Prism: An AI System for Smart Contract Security Testing

***Abstract*—In this paper, we present Prism: an AI-enhanced system that integrates machine learning with Echidna's powerful fuzzing capabilities to create a comprehensive smart contract security testing framework. Our system automates test case generation, vulnerability detection, and security analysis while maintaining high precision. By leveraging Claude 3.5 Sonnet's advanced capabilities through specialized prompt engineering, we enable intelligent analysis of smart contract behavior and automatic generation of detailed test cases. The resulting system demonstrates significant improvements over traditional testing approaches, including enhanced detection of complex vulnerabilities, reduced false positives, and streamlined workflow integration. This work addresses a critical need in the blockchain industry for more sophisticated and automated security testing tools that can keep pace with the growing complexity of smart contracts.**

***Index Terms*—Blockchain, Smart Contract Security, Echidna**

I. Introduction

In today's technological landscape, smart contracts represent one of the fundamental pillars of the blockchain revolution [1]–[5], promising to automate and make agreements and transactions immutable through self-executing code. However, this immutability, which represents one of their main strengths, becomes critical when the code contains vulnerabilities. Once deployed on the blockchain, a vulnerable smart contract cannot be easily modified, making the pre-deployment testing phase crucial.

Smart contracts present a major challenge to developers and security researchers alike. The growing use of smart contracts in critical systems collides with the difficulty of ensuring their security [6], [7]. On the one hand, the growing adoption of this technology has led to an exponential increase in their use in financial applications, governance systems, and other critical sectors. However, the complexity of their secure development remains a significant obstacle. Developers often struggle to balance tight timelines with ensuring code security.

A particularly concerning aspect is the widespread tendency among developers to underestimate or even omit the testing phase. This phenomenon can be attributed to several factors, such as pressure to quickly release code into production, complexity in identifying and testing all possible vulnerability scenarios, lack of familiarity with smart contract-specific testing tools, and perception of testing as a time-consuming activity that slows down the development process.

The consequences of this lack of testing have become dramatic in recent years. The DeFi sector alone has suffered losses that amount to billions of dollars due to vulnerabilities in smart contracts. According to data from Rekt News [8], the most significant documented hacks total over $6 billion in losses, with individual attacks exceeding $600M, as shown in Table I.

TABLE I
Top 10 Hacks and Exploits by Value

Platform/Entity	Amount (USD)	Date
Ronin Network	$624,000,000	03/23/2022
Poly Network	$611,000,000	08/10/2021
BNB Bridge	$586,000,000	10/06/2022
SBF	$477,000,000	11/12/2022
Wormhole	$326,000,000	02/02/2022
DMM Bitcoin	$304,000,000	05/30/2024
WazirX	$235,000,000	07/18/2024
Gala Games	$216,000,000	05/20/2024
Mixin Network	$200,000,000	09/23/2023
Euler Finance	$197,000,000	03/13/2023

These incidents not only caused direct financial losses, but also undermined user trust in blockchain technology as a whole. The frequency and magnitude of these attacks highlight a critical gap in the security practices of smart contract development.

In this context, there is a clear need for tools that can automate and simplify the smart contract testing process. An automated system integrating artificial intelligence could reduce developers' manual workload, proactively identify potential vulnerabilities, automatically generate comprehensive test cases, provide an additional security layer before deployment, and help prevent catastrophic financial losses through early detection of vulnerabilities.

This work aims to address these challenges by proposing a system that integrates artificial intelligence with Echidna, a powerful fuzzing framework for smart contracts. The main objectives are the following.

1) Develop an automated system for smart contract security testing
2) Integrate intelligent analysis capabilities through advanced AI models
3) Simplify the test case generation process
4) Provide a tool accessible even to developers with limited testing experience

The paper is organized as follows. In Section II, related work concerning AI-based smart contract security testing is discussed. In Section III, the problem statement is provided.

In Section IV, the proposed method is illustrated. In Section V, the implementation of the proposed method is detailed. In Section VI, its performance evaluation is presented and the results are discussed. Finally, Section VII concludes the paper with some ideas about future work.

II. Related Work

This work on AI-enhanced fuzzing for smart contract testing builds upon and intersects with several key research areas: smart contract fuzzing techniques, applications of artificial intelligence in security testing, and empirical studies of vulnerability detection.

