM clients apply exponential backoff on rate-limit errors and strip markdown fences before JSON parsing. Under the `gpt-5.3-codex` and `gpt-5.4-mini` endpoints, traditional sampling temperature is not exposed as a configurable hyperparameter; the explicit reasoning effort parameter instead governs token generation. The reasoning effort is set to `high` at every call site — detection, characterization, validation, planning, synthesis and Critic repair — so that the ablation contrasts of Section 6.3 vary the prompt scaffolding rather than the inference budget.

While the pipeline integrates external utilities like Slither and Foundry, it does not utilize native provider-side function-calling APIs (i.e., autonomous tool dispatch by the LLM). Instead, execution is handled deterministically via LangGraph node orchestration. Stage outputs are obtained through schema-constrained generation, where agents emit JSON conforming to fixed Pydantic schemas, and external execution traces are fed back into the context window as standard prompt observations.

Model versioning note.

The reported evaluation uses floating model aliases rather than pinned snapshots, so provider-side updates may affect exact reproducibility. The cross-generation comparison uses pinned snapshots (`gpt-4.1-2025-04-14`, `gpt-4.1-mini-2025-04-14`), so the two configurations differ in reproducibility guarantee as well as in model generation. The implications are discussed in Section 6.5 (Chapter 6).

5.3 Workspace and Environment Management

For each target contract, an isolated Foundry workspace is provisioned and validated with `forge build`. Because the evaluation dataset (SB Curated) contains smart contracts spanning multiple generations of the Solidity compiler, the system dynamically parses the `pragma solidity` directives of the target contract and configures the `foundry.toml` file to utilize the precise matching compiler version via Foundry's built-in toolchain management. Workspaces that fail to compile are excluded from all further processing. Each parallel vulnerability branch operates on a physically distinct workspace copy to prevent compilation race conditions when multiple synthesis attempts run concurrently.

Slither is invoked in printer-only mode once per contract to extract CFGs and data-dependency graphs, both stored in the shared `PipelineState` object. Detection issues one asynchronous LLM call per OpenSCV category; returned candidates are aggregated by the merge-and-deduplicate node.

A lightweight template matching step assigns each candidate a category tag that selects appropriate few-shot examples for the attack planning stage, based on a deterministic lookup from Slither check names with a regex heuristic fallback.

5.4 Exploit Synthesis, Repair, and Confirmation

The codegen tier translates each validated SCoT blueprint into a complete Foundry Solidity test, enforcing the naming collision constraint described in Section 4.7. Generated tests are executed against an ephemeral Anvil EVM node, with each parallel branch allocated an independent port to prevent state cross-contamination.

Forge output is classified into typed outcome labels by an ordered classifier that checks high-signal mitigation indicators before generic failure patterns. Failed tests are routed to the Critic agent, which revises the `SCoTBlueprint` at the specification level rather than patching the failing Solidity directly; the synthesizer re-translates the corrected blueprint from scratch. This approach prevents the Critic from inheriting the synthesizer's syntactic errors.

Passing tests are re-executed against the patched contract for counterfactual confirmation. Tests that pass on both contracts re-enter the repair loop as semantic false positives with the differential result appended to the Critic's context.

5.5 Dataset Augmentation: Patched Contract Generation

The 139 patched contracts in the augmented SB Curated dataset were produced by the authors of this thesis using an LLM-assisted workflow combining Claude and Gemini, followed by manual review and compilation verification of every patch.

The minimal-patch principle was applied: each patch addresses only the flagged vulnerability with no broader refactoring. Because SB Curated contracts are small, single-purpose files each containing one vulnerability, a minimal patch is also a complete fix with no risk of cross-vulnerability interaction. Function signatures are preserved wherever possible, ensuring that counterfactual failures are causally attributable to the vulnerability's absence. Cases requiring signature changes are recorded with the `interface_mismatch` label. Four contracts were excluded because the vulnerability had already been corrected in the deployed version, making a minimal reversal require artificial structural changes.

Spot checks confirmed that exploits produce the expected state changes on vulnerable contracts and that patched contracts block the attack at the intended code path.

Patch construction examples.

Reentrancy. The minimal patch adds a boolean reentrancy guard variable and updates it before the external call, implementing the check-effects-interactions pattern without introducing `ReentrancyGuard` (which would change the inheritance structure).

Arithmetic. For contracts compiled under Solidity below 0.8.0, the minimal patch either bumps the pragma to 0.8+ or wraps the vulnerable operation in a SafeMath call.

Access Control. The minimal patch adds a `require` statement checking `msg.sender == owner` or adds the `onlyOwner` modifier, depending on the contract's existing modifier infrastructure.

5.6 Data and Code Availability

The augmented dataset is available at https://github.com/Aisha8665/smartbugs-curated-clean-upgraded (release). It is derived from SB Curated Durieux *et al.* (2020); SmartBugs Project (2024), retaining the upstream directory layout under `dataset/`, with one subdirectory per DASP category, together with the upstream `vulnerabilities.json` annotation file. Each category directory is paired with a sibling `<category>_patched/` directory holding the counterfactual for every contract, named `<contract>_patched.sol`. For example:

```
dataset/access_control/mycontract.sol
dataset/access_control_patched/mycontract_patched.sol
```

The repository is organised as a Foundry project, so a contract and counterfactual compile and execute under the same toolchain used in the evaluation, and `versions.csv` records the compiler version selected for each contract. The contracts themselves retain the licences under which they were originally published; all other files are released under Apache 2.0. [PIPELINE SOURCE SENTENCE]

5.7 Chapter Summary

This chapter has described the system implementation of ExploitGen, from the LangGraph orchestration architecture through the dataset augmentation process. The key implementation decisions, asynchronous workspace isolation, typed state management, specification-level Critic repair, and minimal-patch augmentation, directly reflect the design requirements established in Chapters 3 and 4. Chapter 6 presents the experimental evaluation.

CHAPTER 6

EVALUATION

This chapter presents the experimental evaluation of ExploitGen, addressing the three research questions defined in the introduction. Section 6.1 describes the experimental setup. Sections 6.2–6.4 address RQ1–RQ3. Section 6.5 discusses threats to validity.

