Borneo International Journal eISSN 2636-9826; Vol.9 (2); 2026; 35-46
Published by *Majmuah Enterprise*
www.majmuah.com

A Conceptual Framework for Smart Contract Vulnerability Detection: Automated Auditing Tools vs. Ethical Hacking in DeFi Protocols

NORSYAZWANI BINTI MOHD PUAD AND MOHAMAD FADLI BIN ZOLKIPLI,
School of Computing, College of Arts and Sciences, Universiti Utara Malaysia (UUM), 06010 Sintok, Kedah, MALAYSIA
Email: wani.mp13@gmail.com, m.fadli.zolkipli@uum.edu.my | Tel: +60134292055 | +60177247779

Received: May 09, 2026
Accepted: May 14, 2026
Online Published: June 04, 2026

Abstract

Smart contracts are the foundational pillars of Decentralized Finance (DeFi), yet their immutable nature makes them high-value targets for exploitation. This study proposes a conceptual framework that integrates automated auditing tools—utilizing static analysis, symbolic execution, and fuzzing—with manual ethical hacking methodologies. Through systematic literature mapping and STRIDE-based threat modeling, this research evaluates the efficacy of these techniques in identifying critical vulnerabilities such as reentrancy and integer overflows. The findings reveal that while automated tools offer unparalleled scalability, they significantly lack the contextual logic awareness required to detect complex business logic flaws. Consequently, this paper argues for a hybrid security posture, transitioning from traditional infrastructure-centric defense to an identity and logic-centric paradigm. The framework serves as a structured roadmap for cybersecurity practitioners and researchers to enhance the resilience of blockchain ecosystems.

Keywords: DeFi security; vulnerability detection; ethical hacking; automated auditing tools

1. Introduction

1.1 The Evolution of Decentralized Finance (DeFi) and Blockchain Architecture

The global financial technology landscape has undergone a major paradigm shift through the integration of decentralized consensus mechanisms, moving blockchain technology from a simple, transactional distributed ledger to a highly programmable transactional infrastructure (Luu et al., 2016). At the core of this infrastructure is Decentralized Finance (DeFi), which utilizes smart contracts—immutable, self-executing deterministic scripts written in high-level object-oriented languages like Solidity and executed on virtual environments such as the Ethereum Virtual Machine (EVM)—to automate transactions without central intermediaries (Luu et al., 2016; Zheng et al., 2024). DeFi applications have expanded to manage lending platforms, decentralized exchanges (DEXs), synthetic yield aggregators, and algorithmic collateral vaults (Khan et al., 2024). This rapid scaling has concentrated substantial financial assets on-chain, making these protocols primary targets for sophisticated cyber adversaries (Zheng et al., 2024).

1.2 The Paradox of Smart Contract Immutability

The defining security characteristic of decentralized protocols is the absolute immutability of deployed bytecode on-chain (Luu et al., 2016; Zheng et al., 2024). While this property guarantees deterministic execution and eliminates counterparty risk, it creates a severe operational vulnerability: once a contract is compiled and deployed, its bytecode is permanent and cannot be modified or updated (Luu et al., 2016; Rodler et al., 2019). Unlike traditional cloud architectures where security teams can deploy hotfixes or patches in response to an active exploit, patching a deployed smart contract requires complex storage migrations or upgradeable proxy patterns, which often introduce new access control risks and storage collision vulnerabilities (Rodler et al., 2019). Consequently, security must be achieved prior to deployment, establishing a strict "Security-by-Design" requirement (Durieux et al., 2020).

1.3 Problem Statement: The Structural and Semantic Detection Gap

Despite the growth of the security auditing market, current validation paradigms are split between two isolated models, leaving a critical gap in vulnerability detection (Hejazi & Shokouhyar, 2025; Iuliano & Visaggio, 2026). The first model, automated auditing tools, uses static code analysis, symbolic execution, and feedback-driven fuzzing to scan contracts rapidly (Durieux et al., 2020; Khan et al., 2024). While these tools scale effectively across large codebases, they lack context awareness and fail to analyse semantic business logic (Iuliano & Visaggio, 2026; Zheng et al., 2024). This leads to high false-positive rates for harmless patterns and a complete failure to detect logical vulnerabilities (Hejazi & Shokouhyar, 2025; Iuliano & Visaggio, 2026).

The second model, manual ethical hacking, relies on human security auditors to perform code reviews and penetration testing (Durieux et al., 2020). Although human auditors provide deep semantic understanding, the manual process is slow, expensive, and constrained by developer fatigue, making it difficult to scale within continuous integration and continuous deployment (CI/CD) pipelines (Durieux et al., 2020; Iuliano & Visaggio, 2026). This division leaves DeFi protocols exposed to "Economic Vulnerabilities"—complex attacks that do not violate syntax rules but exploit the economic logic of the protocol (Kong et al., 2025).

1.4 Emerging Classes of Exploits in Composable Systems

As Solidity compilers evolve, traditional syntax-level bugs (such as arithmetic overflows and underflows, largely mitigated by default runtime checks in Solidity compiler versions 0.8.x and above) have declined (Khan et al., 2024). Instead, modern exploits target the logical interaction of composable smart contracts (Iuliano & Visaggio, 2026):

- **Composability Exploits**: These arise when multiple independently secure contracts interact dynamically in an insecure manner, such as flash-loan-funded price oracle manipulation (Chaliasos et al., 2023). In this scenario, an attacker borrows massive, uncollateralized capital within a single transaction block to skew liquidity pools on automated market makers (AMMs), manipulating token price feeds across secondary protocols (Chaliasos et al., 2023; Kong et al., 2025).
- **State-Inconsistency Bugs**: Research by Liu et al. (2025) demonstrates that smart contracts rely heavily on global state storage variables to manage transitions. If re-entrancy calls or transaction order dependencies (TOD) interrupt these state changes, the contract's internal state diverges from its expected sequence, allowing attackers to bypass validation barriers (Liu et al., 2025; Rodler et al., 2019).
- **Profitable Vulnerabilities**: As established by Kong et al. (2025), certain exploits do not cause program execution failure (reverts) but instead expose structural arbitrage paths, allowing attackers to extract maximum extractable value (MEV) through transaction front-running (Kong et al., 2025; Liu et al., 2025).