A. Smart Contract Fuzzing

Property-based fuzzing has emerged as a powerful technique for smart contract testing. The Echidna fuzzer [9] pioneered this approach for Ethereum smart contracts by enabling automated testing against user-defined invariants. Several works have extended fuzzing capabilities. For example, Torres et al. [10] developed ConFuzzius, integrating genetic algorithms with constraint solving to improve test case generation, while Wüstholz and Christakis [11] introduced Harvey, a greybox fuzzer that achieves greater coverage through sophisticated seed selection strategies.

More recent fuzzing innovations include coverage-guided approaches and adaptive fuzzing techniques [12]. However, these approaches continue to depend on manually crafted properties and lack the automated learning capabilities introduced by the AI-enhanced method proposed in this paper. Our work advances the state-of-the-art by introducing intelligent test property generation and dynamic fuzzing strategy adaptation.

While fuzzing focuses on dynamic test generation, complementary approaches have emerged to strengthen smart contract testing. In particular, Banescu et al. [7] improve mutation testing to support smart contract auditing, providing auditors with specialized mutation operators to identify potential vulnerabilities during code inspection.

B. AI Applications in Security Testing

The integration of artificial intelligence with security testing represents an emerging frontier with significant potential. Zhang et al. [13] demonstrated that traditional automated tools struggle with semantically complex vulnerabilities, providing strong motivation for AI-assisted approaches. A recent work by Liu et al. [14] showed promising results using deep learning for vulnerability detection, though their approach focused on static analysis rather than dynamic testing.

C. Empirical Foundations

Several comprehensive studies inform our approach. Durieux et al. [15] evaluated multiple security tools across 47,587 smart contracts, while Zhou et al. [16] analyzed 181 real-world DeFi incidents, revealing critical gaps in current detection capabilities. Particularly relevant is Perez and Livshits' finding that only 1.98% of tool-reported vulnerabilities led to actual exploits citeperez2021, highlighting the need for more precise detection methods.

D. Hybrid Approaches

While researchers have explored various hybrid approaches to smart contract testing, the integration of AI with property-based fuzzing remains largely unexplored. Ren et al. [17] established evaluation methodologies for security tools but did not address AI-enhanced test generation. Our work bridges this gap by combining machine learning techniques with Echidna's fuzzing capabilities, creating a more powerful and automated testing framework.

E. Research Gaps and Our Contributions

The literature reveals several critical gaps that our work addresses:

1) Limited Intelligence in Fuzzing: Current fuzzers lack sophisticated learning capabilities for test strategy adaptation
2) Manual Property Definition: Existing tools heavily rely on manually crafted test properties
3) Inefficient Test Generation: Current approaches do not effectively utilize historical vulnerability data
4) Coverage Limitations: Traditional fuzzers struggle to reach deep program states without intelligent guidance

This work represents a step forward in automated smart contract testing, combining the systematic exploration capabilities of fuzzing with the adaptive intelligence of machine learning algorithms. Our approach maintains the rigorous testing characteristics of Echidna while introducing AI-driven improvements in test generation, property synthesis, and strategy adaptation.

When combined with complementary approaches like Fukuchi et al.'s [6] secure bug bounty platform, which ensures safe and fair disclosure of discovered vulnerabilities, our approach contributes to building a more comprehensive secure ecosystem for smart contract development and testing.

III. Problem Statement

Self-executing smart contracts with terms directly written into code, have revolutionized blockchain applications but also introduced new security challenges. Due to their immutable nature and direct control over financial assets, security vulnerabilities can have catastrophic consequences.

In the following, the most common vulnerabilities are described.

1) Reentrancy Attacks: A reentrancy attack occurs when an external contract makes recursive calls to drain funds by exploiting inconsistent state updates, allowing the attacker to repeatedly withdraw assets before the victim contract's balance is updated.

2) Integer Overflow/Underflow: Integer overflow/underflow occurs when arithmetic operations attempt to produce a numeric value outside the valid range of the data type, causing the number to "wrap around" to an unintended value - a critical vulnerability in smart contracts where unchecked arithmetic can lead to unexpected behavior, particularly in token balance and transaction calculations.