6.1 Experimental Setup

6.1.1 Benchmark

ExploitGen is evaluated on the SB Curated benchmark Durieux *et al.* (2020), a curated dataset of annotated Solidity contracts organized by the DASP vulnerability taxonomy NCC Group (2018). The evaluation spans eight categories: reentrancy, arithmetic, access control, unchecked low-level calls, denial of service, front-running, bad randomness, and time manipulation, comprising 139 ground-truth vulnerabilities across 278 contracts (139 vulnerable originals and 139 semantically patched counterparts produced by this thesis).

6.1.2 Evaluation Metrics

Two rates are reported throughout. *Confirmation yield* (Y) is the fraction of candidates entering synthesis that reach counterfactual confirmation; *recall* (R) is the fraction of ground-truth vulnerabilities confirmed:

$$Y = \frac{|\text{exploit\_proven}|}{|\text{processed}|}, \quad R = \frac{|\text{exploit\_proven}|}{|\text{existed}|}, \quad F_1 = \frac{2YR}{Y+R} \tag{6.1}$$

where *processed* is the number of candidates entering the synthesis stage from the vulnerable contract subset, and *existed* is the per-category ground-truth count. A candidate is counted as *exploit_proven* only if its PoC test passes on the original contract and fails on the patched version (counterfactual confirmation criterion).

Y occupies the structural position of precision and is combined with recall as F_1 on that basis, but it is deliberately not called precision. The complement of Y is not a set of detector errors: the $116 - 86 = 30$ unconfirmed candidates are `inconclusive` verdicts (Section 4.8), that is, candidates for which the retry budget was exhausted, and an unknown share of them are genuine vulnerabilities the synthesiser failed to exploit rather than false positives. Reported precision, the fraction of the 86 `exploit_proven` verdicts that are genuine, is a separate quantity, bounded below by the counterfactual criterion itself and assessed in Section 7.4.

While generative code synthesis tasks are conventionally evaluated using metrics such as Pass@k, the overarching objective of ExploitGen is vulnerability confirmation. By subjecting the generative output (the PoC test) to a strict binary counterfactual gate, the generation phase is logically reduced to a binary classification mechanism (Exploit Proven vs. Not Proven). This strict falsification boundary permits the rigorous application of Precision, Recall, and F1-score to evaluate the end-to-end efficacy of the verification pipeline.

A generated exploit test is classified as a *Verified Exploit* only if it satisfies the dual-execution counterfactual gate of Definition 4.1: the test must *pass* under `forge test` against the vulnerable contract, meaning its state-delta assertions detect the security violation, and the same test must *fail* against the patched contract, meaning the violation no longer occurs. Tests that pass on both contracts, or that fail on the vulnerable contract, do not satisfy the gate and are returned to the repair loop.

6.1.3 Ablation Configurations

Six configurations are evaluated: Base (detection + single synthesis attempt, no verification stages), Full (all stages enabled), No Attack Info (SCoT planning disabled), No Critic (repair loop disabled), No Validation (falsification stage disabled), and No Diagram (CFG/DFG inputs disabled).

6.1.4 Baselines

Slither Feist *et al.* (2019) is reported as an external reference: recall 0.611 at precision 0.061 under all-severity counting, for an F1 of 0.111 (Table 3.1). Because Slither performs detection only and produces no exploit witness, its figures are not directly comparable to the counterfactually confirmed results reported here. The controlled comparison in this thesis is against ExploitGen-Detection (Chapter 3), the internal detection-only baseline, at recall 0.93 and precision 0.47.

6.2 RQ1: End-to-End Pipeline Effectiveness

Table 6.1 reports precision, recall, and F1 for the full pipeline across all eight vulnerability categories. Within the table, "Existed" refers to the ground-truth count, "Proc." indicates candidates entering synthesis, and "Exploit." denotes confirmed exploits.

Table 6.1 Full pipeline performance per vulnerability category

Category	Existed	Proc.	Exploit.	Prec.	Recall	F1
Reentrancy	31	29	27	0.931	0.871	0.900
Arithmetic	15	13	11	0.846	0.733	0.786
Access Control	18	15	12	0.800	0.667	0.727
Unchecked Low-Level	52	44	28	0.636	0.538	0.583
Denial of Service	6	3	2	0.667	0.333	0.444
Front-Running	4	4	1	0.250	0.250	0.250
Bad Randomness	8	5	3	0.600	0.375	0.462
Time Manipulation	5	3	2	0.667	0.400	0.500
Overall	**139**	**116**	**86**	**0.741**	**0.619**	**0.675**

Across all 139 ground-truth vulnerabilities, ExploitGen achieves a recall of 0.619, a confirmation yield of 0.741, and F1 of 0.675. All 86 reported exploits were confirmed via counterfactual validation.

Within the controlled evaluation, the verification pipeline reduces synthesis-stage false positives from 39 to 30, a 23.1% reduction relative to the single-synthesis Base configuration, while increasing confirmed true positives from 73 to 86, demonstrating that the additional verification stages simultaneously filter unexploitable candidates and improve exploit synthesis coverage. This within-system reduction is distinct from the 91.2% reduction (342 to 30) reported against the detection-only ExploitGen-Detection paradigm in Chapter 3, which has a different, detection-level denominator.

Performance partitions by synthesizability. Reentrancy achieves the highest F1 (0.900) because the exploit is expressible as a self-contained, single-transaction interaction that maps directly onto the pipeline's three-phase blueprint structure. Arithmetic and access control are similarly well-suited to single-actor synthesis, achieving F1 of 0.786 and 0.727 respectively.

Performance degrades for categories requiring environmental preconditions outside the pipeline's execution model. Front-running, bad randomness, and time manipulation depend on miner-controlled values or transaction ordering that cannot be reproduced in a deterministic Foundry environment, reflecting a structural scope limitation rather than a calibration failure.

Unchecked low-level calls present a distinct problem: the category is the largest (52 contracts, 37.4% of the benchmark) and most heterogeneous. Its high inconclusive count (16 of 44 processed) indicates that the pipeline regularly generates tests that pass on both the vulnerable and patched contracts, a counterfactual failure indicating that test assertions are not causally tied to the vulnerability.