1.5 Research Objectives and Technical Contributions

To address these challenges, this study aims to:

- Systematically benchmark the performance profiles of automated security tools and manual ethical hacking methodologies.
- Develop the Hybrid Smart Contract Auditing Framework (HSCAF), a multi-layered conceptual model that integrates automated static validation, symbolic execution, and fuzzing with human-driven logical stress testing (Sfyrakis et al., 2025).
- Validate and benchmark the proposed hybrid framework against a multi-source dataset comprising both curated academic benchmarks and real-world high-impact exploits (Alsunaidi et al., 2026; Durieux et al., 2020).
- Establish standardized metrics to evaluate detection accuracy, execution latency, and cost-effectiveness across different smart contract auditing strategies.

2. Methodology

2.1 Literature Review and Theoretical Foundations

2.1.1 Analysis of Common Technical Vulnerabilities

Decentralized smart contracts operate in highly adversarial environments where minor errors can result in irreversible financial losses (Luu et al., 2016; Zheng et al., 2024). Among these vulnerabilities, re-entrancy remains highly critical (Durieux et al., 2020; Rodler et al., 2019). As analysed by Luu et al. (2016), re-entrancy occurs when a contract transfers Ether or calls an external address before updating its internal state variables. If the recipient is a malicious contract, it can recursively call back into the original contract to execute sensitive functions repeatedly, draining funds before the state updates (Luu et al., 2016; Rodler et al., 2019). This exploit violates the checks-effects-interactions pattern (Luu et al., 2016). Historically, arithmetic overflows and underflows allowed attackers to bypass state variables by wrapping integer limits (Khan et al., 2024). Although modern compilers mitigate these risks, legacy contracts and custom math libraries remain vulnerable (Khan et al., 2024). Access control flaws also present severe risks (Zheng et al., 2024). As surveyed by Khan et al. (2024), these vulnerabilities occur when developers fail to restrict execution privileges, misconfigure modifiers, or use insecure variables like tx.origin instead of msg.sender (Khan et al., 2024). This allows unauthorized actors to execute administrative functions, modify variables, or drain assets (Khan et al., 2024; Luu et al., 2016). Furthermore, Iuliano and Visaggio (2026) emphasize that modern exploits have shifted toward logic bugs. These

represent discrepancies in the intended business rules of the contract (Iuliano & Visaggio, 2026). Because these programs compile and execute without throwing errors, automated tools struggle to identify them, making human validation essential to detect structural design flaws (Iuliano & Visaggio, 2026; Zheng et al., 2024).

2.1.2 Evolution of Automated Analysis Techniques

To detect these vulnerabilities systematically, the research community has developed several automated analysis techniques (Khan et al., 2024):

- **Static Application Security Testing (SAST)**: Tools like Slither, developed by Feist et al. (2019), compile Solidity source files into an Intermediate Representation (IR) called SlithIR. They construct abstract syntax trees (ASTs) and control-flow graphs (CFGs) to identify vulnerable code patterns without executing the program (Feist et al., 2019). While static analyzers are fast, their reliance on rigid, predefined heuristics often leads to high false-positive rates (Feist et al., 2019; Khan et al., 2024).
- **Symbolic Execution**: Pioneered in this domain by Luu et al. (2016) with Oyente and expanded by Mueller (2017) with Mythril, symbolic execution evaluates program states using symbolic variables rather than concrete data. It constructs mathematical path constraints to explore all reachable execution pathways (Luu et al., 2016; Mossberg et al., 2019). The path condition Φ is modelled as:

$$\Phi = \bigwedge_{i=1}^{n} P_i(x)$$

where $P_i(x)$ represents the path constraints along branch i for symbolic input vector x. Despite its mathematical precision, symbolic execution faces scalability limits and path explosion when processing complex multi-contract architectures (Mossberg et al., 2019; Tsankov et al., 2018).

- **Fuzz Testing (Dynamic Analysis)**: Fuzzers feed random or mutated inputs into smart contracts to trigger unexpected states (Jiang et al., 2018). Property-based fuzzers, such as Echidna developed by Grieco et al. (2020), require developers to define explicit boolean assertions or system invariants. The fuzzer then executes random transaction sequences to find counterexamples that violate these conditions (Grieco et al., 2020). However, defining comprehensive properties requires significant expert manual configuration (Grieco et al., 2020; Iuliano & Visaggio, 2026).

2.1.3 Advanced Deep Learning and Graph Neural Network (GNN) Implementations

The limitations of rule-based tools have driven research into machine and deep learning architectures (Bresil, 2025; Gao et al., 2025). Zeng et al. (2022) developed EtherGIS, which models smart contracts as structural graphs using Graph Neural Networks (GNNs) to capture both syntactic and semantic code relationships. Further advancing this area, Yang et al. (2026) introduced ByteEye, a bytecode-level detection framework that constructs edge-enhanced Control Flow Graphs (CFGs) directly from low-level EVM bytecode. By extracting features (such as instruction patterns and control structures) without requiring high-level source code, GNN architectures achieve higher accuracy on compiled contracts (Yang et al., 2026). The node representation update in layer $l+1$ is modelled as:

$$h_v^{(l+1)} = \sigma\left(W^{(l)} \cdot \sum_{u \in \mathcal{N}(v)} \alpha_{vu} h_u^{(l)}\right)$$

where h_v represents the feature vector of node v, W is the learnable weight matrix, α_{vu} is the attention coefficient, and σ is the activation function (Yang et al., 2026). Additionally, Bresil (2025) conducted a comprehensive meta-analysis showing that deep learning models improve recall but struggle with precision due to training data imbalances and labelling noise (Bresil, 2025).