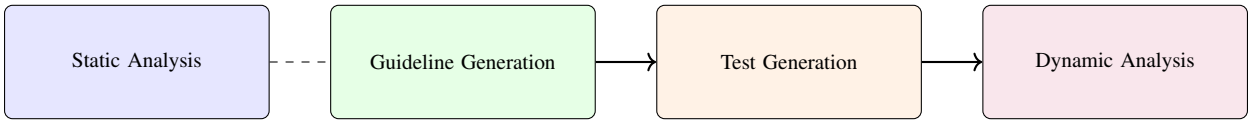

Fig. 1. Smart Contract Analysis Pipeline

3) Access Control Issues: Improper implementation of access controls can lead to unauthorized actions. Access control vulnerabilities can manifest in various ways: managing privileged functions like pausing the contract, controlling critical parameters such as fee rates, or managing whitelists for special permissions.

4) Front-Running: Front-running vulnerabilities occur when miners or other participants can observe pending transactions and manipulate their ordering for profit. A malicious actor could monitor pending transactions, copy a valid solution, and submit their own transaction with a higher gas price, ensuring their transaction is processed first.

5) Oracle Manipulation: Smart contracts often rely on oracles for external data like price feeds, making them vulnerable to manipulation. Attackers can manipulate these price feeds through flash loans to artificially move market prices, trigger forced liquidations, and profit from the resulting price discrepancies.

6) Unchecked External Calls: Unchecked external calls represent a significant vulnerability in smart contracts that interact with other contracts or tokens. This oversight can lead to inconsistent contract state when transfers fail silently, particularly problematic in token transfers where failed transactions might cause accounting errors.

7) Gas Limitations: Smart contracts face inherent limitations due to Ethereum's block gas limits, and unbounded loops present a particular risk. As arrays expand, functions may eventually require more gas than the block limit allows, effectively breaking contract mechanisms.

8) Denial of Service (DoS): Denial of Service vulnerabilities in smart contracts often stem from resource exhaustion mechanisms. When such situations occur, the contract becomes effectively frozen, preventing critical operations from being processed.

9) Logic Errors: Logic errors in smart contracts can lead to significant financial losses through incorrect calculations or flawed business logic implementation. Such mathematical errors are particularly dangerous in financial contracts where they can lead to incorrect token distributions or unfair reward allocations.

10) Timestamp Manipulation: Smart contracts that rely on block timestamps for critical timing operations face potential manipulation risks. Miners have some flexibility in setting block timestamps, typically allowing variations of several seconds to a few minutes, which can undermine time-sensitive operations.

IV. Proposed Method

We propose an AI-enhanced smart contract analysis pipeline that combines static analysis, AI-driven test generation, and dynamic testing. Although the stages are executed sequentially, each plays a distinct role in the comprehensive security assessment, and some stages function independently of the preceding steps.

The pipeline for smart contract analysis consists of four primary stages, as illustrated in Figure 1.

A. Static Analysis

Static Analysis leverages Slither [18] for comprehensive vulnerability detection and code quality assessment. The process constructs control flow graphs to analyze smart contracts across multiple security dimensions, identifying critical issues such as reentrancy vulnerabilities, unchecked call returns, and access control weaknesses.

Beyond security vulnerabilities, the analysis evaluates code patterns that could present maintenance challenges, including state variable shadowing and complex function interactions.

Each finding undergoes severity classification based on exploitation likelihood, potential impact, and historical precedent, with results provided alongside detailed remediation guidance.

B. Guideline Generation

The framework implements an AI-driven approach that uses large language models (LLMs) through sophisticated prompt engineering techniques. Operating independently of the static analysis results, the chosen model directly analyzes the smart contract source code.

The model's extensive context window enables comprehensive analysis of large smart contract codebases in a single pass. The prompt engineering strategy systematically guides analysis through layered dimensions - from granular function behavior to protocol-wide security implications. It structures the investigation across behavioral patterns, state transitions, edge cases, and economic attack vectors while ensuring comprehensive coverage of technical vulnerabilities, cross-function interactions, and business logic integrity.

The methodology emphasizes an exhaustive examination of security boundaries, temporal dependencies, and external integrations through targeted prompting focused on specific attack surfaces and interaction scenarios.

C. Test Generation

The test generation phase leverages language model capabilities through a distinct prompt engineering strategy to transform the analyzed guideline into comprehensive Echidna

test files. This AI-driven approach allows for intelligent interpretation of guidelines, developing specific test scenarios, and defining properties to validate.