Comparison with baselines. The full pipeline achieves an F1 of 0.675 under counterfactual confirmation, against 0.111 for Slither under all-severity detection counting; the two tasks differ and the comparison is indicative only. Against the internal detection-only baseline, which achieves recall of 0.93 at precision 0.47, verification substantially improves precision ($0.47 \rightarrow 0.741$) at a deliberate and quantified cost in recall ($0.93 \rightarrow 0.619$): candidates that cannot be demonstrated through an executable, counterfactually confirmed exploit are not reported, even when genuinely vulnerable. This trade is intentional: the system optimizes for

auditor-actionable, pre-validated findings rather than maximal coverage, and the recall reduction is the direct mechanism by which the 342 false positives of the detection-only paradigm are eliminated.

Why prioritize false positives. In many detection settings a false negative is the graver error, and for a tool intended as the *sole* line of defense that ordering would hold here too. ExploitGen is positioned differently: as a pre-audit screening stage whose output is consumed by a human auditor, not as a replacement for one. In that role the binding constraint is auditor attention. A detection-only system that emits 648 findings to surface 306 real ones imposes a review burden—and, more corrosively, a false-positive rate that trains auditors to distrust and eventually ignore the tool, at which point its true positives are lost regardless of recall. High precision with an executable witness per finding is what makes the output actionable at all. Crucially, our design does not discard the missed vulnerabilities silently: candidates that are not confirmed are retained with an `inconclusive` verdict rather than a negative one (Section 4.8), preserving them for the human auditor's attention and keeping recall a reported, inspectable quantity rather than a hidden loss. The false negatives of the verification stage are thus surfaced as unconfirmed candidates. In contrast, the false positives of the detection stage, if unfiltered, would silently consume the auditing effort the system exists to conserve.

6.3 RQ2: Component Contribution via Ablation Study

Table 6.2 reports precision, recall, and F1 per ablation configuration and vulnerability category. Bold values denote the highest F1 per category. Note that the No Critic configuration for Unchecked Calls (UC) is confounded by a higher detection count (100 vs. 88 for the Full pipeline) and should be interpreted with caution.

Counterfactual confirmation. Stage 7 is present in every configuration, including Base, so it cannot be ablated within this study. Its discriminating effect is instead measured directly: across the Full configuration, 17 generated tests passed on the vulnerable contract but also passed on the patched counterpart and were therefore rejected as semantic false positives rather than

Table 6.2 Ablation results per configuration and vulnerability category

Config	Precision / Recall / F1							
	RE	AR	AC	UC	DOS	FR	BR	TM
Base	0.414/0.387/0.400	0.929/0.867/0.897	0.647/0.611/0.629	0.811/0.577/0.674	0.667/0.333/0.444	0.500/0.500/0.500	0.200/0.125/0.154	0.667/0.400/0.500
Full	**0.931/0.871/0.900**	0.846/0.733/0.786	0.800/0.667/0.727	0.636/0.538/0.583	0.667/0.333/0.444	0.250/0.250/0.250	0.600/0.375/0.462	0.667/0.400/0.500
No Attack Info	0.724/0.677/0.700	0.857/0.800/0.828	0.933/0.778/0.848	0.919/0.654/0.764	0.500/0.333/0.400	0.250/0.250/0.250	0.500/0.250/0.333	0.667/0.400/0.500
No Critic	0.900/0.871/0.885	0.786/0.733/0.759	0.786/0.611/0.688	0.840/0.808/0.824[†]	0.333/0.167/0.222	0.500/0.500/0.500	0.400/0.250/0.308	0.667/0.400/0.500
No Validation	0.857/0.774/0.814	0.929/0.867/0.897	**1.000/0.889/0.941**	0.821/0.615/0.703	0.667/0.333/0.444	0.500/0.500/0.500	0.800/0.500/0.615	0.667/0.400/0.500
No Diagram	0.929/0.839/0.881	**1.000/0.933/0.966**	0.923/0.667/0.774	0.773/0.327/0.459	0.500/0.167/0.250	0.250/0.250/0.250	0.800/0.500/0.615	0.667/0.400/0.500

reported. These are concentrated in unchecked low-level calls (8) and front-running (5), the two categories where assertions are hardest to tie causally to the flaw; access control, denial of service and bad randomness account for 1, 2 and 1 respectively, and reentrancy, arithmetic and time manipulation for none. Without the pass-on-both check these 17 tests would have been reported as confirmed exploits, raising the apparent exploit count by roughly a fifth while introducing exactly the class of unverifiable finding the counterfactual criterion is designed to prevent.

SCoT-based attack planning. Removing attack planning (No Attack Info) is associated with a reduction in reentrancy F1 from 0.900 to 0.700, accompanied by a fall in confirmation yield ($0.931 \rightarrow 0.724$) and recall loss ($0.871 \rightarrow 0.677$). Structured pre-planning is essential for reentrancy because the exploit requires a precisely sequenced multi-step interaction across two contracts. Conversely, for access control, removing attack planning increases F1 from 0.727 to 0.848. Inspection of the failing tests suggests a mechanism: SCoT blueprints for access control produce structurally complex Solidity prone to synthesis-level failures that consume the retry budget before a passing test is reached.

Falsification-first validation. No Validation/Access Control achieves F1 = 0.941 with a confirmation yield of 1.000 and recall = 0.889 — the highest F1 in any configuration in any category. Every access control candidate that reached synthesis in that run was successfully exploited, four more than in the Full configuration. The difference cannot be attributed to the validation stage alone: the two runs also differ upstream, with 32 detections in the No Validation run against 30 in Full, and in the Full run the number of candidates entering synthesis is consistent with no validation-stage rejection at all. The observed gap is therefore a combination

of detection non-determinism and synthesis outcome variance, on a category with only 18 ground-truth instances, and it should be read as an association rather than as a measured effect of removing validation. The direction is nonetheless consistent across configurations and with the qualitative observation that access control vulnerabilities are readily distinguishable as exploitable from source code, which makes an adversarial mitigation pass more likely to produce spurious rejections than useful filtering for this category.

CFG/DFG graph information. Graph removal reduces unchecked low-level call F1 from 0.583 to 0.459 and raises arithmetic F1 from 0.786 to 0.966 (precision = 1.000). These opposing effects are consistent with a single principle: graph inputs are beneficial for categories requiring non-local state dependency reasoning (unchecked calls, reentrancy) and harmful for localized single-expression faults (arithmetic).