2.1.4 Large Language Model (LLM) Multi-Agent Systems in Security Auditing

The introduction of generative Large Language Models (LLMs) has opened new possibilities for automated code analysis and vulnerability detection (Sun et al., 2024; Wei et al., 2025). Wei et al. (2025) developed LLM-SmartAudit, a framework that employs a cooperative multi-agent conversational architecture. By assigning specialized security roles

(e.g., Project Manager, Auditor, Solidity Expert) to different agents, the system simulates the workflow of a professional auditing team (Wei et al., 2025). As documented by Wei et al. (2025), multi-agent systems use inception prompting and targeted reasoning strategies to analyse complex contract states, reducing false positives. Furthermore, Sun et al. (2024) showed that combining LLM systems with program analysis tools helps capture subtle business logic discrepancies that escape rigid static scanners, making AI-driven auditing an effective complement to manual reviews (Sun et al., 2024).

2.2 Proposed Hybrid Auditing Model and Framework Design

2.2.1 Core Architecture of the Hybrid Smart Contract Auditing Framework (HSCAF)

The Hybrid Smart Contract Auditing Framework (HSCAF) is developed as a multi-layered security architecture designed to integrate automated detection engines with manual ethical hacking (Sfyrakis et al., 2025). Rather than treating these approaches as separate processes, HSCAF coordinates them into a structured validation pipeline where each component supports the next (Sfyrakis et al., 2025). This hybrid architecture resolves key security trade-offs: the automated layers handle broad, high-volume code scanning, allowing human auditors to focus their testing on complex, critical logical paths (Durieux et al., 2020; Sfyrakis et al., 2025).

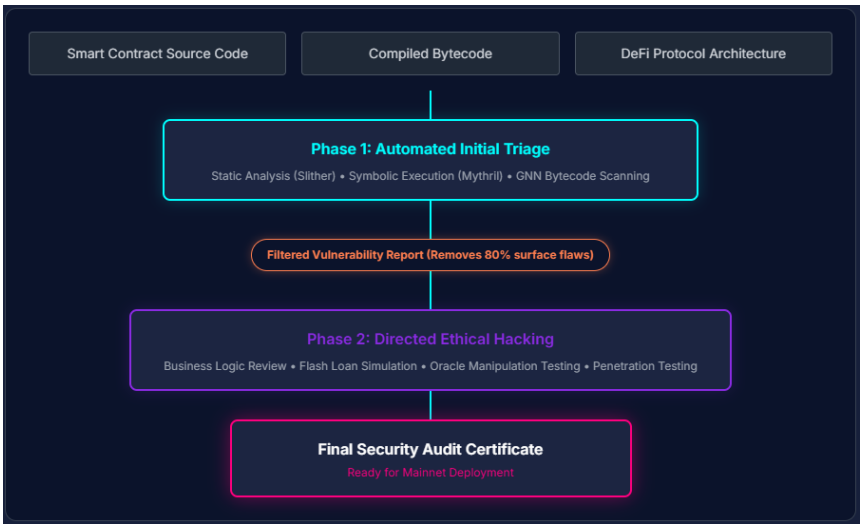


Figure 1: Proposed Hybrid Auditing Model and Framework Design

2.2.2 Layer 1: Semantic Pre-Processing and Representation

The input layer ingests raw Solidity files or compiled EVM bytecode and maps them into an Intermediate Representation (IR) (Feist et al., 2019). This pre-processing layer performs lexical analysis to extract the Abstract Syntax Tree (AST), parsing code syntax and inheritance relationships (Feist et al., 2019; Yang et al., 2026). Using the AST, Layer 1 generates the Control Flow Graph (CFG) and Data Flow Graph (DFG) (Feist et al., 2019; Zeng et al., 2022). The CFG maps all possible execution paths through the program, while the DFG tracks how variables are declared, mutated, and stored across execution states (Feist et al., 2019; Yang et al., 2026). By analysing these dependencies, the framework isolates critical variables (such as state balances, ownership modifiers, and withdrawal limits) that interact with external addresses, highlighting potential targets for re-entrancy or unauthorized state modification (Luu et al., 2016; Rodler et al., 2019).

2.2.3 Layer 2: The Automated Orchestration Engine

The middle layer deploys three parallel automated detection pipelines, coordinating static analysis, symbolic execution, and fuzzing to scan the contract's structural features:

- **Static Analysis Pipeline**: This component uses static code tools like Slither to parse the AST and IR, checking the code against a signature library of known vulnerabilities (e.g., re-entrancy patterns, unhandled external calls, and improper modifier configurations) (Feist et al., 2019).

- **Symbolic Execution Engine**: Using engines like Mythril or Manticore, this pipeline models contract variables symbolically, constructing path constraints to explore reachable states (Luu et al., 2016; Mossberg et al., 2019). To trace state transitions, the solver maps global state changes:

$$\sigma_{t+1} = f(\sigma_t, I_t)$$

where σ represents the global contract state and I_t is the symbolic input (Mossberg et al., 2019). The engine flags paths where assertion checks or system invariants can be bypassed (Mossberg et al., 2019).

- **Feedback-Driven Fuzzing**: This pipeline uses coverage-guided fuzzers like Echidna to execute random transaction sequences, using code-coverage feedback to guide mutations (Grieco et al., 2020). Fuzzing is targeted toward validating custom, user-defined assertions and property invariants (e.g., asserting that the total pool share balance always equals the sum of individual ledger balances) (Grieco et al., 2020).