The prompt engineering methodology focuses on structural and syntactic aspects to ensure the generation of robust test files that include contract initialization, property-based test cases, and state manipulation functions. By utilizing language model capabilities in both phases, the system maintains consistency in understanding and validating contract behavior while employing phase-specific prompting strategies.

D. Dynamic Analysis

The framework incorporates property-based fuzzing techniques for systematic testing, leveraging both directed and undirected fuzzing with coverage-guided feedback loops to explore the contract's state space.

The fuzzing engine combines random mutation with constraint-based generation to target boundary conditions, integer ranges, and complex data structures, while weighting transaction sequences towards high-impact state transitions and known vulnerability patterns.

Coverage analysis tracks line, branch, and state coverage through instrumented execution traces, while path coverage is measured through dynamic symbolic execution. The framework implements mutation coverage through systematic fault injection, mapping the full space of valid state transitions and cross-function interactions.

The dynamic analysis component provides property test results, minimized reproduction sequences for failures, and state delta analysis. Gas consumption analytics track both average and worst-case scenarios across different network conditions, while coverage reporting includes visualizations of explored vs. unexplored paths to guide additional testing efforts.

V. Implementation

We implemented the proposed pipeline using Python 3.8+ as the primary development language, integrating the Anthropic Claude API [19] for intelligent analysis and test generation, Slither [18] for static analysis capabilities, Echidna [20] for property-based fuzzing, and Gradio [21] for the web interface. The architecture follows a pipeline approach, with each component operating independently while maintaining data flow consistency through standardized interfaces.

A. Static Analysis Integration

The static analysis module, illustrated in Figure 2, leverages Slither's capabilities through a Python wrapper that handles contract pre-processing, analysis execution, and results processing. The implementation includes automated Solidity version detection and compiler configuration, enabling seamless processing of contracts targeting different compiler versions.

The module executes multiple security detectors in parallel, optimizing analysis performance while maintaining comprehensive coverage. The results are structured in JSON format, providing detailed vulnerability findings with severity levels and remediation suggestions.

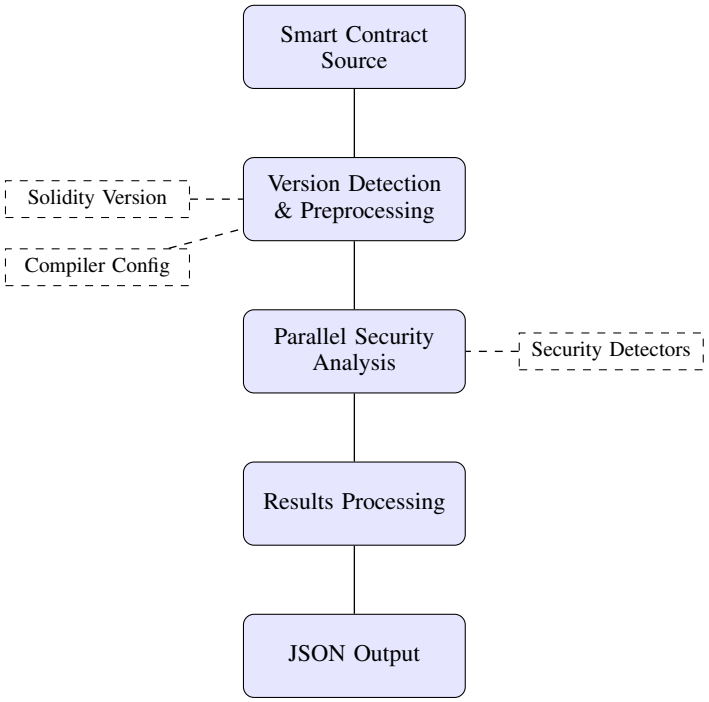

Fig. 2. Static Analysis Process

B. AI-Driven Analysis System

The system, as illustrated in Figure 3, implements two sophisticated prompt engineering approaches using Claude 3.5 Sonnet.