Critic-driven repair. The Critic contributes primarily to efficiency. For reentrancy, Full and No Critic achieve the same exploit count but the full pipeline requires 22 execution retries versus 28 for No Critic, a 21% reduction at comparable F1. For structural environment-dependent categories, the Critic introduces overhead on repairs that cannot converge in a deterministic sandbox.

Table 6.3 summarizes component recommendations per category based on the ablation study, where ✓ indicates the component should be enabled and × indicates it should be disabled. Because the No Critic result for Unchecked Calls (UC) is confounded, No Attack is used as the conservative recommendation. Environment-dependent categories (Front-Running, Bad Randomness, Time Manipulation) remain outside execution scope.

6.4 RQ3: Computational Cost and Runtime Profile

Table 6.4 details pipeline execution time (in seconds) and estimated API cost (in USD) across the Base (detection with a single synthesis attempt), Full (all stages enabled), and No Critic (repair loop disabled) configurations.

Table 6.3 Category-adaptive component recommendations from ablation

Category	Attack Plan	Validation	CFG/DFG	Best Config	F1
Reentrancy	✓	✓	✓	Full	0.900
Arithmetic	✓	✓	×	No Diagram	0.966
Access Control	✓	×	✓	No Validation	0.941
Unchecked Low-Level	×	✓	✓	No Attack‡	0.764
Denial of Service	✓	✓	✓	Full (tied)	0.444
Front-Running		—		Scope limit	—
Bad Randomness		—		Scope limit	—
Time Manipulation		—		Scope limit	—

Table 6.4 Pipeline wall-clock time and estimated API cost

Category	**Base**		**Full**		**No Critic**	
	Time (s)	**Cost ($)**	**Time (s)**	**Cost ($)**	**Time (s)**	**Cost ($)**
Reentrancy	4 188	4.30	10 028	13.33	10 276	14.55
Arithmetic	1 146	1.67	2 825	3.48	3 333	4.15
Access Control	1 854	2.63	3 712	5.51	2 858	5.76
Unchecked Calls	4 735	6.93	10 253	16.02	11 993	17.86
Denial of Svc.	612	0.64	1 407	1.85	575	0.72
Front-Running	604	0.63	1 903	2.46	1 156	1.67
Bad Randomness	935	1.24	1 754	2.50	1 743	2.51
Time Manip.	388	0.45	749	1.13	392	0.76
Overall	**14 462**	**18.49**	**32 630**	**46.27**	**32 325**	**47.97**

Relative to Base, the Full pipeline takes 2.26× as long in wall-clock time (14,462 s versus 32,630 s across the benchmark) and costs 2.50× as much in API charges ($18.49 versus $46.27). The pipeline is uniformly LLM-call-bound: exploit generation accounts for 71–92% of Full runtime per category.

Parallelization. Because the pipeline is LLM-call-bound rather than compute-bound, throughput is governed by concurrency of LLM requests, not local resources. ExploitGen exploits two independent axes of parallelism. Across contracts, each target is processed in an isolated Foundry

workspace on a distinct Anvil instance (Chapter 5), so contracts run concurrently with no shared state. Within a contract, the per-candidate branches—each candidate's characterization, validation, planning, and synthesis—are mutually independent and execute as concurrent asynchronous tasks, bounded only by the provider's rate limit. The wall-clock figures are therefore dominated by provider latency and rate-limiting rather than by the framework, so the per-contract cost ($0.33) and per-contract call count are the more stable, platform-independent measures.

Runtime should not, however, be read as a disadvantage relative to static analyzers. A tool such as Slither returns results in seconds because it performs no execution and no LLM inference—but that speed is precisely what produces its 1,866 unconfirmed findings at 0.06 precision on this benchmark. ExploitGen spends orders of magnitude more time per contract to do something Slither does not attempt: execute a candidate and confirm exploitability against a patched counterpart before reporting it. The relevant comparison is therefore not seconds-to-output but effort-to-actionable-finding. Measured that way, the pipeline's runtime buys a reviewable output set, the 86 confirmed exploits reach a human alongside 30 explicitly unconfirmed candidates, whereas the static analyzer's near-instant output transfers the entire triage burden of its 1,866 unconfirmed findings onto the auditor We accordingly report absolute cost rather than a head-to-head speed ranking, which would reward exactly the fast-but-unverified behavior this thesis is designed to avoid.

Ablating the Critic reveals a category-dependent efficiency trade-off. For exploitable categories (reentrancy, arithmetic, unchecked calls), removing the Critic increases total time as unguided retries exhaust the synthesis budget on attempts a Critic diagnosis would have short-circuited. For structurally constrained categories (DoS, front-running, time manipulation), the Critic introduces overhead on repairs that cannot converge in a deterministic sandbox. The No Critic total (32,325s) nearly equals Full (32,630s), masking large opposing category-level effects.

6.5 Threats to Validity

6.5.1 Baseline Comparability

Slither provides a valid external reference (recall = 0.611 at precision = 0.061, all-severity counting) evaluated under identical conditions on SB Curated. The ExploitGen-Detection recall figure (0.93 vs. 0.619 verified) reflects the addition of executable verification rather than a controlled detection comparison.

6.5.2 Dataset Scale

Four categories contain fewer than ten instances. Results for denial of service (6 instances), front-running (4), bad randomness (8), and time manipulation (5) are directional indicators rather than statistically reliable estimates. One missed exploit shifts recall by 13–25 points in these categories.

6.5.3 Single-Run Ablation and Model Non-determinism

Each of the six configurations was executed once per category. The underlying models are stochastic and no seed is exposed by the provider API, so every reported per-category value carries an unquantified run-to-run variance. The magnitude of that variance is visible within the study itself: identical configurations differ at the detection stage across runs, by 12 detections for unchecked low-level calls (100 versus 88) and by 2 for access control (32 versus 30), before any ablated component acts. In categories with fewer than twenty instances, a two-instance swing moves F1 by roughly 0.10, which is the same order as most of the component effects reported in Section 6.3. The ablation results should therefore be read as directional evidence about which components help which categories, not as point estimates of component contribution, and the category-adaptive recommendations of Table 6.3 are correspondingly provisional. Repeated runs with reported mean and range are the appropriate remedy and are left to future work.