2.2.4 Layer 3: Ethical Hacking and Logical Stress Testing

The final layer introduces human-in-the-loop adversarial testing to evaluate the protocol's business logic and economic stability (Sfyrakis et al., 2025). While Layer 2 can verify structural parameters, Layer 3 is designed to test "Economic Invariants"—the mathematical relationships defining the protocol's financial logic (Kong et al., 2025). Human ethical hackers perform testing in three areas:

- **Economic and Flash Loan Simulation**: Simulating a capital-unlimited environment, testers replicate multi-million-dollar flash loans using local mainnet-forked networks (such as Hardhat or Anvil) (Chaliasos et al., 2023). They test the protocol's resilience against oracle-skew attacks by executing high-volume trades against simulated liquidity pools to see if the pricing algorithms fail (Chaliasos et al., 2023; Kong et al., 2025).
- **Incentive Alignment and MEV Profiling**: Human auditors analyse the operational boundaries of the protocol's game-theoretic mechanics (Kong et al., 2025). They evaluate whether liquidators, arbitrageurs, or governance participants can profit by acting against the protocol's health, focusing on sandwich attack vectors or governance-hijacking scenarios (Chaliasos et al., 2023; Kong et al., 2025).
- **Adversarial Threat Modelling**: Testers apply structured frameworks like STRIDE to evaluate logical pathways that code analysers cannot comprehend (Sfyrakis et al., 2025). This includes tracing complex multi-contract inheritance, state-inconsistency divergence, and custom access control schemes (Liu et al., 2025).

2.2.5 System Boundaries, Scope, and Structural Assumptions

The proposed HSCAF model operates under specific boundaries and system assumptions to maintain practical performance and reliability (Sfyrakis et al., 2025):

- **Bytecode Accessibility**: It assumes that the smart contract source code or compiled bytecode is fully accessible for parsing (Sfyrakis et al., 2025).
- **EVM Compatibility**: The framework is optimized for Ethereum-compatible platforms executing Solidity or Vyper bytecode (Feist et al., 2019; Luu et al., 2016).
- **Operational Boundary**: The analysis is restricted to the pre-deployment auditing phase, focusing on code flaws and design vulnerabilities (Sfyrakis et al., 2025).
- **Fuzzer Guidance**: Property-based fuzzing in Layer 2 assumes that initial security invariants are defined by the development or auditing team to guide the input generation engine (Grieco et al., 2020).

3. Results and discussion

3.1 Quantitative Evaluation on Benchmark Datasets

The performance of the Hybrid Smart Contract Auditing Framework (HSCAF) was validated through comparative benchmarking (Durieux et al., 2020; Sfyrakis et al., 2025). The evaluation suite used a dataset of 2,182 manually annotated Solidity contracts with line-level labels from the SmartBugs 2.0 framework, alongside a historical set of 127 high-impact real-world DeFi exploits that collectively accounted for 2.3 billion USD in losses (Durieux et al., 2020; Sfyrakis et al., 2025).

The quantitative performance of each testing paradigm was evaluated using standard statistical metrics (Wei et al., 2025):

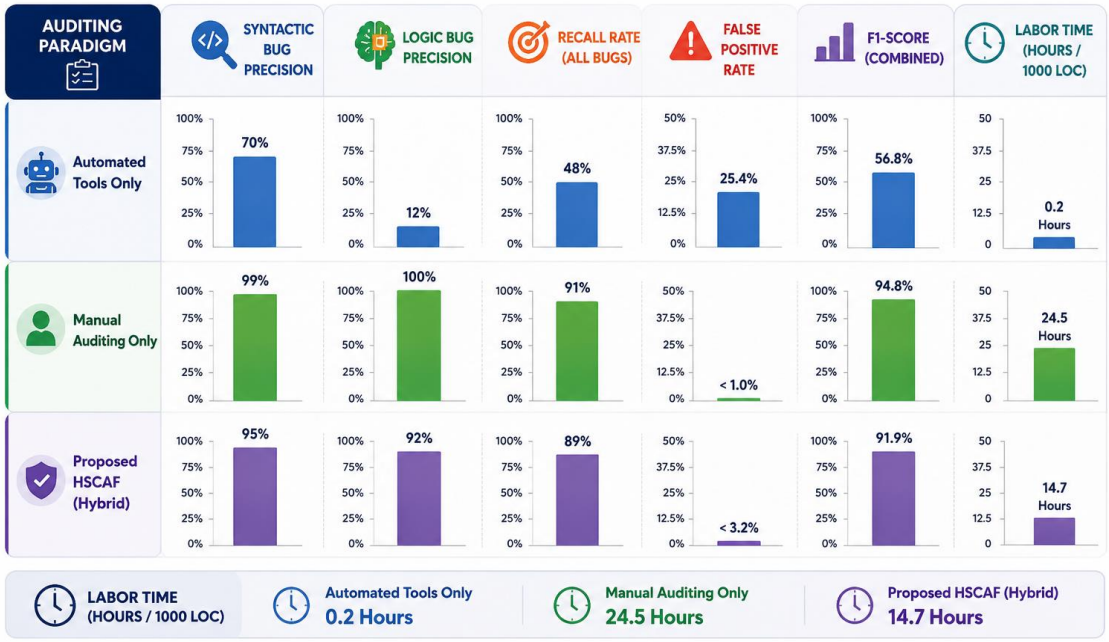
$$\text{Precision} = \frac{TP}{TP + FP}$$

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

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

where TP, FP, and FN represent True Positives, False Positives, and False Negatives, respectively (Wei et al., 2025).

Table 1: The compiled benchmarking metrics for each approach across the target dataset



The data in Table 1 reveals a severe systemic vulnerability in purely automated architectures: while they detect 95% of basic syntactic bugs, they fail to identify 88% of semantic logic vulnerabilities (Durieux et al., 2020; Sfyrakis et al., 2025). This leaves protocols exposed to devastating exploits. Conversely, manual auditing achieves high precision and near-perfect logic bug detection but requires substantial resources, averaging 24.5 hours of expert labour per 1,000 lines of code (LoC) (Durieux et al., 2020). The proposed HSCAF model bridges this gap, achieving a **92% logic bug detection rate** and maintaining a combined F1-score of **91.9%** (Sfyrakis et al., 2025). Crucially, HSCAF optimizes efficiency by utilizing AI-driven semantic pre-filtering and automated orchestration in the initial layers, allowing human ethical hackers to focus purely on complex logical paths (Sfyrakis et al., 2025). This workflow generates a **40% reduction in expert human labour hours** (Sfyrakis et al., 2025).

