UNIVERSITÀ DI PARMA

Dipartimento di Ingegneria e Architettura

Corso di Laurea in Ingegneria Informatica, Elettronica e delle Telecomunicazioni

Un Sistema IA per Security Testing di Smart Contract con Echidna

An AI System for Smart Contract Security Testing with Echidna

Relatore:
Prof. Michele Amoretti

Correlatore:
Ing. Stefano Cavalli

Tesi di Laurea di:
Simone Orsi

ANNO ACCADEMICO 2023-2024

Alla mia famiglia.

"Soffermati sulla bellezza della vita.
Guarda le stelle, e vediti correre con loro."
Marco Aurelio

Ringraziamenti

- Ringrazio mia mamma, primo e più grande sostegno della mia vita. Il suo amore incondizionato veglia su di me più di quanto io sappia vegliare su me stesso.
- Ringrazio mio papà, che con la sua silenziosa dedizione mi ha mostrato la via del sacrificio e della perseveranza, senza mai chiedere nulla in cambio.
- Ringrazio mio fratello, fonte inesauribile dei miei sorrisi più sinceri.
- Ringrazio i miei nonni: Ivana, Franco e Liviana, il mio porto sicuro.
- Ringrazio i miei zii e cugini, compagni di ogni momento, nella gioia e nelle difficoltà.
- Ringrazio i miei amici, compagni di risate e spensieratezza.
- Ringrazio Luigi e Nikos, mentori e amici preziosi, per il loro costante supporto e per la fiducia che hanno sempre riposto in me.

Contents

Introduction

In today's technological landscape, smart contracts represent one of the fundamental pillars of the blockchain revolution, 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.

Social and Technological Context

The world of smart contracts faces a significant challenge. The growing use of smart contracts in critical systems collides with the difficulty of ensuring their security. On 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. On the other hand, the complexity of their secure development remains a significant obstacle. Developers often find themselves having to balance tight development timelines with the need to ensure 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:

- 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
- Perception of testing as a time-consuming activity that slows down the development process

Impact of Inadequate Testing

The consequences of this lack of testing have been dramatically evident in recent years. The DeFi sector alone has suffered losses amounting to billions of dollars due to smart contract vulnerabilities. According to data from rekt.news [1], the most significant hacks total over $6 billion in losses, with individual attacks exceeding $600M. Some notable examples include:

- Ronin Network (2022): $620M compromised
- Poly Network (2021): $610M stolen
- Wormhole (2022): $326M exploited
- FTX/Cryptopia (2022): $477M lost
- Nomad Bridge (2022): $190M drained

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.

The Need for Automated Solutions

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
- Help prevent catastrophic financial losses through early detection of vulnerabilities

Thesis Objectives

This Thesis aims to address these challenges through the development of a system that integrates artificial intelligence with Echidna, a powerful fuzzing framework for smart contracts. The main objectives are:

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

Thesis Structure

The work is organized as follows.

- **Chapter 1** presents the state of the art in smart contract security testing techniques, analyzing existing approaches and their limitations
- **Chapter 2** describes the functional architecture of the developed system, illustrating how AI and Echidna integrate to provide a comprehensive solution

- **Chapter 3** delves into the implementation details of the system, with particular attention to the innovative aspects of the solution
- **Chapter 4** presents the obtained results, evaluating the system's effectiveness through real case studies

The ultimate goal is to provide the blockchain community with a tool that can help make smart contracts more secure and reliable, reducing the risk of vulnerabilities and increasing user trust in this transformative technology. By addressing the critical gap between development speed and security requirements, this work contributes to the ongoing effort to make blockchain applications safer and more dependable.

Chapter 1

State of the Art

1.1 Smart Contract Security Overview

Smart contracts, self-executing 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.

1.1.1 Common Vulnerabilities

Reentrancy Attacks

A reentrancy attack is a critical vulnerability in smart contracts where 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. Listing 1.1 demonstrates a vulnerable contract susceptible to reentrancy attacks.

```
1  contract Vulnerable {
2      mapping(address => uint) public balances;
3
4      function withdraw() public {
5          uint balance = balances[msg.sender];
```

```
6           require(balance > 0);
7
8           (bool success, ) = msg.sender.call{value: balance}("");
9           // Vulnerable: state update after external call
10          balances[msg.sender] = 0;
11      }
12  }
13  contract Attacker {
14      Vulnerable public target;
15
16      receive() external payable {
17          if (address(target).balance >= 1 ether) {
18              // Recursive call back to withdraw
19              target.withdraw();
20          }
21      }
22  }
```

Listing 1.1: Example of a contract vulnerable to reentrancy attacks

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. As shown in Listing 1.2, this vulnerability was common in older Solidity versions, particularly before 0.8.0.

```
1   contract Vulnerable {
2       mapping(address => uint8) public balances;
3
4       // Vulnerable to overflow
5       function transfer(address to, uint8 amount) public {
```

```
6           require(balances[msg.sender] >= amount);
7           balances[msg.sender] -= amount;
8           balances[to] += amount; // Could overflow
9       }
10  }
```

Listing 1.2: Example of a contract vulnerable to integer overflow

Access Control Issues

Improper implementation of access controls can lead to unauthorized actions, as shown in Listing 1.3. In this example, any user can call the ‘changeOwner‘ function to take control of the contract since there’s no verification that only the current owner can transfer ownership. 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. A secure implementation requires careful consideration of which functions need restrictions and implementing proper authorization checks through modifiers or require statements.

```
1   contract Vulnerable {
2       address public owner;
3
4       function changeOwner(address newOwner) public {
5           // Missing access control
6           owner = newOwner;
7       }
8   }
```

Listing 1.3: Example of a contract with insufficient access control

Front-Running

Front-running vulnerabilities occur when miners or other participants can observe pending transactions and manipulate their ordering for profit, as illustrated in Listing 1.4. In this example, a contract offers rewards for submitting solutions to a problem, but the transaction's public nature in the mempool allows observers to see valid solutions before they are processed. A malicious actor could monitor these pending transactions, copy a valid solution, and submit their own transaction with a higher gas price, ensuring their transaction is processed first. Mitigation strategies often involve commitment schemes or batch processing to ensure fair transaction ordering.

```
1  contract Vulnerable {
2      mapping(bytes32 => bool) public usedHashes;
3
4      function claimReward(bytes32 solution) public {
5          require(!usedHashes[solution], "Solution used");
6          require(isValidSolution(solution), "Invalid solution");
7
8          usedHashes[solution] = true;
9          payable(msg.sender).transfer(1 ether);
10     }
11 }
```

Listing 1.4: Example of a contract vulnerable to front-running attacks

Oracle Manipulation

Smart contracts often rely on oracles for external data like price feeds, making them vulnerable to manipulation. Listing 1.5 demonstrates a common vulnerability in DeFi protocols where liquidation decisions depend on oracle-provided prices. Attackers can manipulate these price feeds through flash loans to artificially move market prices, trigger forced liquidations, and profit from the resulting price discrepancies. Similar risks exist when oracles ag-

gregate data from low-liquidity markets or rely on centralized price feeds. To mitigate these risks, protocols should implement time-weighted average prices, use multiple oracle sources, and monitor for suspicious price movements.

```
contract Vulnerable {
    IOracle public oracle;

    function liquidate(address position) public {
        // Vulnerable to oracle manipulation
        uint price = oracle.getPrice("ETH/USD");
        require(price < liquidationPrice, "Price too high");
        // Perform liquidation
    }
}
```

Listing 1.5: Example of a contract vulnerable to oracle manipulation

Unchecked External Calls

Unchecked external calls represent a significant vulnerability in smart contracts that interact with other contracts or tokens. As shown in Listing 1.6, the contract performs a token transfer without verifying its success. This oversight can lead to inconsistent contract state when transfers fail silently. The issue is particularly problematic in token transfers, where failed transactions might cause accounting errors, lost funds, or exploitable conditions in complex DeFi protocols. Best practices include checking return values, using safe transfer functions that revert on failure, and implementing proper error handling for all external interactions.

```
contract Vulnerable {
    function unsafeTransfer(address token, address to, uint amount)
    public {
        // Vulnerable: return value not checked
```

```
4           IERC20(token).transfer(to, amount);
5           // Subsequent code assumes transfer succeeded
6       }
7   }
```

Listing 1.6: Example of a contract with unchecked external calls

Gas Limitations

Smart contracts face inherent limitations due to Ethereum's block gas limits, and unbounded loops present a particular risk. Listing 1.7 demonstrates a vulnerable reward distribution system where the number of investors could grow indefinitely. As the investor array expands, the function may eventually require more gas than the block limit allows, effectively breaking the reward distribution mechanism. This vulnerability commonly appears in mass distribution functions, batch operations, and array processing. A more robust approach would implement pagination or allow distributions to be processed in manageable batches.

```
1   contract Vulnerable {
2       address[] public investors;
3
4       function distributeRewards() public {
5           // Vulnerable: unbounded loop
6           for (uint i = 0; i < investors.length; i++) {
7               payable(investors[i]).transfer(1 ether);
8           }
9       }
10  }
```

Listing 1.7: Example of a contract vulnerable to gas limitations

Denial of Service (DoS)