6.5.4 Detection vs. Generation Count Discrepancy

A single exploit may confirm multiple flagged instances simultaneously (e.g., three bad-randomness annotations covered by one PoC), so the generation phase counts such cases as one exploitable vulnerability, contributing to differences between raw detection counts and the *existed* figures.

6.5.5 Execution Model Constraints

Counterfactual confirmation requires an exploit to fail on a signature-preserving patched contract. This is not constructible for front-running (requires transaction-ordering) or bad randomness (depends on miner-controlled values fixed in Anvil), making both categories structurally out of scope.

6.5.6 Cross-Generation Sensitivity and Generalizability

The models used in the evaluation of Chapter 6 were not snapshot-pinned, so silent provider-side updates to the underlying models may affect the exact reproducibility of the reported quantitative results. More fundamentally, the pipeline's performance depends on frontier-class code-generation capability with native reasoning tokens. Repeating the full ablation with both tiers bound to pinned prior-generation snapshots (Table 5.1) reduces Full-configuration F1 from 0.900 to 0.230 for reentrancy and from 0.727 to 0.467 for access control (Table 6.5). The degradation is not uniform across configurations. For reentrancy the ordering is preserved: Full still exceeds Base by 0.20, so the components retain their direction of effect even when absolute performance collapses. For access control the ordering inverts: No Attack Info becomes the strongest arm at 0.563 while Full falls to fourth, so the category-adaptive recommendation of Table 6.3 does not transfer across model generations. Both comparisons are single-run and carry the same non-determinism caveat as Section 6.3; detection counts differ between arms within the prior-generation configuration by up to a factor of two. Two consequences follow.

Table 6.5 Cross-generation ablation: F1 per configuration under the reported model pair and under the pinned prior-generation pair. Single run per cell.

Config	Reentrancy		Access Control	
	gpt-5.x	gpt-4.1	gpt-5.x	gpt-4.1
Base	0.400	0.033	0.629	0.323
Full	0.900	0.230	0.727	0.467
No Attack Info	0.700	0.066	0.848	0.563
No Critic	0.844	0.333	0.800	0.533
No Validation	0.814	0.333	0.941	0.452
No Diagram	0.867	0.133	0.774	0.533

First, for reproducibility, future evaluations should pin specific model snapshots and report provider update dates. Second, for generalizability, because the prompt templates and SCoT blueprint formats are calibrated to frontier-class reasoning models, sustaining performance as model providers and capabilities evolve will require periodic recalibration of these templates and synthesis constraints; developing category-specific prompts that perform robustly across a wider range of models is identified as a direction for future work (Conclusion).

6.5.7 Metric Definition

Precision is defined over synthesis-stage candidates; a total-detection denominator would yield lower values. The No Critic unchecked-calls run produced 100 detections versus 88 in the full pipeline, attributable to LLM non-determinism in the detection phase rather than Critic influence.

6.5.8 Adversarial Inputs and Prompt Injection

When autonomous agents process smart contracts, especially those sourced from the wild, there is an inherent risk that the source code contains adversarial comments, string literals, or variable names designed to manipulate the LLM's behavior (i.e., prompt injection). While the benchmark evaluated in this work consists of known, benignly formatted vulnerabilities, a

production deployment of this pipeline faces this threat. The current architecture mitigates this risk through rigid schema-constrained generation (enforced via Pydantic); by strictly limiting the agent's output format to predefined JSON structures, the pipeline reduces the attack surface for instruction override during the generation and critic phases.

6.5.9 Data Contamination and Benchmark Memorization

A fundamental threat to validity when evaluating large language models is the potential inclusion of the SB Curated benchmark within the models' pre-training corpora. To mitigate the impact of memorization, our evaluation relies exclusively on the pipeline's ability to synthesize novel, executable exploit logic that passes a rigorous counterfactual verification gate (succeeding on vulnerable contracts while failing on patched versions), rather than merely generating static payload strings. While exact training compositions for frontier models remain proprietary, this dynamic execution requirement ensures that the system is evaluated on its functional reasoning rather than pure regurgitation.

6.5.10 Oracle Category Scoping

During the evaluation on SB Curated, the known category structure of the benchmark was used to scope each run to the relevant detection agent (Section 4.2). This isolates the exploit-generation pipeline from cross-category detection noise, but it is an oracle: it is not available in deployment, where all eight agents run in parallel. Three consequences follow. First, the 91.2% false-positive reduction reported against ExploitGen-Detection compares a system without scoping to a system with it, so it reflects both the addition of verification and the change in detection scoping, and an unquantified share is attributable to the latter, in particular, the *category boundary bleed* failure mode identified in Section 3.3 is eliminated by scoping rather than by verification. Second, every per-category precision in Table 6.1 is measured with the category known in advance. Third, the 0.93 $\rightarrow$ 0.619 recall figure likewise spans the scoping boundary. The within-system comparison is unaffected: Full versus Base uses identical scoping on both arms and yields a 23.1% synthesis-stage false-positive reduction and an F1 improvement from 0.582

to 0.675. Quantifying the oracle's contribution directly, via an all-agents-in-parallel arm, is left to future work.

6.6 Chapter Summary

This chapter has established that ExploitGen achieves an overall F1 of 0.675 at a confirmation yield of 0.741 on SB Curated, substantially improving on the detection-only baseline's precision of 0.47, with Slither reported as an indicative external reference (recall 0.611, precision 0.061, F1 0.111). The ablation study indicates that component selection may need to be category-adaptive: SCoT planning is essential for reentrancy but harmful for access control and unchecked calls. Chapter 7 interprets these results and discusses their implications for practical deployment.

CHAPTER 7

DISCUSSION

This chapter interprets the evaluation results, examines component interaction effects, discusses the implications for practical deployment, and identifies the limitations of the current approach.

7.1 Interpreting End-to-End Results

The overall F1 of 0.675 and confirmation yield of 0.741 represent a substantial improvement over the 0.47 precision of the detection-only paradigm, and are reported alongside Slither's F1 of 0.111 as an indicative external reference on the same benchmark. More importantly, every reported exploit is differentially confirmed against a patched counterpart: the system makes no unverified claims. This is a qualitative improvement over all existing approaches, which report unconfirmed candidates whose exploitability the human auditor must manually assess.