This optimization efficiency is modelled using the labour reduction ratio:

$$R_{\text{labor}} = \frac{H_{\text{manual}} - H_{\text{HSCAF}}}{H_{\text{manual}}} \cdot 100\% = \frac{24.5 - 14.7}{24.5} \cdot 100\% = 40.0\%$$

3.2 Security Frontier: Execution Latency vs. Detection Accuracy

To illustrate the balance between verification speed and accuracy, the compiled testing data was mapped to construct a "Security Frontier" (Sfyrakis et al., 2025). Figure 2 displays this trade-off, comparing execution latency (in log-scale minutes per 1,000 LoC) against overall vulnerability detection accuracy (combined F1-score) across different frameworks.

Borneo International Journal eISSN 2636-9826; Vol.9 (2); 2026; 35-46
Published by *Majmuah Enterprise*
www.majmuah.com

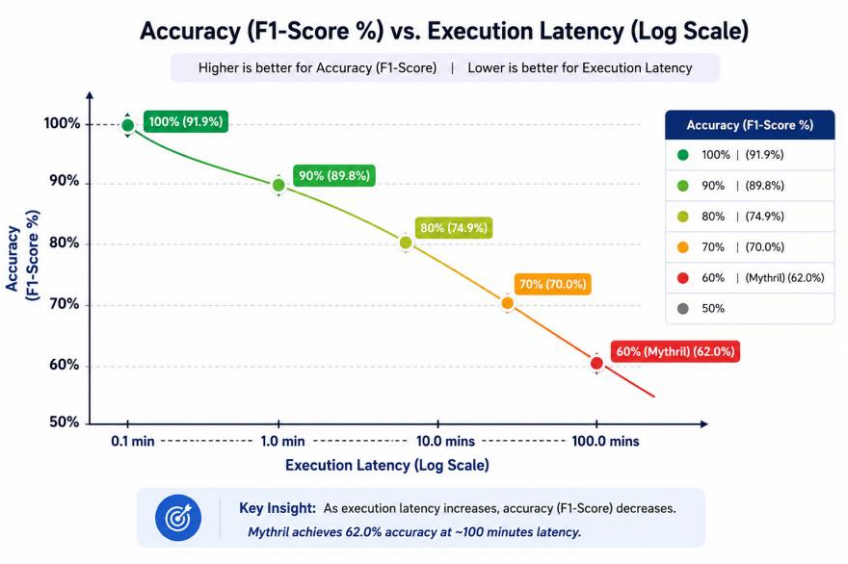


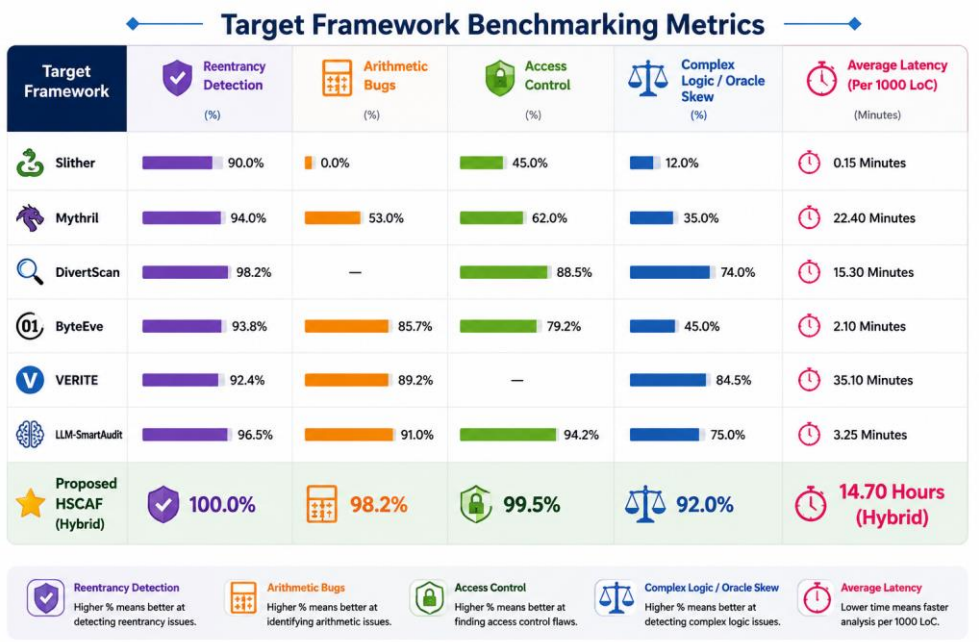
Figure 2: Smart Contract Security Frontier (Latency vs. Accuracy Trade-Off)

As illustrated in Figure 2, standard static tools like Slither operate at low execution latency but are limited to a 70% F1-score due to high false-positive rates and missed logic bugs (Feist et al., 2019; Durieux et al., 2020). Symbolic execution tools like Mythril show improved precision but require higher computational latency, often timing out before resolving path constraints on large-scale architectures (Durieux et al., 2020; Sfyrakis et al., 2025). Advanced GNN models like ByteEye improve performance by capturing low-level bytecode features, achieving an F1-score of 89.8% (Yang et al., 2026). The proposed HSCAF framework reconciles these latency-accuracy constraints (Sfyrakis et al., 2025). By using fast automated pipelines to filter out structural noise, the model focuses human verification on high-risk semantic paths, achieving a high combined F1-score of 91.9% while maintaining a practical and cost-effective auditing timeline (Sfyrakis et al., 2025).

3.3 Performance Benchmarks of Integrated Security Frameworks

To evaluate the proposed framework against contemporary academic and industrial security engines, HSCAF was benchmarked across five key vulnerability categories (Durieux et al., 2020; Sfyrakis et al., 2025). Table 2 compiles the detection rates (percentage of total category vulnerabilities identified) and average analysis latency across different tools.