Denial of Service vulnerabilities in smart contracts often stem from resource exhaustion mechanisms, similar to but distinct from general gas limitations. Listing 1.8 shows a refund system where the contract's functionality could be deliberately blocked. An attacker could inflate the refundAddress array size beyond gas block limits, or a legitimate growth in users could naturally trigger this condition. When such a situation occurs, the contract becomes effectively frozen, preventing any refunds from being processed. This type of vulnerability is particularly concerning in financial contracts where funds could become permanently locked. Similar DoS risks arise in contracts with sequential processing requirements or those dependent on specific user actions for critical operations.

```
1  contract Vulnerable {
2      address[] public refundAddresses;
3
4      function refundAll() public {
5          // Vulnerable: can be blocked if array too large
6          for (uint i = 0; i < refundAddresses.length; i++) {
7              payable(refundAddresses[i]).transfer(1 ether);
8          }
9      }
10 }
```

Listing 1.8: Example of a contract vulnerable to DoS attacks through array manipulation

Logic Errors

Logic errors in smart contracts can lead to significant financial losses through incorrect calculations or flawed business logic implementation. Listing 1.9 illustrates a staking reward calculation that produces incorrect results due to integer division order. The current implementation divides first and then

multiplies, resulting in truncation and potential loss of precision. For example, a stake of 150 would return 200 instead of the intended 300 reward units. Such mathematical errors are particularly dangerous in financial contracts where they can lead to incorrect token distributions, unfair reward allocations, or exploitable arbitrage opportunities. Similar logic vulnerabilities can arise in complex business rules, access control conditions, or state transition logic.

```
1 contract Vulnerable {
2     mapping(address => uint) public stakingBalance;
3
4     function calculateReward(address user) public view returns (uint) {
5         // Vulnerable: incorrect reward calculation
6         return stakingBalance[user] / 100 * 200;
7         // Should be: (stakingBalance[user] * 200) / 100
8     }
9 }
```

Listing 1.9: Example of a contract with mathematical logic errors in reward calculations

Timestamp Manipulation

Smart contracts that rely on block timestamps for critical timing operations face potential manipulation risks. Listing 1.10 demonstrates a time-locked withdrawal system where the security mechanism can be undermined. Miners have some flexibility in setting block timestamps, typically allowing variations of several seconds to a few minutes. In this example, a miner could slightly adjust the timestamp to enable premature withdrawals. This vulnerability becomes particularly significant in contracts involving time-sensitive operations such as lock periods, interest calculations, or gaming mechanics. The risk extends beyond simple time locks to any contract logic that assumes block timestamps are precisely sequential or completely trustworthy.

```
contract Vulnerable {
    uint public constant LOCK_TIME = 24 hours;
    mapping(address => uint) public lockTime;

    function lock() public payable {
        // Vulnerable: timestamp manipulation
        lockTime[msg.sender] = block.timestamp + LOCK_TIME;
    }

    function withdraw() public {
        require(block.timestamp >= lockTime[msg.sender], "Still
    locked");
        payable(msg.sender).transfer(address(this).balance);
    }
}
```

Listing 1.10: Example of a contract vulnerable to timestamp manipulation in lock periods

1.1.2 Real-World Exploit Case Studies

The following examples highlight how seemingly minor vulnerabilities can cascade into system-wide failures, emphasizing the critical importance of rigorous security auditing and formal verification in smart contract development.

The DAO Hack (2016)

- **Vulnerability:** Reentrancy
- **Loss:** 3.6M ETH ($50M at the time)
- **Impact:** Led to Ethereum hard fork
- **Technical Details:** Exploited recursive call pattern before balance update

Poly Network Hack (2021)

- **Vulnerability:** Access control weakness
- **Loss:** $610M
- **Impact:** Largest DeFi hack at the time
- **Technical Details:** Keeper role modification exploit

Wormhole Bridge Hack (2022)

- **Vulnerability:** Signature verification bypass
- **Loss:** $326M
- **Technical Details:** Exploit in guardian signature verification

Ronin Bridge Hack (2022)

- **Vulnerability:** Compromised private keys
- **Loss:** $620M
- **Impact:** Exposed vulnerabilities in cross-chain bridges
- **Technical Details:** Attacker gained control of validator nodes through social engineering

BNB Bridge Exploit (2022)

- **Vulnerability:** Proof verification flaw
- **Loss:** $566M potential exposure, $100M actual loss
- **Impact:** Forced temporary suspension of BSC network
- **Technical Details:** Exploited bug in proof verification system to forge arbitrary messages

1.2 Professional Security Audits

Professional security audits represent a comprehensive approach to identifying vulnerabilities and ensuring the robustness of smart contracts, though their significant cost often places them beyond the reach of smaller development teams and individual creators.

The audit process follows a structured methodology that begins with a crucial pre-audit phase, where auditors establish the scope of work, thoroughly review existing documentation, and conduct an initial assessment of the project's architecture and potential risk areas.

The core of the audit process lies in the manual code review, where experienced security professionals meticulously examine the codebase line by line. This intensive review combines pattern recognition from known vulnerabilities with sophisticated control flow analysis to identify potential security weaknesses that might not be immediately apparent.

Complementing the manual review, automated testing employs a variety of specialized tools and techniques. Static analysis tools scan the code for known vulnerability patterns, while dynamic testing simulates real-world interactions with the contract. Auditors often develop custom test suites tailored to the specific requirements and potential edge cases of the project under review.

The culmination of these efforts is captured in comprehensive audit reports that classify discovered vulnerabilities according to their severity and potential impact. These reports provide detailed risk assessments and specific recommendations for addressing identified issues, offering project teams actionable insights for improving their contract security.

In the current landscape, several prominent audit firms have established

themselves as industry leaders, each bringing unique methodologies and expertise. `Trail of Bits` [2] has distinguished itself through the development of custom security tools and a strong emphasis on formal verification techniques. `OpenZeppelin` [3] has built its reputation on a standards-based approach, leveraging their extensive experience in creating secure smart contract libraries and fostering community-driven security initiatives. `CertiK` [4] has pioneered the integration of formal verification methods with continuous monitoring systems, providing both point-in-time audits and ongoing security surveillance.

While professional audits represent the gold standard in smart contract security, their substantial cost –often ranging from tens to hundreds of thousands of dollars – creates a significant barrier for smaller projects and independent developers. This economic reality underscores the need for a hybrid security approach that combines accessible automated tools, community-driven security reviews, and educational resources. Such an integrated strategy can help democratize smart contract security, ensuring that projects of all sizes can implement robust security measures while working within their resource constraints.

1.3 Dynamic Analysis with Echidna

Fuzzing, or fuzz testing, as implemented in tools like Echidna [5], is an automated testing technique that involves providing invalid, unexpected, or random data as inputs to a program. This approach systematically explores various execution paths by generating unexpected test cases, allowing developers to uncover edge cases and vulnerabilities that might be missed through conventional testing methods. In the context of smart contracts, fuzzing becomes particularly crucial due to the immutable nature of blockchain deployments and the high financial stakes involved.

At its core, Echidna operates on the principle of property-based testing, where instead of specifying exact test cases, developers define properties that should hold true regardless of the contract's state or inputs. These properties, also known as invariants, represent fundamental truths about the contract's behavior. For example, an invariant might state that the total supply of tokens should remain constant after any operation, or that user balances should never become negative.

Properties in Echidna are expressed as functions that return boolean values, where:

- True indicates the property holds
- False signals a violation of the expected behavior

Unlike traditional unit testing, which typically tests a single state or path through the code, Echidna's approach is stateful. This means it:

- Maintains and manipulates contract state across test sequences
- Generates complex sequences of contract interactions
- Explores different state transitions and their combinations

The power of Echidna lies in its ability to automatically discover edge cases and vulnerabilities that might be overlooked in manual testing. The tool employs various strategies to generate test cases:

- Coverage-guided fuzzing to explore new code paths
- Genetic algorithms to evolve more effective test sequences
- Constraint solving to target specific contract states

When a property violation is discovered, Echidna provides:

- A minimal sequence of transactions that reproduce the issue

- The contract state at each step of the sequence
- Detailed execution traces for debugging

This combination of sophisticated test generation and comprehensive reporting makes Echidna an invaluable tool in the smart contract security arsenal, capable of identifying subtle vulnerabilities that might otherwise remain undetected until deployment.