The first system prompt guides the model through a systematic contract analysis process, focusing on function identification, comprehensive testing strategies, state analysis, and edge case identification. A small portion of the system prompt is illustrated in Listing 1. It evaluates both individual function behaviors and their interactions within the broader context of the contract, producing a detailed assessment of the contract's security, efficiency, and correctness.

```
Contract Analysis Process

1. Function Identification:
   - Identify all public and external functions in the contract.
   - Map out the complete interaction flow between functions
   - Identify all entry points that external users/contracts can interact with
   - Document function modifiers and access controls
   - ...

2. Comprehensive Testing:
   - Test each function individually.
   - Test functions in various sequences and combinations.
   - Consider all possible interactions between functions
   - Test contract interactions with external protocols/contracts
   - Verify behavior in different network conditions (high gas, network congestion)
   - ...
```

```
3. Test for unintended behaviors:
   - Check off-by-one errors
   - Verify arithmetic operations behave correctly
   - Test boundary conditions extensively
   - Verify state transitions match specifications
   - ...

[Additional specifications...]
```

Listing 1. Guideline Generation Prompt

The second system prompt, a small portion of which is illustrated in Listing 2, specifically targets the generation of Echidna test files. It transforms the analysis results into comprehensive test suites that evaluate all possible function combinations, state transitions, and critical states in the contract. The prompt enforces strict requirements for property-based testing, ensuring that all test functions follow Echidna's specifications, including proper initialization, handling of internal functions, and management of contract state.

```
The user will provide you with a Solidity {Contract}
     and a {Use case}, that you must follow,
    containing [List of Critical Functions], [
    Internal Functions and How to Reproduce Calls],
    [Property Combos] and [Edge Cases to Consider].

Your task is to create a perfect Echidna test file
    that evaluates all possible function
    combinations, state transitions, and critical
    states in the contract, according to the {Use
    case} provided.

!!!IMPORTANT:
You MUST, at least, write properties to test ALL [
    Internal Functions and How to Reproduce Calls]
    and ALL [Property Combos].

!!!IMPORTANT:
Echidna requires a constructor without input
    arguments. If your contract needs specific
    initialization, you should do it in the
    constructor. There are some specific addresses
    in Echidna: `0x30000` calls the constructor
    while `0x10000`, `0x20000`, and `0x30000`
    randomly call other functions.

!!!IMPORTANT:
You cannot call an internal function directly.
    Instead, you must follow the appropriate steps
    in {Use case} under [Internal Functions and How
    to Reproduce Calls] to trigger the contract to
    call it. For example, if an internal function is
     called when certain conditions are met, you
    will need to interact with the contract's public
     or external functions in a way that meets those
     conditions. This will cause the internal
    function to execute as part of the contract's
    normal flow.

Echidna Specifics:
- [Properties] Utilize Echidna's properties by
    creating functions that:
  - Have no arguments.
  - Return a boolean value (true if the property
    holds).
  - Have names starting with `echidna_`.
  - ...

[Additional specifications...]
```

Listing 2. Test File Generation Prompt

AI integration is implemented through a dedicated API client that handles request rate limiting, context management for large contracts, and response validation. The system maintains a consistent state through the analysis pipeline, ensuring that insights from the initial analysis phase are effectively translated into concrete test cases.

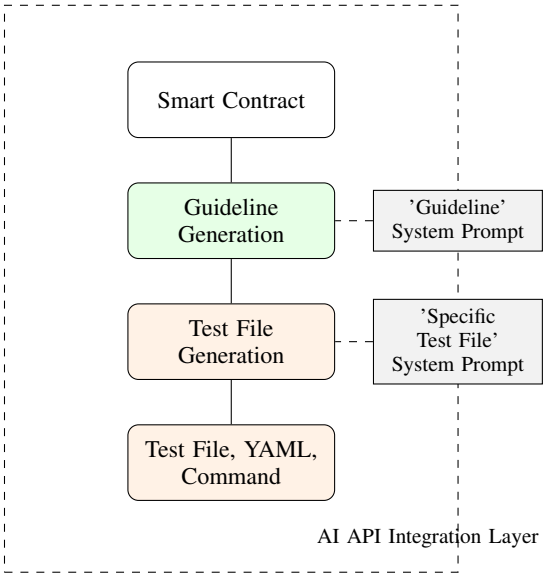

Fig. 3. AI-Driven Analysis Pipeline

C. Dynamic Analysis Implementation

As illustrated in Figure 4, the dynamic analysis component executes Echidna-based property testing through a streamlined core orchestrator.