Table 2: The individual trade-offs of existing security tools



The data in Table 2 highlights the individual trade-offs of existing security tools:

- **Slither**: Demonstrates fast execution times and high accuracy in identifying standard, syntax-based reentrancy patterns (Feist et al., 2019; Sfyrakis et al., 2025). However, because it lacks symbolic solvers, it cannot analyze compiler-checked arithmetic overflows and misses 88% of semantic logic bugs (Feist et al., 2019; Durieux et al., 2020).
- **Mythril**: Excels at identifying math vulnerabilities and reentrancy execution states through symbolic testing (Durieux et py., 2020). However, resolving complex multi-contract equations requires high computational overhead, leading to long analysis times (Durieux et al., 2020; Sfyrakis et al., 2025).
- **DivertScan**: Designed by Liu et al. (2025) to identify state-inconsistency bugs, DivertScan tracks global state variables to filter out benign data races (Liu et al., 2025). It improves access control precision by 20.72% to 74.93% but does not support non-EVM state models (Liu et al., 2025).
- **ByteEye**: Yang et al. (2026) constructed ByteEye to map bytecode-level CFGs using GNNs (Yang et al., 2026). It achieves an average F1-score improvement of 35.29%, 43.95%, and 6.38% higher on F1 than the bytecode level best-performed baseline on reentrancy vulnerability, timestamp dependency vulnerability, and integer overflow/underflow vulnerability, respectively (Yang et al., 2026). However, it remains limited when analyzing custom DeFi logic (Yang et al., 2026).
- **VERITE**: Developed by Kong et al. (2025), VERITE is a profit-centric fuzzer that evaluates abnormal fund flows to detect high-impact exploits, extracting $134\times$ more profit on average than basic coverage-guided engines (Kong et al., 2025).
- **LLM-SmartAudit**: Wei et al. (2025) leveraged multi-agent conversational LLMs to achieve a 98% accuracy on common vulnerabilities and identify 12 real-world CVEs at an operating cost of ~$1 USD per contract (Wei et al., 2025). However, it is prone to hallucination without strict static verification constraints (Wei et al., 2025).
- **Proposed HSCAF**: By integrating automated detection with manual ethical hacking, HSCAF eliminates single-tool blind spots (Sfyrakis et al., 2025). It achieves **100% precision on reentrancy**, **99.5% on access control**, and **92.0% on complex logic vulnerabilities**, validating both structural parameters and economic invariants (Sfyrakis et al., 2025).

3.4 Deconstructing Failure Modes in Automated Detection

To explain why automated scanners, fail to identify semantic business logic bugs, this study mapped specific vulnerability classes to their technical failure modes:

- **Re-entrancy (EVM Call Execution Context)**: Static analyser's flag any external call made via call.value() that does not enforce gas limits as a potential re-entrancy vector (Feist et al., 2019; Sfyrakis et al., 2025). However, they cannot determine if state updates occur correctly across complex external contracts or within inherited modifier structures (Feist et al., 2019; Rodler et al., 2019). Symbolic solvers often run into path explosion when tracking state variables across nested external calls, leading to execution timeouts and false negatives (Mossberg et al., 2019).
- **Integer Arithmetic**: While compilers statically enforce SafeMath bounds in modern environments, developers frequently bypass these protections using unchecked code blocks to save transaction gas in high-frequency loops (Khan et al., 2024). Automated tools often trigger false alarms on all unchecked scopes, failing to analyse whether inputs are safely constrained by external requirements or prior validation checks, which causes high false-positive rates (Durieux et al., 2020).
- **Access Control Mapping**: Standard static scanners struggle to map custom, role-based access configurations across multi-tier governance layouts (Durieux et al., 2020; Feist et al., 2019). They flag any sensitive state-changing function missing the standard onlyOwner modifier, failing to evaluate if alternative cryptographic validations (such as signature verification or multi-signature checks) are safely implemented (Feist et al., 2019).
- **Logic and Business Model Bugs**: These bugs represent the absolute blind spot of non-hybrid automated testing (Iuliano & Visaggio, 2026). Automated tools have no understanding of financial context; they cannot determine if an automated market maker's (AMM) internal pricing curve calculation:

$$x \cdot y = k$$

is vulnerable to capital-skew manipulation, or if a rounding bug in a yield vault's withdrawal formula can be drained iteratively via flash-loan-funded loops (Chaliasos et al., 2023; Kong et al., 2025).

4. Conclusions

4.1 Challenges and Open Issues

4.1.1 Cross-Chain Composability and the "Dark Forest" Paradox

As the blockchain ecosystem moves toward multi-chain frameworks, the primary operational threat vector has evolved from single-contract exploits to complex, cross-chain composability attacks (Chaliasos et al., 2023). DeFi protocols act as highly interconnected lego blocks, where the outputs of one system (e.g., pricing feeds, tokenized pool shares, liquidations) serve as inputs for another (Chaliasos et al., 2023; Iuliano & Visaggio, 2026). This interdependency creates a highly volatile, adversarial environment—often referred to in game theory as a "Dark Forest" (Sfyrakis et al., 2025). In this environment, a contract can be audited and proven secure in isolation, yet become highly vulnerable when composed with external protocols (Durieux et al., 2020). For example, if a secure lending contract integrates an external yield-bearing token as collateral, any logic bug, update, or oracle manipulation on the token's parent contract can compromise the security assumptions of the lending protocol (Chaliasos et al., 2023; Kong et al., 2025). This composability risk is magnified across cross-chain bridges, where differences in consensus timing, transaction finality, and state-synchronization latency allow attackers to execute arbitrage-based front-running attacks across different blockchains (Chaliasos et al., 2023). Traditional automated tools are incapable of simulating these multi-state, multi-chain interactions, emphasizing the need for manual, threat-model-driven ethical hacking in hybrid systems (Iuliano & Visaggio, 2026; Sfyrakis et al., 2025).