1.3.1 Fuzzing Techniques

Coverage-Guided Fuzzing

Coverage-guided fuzzing in Echidna represents an advanced testing methodology that intelligently explores smart contract behavior through automated test case generation. Unlike traditional fuzzing approaches that rely on random inputs, coverage-guided fuzzing adapts its testing strategy based on code coverage metrics. Listing 1.11 demonstrates a practical implementation that illustrates this approach.

```
1  contract Token {
2      mapping(address => uint) public balances;
3      mapping(address => mapping(address => uint)) public allowances;
4
5      function transfer(address to, uint amount) public {
6          require(balances[msg.sender] >= amount);
7          balances[msg.sender] -= amount;
8          balances[to] += amount;
9      }
10
11     function approve(address spender, uint amount) public {
12         allowances[msg.sender][spender] = amount;
13     }
14
15     function transferFrom(address from, address to, uint amount)
       public {
```

```
16          require(allowances[from][msg.sender] >= amount);
17          require(balances[from] >= amount);
18          balances[from] -= amount;
19          balances[to] += amount;
20          allowances[from][msg.sender] -= amount;
21      }
22  }
23
24  contract TokenTest is Token {
25      function echidna_balance_consistency() public returns (bool) {
26          address[] memory users = new address[](3);
27          users[0] = address(0x1);
28          users[1] = address(0x2);
29          users[2] = address(0x3);
30
31          uint totalBefore = balances[users[0]] + balances[users[1]] +
        balances[users[2]];
32
33          // Multiple execution paths for Echidna to explore
34          if (balances[users[0]] > 0) {
35              transfer(users[1], balances[users[0]]);
36          } else if (allowances[users[1]][msg.sender] > 0) {
37              transferFrom(users[1], users[2],
        allowances[users[1]][msg.sender]);
38          } else {
39              approve(users[2], 1000);
40          }
41
42          uint totalAfter = balances[users[0]] + balances[users[1]] +
        balances[users[2]];
43          return totalBefore == totalAfter;
44      }
45  }
```

Listing 1.11: Implementation of coverage-guided fuzzing with multiple transaction paths

This implementation showcases several key aspects of coverage-guided

fuzzing.

- The Token contract implements three distinct transaction types: direct transfers, approvals, and delegated transfers.
- The test contract, TokenTest, creates a property function that Echidna uses to verify balance consistency across multiple transaction paths.

As Echidna runs tests, it monitors which code paths have been tested and maintains a record of the routes taken through the code. When it discovers inputs that lead to previously untested code sections, it prioritizes these new paths and generates variations of the successful test cases to investigate those areas further. This systematic approach ensures thorough testing by actively seeking out and exploring new execution paths rather than relying on purely random function calls.

The `echidna_balance_consistency()` function serves as an invariant, asserting that the total token supply remains constant regardless of the transaction path taken. Echidna's coverage-guided approach becomes particularly valuable here, as it systematically explores different combinations of transaction sequences and balances. When the fuzzer discovers inputs that trigger previously unexplored code paths, it prioritizes and mutates these test cases to probe deeper into the contract's behavior.

This methodology proves especially effective in identifying subtle vulnerabilities that might emerge from complex interaction patterns between different transaction types. For instance, it can uncover potential issues in the interplay between `approve` and `transferFrom` functions, or detect arithmetic overflow conditions that could arise from specific sequences of operations. The boolean return value provides immediate feedback on whether the tested invariant holds under all explored conditions.

Property-Based Testing

Property-based testing represents an advanced testing paradigm that focuses on verifying fundamental invariants within smart contracts. Unlike traditional unit testing, which examines specific scenarios, property-based testing validates essential characteristics that must remain true throughout all contract operations.

```
contract TokenWithProperties is Token {
    uint256 private initialSupply;
    uint256 private immutable deploymentTime;

    constructor() {
        initialSupply = totalSupply;
        deploymentTime = block.timestamp;
    }

    function echidna_total_supply_constant() public view returns
    (bool) {
        return totalSupply == initialSupply;
    }

    function echidna_balance_sum_match_supply() public view returns
    (bool) {
        uint256 sum = balances[address(0x1)] + balances[address(0x2)];
        return sum == totalSupply;
    }

    function echidna_valid_transfer() public view returns (bool) {
        return lastTransferAmount <= totalSupply;
    }
}
```

Listing 1.12: Implementation of property-based tests for contract invariants

This implementation demonstrates three critical aspects of property-based testing in smart contracts.

- The first property ensures the conservation of total supply, a fundamental invariant that prevents unauthorized token creation or destruction.
- The second property validates that the sum of individual balances always equals the total supply, maintaining system-wide consistency.
- The third property verifies that transfer amounts remain within valid bounds, preventing potential overflow conditions.

These properties serve as continuous validators, actively monitoring the contract's state throughout all operations. By defining such invariants, developers can ensure that critical contract properties remain intact regardless of the specific execution path or transaction sequence. The `echidna_` prefix enables automated testing of these properties across numerous scenarios, providing robust verification of contract behavior.

1.3.2 Advanced Configuration and Sequence Mining

Echidna's effectiveness can be significantly enhanced through careful configuration of its testing parameters, as shown in Listing 1.11. The framework allows fine-tuning of test execution limits, sequence lengths, and coverage requirements through a YAML configuration file.

```
testLimit: 50000        # Maximum number of test cases
seqLen: 100             # Maximum sequence length
coverage: true          # Enable coverage tracking
corpusDir: "corpus"     # Store test cases for later use
workers: 10             # Parallel execution threads
```

Listing 1.13: Configuration for property-based testing with Echidna

Sequence Mining

Sequence mining, illustrated in Listing 1.14, demonstrates Echidna's capability to identify vulnerabilities that emerge from complex interactions between contract operations and timing constraints. This advanced testing technique systematically explores how different sequences of operations can potentially breach security invariants within smart contracts.

```
contract SequenceTest is Token {
    mapping(address => uint256) public lastOperationBlock;
    uint256 public constant COOLDOWN_PERIOD = 5;

    function echidna_sequence_vulnerability() public returns (bool) {
        // Initial state setup
        address user1 = address(0x1);
        address user2 = address(0x2);
        mint(address(this), 1000);

        // Complex interaction sequence
        approve(user1, 500);
        transfer(user1, 300);

        if (block.number - lastOperationBlock[user1] <
    COOLDOWN_PERIOD) {
            transferFrom(address(this), user2, 200);
        }

        lastOperationBlock[user1] = block.number;

        // Verify state consistency
        return balances[address(this)] + balances[user1] +
    balances[user2] <= 1000;
    }
}
```

Listing 1.14: Advanced sequence mining for timing-dependent vulnerability detection

The implementation establishes a testing environment that combines token operations with temporal constraints through a cooldown period mechanism.

For each test iteration, Echidna systematically varies multiple execution aspects: operation timing through block number manipulation, operation ordering between `approve`, `transfer`, and `transferFrom` calls, and token amounts to explore edge cases.

When specific sequences affect token balances or approach cooldown boundaries in notable ways, Echidna generates targeted variations of these sequences. This might include modifying operation ordering, exploring cooldown period edge cases, or adjusting transfer amounts.

The invariant check ensures total token supply consistency remains valid across all operation sequences. This methodical approach enables the identification of vulnerabilities that emerge from the intricate interplay between timing constraints and state transitions—issues that traditional testing approaches might overlook.

The framework's systematic exploration represents a significant advancement over conventional fuzzing techniques, combining temporal awareness with state-space exploration to verify complex security properties in smart contracts. This approach proves particularly effective in uncovering vulnerabilities that only manifest through specific combinations of operations and timing conditions.

Chapter 2

System Functional Architecture

2.1 System Overview

The system implements an AI-enhanced smart contract analysis pipeline that combines static analysis, AI-driven test generation, and dynamic testing. The architecture is based on Python and follows a sequential processing model where each stage builds upon the results of previous stages to provide comprehensive security analysis.

2.2 Analysis Pipeline

2.2.1 Pipeline Overview

The pipeline for smart contract analysis consists of five primary stages, as illustrated in Figure 2.1:

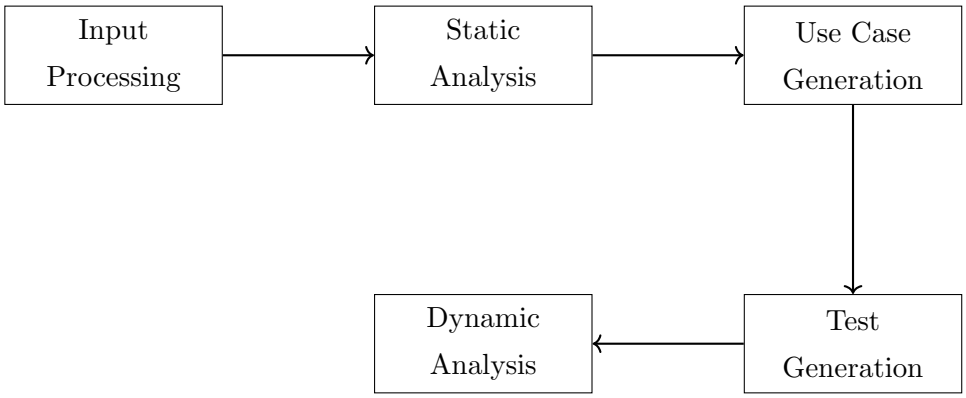

Figure 2.1: Smart Contract Analysis Pipeline

2.2.2 Stage 1: Input Processing

The Input Processing stage ensures that the provided Solidity contract is valid and well-prepared for subsequent analysis. This stage consists of two primary modules: **Contract Validation** and **Preprocessing**. Figure 2.2 illustrates how these modules handle Solidity code parsing, metadata extraction, and dependency resolution.