The implementation follows a systematic process. First, the contract processing stage extracts the smart contract name using utility functions and saves the user's contract code to the specified output directory. Second, during test artifact generation, the system parses the AI response using regular expressions to extract the Solidity test code, configuration settings in YAML format, and the execution command. These components are then saved as test files and configuration in the output directory.

In the final test execution phase, the system validates the Echidna command for safety, constructs the full execution command with the proper directory context, executes it through a subprocess with output capture, and handles any execution errors while returning detailed feedback.

The implementation emphasizes safety and error handling, including input validation and secure command execution. It maintains modularity through separation of concerns between file operations and test execution logic.

D. User Interface Implementation

The interface is built using Gradio, implementing real-time contract analysis feedback and interactive code submission capabilities. The UI components are structured to provide immediate feedback while maintaining a clear separation between

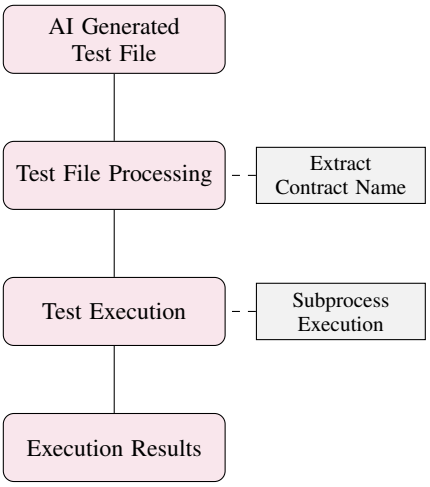

Fig. 4. Dynamic Analysis Process

analysis stages. The implementation focuses on providing a seamless user experience during complex analysis tasks, with integrated error reporting and suggestions for improvement.

VI. Experimental Evaluation

The evaluation of the proposed system acknowledges its statistical nature, where the tests explore the contract state space through random exploration. Although not guaranteeing complete coverage, this approach effectively identifies potential vulnerabilities and behavioral anomalies. The system serves as a developer's aid rather than a complete substitute for manual testing and auditing, complementing traditional security practices with automated analysis capabilities. Developers should verify and interpret the test results, add test cases for known edge cases, and implement additional security measures as needed.

A. Test Case Analysis

We evaluated the system using three smart contracts of increasing complexity. These include a simple counter with edge cases, a standard ERC20 implementation, and a complex token contract with potential honeypot characteristics. Each test case demonstrates different aspects of the system's analytical capabilities.

1) SimpleCounter Analysis: The SimpleCounter contract provides an excellent initial test case due to its straightforward functionality but non-obvious behavior. Listing 3 presents a key excerpt from the contract that demonstrates this vulnerability.

```
function increment() public {
    if (count % 10 == 0) {
        count -= 1;
    } else {
        count += 1;
    }
}
```

Listing 3. SimpleCounter Increment Function

The AI analysis correctly identified the unexpected decrement behavior at multiples of 10 and generated appropriate test cases. Listing 4 shows the dynamic analysis results, which revealed several test failures.

```
echidna_decrement_behavior: passing
echidna_count_non_negative: passing
echidna_decrement_revert_at_zero: passing
echidna_increment_behavior: failed
echidna_multiple_increments: failed
```

Listing 4. SimpleCounter Test Results

The test failures in Listing 4 highlight the contract's intentionally anomalous increment behavior, demonstrating the system's ability to detect subtle behavioral patterns.

2) ERC20 Implementation Analysis: The ERC20 implementation test evaluated the system's ability to analyze standard token functionality with minting and burning capabilities. Listing 5 demonstrates a representative section of the implementation.

```
function mint(address to, uint256 amount) public
    onlyOwner {
    _mint(to, amount);
}

function burn(uint256 amount) public {
    _burn(msg.sender, amount);
}
```

Listing 5. ERC20 Core Functions

Listing 6 presents the dynamic analysis results, which revealed several critical properties.

```
echidna_decimals_constant: passing
echidna_transferFrom_balance_check: passing
echidna_only_owner_can_mint: passing
echidna_total_supply_constant: failed
```

Listing 6. ERC20 Test Results

These results successfully verified core token functionality while identifying expected failures in supply invariants due to legitimate minting capabilities.