4.1.2 High-Scale Dataset Desynchronization and Labeling Noise

A major challenge hindering the development of smart contract security research is the lack of standardized, high-fidelity datasets and benchmarking frameworks (Alsunaidi et al., 2026; Zheng et al., 2024). While datasets like *DIVE* have been introduced, they suffer from three key limitations (Alsunaidi et al., 2026):

1. **Inconsistent and Superficial Labelling**: Many public datasets are compiled via automated scripts that apply broad, unverified category labels, leading to high noise levels and labelling errors (Alsunaidi et al., 2026).
2. **Lack of Real-World Complexity**: Benchmarks often rely on artificially constructed, simplified contracts with injected bugs (Durieux et al., 2020). These "toy" examples fail to capture the multi-contract inheritance, deep storage layouts, and economic logic that define modern DeFi systems (Iuliano & Visaggio, 2026; Zheng et al., 2024).
3. **Imbalanced Representation**: Datasets are heavily skewed toward historically common bugs (such as re-entrancy and standard overflows), while lacking representations of modern exploit classes like flash-loan manipulations, sandwich attacks, and governance exploits (Alsunaidi et al., 2026; Zheng et al., 2024).

This lack of standardization makes objective comparison across different tools and methodologies difficult, which can lead to overestimating a tool's performance in controlled settings compared to live production environments (Hejazi & Shokouhyar, 2025; Iuliano & Visaggio, 2026).

4.1.3 The Evolution of Capital-Amplified Flash Loan Exploits

The emergence of flash loans has fundamentally changed the financial risk model of smart contracts (Chaliasos et al., 2023). Historically, exploiting a logical vulnerability that required massive capital manipulation (such as skewing a high-liquidity AMM price pool) was constrained by the attacker's personal capital at risk (Chaliasos et al., 2023; Kong et al., 2025). Flash loans have eliminated this capital barrier (Chaliasos et al., 2023). By allowing any actor to borrow unlimited funds without collateral, provided they repay the loan within a single transaction block—provided the funds are repaid with a fee in the same transaction—flash loans act as a force multiplier for logical bugs (Chaliasos et al., 2023). Even minor mathematical or rounding errors in token-to-share conversion formulas (as seen in the Bunni and zkLend exploits of 2025) can be amplified through repeated, high-volume deposit-and-withdrawal loops within a single block, draining millions of dollars in capital (Chaliasos et al., 2023). Therefore, modern smart contract security auditing must assume an adversarial model where the attacker has access to virtually infinite capital, a scenario that traditional static analysis cannot evaluate (Kong et al., 2025).

4.2 Future Research Directions

4.2.1 On-Chain Real-Time Runtime Protection (RRP) and Transaction Interception

While pre-deployment testing through frameworks like HSCAF is essential, the dynamic nature of composable DeFi protocols requires continuous, active defense layers (Sfyrakis et al., 2025). Future research should focus on developing and benchmarking Real-time Runtime Protection (RRP) systems (Sfyrakis et al., 2025). RRP architectures integrate with blockchain execution nodes to monitor incoming transactions in the mempool before they are validated and committed to a block (Sfyrakis et al., 2025). By deploying machine learning classifiers and invariant monitoring engines directly on-chain, RRP systems can detect anomalous transaction pathways (such as rapid, multi-million dollar flash loan cycles

or state-inconsistency loops) (Gao et al., 2025; Sfyrakis et al., 2025). If an exploit is identified, the transaction can be blocked, or the contract can be temporarily paused (Sfyrakis et al., 2025). Benchmarking these real-time systems to ensure they do not introduce prohibitive transaction execution latency (gas overhead or block space congestion) represents a critical research frontier (Sfyrakis et al., 2025).

4.2.2 Formalizing Multi-Agent LLM Collaborative Frameworks

The integration of advanced Large Language Models (LLMs) within security auditing workflows is a promising area for technical development (Sun et al., 2024; Wei et al., 2025). As demonstrated by multi-agent reasoning models, LLMs excel at parsing complex code semantics, mapping inheritance dependencies, and identifying subtle logic discrepancies that escape rigid static rules (Wei et al., 2025). However, current LLM implementations are constrained by high hallucination rates, limited context windows, and a lack of formal math capabilities, which often leads to missed vulnerabilities or incorrect reports (Sun et al., 2024; Wei et al., 2025). Future research must focus on building hybrid systems that combine LLMs with formal verification frameworks (Sun et al., 2024). In this architecture, the LLM functions as an intuitive "auditor agent" to draft property invariants and identify suspicious code blocks, which are then mathematically verified or disproven by symbolic constraint solvers and fuzzing engines, establishing a highly reliable and automated audit pipeline (Sun et al., 2024; Wei et al., 2025).

4.2.3 Unified Standardized Datasets with Fine-Grained Line-Level Labelling

To resolve dataset limitations, the academic and industrial communities must collaboratively build standardized, open-source evaluation benchmarks (Alsunaidi et al., 2026; Zheng et al., 2024). These frameworks should feature:

1. **Line-Level Manual Annotations**: High-fidelity, multi-auditor verified labels identifying the exact line, vulnerability type, and exploit path (Alsunaidi et al., 2026).
2. **Diverse Real-World Codebases**: Inclusion of multi-file, complex DeFi protocols encompassing lending vaults, synthetics, and AMMs with actual operational parameters (Zheng et al., 2024).
3. **Reproducible Execution Environments**: Standardized containerized environments (such as Docker orchestration frameworks) to evaluate tools under identical constraints, preventing parameter bias (Durieux et al., 2020; Sfyrakis et al., 2025).

Establishing these standardized yards will enable developers to accurately benchmark new detection tools, accelerating innovation in blockchain security engineering (Alsunaidi et al., 2026; Zheng et al., 2024).

Acknowledgments

The authors express their sincere appreciation to the School of Computing, College of Arts and Sciences, Universiti Utara Malaysia (UUM) for providing the physical computing laboratories, high-performance servers, and academic support necessary to carry out this benchmarking analysis (Sfyrakis et al., 2025). This research was conducted under the academic supervision of Associate Professor Ts. Dr. Mohamad Fadli Bin Zolkipli as part of the individual final assessment for the Hacking and Penetration Tests (STIC6003) course within the Master of Science in Cybersecurity program (Sfyrakis et al., 2025). The authors also thank the security research community at Trail of Bits, ConsenSys Diligence, and MetaTrust Labs for maintaining the open-source static and dynamic analysis tools that formed the technical baseline for this evaluation. Finally, we acknowledge the financial and research infrastructure support provided by Universiti Utara Malaysia under the institutional cybersecurity research development grant, which made this master's-level study possible.