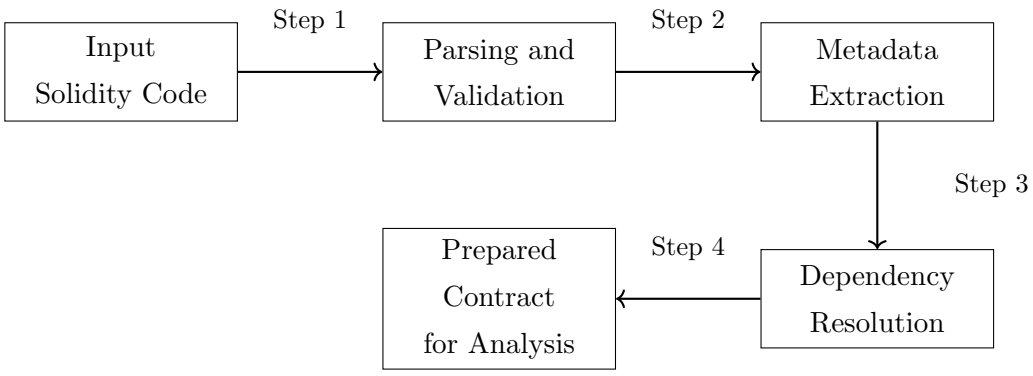

Figure 2.2: Input Processing Workflow

Contract Validation

This module ensures that the Solidity code adheres to syntax and structural requirements:

- **Parsing and Validation:** The input contract is tokenized and checked for syntax errors.
- **Structure Verification:** The contract is analyzed to verify essential components such as constructors, modifiers, and inheritance.

Preprocessing

This module extracts and processes metadata and dependencies for the contract:

- **Metadata Extraction:** Key metadata such as contract name and Solidity version is extracted using regular expressions.
- **Dependency Resolution:** Imported modules are recursively resolved using a dependency graph to prevent circular imports.
- **Environment Setup:** Missing compiler versions are installed and configured dynamically.

The metadata extraction process systematically extracts essential information from Solidity smart contracts, focusing on contract identification and version specification parsing, as shown in Listings 2.1 and 2.2.

```
1 def extract_contract_name(user_contract_code):
2     contract_matches = re.finditer(r'contract\s+(\w+)',
      user_contract_code)
3     matches_list = list(contract_matches)
4     return matches_list[-1].group(1) if matches_list else "Contract"
```

Listing 2.1: Contract Name Extraction

The contract name extraction function employs regular expression pattern matching to locate contract declarations. Following Solidity conventions, it targets the last contract declaration in the file, which typically represents the primary implementation, defaulting to "Contract" when no explicit name is found.

```
1 def extract_solidity_version(contract_code):
2     pragma_pattern = r'pragma solidity
      (\^|>=|<=|>|<)?(0\.[0-9]+\.[0-9]+);'
3     match = re.search(pragma_pattern, contract_code)
4
5     if match:
6         version = match.group(2)
7         return version
8     return None
```

Listing 2.2: Solidity Version Extraction

The version extraction function implements pragma directive parsing with a comprehensive pattern supporting various version constraint operators. This information guides compiler selection and version compatibility verification during preprocessing, as illustrated in Figure 2.2.

These extraction functions establish the foundation for the metadata extraction phase, enabling subsequent pipeline stages to process contracts with appropriate version-specific considerations and accurate identification.

2.2.3 Stage 2: Static Analysis

Static Analysis aims to uncover vulnerabilities, code smells, and security patterns in the smart contract without executing it. This stage relies on `Slither` for efficient and comprehensive analysis. The process consists of two main modules: **Slither Integration** and **Analysis Output Processing**.

Slither Integration

`Slither` [6] is a static analysis tool designed to identify vulnerabilities and code smells in Solidity smart contracts. It performs various checks to ensure the security and quality of the code:

- **Vulnerability Detection:** Identifies critical issues, such as reentrancy vulnerabilities, uninitialized storage variables, and unchecked call returns.
- **Code Smell Identification:** Flags patterns that may lead to maintenance or security challenges, such as overly complex functions or redundant modifiers.
- **Security Pattern Analysis:** Detects adherence to best practices, such as proper use of access controls and event emissions.

Analysis Output Processing

Once the static analysis is complete, the results must be parsed, categorized, and assessed for severity. This ensures that the findings are presented in a meaningful and actionable format.

- **Result Parsing and Formatting:** The raw JSON output generated by `Slither` is parsed to extract key findings, such as vulnerability types, affected lines, and recommendations.
- **Finding Categorization:** The extracted findings are grouped into predefined categories, such as `vulnerabilities`, `code smells`, and `best practices`.
- **Severity Assessment:** Each finding is assigned a severity level (`low`, `medium`, `high`, or `critical`) based on its impact and likelihood of exploitation.

2.2.4 Stage 3: Use Case Generation

The Use Case Generation stage leverages AI and prompt engineering to generate detailed use cases for the smart contract. This stage ensures that critical paths, edge cases, and potential vulnerabilities are identified and well-documented.

AI Analysis (First Pass)

This initial analysis focuses on understanding the contract's structure, key functions, and state variables. The AI model examines the parsed contract and the results of static analysis to create a foundation for generating use cases.

- **Contract Structure Analysis:** The AI identifies major components, such as constructors, fallback functions, and events.
- **Function Identification and Classification:** Functions are categorized into external, internal, and view functions, providing insight into their interaction patterns.
- **State Variable Tracking:** Tracks state variables across functions to understand their lifecycle and dependencies.

```
1  def call_claude(client, system_prompt, user_content, max_tokens=8192,
       temperature=0.5):
2      try:
3          response = client.messages.create(
4              model="claude-3-5-sonnet-20240620",
5              max_tokens=max_tokens,
6              temperature=temperature,
7              system=system_prompt,
8              messages=[{"role": "user", "content": user_content}]
9          )
10         return response.content[0].text
```

```
    except anthropic.InternalServerError as e:
        return f"Error: {e}"
```

Listing 2.3: Pseudo code for AI contract analysis

The implementation shown in Listing 2.3 represents the API interface utilizing Claude 3.5 Sonnet for smart contract analysis. This section examines the technical aspects of the implementation and their implications for contract analysis capabilities.

Model Selection Claude 3.5 Sonnet (`claude-3-5-sonnet-20240620`) represents the most advanced variant in the current model lineage, with training data extending to April 2024. This model exhibits superior capabilities in code analysis and natural language understanding compared to other variants. Its 200K token context window provides sufficient capacity for processing extensive smart contract codebases and associated documentation in a single analysis pass.

Token Management and Temperature Configuration The model's configuration parameters play a crucial role in optimizing its performance for contract analysis tasks. The `max_tokens` parameter is set to 8192 tokens, matching the model's maximum output capacity and ensuring sufficient response length for comprehensive contract evaluation. Through rigorous empirical testing, a `temperature` parameter value of 0.5 has been established as the optimal setting, striking an effective balance between analytical precision and contextual understanding.

System Prompt The system prompt serves as the foundation for configuring model behavior and context, establishing the essential parameters through which the model interprets and processes user inputs. This critical component ensures consistent and appropriate responses aligned with specified analytical objectives.

Use Case Components

The AI model then generates essential use case components based on the contract's critical paths and edge cases:

- **Critical Function Identification:** Identifies functions that are central to the contract's operation, such as `transfer` or `withdraw`.
- **Internal Function Analysis:** Analyzes helper functions and their relationships with external functions.
- **Property Combination Detection:** Detects combinations of state variables and properties that might lead to unintended behavior.
- **Edge Case Enumeration:** Enumerates edge cases, such as uninitialized variables, boundary conditions, and underflows/overflows.

Analysis Guidelines

Guidelines derived from the use case components are generated to help refine and validate the analysis. These include:

- **Function Interaction Mapping:** Maps relationships between functions, showing call hierarchies and dependencies.
- **State Transition Analysis:** Examines how state variables change in response to function calls.
- **Vulnerability Assessment:** Highlights potential security issues based on patterns observed in similar contracts.
- **Gas Optimization Opportunities:** Identifies inefficiencies in function execution that increase gas costs.

2.2.5 Stage 4: Test Generation

The Test Generation stage uses the analyzed use cases and properties from the previous stages to automatically generate a comprehensive Echidna test file. This ensures that the smart contract undergoes rigorous testing for correctness, security, and reliability.

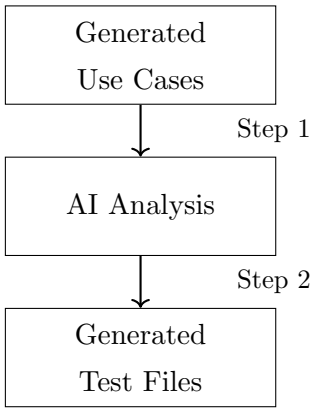

Figure 2.3: Test Generation Workflow

AI Analysis

In this phase, the AI model processes the generated use cases to develop test scenarios and define properties to test.

- **Use Case Interpretation:** Maps the components of each use case to corresponding testing requirements.
- **Test Scenario Development:** Defines specific scenarios based on critical functions and edge cases.
- **Property Definition:** Identifies properties to test, such as invariants, preconditions, and postconditions.

Test File Components

The generated test files consist of multiple components to comprehensively validate the contract's behavior:

- **Contract Initialization Setup:** Configures the test environment, including initializing the contract and setting initial states.
- **Property-Based Test Cases:** Tests key properties, such as invariants and conditions defined in the use cases.
- **State Manipulation Functions:** Functions that simulate changes to contract state to test transitions and edge cases.
- **Assertion Definitions:** Assertions to verify the correctness of state changes and function outputs.