The reentrancy category's F1 of 0.900 confirms the initial hypothesis that categories with well-defined, single-transaction exploit structures are most amenable to automated PoC generation. The exploit maps directly onto the three-phase SCoT blueprint: a setup phase that deploys an attacker contract, an exploit phase that triggers the callback, and a verification phase that computes the balance delta. This structural alignment between the pipeline's planning format and the exploit's execution pattern is the primary driver of reentrancy's strong performance.

The unchecked low-level calls category's relatively low F1 (0.583), despite being the largest category, reveals a fundamental challenge for causal verification: the *effect* of an unchecked failed call is often an incorrect state that is difficult to characterize in assertions that fail specifically when the call does not fail. Unlike reentrancy (where the exploit drains a measurable balance), the harmful outcome of an unchecked call may be an absence of expected state update rather than a positive state change, making state-delta assertions harder to construct.

7.2 Component Interaction Effects

The ablation study reveals non-trivial component interaction effects that a single-component analysis cannot capture. The most important is the interaction between SCoT planning and the synthesis retry budget.

For access control, the SCoT blueprint for a phishing attack (requiring a forwarding contract to exploit `tx.origin` authentication) produces structurally complex Solidity with multiple contract definitions and complex initialization sequences. This complexity increases the probability of compilation failure on the first synthesis attempt, consuming retry slots before a passing test is reached. Without SCoT planning, the synthesizer produces simpler code that may not capture the full attack nuance but succeeds syntactically, yielding a higher F1 at the cost of some false positive risk.

This finding has a direct implication for system design: the SCoT blueprint format should be calibrated to the synthesis complexity appropriate for each vulnerability category. For categories where the exploit is a simple two-step call, a lightweight blueprint format reduces synthesis overhead without sacrificing planning quality.

7.3 The Detection–Verification Gap: Quantified Impact

The preliminary study and full evaluation together provide quantitative bounds on the detection–verification gap for the SB Curated benchmark:

- Without verification: 342 false positives, precision 0.47, F1 (recall-weighted) limited by noise
- With verification: 86 confirmed exploits and 30 explicitly unconfirmed candidates, confirmation yield 0.741, F1 0.675
- False positive reduction: 91.2%

In practical terms, this means that a human auditor using the detection-only system would need to manually investigate 342 + 306 = 648 findings to confirm 306 true vulnerabilities. Using the

full pipeline, the same auditor receives 116 items: 86 confirmed exploits requiring only assertion review, plus 30 unconfirmed candidates requiring full manual assessment, a 5.6× reduction in volume and a qualitative change in the effort each item demands, with every confirmed finding pre-validated by an executable exploit.

7.4 Limitations of the Current Approach

7.4.1 Single-Transaction Execution Model

The pipeline's counterfactual confirmation requirement cannot be satisfied for vulnerabilities whose exploitation requires multiple separate transactions (flash loan attacks, governance manipulation) or environmental conditions (transaction ordering, miner collusion) that cannot be reproduced in a local Anvil context. This limits coverage to vulnerabilities expressible as single-transaction EVM interactions.

7.4.2 Benchmark Scale and Distribution

SB Curated contains 139 vulnerabilities across eight categories, with four categories having fewer than ten instances. The results for DoS, front-running, bad randomness, and time manipulation are directional indicators rather than statistically reliable estimates. Evaluation on a larger, more diverse benchmark is required to establish statistical reliability across all categories.

7.4.3 Component Calibration

The ablation study identifies miscalibration of the validation stage for access control, arithmetic, bad randomness, and front-running categories. Category-adaptive falsification heuristics, developed through targeted prompt engineering and evaluation on a held-out validation set, represent the highest-priority improvement for the current system.

7.4.4 Dependence on Patch Correctness

Counterfactual confirmation establishes that a proof-of-concept succeeds on the vulnerable contract C and fails on the patched counterpart C^*. This is evidence of exploitability only under the assumption that C^* is a *correct and minimal* fix: one that removes the target vulnerability and alters no other behavior. Two deviations would undermine the verdict. An over-broad patch that changes behavior beyond the fix could cause the PoC to fail on C^* for reasons unrelated to the vulnerability, yielding a false confirmation; an incomplete patch that addresses only one sub-condition could cause a PoC exercising that sub-path to fail while the vulnerability persists, an instance of the coincidental-invalidation risk also noted for patch-differential validation elsewhere Andersson *et al.* (2025). In this work patch correctness is controlled rather than proven: patches are signature-preserving, single-vulnerability, and manually reviewed by the authors, and the single-purpose structure of SB Curated contracts makes a minimal patch a complete fix with no cross-vulnerability interaction. As an additional check, a sample of confirmed cases was inspected manually, and in each the generated proof-of-concept was found to be a valid exploit that exercises the intended vulnerability rather than passing on an incidental assertion; these are informal spot checks, not an exhaustive audit. This does not constitute a formal guarantee. Confirmation therefore does not eliminate manual verification but relocates it: rather than assessing whether each finding is exploitable, the auditor confirms that the generated assertion captures a genuine security violation and that the patch is a faithful fix — a substantially smaller task, reflected in the reduction from 648 findings to 116 items requiring review (Chapter 6), but not a null one. Automatically establishing patch correctness, or generating a patch together with a proof of its correctness, is left to future work.

7.4.5 Contract Complexity and Dependency Scope

Evaluation uses the SB Curated benchmark, consisting of simple, self-contained smart contracts. The current exploit test generation pipeline is optimized for these single-file targets and does not natively resolve deep multi-file inheritance trees or external package libraries. Additionally, the confirmation environment relies on an isolated, clean-room local EVM node. Exploits whose

execution feasibility depends on pre-existing mainnet state, live liquidity, or external oracle values would require fork-based execution, which is left to future work.

7.5 Implications for Practical Deployment

For practical deployment in a CI/CD pipeline or manual audit workflow, the ablation findings suggest that a category-adaptive configuration should be applied. Reentrancy should use the full pipeline; arithmetic should disable graph inputs; access control should disable validation (the No Validation configuration achieves the highest measured F1 of 0.941 for this category, while attack planning remains beneficial relative to its own ablation). Taking the best-performing configuration per category from Table 6.2 would give a mean F1 of 0.893 across the four in-scope, non-small categories. That figure is an upper bound rather than a performance estimate: it selects each configuration using the same data on which it is then evaluated, so it capitalises on run-to-run variance in categories with as few as fifteen instances.