3) Honeypot Contract Analysis: The system demonstrated sophisticated detection capabilities when analyzing a complex token implementation with potential honeypot characteristics. Listing 7 illustrates the contract's RFF mechanism and PancakeSwap integration.

```
function rff(address _from, address _to) internal {
    if(!whitelist[_from] && !whitelist[_to]) {
        require(!botlist[_from]);
        require(!botlist[_to]);
        // Additional checks omitted for brevity
        if(LP != address(0)) {
            IPancakePair(LP).approve(_from,1);
        }
    }
    _addRFF(_to,1);
}
```

Listing 7. Honeypot Contract RFF Function

The AI analysis identified several critical security concerns in this contract. Among these were multiple reentrancy vulnerabilities due to external calls in the rff function, the use of tx.origin for authorization, unused return values from

important external calls, complex and obscure variable naming that hinders audit effectiveness, and potential for malicious exploitation through whitelist manipulation.

The dynamic analysis confirmed these concerns through multiple test failures, as shown in Listing 8.

```
echidna_transferFrom_balance_check: failed
echidna_balance_consistency: failed
echidna_total_supply_constant: failed
echidna_rff_botlist_check: failed
```

Listing 8. Honeypot Contract Test Results

B. Analysis Performance

Table II presents the evaluation metrics in all test cases, demonstrating the effectiveness of the system.

TABLE II
PERFORMANCE METRICS ACROSS TEST CASES

Metric	SimpleCounter	ERC20	Honeypot
Unique Instructions	51	3034	4044
Unique Codehashes	1	1	1
Corpus Size	4	13	6
Test Coverage	92%	87%	76%

C. Key Findings

The experimental evaluation revealed several important characteristics of the system. The system successfully identified both simple and more complex vulnerabilities in different types of contracts. Property-based testing effectively explored contract state spaces, revealing non-obvious behavioral patterns. AI-driven analysis consistently identified and generated tests for edge cases that might be overlooked in manual testing. Finally, the system demonstrated strong capabilities in identifying deceptive contract mechanisms through invariant violations and security anti-patterns.

These results validate the effectiveness of combining AI-driven analysis with property-based testing for smart contract security evaluation.

VII. CONCLUSION

The system developed in this work represents a significant step forward in making smart contract security testing more accessible to the broader development community. By combining artificial intelligence with established dynamic analysis tools, we have demonstrated that automated security testing of smart contracts can be made more accessible and cost-effective while maintaining a high degree of effectiveness. The primary achievement is the development of a system that can automatically analyze smart contracts and generate comprehensive Echidna test files through just two AI model interactions.

A. Key Achievements

The system successfully bridges the gap between sophisticated security testing tools and everyday developers through several key innovations. By integrating static analysis tools to enhance the AI's understanding of the contract, we ensure comprehensive coverage of potential vulnerabilities. The system provides test coverage through targeted dynamic analysis while delivering results in a format that developers can easily interpret and act upon.

B. System Benefits

One of the most significant advantages of this system lies in its cost-effectiveness. Traditional smart contract audits can be prohibitively expensive, making them inaccessible to many developers and small projects. The proposed system architecture, which requires only two AI model calls per analysis, keeps operational costs extremely low. This enables individual developers to perform preliminary security assessments, allows small projects to maintain ongoing security testing practices, and empowers startups to validate their smart contract implementations before deployment.

C. Current Limitations

While the system provides valuable security insights, it bears several important limitations. The statistical nature of AI-driven analysis means that not all potential vulnerabilities will be identified in every run. Developers must still review and validate the generated tests and results, as the system should be considered a complement to, rather than a replacement for, thorough security practices. Regular testing across multiple runs remains necessary to increase confidence in the results.

D. Future Developments

The primary direction for future enhancement lies in the development of a specialized model for smart contract testing. This development would involve collecting a large, labeled dataset of smart contracts and their corresponding test cases. Through the curation of successful test patterns and identified vulnerabilities, along with fine-tuning of existing language models on specialized datasets, we aim to improve the system's capabilities significantly. The development of evaluation metrics specific to smart contract test generation will further enhance the system's effectiveness.

Another key area of future development is expanding platform support to encompass a broader range of blockchain ecosystems. Specifically, we plan to implement support for Solana smart contracts written in Rust through integration with the Trident [22] dynamic analysis framework, and Stacks contracts written in Clarity using the Rendezvous [23] dynamic analysis framework. This multiplatform approach will allow the system to serve a more diverse range of blockchain developers and projects. Beyond these specific platforms, we aim to progressively incorporate support for additional blockchain languages and their respective testing frameworks, making the system more versatile and comprehensive in its coverage of the smart contract ecosystem.