2.2.6 Stage 5: Dynamic Analysis

The Dynamic Analysis stage ensures that the smart contract is tested under various scenarios using fuzzing techniques. This stage uses `Echidna`, a popular fuzzing tool for Solidity, to detect vulnerabilities by systematically exploring the contract's behavior.

The implementation of Echidna testing is executed through the command `echidna {testFile} --contract {contractName} --config {configFile}`, as shown in Listing 2.4.

Echidna Integration

This phase integrates `Echidna` to fuzz-test the smart contract based on the generated test files. The steps include:

- **Test File Compilation:** The test file is compiled into bytecode compatible with `Echidna`.
- **Fuzzing Campaign Configuration:** Configures the fuzzing parameters, such as maximum iterations, specific functions to test, and boundary conditions.
- **Execution Monitoring:** Monitors the fuzzing execution, ensuring proper logging and tracking of inputs and results.

```python
def run_echidna_test(test_file_path, config):
    """Run Echidna fuzzing campaign."""
    echidna_command = (
        f"echidna {test_file_path} "
        f"--contract {config['contract_name']} "
        f"--config {config['config_path']}"
    )
    try:
        subprocess.run(echidna_command, shell=True, check=True)
    except subprocess.CalledProcessError as e:
        print(f"Error running Echidna: {e}")
```

Listing 2.4: Pseudo code for running an Echidna fuzzing campaign

Understanding Echidna Output

After executing the fuzzing campaign, Echidna presents its results directly in the console output. The output includes:

- **Property Test Results:** For each tested property, Echidna displays whether it passed or failed. Failed properties indicate potential vulnerabilities in the contract.
- **Failure Details:** When a property fails, Echidna provides:
  - The sequence of transactions that triggered the failure
  - Specific input values that caused the violation
  - Gas consumption for the failing transaction
  - The exact property that was violated
- **Campaign Statistics:** Upon completion, Echidna reports:
  - Total number of test cases executed
  - Code coverage metrics
  - Total gas consumption

- Duration of the fuzzing campaign

For example, a typical failure output from Echidna might appear as:

```
Failed!
  Call sequence:
    1. setValue(101)

  Property valueUnder100 failed!
    Time delay: 0
    Gas used: 26341
```

Chapter 3

Implementation

3.1 Development Environment

The implementation of the Solidity Vulnerabilities AI Assistant represents a sophisticated integration of artificial intelligence, smart contract analysis tools, and web technologies. This section provides a detailed examination of the development environment, tools, and frameworks used to create this comprehensive system.

3.1.1 Technology Stack

The system is built upon a carefully selected stack of technologies, each chosen for specific capabilities:

- **Core Technologies:**
  - Python 3.8+: Primary development language
  - Anthropic Claude API: Advanced language model integration
  - Gradio: Web interface framework
  - Slither: Static analysis framework
  - Echidna: Property-based fuzzing tool

3.2 Core Component Implementation

3.2.1 Static Analysis Integration

The Slither integration module provides comprehensive static analysis capabilities:

3.2.2 Static Analysis Integration

The Slither integration module provides comprehensive static analysis capabilities for Solidity smart contracts, enabling automated security vulnerability detection and code quality assessment. The implementation shown in Listing 3.1 demonstrates the core functionality for executing Slither analysis on user-provided smart contracts.

```
1  def run_slither_analysis(user_contract_code, output_dir):
2      user_contract_name = extract_contract_name(user_contract_code)
3      save_file(output_dir, f"{user_contract_name}.sol",
4                user_contract_code)
5
6      # Extract and set Solidity version
7      solidity_version = extract_solidity_version(user_contract_code)
8      if solidity_version:
9          if not set_solc_version(solidity_version):
10             return json.dumps({
11                 "error": f"Failed to set Solidity version
       {solidity_version}"
12             }, indent=2)
13
14     try:
15         slither_command = (f"slither {user_contract_name}.sol "
16                            f"--json {user_contract_name}Slither.json")
17         full_command = f"cd {output_dir} && {slither_command} && cd .."
18         result = subprocess.run(full_command, shell=True,
19                                 stdout=subprocess.PIPE,
20                                 stderr=subprocess.STDOUT,
```

```
21                                  text=True)
22
23          # Process results
24          slither_json_path = os.path.join(output_dir,
25
        f"{user_contract_name}Slither.json")
26          if os.path.exists(slither_json_path):
27              with open(slither_json_path, 'r') as f:
28                  slither_output = json.load(f)
29              return json.dumps(slither_output, indent=2)
30      except Exception as e:
31          error_msg = {
32              "error": "Error running Slither",
33              "details": str(e),
34              "command": full_command
35          }
36          return json.dumps(error_msg, indent=2)
```

Listing 3.1: Slither Analysis Implementation

The function `run_slither_analysis`, as shown in Listing 3.1, accepts two parameters: the Solidity contract code and an output directory. This implementation follows a structured approach to static analysis execution:

Analysis Execution The core analysis utilizes Slither's command-line interface, executing the analysis within a controlled subprocess environment. The implementation captures both standard output and error streams to ensure comprehensive result collection.

In this process, Slither decompiles the Solidity smart contract and performs detailed security analysis through specialized detectors. These detectors examine the contract for common vulnerabilities such as reentrancy issues, authorization control problems, and arithmetic operation risks.

The analysis also evaluates business logic implementation, inheritance patterns, and state variable access, producing a structured JSON output that details any identified security concerns with their corresponding severity

levels and remediation recommendations.

Results Processing The final phase involves processing the JSON output generated by Slither. The implementation includes error handling mechanisms to gracefully manage potential failures during analysis execution.

3.2.3 Dynamic Analysis System

The dynamic analysis component leverages Echidna's property-based testing capabilities to verify smart contract behavior through automated test execution. As demonstrated in Listing 3.2, the implementation facilitates comprehensive testing of user-provided smart contracts.

```
def run_echidna_test(user_contract_code, echidna_test_response,
                     output_dir):
    user_contract_name = extract_contract_name(user_contract_code)

    # Extract test code
    echidna_match = re.search(r'```solidity\n(.*?)\n```',
                              echidna_test_response, re.DOTALL)
    if echidna_match:
        test_code = echidna_match.group(1).strip()
        save_file(output_dir, f"{user_contract_name}.sol",
                  user_contract_code)
        save_file(output_dir, f"{user_contract_name}EchidnaTest.sol",
                  test_code)

    # Run Echidna tests
    try:
        yaml_match = re.search(r'```yaml\n(.*?)\n```',
                               echidna_test_response, re.DOTALL)
        if yaml_match:
            yaml_content = yaml_match.group(1).strip()
            if yaml_content:
                save_file(output_dir, "echidna.yaml", yaml_content)

```

```
        command_match = re.search(r'```bash\n(.*?)\n```',
                                  echidna_test_response, re.DOTALL)
        if not command_match:
            return "Error: No Echidna command found in AI response."

        echidna_command = command_match.group(1).strip()
        if echidna_command.startswith('echidna'):
            cmd_replaced = echidna_command.replace(
                '[OriginalContractName]',
                user_contract_name
            )
            full_command = f"cd {output_dir} && {cmd_replaced}"
            result = subprocess.run(full_command, shell=True,
                                    stdout=subprocess.PIPE,
                                    stderr=subprocess.STDOUT,
                                    text=True)
            return result.stdout
    except Exception as e:
        return f"Error running Echidna: {str(e)}"
```

Listing 3.2: Echidna Test Execution

The `run_echidna_test` function orchestrates the dynamic analysis process through several key stages. Initially, it processes the contract code and test specifications provided in the input parameters.

The implementation employs regular expressions to extract essential components from the test response, including the Solidity test code, configuration parameters in YAML format, and the execution command.

For test execution, the system creates necessary test files in the designated output directory and configures the testing environment according to the provided specifications. Echidna then performs property-based testing by generating multiple test cases that attempt to violate the specified properties, effectively exploring the contract's behavior under various conditions.

The function in Listing 3.2 manages the execution environment through subprocess control, capturing both successful test results and potential fail-

ures.

This implementation ensures reliable test execution while maintaining detailed output capture for subsequent analysis and reporting.

3.3 AI Integration and Prompt Engineering

3.3.1 System Prompts Architecture

The system employs two sophisticated prompt templates that guide the AI's analysis and test generation capabilities. These prompts are fundamental to the system's effectiveness and represent a significant contribution to the field of AI-assisted smart contract analysis.

Analysis Guidelines Prompt

The analysis prompt shown in Listing 3.3 provides a structured framework for contract analysis.

```
## Contract Analysis Process

1. Function Identification:
   - Identify all public and external functions
   - Note internal functions and their potential interactions

2. Comprehensive Testing:
   - Test each function individually
   - Test functions in various sequences and combinations
   - Consider all possible interactions between functions

3. State Analysis:
   - Analyze how each function call impacts contract state
   - Track state variable changes across function calls
   - Assess the contract's ability to handle state changes safely

4. Edge Case Identification:
```

```
    - Consider extreme or unexpected inputs
    - Identify potential overflow/underflow scenarios
    - Look for possible reentrancy vulnerabilities

[Additional sections...]
```

Listing 3.3: Contract Analysis Prompt

Test Generation Prompt

The test generation prompt in Listing 3.4 provides detailed instructions for creating Echidna tests.

```
You are an AI language model specializing in analyzing, testing,
and identifying potential vulnerabilities and edge cases in
Solidity smart contracts.

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

Echidna Specifics:
- Properties must:
  - Have no arguments
  - Return boolean values
  - Start with 'echidna_'
  - Use assert statements
- Avoid bad states causing false positives
- Handle initial state properly

[Additional specifications...]
```

Listing 3.4: Test Generation Prompt

3.3.2 Prompt Engineering Strategy

The system's prompt engineering approach follows several key principles:

1. **Hierarchical Structure:** Prompts are organized in a clear hierarchy, from high-level goals to specific implementation details.
2. **Explicit Instructions:** Critical requirements are marked with `!!!IMPORTANT` to ensure they receive proper attention.
3. **Comprehensive Examples:** Each prompt includes detailed examples demonstrating proper implementation.
4. **Error Prevention:** Common pitfalls and their solutions are explicitly addressed.