The cost profile ($46.27 total for 139 contracts) suggests that the full pipeline is economically viable as a pre-audit screening tool for protocols with assets under management exceeding the cost of a manual audit ($10,000–$100,000+). The per-contract cost of approximately $0.33 compares favorably to the hourly rate of a senior smart contract auditor.

7.6 Chapter Summary

This chapter has interpreted the evaluation results, identified component interaction effects, quantified the practical impact of the detection–verification gap, and discussed the limitations and deployment implications of the current approach. The conclusion synthesizes these findings into directions for future research.

CONCLUSION AND RECOMMENDATIONS

Summary of Contributions

This thesis has addressed the detection–verification gap in automated smart contract auditing: the failure of LLM-based detection systems to distinguish genuinely exploitable vulnerabilities from benign code patterns that superficially resemble them. The core insight is that this gap can be closed by requiring every reported vulnerability to be demonstrated via an executable proof-of-concept test that passes on the vulnerable contract and fails on a semantically patched counterpart.

ExploitGen operationalizes this insight through a LangGraph-orchestrated multi-agent pipeline integrating eight taxonomy-specialized detection agents with a multi-stage verification sequence spanning static graph extraction, falsification-first validation, semantic characterization, SCoT-based attack planning, exploit synthesis with iterative repair, and counterfactual confirmation. Evaluated on the SB Curated benchmark, the system counterfactually confirms 86 exploits, with detection spanning all eight categories and counterfactual verification in scope for six,[1] achieving an overall F1 of 0.675 and a confirmation yield of 0.741, with every result causally verified. False positives are reduced from 342 (detection-only paradigm) to 30 (91.2% reduction), demonstrating that the verification pipeline is structurally effective at its primary purpose.

The ablation study establishes that no single component configuration is optimal across all vulnerability classes. SCoT-based attack planning yields the largest gain for reentrancy (+0.200 F1) but degrades performance for access control where synthesis complexity outweighs planning benefit. Falsification-first validation improves reentrancy precision but over-rejects access control candidates. Graph grounding is beneficial for non-local call chain analysis (unchecked

[1] Four of the 86 confirmations arise in front-running and bad randomness, which Section 6.5 identifies as structurally out of scope for counterfactual verification. They are retained in the totals but are not counterfactual demonstrations of equal strength.

calls) and harmful for localized expression-level faults (arithmetic). These findings establish that *component selection must be adapted to the vulnerability class*: a category-adaptive pipeline is expected to substantially outperform the current uniform configuration.

The four contributions of this thesis, the LangGraph architecture, broad-taxonomy counterfactual verification, the SCoT trade-off analysis, and the augmented evaluation dataset, collectively advance the state of the art in verification-backed smart contract auditing.

Future Research Directions

Multi-Transaction Exploit Synthesis. Extending the pipeline's execution model to support multi-transaction exploit sequences would enable coverage of flash loan attacks, governance manipulation, and other complex DeFi exploit patterns that currently produce inconclusive verdicts. This requires a stateful replay mechanism beyond the current single-transaction Foundry test format.

Category-Adaptive Component Selection. The ablation study provides a per-category component recommendation. Automating this selection, learning which component configuration to apply based on properties of the detected vulnerability, would eliminate the need for manual calibration and potentially close the gap between the current uniform configuration and the estimated adaptive optimum.

Larger and More Diverse Benchmarks. Evaluation on a benchmark with more instances per category (particularly DoS, front-running, bad randomness, and time manipulation) is required to establish statistical reliability across all categories. The CodeArena and Solodit historical audit databases represent potential sources for a significantly larger evaluation set.

Model-Agnostic Calibration. The current prompt templates and SCoT blueprint formats are calibrated to frontier-class models with native reasoning. Developing category-specific

prompt templates that perform well across a wider range of models would improve accessibility and robustness to model provider changes.

Integration with Formal Verification. Combining ExploitGen's executable verification with formal verification tools for cases where a PoC cannot be constructed (inconclusive verdicts) would provide stronger completeness guarantees. A hybrid pipeline could use formal verification as a fallback when the PoC synthesis budget is exhausted.

Generalization Beyond Smart Contracts. The counterfactual confirmation principle at the core of this thesis, reporting a flagged vulnerability only when an executable proof-of-concept succeeds on the vulnerable code and fails on a patched counterpart, is not specific to smart contracts. It transfers wherever two conditions hold: a candidate exploit can be executed deterministically, and a fixed version exists to serve as the counterfactual. The most direct extension is to other blockchain platforms (Move on Aptos and Sui, Rust-based Solana programs), which share the immutability and financial criticality that motivate executable confirmation and provide analogous local-node sandboxes. More broadly, the principle is well suited to vulnerability *reproduction*, where the security patch already exists in a project's commit history and the project's own test harness or a container supplies the execution environment; this is precisely the patch-differential setting in which prior proof-of-concept generators such as PoCo Andersson *et al.* (2025) and PoCShift Sun *et al.* (2025) operate, and extending it from smart contracts to conventional software is a natural next step. The taxonomy-specialized detection stage generalizes more readily still, to any codebase with an established weakness taxonomy such as CWE. What must be rebuilt per domain is the execution sandbox and the per-category success oracle: a state-delta assertion here, a sanitizer trip for memory-safety bugs, an unauthorized-action assertion for access-control flaws in web services. The method is correspondingly weakest for *discovering* entirely new vulnerability classes, where no patch yet exists to difference against, and for targets whose exploitation cannot be reproduced deterministically, the

same limitation identified for front-running and bad randomness in Chapter 6, generalized to concurrency bugs, timing side-channels, and distributed-system faults.

Real-World Protocol Evaluation. Evaluating ExploitGen on a corpus of deployed DeFi protocols, where ground truth is established by historical audit reports rather than benchmark annotations, would provide the most direct evidence of practical utility. Responsible disclosure protocols must be established for any newly discovered vulnerabilities.