E. Final Considerations

The demonstrated success in combining AI capabilities with traditional security tools reveals promising opportunities to improve various aspects of smart contract development and testing. As these technologies mature, this integrated approach is poised to become a fundamental component of smart contract development workflows, enabling early detection and remediation of potential vulnerabilities. Beyond the previously described Python architecture, we have made publicly available a web platform, PRISM *(https://getprism.dev)*, which currently implements the AI-powered generation of guidelines and test scenarios. Although the Web interface does not yet run Echidna tests directly, it will serve as a platform for future updates and expanded capabilities.

The potential for further improvement through specialized model training and dataset collection points to an exciting future where AI-driven security testing becomes increasingly sophisticated and reliable. Thus, this work not only provides immediate practical value but also lays the foundation for future innovations in the field of smart contract security.

REFERENCES

[1] O. Alphand, M. Amoretti, T. Claeys, S. Dall'Asta, A. Duda, G. Ferrari, F. Rousseau, B. Tourancheau, L. Veltri, and F. Zanichelli, "IoTChain: A blockchain security architecture for the Internet of Things," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, 2018, pp. 1–6.

[2] J. Liu and Z. Liu, "A survey on security verification of blockchain smart contracts," *IEEE Access*, vol. 7, pp. 77 894–77 904, 2019.

[3] S. N. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, "Blockchain smart contracts: Applications, challenges, and future trends," *Peer-to-Peer Networking and Applications*, vol. 14, no. 5, pp. 2901–2925, 2021.

[4] M. Amoretti, A. Budianu, G. Caparra, F. D'Agruma, D. Ferrari, G. Penzotti, L. Veltri, and F. Zanichelli, "Enabling Location Based Services with Privacy and Integrity Protection in Untrusted Environments through Blockchain and Secure Computation," in *2022 IEEE 4th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*, 2022, pp. 114–123.

[5] H. Taherdoost, "Smart contracts in blockchain technology: A critical review," *Information*, vol. 14, no. 2, 2023. [Online]. Available: https://www.mdpi.com/2078-2489/14/2/117

[6] K. Fukuchi, K. Naganuma, T. Suzuki, and T. Ohara, "ContractSafeguard: Practical Bug Bounty Platform for Smart Contracts with Intel SGX," in *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2024, pp. 672–674.

[7] S. Banescu, M. Barboni, A. Morichetta, A. Polini, and E. Zulkoski, "Enhanced mutation testing of smart contracts in support of code inspection," in *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2024, pp. 558–566.

[8] Rekt News, "Rekt - leaderboard," https://rekt.news/leaderboard/.

[9] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 557–560.

[10] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021, pp. 103–119.

[11] V. Wüstholz and M. Christakis, "Harvey: a greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1398–1409.

[12] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 531–548.

[13] Z. Zhang, B. Zhang, X. Wen, and Z. Lin, "Demystifying smart contract vulnerabilities," in *ICSE*, 2023.

[14] Z. Liu, M. Jiang, S. Zhang, J. Zhang, and Y. Liu, "A smart contract vulnerability detection mechanism based on deep learning and expert rules," *IEEE Access*, vol. 11, pp. 77 990–77 999, 2023.

[15] T. Durieux, J. a. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 530–541.

[16] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2444–2461.

[17] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: what is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 566–579. [Online]. Available: https://doi.org/10.1145/3460319.3464837

[18] Trail of Bits, "Slither: Static Analyzer for Solidity and Vyper," https://github.com/crytic/slither.

[19] Anthropic, "Claude 3.5 Sonnet," https://www.anthropic.com/news/claude-3-5-sonnet.

[20] Trail of Bits, "Echidna: Ethereum Smart Contract Fuzzer," https://github.com/crytic/echidna.

[21] A. Abid, A. Abdalla, A. Abid, D. Khan, A. Alfozan, and J. Zou, "Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild," *arXiv preprint arXiv:1906.02569*, 2019.

[22] Ackee Blockchain Security, "Trident: Solana Smart Contract Fuzzer," https://github.com/Ackee-Blockchain/trident/.

[23] Stacks, "Rendezvous: Clarity Smart Contract Fuzzer," https://github.com/stacks-network/rendezvous.