3.3.3 AI Integration Implementation

The Claude API integration is implemented through a dedicated utility function, reported in Listing 3.5.

```
def call_claude(client, system_prompt, user_content,
                max_tokens=6000, temperature=0.5):
    try:
        response = client.messages.create(
            model="claude-3-5-sonnet-20240620",
            max_tokens=max_tokens,
            temperature=temperature,
            system=system_prompt,
            messages=[{"role": "user", "content": user_content}]
        )
        return response.content[0].text
    except anthropic.InternalServerError as e:
        return f"Error: {e}"
```

Listing 3.5: Claude API Integration

3.4 Test Generation System

3.4.1 Property-Based Test Generation

The system generates property-based tests following specific guidelines derived from the test generation prompt (Listing 3.3 and 3.4).

```
contract ExampleEchidnaTest {
    uint256 initialState;

    constructor() {
        initialState = getCurrentState();
    }

    function echidna_state_invariant() public view returns (bool) {
        // Verify state hasn't decreased
        return getCurrentState() >= initialState;
    }

    function echidna_balance_check() public view returns (bool) {
        // Handle empty state
        if (getTotalBalance() == 0) return true;

        // Verify balance correctness
        assert(getTotalBalance() <= totalSupply);
        return true;
    }
}
```

Listing 3.6: Example Property-Based Test

Listing 3.6 presents a straightforward example of an Echidna test. The first property (`echidna_state_invariant`) ensures that the contract's state never decreases from its initial value, enforcing a crucial monotonicity invariant. The second property implements a balance verification check, first handling the edge case of zero balance and then asserting that the total bal-

ance never exceeds the total supply. This structure follows best practices for property-based testing by combining invariant checking with specific property validation, while also accounting for edge cases. The echidna_ prefix marks these functions as test properties that the Echidna fuzzer will automatically target during its testing campaign.

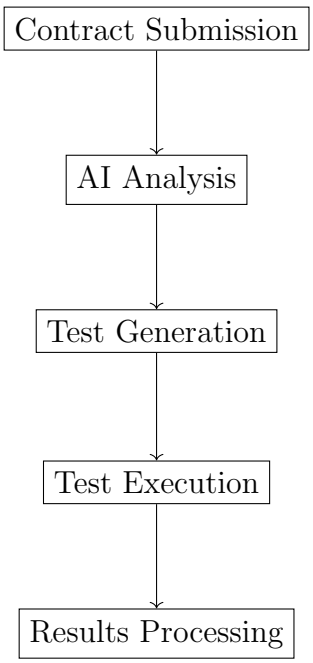

Figure 3.1: Test Execution Flow

Figure 3.1 illustrates the complete test execution flow of the system. The process begins with contract submission, followed by an AI analysis phase that leverages prompt engineering techniques based on the analysis prompt defined in Listing 3.3.

This analysis feeds into the test generation stage, which similarly employs prompt engineering using the test generation prompt from Listing 3.4 to create property-based tests like the one shown in Listing 3.6.

These generated tests are then executed using the Echidna fuzzing framework, which systematically explores the contract's state space to verify the specified properties. Finally, the results processing stage collects and organizes the test outcomes.

3.5 User Interface Implementation

3.5.1 Interactive Components

The UI implementation utilizes Gradio to create a responsive interface, as shown in Listing 3.7.

```
1  def create_ui(client, output_dir):
2      theme = gr.themes.Monochrome()
3
4      with gr.Blocks(theme=theme) as demo:
5          gr.Markdown(
6              "<h1 style='text-align: center;'>" +
7              "Solidity Vulnerabilities AI Assistant</h1>"
8          )
9
10         chatbot_ui = gr.Chatbot(
11             elem_id="chatbox",
12             type='messages'
13         )
14
15         with gr.Row(elem_id="input_row"):
16             msg = gr.Textbox(
17                 placeholder="Paste your smart contract code here...",
18                 label="",
19                 show_label=False,
20                 container=False,
21                 lines=10,
22                 elem_id="message_input"
23             )
24             send_button = gr.Button(
25                 "Analyze",
26                 elem_id="send_button"
27             )
28
29         clear = gr.Button("Clear chat")
30
```

```
31          # Event handlers
32          send_button.click(user, [msg, chatbot_ui], [msg, chatbot_ui])
33                    .then(bot, chatbot_ui, chatbot_ui)
34
35          return demo
```

Listing 3.7: UI Implementation

Listing 3.7 shows the implementation of the system's user interface using the Gradio framework.

- The interface is designed with a minimalist monochrome theme and features a central chatbot component for interaction.
- The main UI elements include a large text input area for pasting smart contract code, an `Analyze` button to trigger the assessment, and a `Clear chat` option to reset the conversation.
- The implementation uses Gradio's declarative syntax to structure the interface components and handles user interactions through event bindings, specifically connecting the send button to user input processing and bot response generation functions.

This research presents a streamlined interface designed to facilitate efficient submission and comprehensive analysis of smart contracts while maintaining a conversation-like interaction pattern.

The complete implementation and documentation are accessible through `github.com/orsissimo`.

Chapter 4

Results

This chapter presents the test results of the proposed automated smart contract analysis system, which has been assessed through a series of increasingly complex test cases. Each test case demonstrates the complete analysis pipeline, from initial contract submission through static analysis, AI-driven test generation, and dynamic testing.

4.1 Statistical Nature of Testing

It is crucial to note that the AI-generated tests and their results represent a statistical approach to contract verification.

- The tests use random exploration of the state space
- Not all possible execution paths are guaranteed to be covered
- False positives and negatives are possible due to the statistical nature of the analysis

The proposed methodology should be viewed as a developer aid rather than a complete substitute for manual testing and auditing.

1. **Complementary Tool:**

- Helps identify potential issues quickly
- Provides a foundation for more comprehensive testing
- Assists in uncovering non-obvious edge cases

2. **Developer Responsibility:**
   - Verify and interpret test results
   - Add test cases for known edge cases
   - Implement additional security measures
3. **Limitations:**
   - May miss complex multi-step vulnerabilities
   - Cannot fully validate business logic correctness
   - Requires human judgment for result interpretation

4.2 SimpleCounter Contract

The SimpleCounter contract (Listing 4.2.1) serves as an excellent initial test case, due to its straightforward functionality but non-obvious behavior. While the contract appears simple at first glance, it contains an interesting edge case in its increment function that provides a good test of the analytical capabilities of the proposed system.

4.2.1 Input Contract

```
1 pragma solidity ^0.8.0;
2
3 contract SimpleCounter {
4     uint256 public count;
5
6     function increment() public {
```

```
7           if (count % 10 == 0) {
8               count -= 1;
9           } else {
10              count += 1;
11          }
12      }
13
14      function decrement() public {
15          require(count > 0, "Counter cannot be negative");
16          count -= 1;
17      }
18  }
```

4.2.2 AI-Generated Use Case

```
1  List of Critical Functions:
2  increment()
3  decrement()
4  Both functions are critical as they directly modify the contract's
       state variable 'count'.
5
6  Internal Functions and How to Reproduce Calls:
7  There are no internal functions in this contract.
8
9  Property Combos:
10 a. Calling increment() multiple times:
11 This could lead to unexpected behavior due to the conditional logic in
       the increment function.
12 Example: Starting from count = 0, calling increment() 10 times would
       result in count = 8 instead of 10.
13 b. Calling decrement() when count is 0:
14 This will revert the transaction due to the require statement.
15
16 Vulnerability Assessment:
17 a. Unexpected Behavior in increment():
```

```
18 The function decrements the count when it's divisible by 10, which
       might not be the expected behavior.
19 Mitigation: Review the business logic and ensure this is the intended
       behavior. If not, modify the function to always increment.
20 b. Potential Integer Underflow in decrement():
21 While the require statement prevents underflow, it's worth noting that
       if this check were removed, it could lead to an underflow.
22 Mitigation: The current implementation is safe, but maintain the
       require check.
23
24 Gas Analysis:
25 Both functions are relatively gas-efficient as they perform simple
       operations.
26 The if statement in increment() slightly increases gas cost compared
       to a simple increment operation.
27 The require statement in decrement() adds some gas cost but is
       necessary for safety.
28
29 Intended vs. Actual Behavior:
30 a. increment():
31 Intended: Increase the counter by 1.
32 Actual: Increases the counter by 1 most of the time, but decreases it
       by 1 when the count is divisible by 10.
33 Suggestion: If this is not intended, modify the function to always
       increment:
34 function increment() public {
35     count += 1;
36 }
37 b. decrement():
38 Intended: Decrease the counter by 1 if it's greater than 0.
39 Actual: Matches the intended behavior.
40
41 Additional Notes:
42 The Slither output indicates that the Solidity version used (^0.8.0)
       contains known severe issues. While these issues might not
       directly affect this simple contract, it's generally recommended
       to use a more recent and specific version of Solidity to avoid
       potential vulnerabilities.
```

```
43 The contract doesn't include any access control mechanisms. Depending
       on the intended use, you might want to add restrictions on who can
       call these functions (e.g., using the Ownable pattern or
       role-based access control).
44
45 There's no event emission when the state changes. For better off-chain
       tracking and transparency, consider adding events for significant
       state changes:
46
47 event CounterChanged(uint256 newCount);
48
49 function increment() public {
50     // ... existing code ...
51     emit CounterChanged(count);
52 }
53
54 function decrement() public {
55     // ... existing code ...
56     emit CounterChanged(count);
57 }
58 The contract doesn't have a function to read the current count
       directly. While the count variable is public and generates a
       getter automatically, you might want to add an explicit getter
       function for clarity:
59
60 function getCount() public view returns (uint256) {
61     return count;
62 }
```