BIBLIOGRAPHY

Alam, M. T., Paswan, P. K., Halder, R. & Maiti, A. (2026). When Tools Disagree: Agentic Adjudication and Repair of Smart Contract Vulnerabilities. *2026 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*.

Andersson, V., Bobadilla, S., Hobbelhagen, H. & Monperrus, M. (2025). PoCo: Agentic Proof-of-Concept Exploit Generation for Smart Contracts. *ACM Transactions on Software Engineering and Methodology*. doi: 10.1145/3816704. Just Accepted.

Chen, C., Su, J., Chen, J., Wang, Y., Bi, T., Yu, J., Wang, Y., Lin, X., Chen, T. & Zheng, Z. (2025). When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *ACM Transactions on Software Engineering and Methodology*, 34(4), 1–30.

Choi, J., Kim, D., Kim, S., Grieco, G., Groce, A. & Cha, S. K. (2021). SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 227–239. doi: 10.1109/ASE51524.2021.9678888.

DefiLlama. [Available: defillama.com/hacks (accessed 2025)]. (2025). DeFi Hacks.

Deng, W., Wei, H., Huang, T., Cao, C., Peng, Y. & Hu, X. (2023). Smart Contract Vulnerability Detection Based on Deep Learning and Multimodal Decision Fusion. *Sensors*, 23(16), 7246.

Ding, H., Liu, Y., Piao, X., Song, H. & Ji, Z. (2025). SmartGuard: An LLM-Enhanced Framework for Smart Contract Vulnerability Detection. *Expert Systems with Applications*, 269, 126479.

Durieux, T., Ferreira, J. a. F., Abreu, R. & Cruz, P. (2020). Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*, pp. 530–541. doi: 10.1145/3377811.3380364.

Ethereum Foundation. [Available: docs.soliditylang.org (accessed 2025)]. (2024). Solidity Programming Language Documentation.

Feist, J., Grieco, G. & Groce, A. (2019). Slither: A Static Analysis Framework for Smart Contracts. *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15.

Fezari, M. & Ali-Al-Dahoud, A. A. D. (2023). From GPT to AutoGPT: A Brief Attention in NLP Processing Using DL. Retrieved from: Preprint.

Grieco, G., Song, W., Cygan, A., Feist, J. & Groce, A. (2020). Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 557–560.

Hamiaz, M. K. & Driss, M. (2025). Ethereum Smart Contracts Under Scrutiny: A Survey of Security Verification Tools, Techniques, and Challenges. 14(6), 226. doi: 10.3390/computers14060226.

Jin, Y., Li, C., Fan, P., Liu, P., Li, X., Liu, C. & Qiu, W. (2025). LLM-BSCVM: LLM-Based Blockchain Smart Contract Vulnerability Management Framework. *Algorithms and Architectures for Parallel Processing (ICA3PP)*, pp. 44–56.

Karanjai, R., Xu, L. & Shi, W. (2025, Nov). Securing Smart Contract Languages with a Unified Agentic Framework for Vulnerability Repair in Solidity and Move. *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*, pp. 223–232. doi: 10.1109/AIware69974.2025.00032.

Kezadri Hamiaz, M. & Driss, M. (2025). Ethereum Smart Contracts Under Scrutiny: A Survey of Security Verification Tools, Techniques, and Challenges. *Computers*, 14(6), 226. doi: 10.3390/computers14060226.

Krichen, M. (2023). Strengthening the Security of Smart Contracts Through the Power of Artificial Intelligence. *Computers*, 12(5), 107.

Krupp, J. & Rossow, C. (2018). teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pp. 1317–1333. Retrieved from: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.

LangChain. [GitHub: github.com/langchain-ai/langgraph (accessed 2025)]. (2024). LangGraph: Build Stateful, Multi-Actor Applications.

Li, J., Li, G., Li, Y. & Jin, Z. (2025). Structured Chain-of-Thought Prompting for Code Generation. *ACM Transactions on Software Engineering and Methodology*, 34(2), 1–23.

Mueller, B. (2018). Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit. *9th Annual HITB Security Conference (HITBSecConf)*.

NCC Group. (2018). DASP — Decentralized Application Security Project Top 10. Retrieved from: https://dasp.co/.

Nikolić, I., Kolluri, A., Sergey, I., Saxena, P. & Hobor, A. (2018). Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *Proceedings of the 2018 Annual Computer Security Applications Conference (ACSAC '18)*, pp. 653–663. doi: 10.1145/3274694.3274743.

SmartBugs Project. [Available: github.com/smartbugs/smartbugs-curated (accessed 2025)]. (2024). SmartBugs Curated: A Dataset of Vulnerable Solidity Smart Contracts.

SmartContractSecurity. (2020). SWC Registry: Smart Contract Weakness Classification. Retrieved from: https://swcregistry.io/.

Sun, K., Xu, Z., Li, K., Zhang, L., Feng, Y., Wu, D. & Liu, Y. (2025). Learning from the Past: Real-World Exploit Migration for Smart Contract PoC Generation. *40th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*.

Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X. & Liu, Y. (2024). GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. doi: 10.1145/3597503.3639117.

Tamberg, K. & Bahsi, H. (2025). Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study. *IEEE Access*.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F. & Vechev, M. (2018). Securify: Practical Security Analysis of Smart Contracts. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 67–82.

Vidal, F. R., Ivaki, N. & Laranjeiro, N. (2024). OpenSCV: An Open Hierarchical Taxonomy for Smart Contract Vulnerabilities. *Empirical Software Engineering*, 29(4), 101. doi: 10.1007/s10664-024-10446-8.

Wang, S. J., Pei, K. & Yang, J. (2024). SmartInv: Multimodal Learning for Smart Contract Invariant Inference. *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 2217–2235. doi: 10.1109/SP54263.2024.00126.

Wei, Z. et al. (2025). Advanced Smart Contract Vulnerability Detection via LLM-Powered Multi-Agent Systems. *IEEE Transactions on Software Engineering*, 51(10), 2830–2846. doi: 10.1109/TSE.2025.3597319.

Wood, G. (2014). *Ethereum: A Secure Decentralised Generalised Transaction Ledger*.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R. & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *The Eleventh International Conference on Learning Representations (ICLR)*.

Zhang, Q., Wang, Y., Li, J. & Ma, S. (2020). EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 116–126. doi: 10.1109/SANER48275.2020.9054822.