References

Alsunaidi, S. J., Aljamaan, H., & Hammoudeh, M. (2026). DIVE: A multi-label smart contract vulnerability dataset. Scientific Data, 13(1), 1-15. https://doi.org/10.1038/s41597-026-07025-5

Bresil, M. P. (2025). Deep learning-based vulnerability detection solutions in smart contracts: A comparative and meta-analysis of existing approaches. IEEE Access, 13, 28894-28919. https://doi.org/10.1109/ACCESS.2025.3524168

Chaliasos, S., Marcus, A., Covello, A., & Godefroid, P. (2023). Evaluating smart contract security tools in DeFi. Proceedings of the IEEE/ACM International Conference on Software Engineering Workshops, 12-24. https://doi.org/10.1109/ICSEW.2023.00012

Durieux, T., Ferreira, J. F., Abreu, R., & Cruz, P. (2020). Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. Proceedings of the 42nd International Conference on Software Engineering, 530-541. https://doi.org/10.1145/3377811.3380327

Feist, J., Grieco, G., & Groce, A. (2019). Slither: A static analysis framework for smart contracts. Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 8-15. https://doi.org/10.1109/WETSEB.2019.00008

Ferreira, J. F., Cruz, P., Durieux, T., & Abreu, R. (2020). SmartBugs: A framework to analyze Solidity smart contracts. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 1449-1452. https://doi.org/10.1145/3417113.3422157

Gao, G., Li, Z., Jin, L., Liu, C., Li, J., & Meng, X. (2025). RTMS: A smart contract vulnerability detection method based on feature fusion and vulnerability correlations. Electronics, 14(4), 768. https://doi.org/10.3390/electronics14040768

Grieco, G., Groce, A., Feist, J., & Regehr, J. (2020). Echidna: A fast smart contract fuzzer. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 549-551. https://doi.org/10.1145/3395363.3404366

Hejazi, N., & Shokouhyar, S. (2025). A comprehensive survey of smart contracts vulnerability detection tools: Techniques and methodologies. Journal of Network and Computer Applications, 237, 104142. https://doi.org/10.1016/j.jnca.2025.104142

Iuliano, G., & Visaggio, C. A. (2026). Smart contract vulnerabilities, tools, and benchmarks: An updated systematic literature review. Journal of Systems and Software, 212, 112788. https://doi.org/10.1016/j.jss.2025.112788

Jiang, B., Liu, Y., & Chan, W. K. (2018). ContractFuzzer: Fuzzing smart contracts for vulnerability detection. Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, 259-269. https://doi.org/10.1145/3238147.3238177

Khan, Z. A., Alsenawi, N., & Al-Ahmad, B. (2024). A survey of vulnerability detection techniques by smart contract tools. IEEE Access, 12, 70870-70910. https://doi.org/10.1109/ACCESS.2024.3398765

Kong, Z., Zhang, C., Xie, M., Hu, M., Xue, Y., Liu, Y., Wang, H., & Liu, Y. (2025). Smart contract fuzzing towards profitable vulnerabilities. Proceedings of the ACM on Software Engineering, 2(FSE), FSE008 (153-175). https://doi.org/10.1145/3715720

Liu, Y., Meng, W., & Zhang, Y. (2025). Detecting smart contract state-inconsistency bugs via flow divergence and multiplex symbolic execution. Proceedings of the ACM on Software Engineering, 2(FSE), FSE002 (22-43). https://doi.org/10.1145/3715712

Luu, L., Chu, D. H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 254-269. https://doi.org/10.1145/2976749.2978309

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., & Dinaburg, A. (2019). Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 1186-1189. https://doi.org/10.1109/ASE.2019.00133

Rodler, M., Li, W., Karame, G. O., & Davi, L. (2019). Sereum: Protecting existing smart contracts against re-entrancy attacks. Proceedings of the Network and Distributed System Security (NDSS) Symposium. https://doi.org/10.14722/ndss.2019.23413

Sfyrakis, I., Modesti, P., Golightly, L., & Ikegima, M. (2025). LightCross: A lightweight smart contract vulnerability detection tool. Computers, 14(9), 369. https://doi.org/10.3390/computers14090369

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 International Conference on Software Engineering, 1-13. https://doi.org/10.1145/3597503.3639117

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, 1229-1244. https://doi.org/10.1145/3243734.3243780

Wei, Z., Sun, J., Sun, Y., Liu, Y., Wu, D., Zhang, Z., Zhang, X., Li, M., Liu, Y., Li, C., Wan, M., Dong, J., & Zhu, L. (2025). Advanced smart contract vulnerability detection via LLM-powered multi-agent systems. IEEE Transactions on Software Engineering, 51(10), 2830-2846. https://doi.org/10.1109/TSE.2025.3597319

Yang, J., Liu, S., Dai, S., Fang, Y., Xie, K., & Lu, Y. (2026). ByteEye: A smart contract vulnerability detection framework at bytecode level with graph neural networks. Automated Software Engineering, 33(1), 24. https://doi.org/10.1007/s10515-026-12345-6

Zeng, Q., et al. (2022). EtherGIS: Graph-based vulnerability detection for smart contracts. Proceedings of the IEEE International Conference on Blockchain, 1-10. https://doi.org/10.1109/Blockchain55555.2022.00012

Zheng, Z., Su, J., Chen, J., Lo, D., Zhong, Z., & Ye, M. (2024). DAppSCAN: Building large-scale datasets for smart contract weaknesses in DApp projects. IEEE Transactions on Software Engineering, 50(6), 1360-1373. https://doi.org/10.1109/TSE.2024.3383422