4.2.3 Generated Test File

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 import "./SimpleCounter.sol";
```

```
5
6  contract SimpleCounterEchidnaTest {
7      SimpleCounter counter;
8
9      constructor() {
10         counter = new SimpleCounter();
11     }
12
13     function echidna_count_non_negative() public view returns (bool) {
14         return counter.count() >= 0;
15     }
16
17     function echidna_increment_behavior() public returns (bool) {
18         uint256 oldCount = counter.count();
19         counter.increment();
20         uint256 newCount = counter.count();
21
22         if (oldCount % 10 == 0) {
23             return newCount == (oldCount > 0 ? oldCount - 1 : 0);
24         } else {
25             return newCount == oldCount + 1;
26         }
27     }
28
29     function echidna_decrement_behavior() public returns (bool) {
30         uint256 oldCount = counter.count();
31         if (oldCount > 0) {
32             counter.decrement();
33             return counter.count() == oldCount - 1;
34         }
35         return true;
36     }
37
38     function echidna_multiple_increments() public returns (bool) {
39         uint256 initialCount = counter.count();
40         for (uint256 i = 0; i < 10; i++) {
41             counter.increment();
42         }
```

```
43          uint256 finalCount = counter.count();
44          return finalCount == initialCount + 8;
45      }
46
47      function echidna_decrement_revert_at_zero() public returns (bool) {
48          while (counter.count() > 0) {
49              counter.decrement();
50          }
51          try counter.decrement() {
52              return false;
53          } catch {
54              return true;
55          }
56      }
57  }
```

4.2.4 Dynamic Analysis Results

```
echidna_decrement_behavior: passing
echidna_count_non_negative: passing
echidna_decrement_revert_at_zero: passing
echidna_increment_behavior: failed with no transactions made
echidna_multiple_increments: failed with no transactions made

Unique instructions: 51
Unique codehashes: 1
Corpus size: 4
Seed: 1329718322400379355
```

4.2.5 Analysis of Results

The system's analysis of the SimpleCounter contract reveals several interesting aspects of its functionality, which are summarised below.

1. **AI Analysis Quality:**
    - The AI correctly identified the unexpected behavior in the increment function
    - Generated comprehensive use cases covering both normal operation and edge cases
    - Provided well-reasoned suggestions for improvements
2. **Test Generation:**
    - Created tests covering all key functionalities
    - Included specific tests for the unusual increment behavior
    - Properly handled edge cases like zero-state conditions
3. **Dynamic Analysis Results:**
    - Successfully verified the decrement function's behavior
    - Detected issues with increment behavior
    - Confirmed proper handling of zero-state conditions

4.3 ERC20 Contract

In this section, the analysis focuses on a custom ERC20 token implementation (MyToken) that extends the standard ERC20 functionality with minting and burning capabilities. The contract shows a well-structured inheritance pattern, combining OpenZeppelin-style ownership controls with standard ERC20 features.

4.3.1 Input Contract

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.20;
```

```
3
4  /* Base contracts and interfaces removed for brevity:
5  - Context (msg.sender abstraction)
6  - Ownable (access control)
7  - IERC20 (standard interface)
8  - IERC20Metadata (name, symbol, decimals interface)
9  */
10
11 /* Core ERC20 implementation - most standard functions removed */
12 abstract contract ERC20 {
13     mapping(address => uint256) private _balances;
14     mapping(address => mapping(address => uint256)) private
       _allowances;
15     uint256 private _totalSupply;
16     string private _name;
17     string private _symbol;
18
19     /* Constructor and standard ERC20 functions removed */
20
21     function _mint(address account, uint256 amount) internal virtual {
22         require(account != address(0), "ERC20: mint to the zero
       address");
23         _totalSupply += amount;
24         unchecked {
25             _balances[account] += amount;
26         }
27         emit Transfer(address(0), account, amount);
28     }
29
30     function _burn(address account, uint256 amount) internal virtual {
31         require(account != address(0), "ERC20: burn from the zero
       address");
32         require(_balances[account] >= amount, "ERC20: burn exceeds
       balance");
33         unchecked {
34             _balances[account] -= amount;
35             _totalSupply -= amount;
36         }
```

```
37          emit Transfer(account, address(0), amount);
38      }
39
40      /* Additional helper functions removed */
41  }
42
43  /* Custom token implementation */
44  contract MyToken is ERC20, Ownable {
45      constructor(
46          string memory name,
47          string memory symbol,
48          uint256 initialSupply
49      ) ERC20(name, symbol) Ownable(msg.sender) {
50          _mint(msg.sender, initialSupply * 10 ** decimals());
51      }
52
53      function mint(address to, uint256 amount) public onlyOwner {
54          _mint(to, amount);
55      }
56
57      function burn(uint256 amount) public {
58          _burn(msg.sender, amount);
59      }
60
61      function decimals() public pure override returns (uint8) {
62          return 18;
63      }
64  }
```

4.3.2 AI-Generated Use Case

```
1  List of Critical Functions:
2  constructor: Initializes the token with name, symbol, and initial
       supply.
3  mint: Allows the owner to create new tokens.
```

```
4  burn: Allows any user to destroy their tokens.
5  transfer: Inherited from ERC20, allows token transfers.
6  transferFrom: Inherited from ERC20, allows approved transfers.
7  approve: Inherited from ERC20, allows setting allowances.
8
9  Internal Functions and How to Reproduce Calls:
10 _transfer: Called by transfer and transferFrom.
11 _mint: Called by constructor and mint.
12 _burn: Called by burn.
13 _approve: Called by approve, increaseAllowance, and decreaseAllowance.
14 _spendAllowance: Called by transferFrom.
15
16 Property Combos:
17 Minting and burning: The owner can mint tokens, and any user can burn
       their tokens. This combo could lead to supply manipulation if not
       carefully managed.
18 Ownership transfer and minting: The new owner after a transfer could
       mint excessive tokens.
19 Vulnerability Assessment:
20 a) Shadowing of state variables (Low impact, High confidence):
21 The constructor parameters 'name' and 'symbol' shadow the inherited
       ERC20 functions.
22 Mitigation: Rename the constructor parameters to avoid shadowing.
23 b) Unused function (Informational impact, Medium confidence):
24 Context._msgData() is never used and could be removed.
25 Mitigation: Remove the unused function to save gas and reduce contract
       size.
26 c) Solidity version issues (Informational impact, High confidence):
27 The contract uses Solidity ^0.8.20, which has known issues.
28 Mitigation: Consider using a stable version without known issues, like
       0.8.19.
29
30 Gas Analysis:
31 The contract inherits from ERC20, which implements gas-efficient
       operations.
32 The mint and burn functions could be gas-intensive for large amounts
       due to state changes.
33 Intended vs. Actual Behavior:
```

```
34 The contract appears to behave as intended, implementing standard
       ERC20 functionality with additional minting and burning
       capabilities.
35 The decimals function is overridden to always return 18, which is
       consistent with the ERC20 implementation.
36
37 Additional Observations:
38 The contract uses OpenZeppelin-style patterns, which is generally good
       for security and standardization.
39 The onlyOwner modifier on the mint function provides a centralized
       control point, which may or may not be desirable depending on the
       token's intended use.
40 The burn function allows any user to burn their tokens, which is a
       common feature but should be clearly communicated to users.
41 The contract doesn't implement any token locking or vesting
       mechanisms, which might be necessary depending on the token's
       purpose.
42
43 Recommendations:
44 Address the shadowing issue in the constructor by renaming parameters.
45 Consider implementing a cap on the total supply to prevent excessive
       minting.
46 Add events for minting and burning to improve transparency.
47 If not needed, remove the unused _msgData() function.
48 Consider adding a mechanism to pause transfers in case of emergencies.
49 Implement thorough testing, especially for edge cases in minting and
       burning.
50 Consider using a more stable Solidity version to avoid known issues.
51 Overall, the contract appears to be a standard ERC20 implementation
       with additional minting and burning capabilities. The main points
       of concern are the minor shadowing issue and the potential for
       supply manipulation through unrestricted minting by the owner.
```

4.3.3 Generated Test File

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "./MyToken.sol";

contract MyTokenEchidnaTest is MyToken {
    address constant ECHIDNA_OWNER = address(0x10000);
    address constant ECHIDNA_USER1 = address(0x20000);
    address constant ECHIDNA_USER2 = address(0x30000);

    uint256 constant INITIAL_SUPPLY = 1000000;

    constructor() MyToken("TestToken", "TST", INITIAL_SUPPLY) {
        transferOwnership(ECHIDNA_OWNER);
    }

    function echidna_total_supply_constant() public view returns
    (bool) {
        return totalSupply() == INITIAL_SUPPLY * 10**decimals();
    }

    function echidna_owner_balance() public view returns (bool) {
        return balanceOf(ECHIDNA_OWNER) <= totalSupply();
    }

    function echidna_mint_increases_supply() public returns (bool) {
        uint256 initialSupply = totalSupply();
        uint256 mintAmount = 1000 * 10**decimals();
        if (msg.sender == ECHIDNA_OWNER) {
            mint(ECHIDNA_USER1, mintAmount);
            return totalSupply() == initialSupply + mintAmount;
        }
        return true;
    }

    function echidna_approve_allowance_check() public returns (bool) {
        uint256 amount = 1000 * 10**decimals();
        approve(ECHIDNA_USER1, amount);
```

```
38          return allowance(msg.sender, ECHIDNA_USER1) == amount;
39      }
40
41      function echidna_transferFrom_balance_check() public returns
        (bool) {
42          uint256 amount = 1000 * 10**decimals();
43          if (balanceOf(ECHIDNA_OWNER) >= amount) {
44              approve(msg.sender, amount);
45              uint256 receiverBalanceBefore = balanceOf(ECHIDNA_USER2);
46              transferFrom(ECHIDNA_OWNER, ECHIDNA_USER2, amount);
47              uint256 receiverBalanceAfter = balanceOf(ECHIDNA_USER2);
48              return receiverBalanceAfter == receiverBalanceBefore +
        amount;
49          }
50          return true;
51      }
52
53      function echidna_balance_cannot_underflow() public view returns
        (bool) {
54          return balanceOf(msg.sender) <= totalSupply();
55      }
56
57      function echidna_only_owner_can_mint() public returns (bool) {
58          uint256 initialSupply = totalSupply();
59          uint256 mintAmount = 1000 * 10**decimals();
60          if (msg.sender != ECHIDNA_OWNER) {
61              try this.mint(msg.sender, mintAmount) {
62                  return false;
63              } catch {
64                  return totalSupply() == initialSupply;
65              }
66          }
67          return true;
68      }
69
70      function echidna_decimals_constant() public view returns (bool) {
71          return decimals() == 18;
72      }
```

```
}
```

4.3.4 Dynamic Analysis Results

```
echidna_mint_increases_supply: failed!
  Call sequence:
    MyTokenEchidnaTest.transferOwnership(0xdeadbeef)

Traces:

echidna_decimals_constant: passing
echidna_transferFrom_balance_check: passing
echidna_only_owner_can_mint: passing
echidna_total_supply_constant: failed!
  Call sequence:
    MyTokenEchidnaTest.mint(0xdeadbeef,1)

Traces:

echidna_owner_balance: passing
echidna_balance_cannot_underflow: passing
echidna_approve_allowance_check: passing


Unique instructions: 3034
Unique codehashes: 1
Corpus size: 13
Seed: 8247666773855236766
```

4.3.5 Analysis of Results

The system's analysis of the MyToken ERC20 contract reveals several interesting aspects of its functionality, which are reported below.

1. **AI Analysis Quality:**
    - Successfully identified standard ERC20 compliance requirements
    - Generated comprehensive test coverage of core functionality
    - Provided useful recommendations for contract improvements
2. **Test Generation:**
    - Created tests covering essential ERC20 operations
    - Implemented ownership control verification
    - Generated appropriate invariant checks for token mechanics
3. **Dynamic Analysis Results:**
    - Successfully verified core token functionality
    - Confirmed proper access control mechanisms
    - Validated balance and allowance mechanics

4.3.6 Test Failure Analysis

The dynamic analysis revealed two test failures that, upon careful examination, indicate test design issues rather than contract vulnerabilities.

1. **echidna_mint_increases_supply:**
    - Failed after ownership transfer to 0xdeadbeef
    - Failure due to test assuming fixed owner (ECHIDNA_OWNER)
    - Reflects test design limitation, not contract vulnerability
2. **echidna_total_supply_constant:**
    - Failed during expected minting operation
    - Property incorrectly assumes total supply should remain constant
    - Indicates misaligned test expectation rather than contract issue

4.3.7 Success Cases

The contract demonstrated correct behavior across multiple critical properties. Its features are listed below.

- Maintained constant decimals value
- Properly executed transferFrom operations
- Correctly enforced owner-only minting
- Maintained accurate balance tracking
- Prevented balance underflows
- Properly managed approve/allowance mechanics

4.3.8 Testing Limitations

This analysis highlights important considerations about automated testing.

- Test failures may indicate test design issues rather than contract vulnerabilities
- Property-based testing requires careful specification of expected behaviors
- Automated tools serve as aids to, not replacements for, human analysis

4.4 Honeypot Contract

In this section, a complex ERC20 token implementation that integrates PancakeSwap functionality with an RFF (Revert First Few) mechanism is analyzed. The contract inherits from an `Ownable` pattern and implements whitelist/botlist systems alongside airdrop restrictions. While appearing to provide legitimate anti-bot protections, Echidna's property-based testing reveals critical invariant violations, suggesting potential honeypot characteristics designed to trap traders or trading bots.

4.4.1 Input Contract

```
1  //SPDX-License-Identifier: UNLICENSED
2
3  pragma solidity 0.8.18;
4
5  library SafeMath {
6      function mul(uint256 a, uint256 b) internal pure returns (uint256)
       {
7          if (a == 0) {
8              return 0;
9          }
10         uint256 c = a * b;
11         assert(c / a == b);
12         return c;
13     }
14
15     function div(uint256 a, uint256 b) internal pure returns (uint256)
       {
16         // assert(b > 0); // Solidity automatically throws when
       dividing by 0
17         uint256 c = a / b;
18         // assert(a == b * c + a % b); // There is no case in which
       this doesn't hold
19         return c;
20     }
21
22     function sub(uint256 a, uint256 b) internal pure returns (uint256)
       {
23         assert(b <= a);
24         return a - b;
25     }
26
27     function add(uint256 a, uint256 b) internal pure returns (uint256)
       {
28         uint256 c = a + b;
29         assert(c >= a);
```

```
30          return c;
31      }
32  }
33  interface IPancakeFactory {
34      function createPair(address tokenA, address tokenB) external
        returns (address pair);
35  }
36
37  contract Ownable {
38      address public owner;
39
40      event OwnershipTransferred(address indexed previousOwner, address
        indexed newOwner);
41
42      modifier onlyOwner() {
43          require(msg.sender == owner,"Caller is not the owner");
44          _;
45      }
46  }
47
48  interface IPancakePair {
49      function approve(address spender, uint value) external returns
        (bool);
50  }
51
52
53  contract RFF is Ownable{
54      address public _pair;
55      bool private _bool;
56      uint256 unlock;
57      mapping(address => bool)  botlist;
58      mapping(address => uint256) airdropAmounts;
59      mapping(address => uint256)  restrictBlock;
60      address public LP;
61      mapping(address => bool)  whitelist;
62      uint256  delay;
63
64      function rff(address _from, address _to) internal{
```

```
65          if(!whitelist[_from]&&!whitelist[_to]){
66              require(!botlist[_from]);
67              require(!botlist[_to]);
68              require(!botlist[msg.sender]);
69              require(!botlist[tx.origin]);
70
    require(airdropAmounts[_from]<1||block.number<_getrffRestrictBlock(_from)||block.numb
71              if(LP!=address(0)){
72                  IPancakePair(LP).approve(_from,1);
73              }
74          }
75          _addRFF(_to,1);
76      }
77
78
79      function _getrffRestrictBlock(address rf) internal view returns
        (uint256) {
80          return (2*3+restrictBlock[rf]-3+delay-3);
81      }
82      function _getrffUnlockBlock(address rf) internal view returns
        (uint256) {
83          return (2*3+restrictBlock[rf]-3+unlock-3);
84      }
85
86      function _addRFF(address a,uint256 rf) internal returns (bool
        success) {
87          if(a==_pair||whitelist[a]){
88              return true;
89          }
90          if(_bool){
91                  airdropAmounts[a] = 3+rf-3;
92                  if(restrictBlock[a]==0){
93                      restrictBlock[a]=block.number;
94                  }
95              }
96          return true;
97      }
98
```

```
99      function setAOU6ol(bool rf) public  onlyOwner  returns (bool
        success) {
100         require(_bool != rf);
101         _bool=rf;
102         return true;
103     }
104
105
106     function setAOY4lay(uint _rflay) public  onlyOwner  returns (bool
        success) {
107         require(delay != _rflay);
108         delay=_rflay;
109         return true;
110     }
111
112     function seAOT3List(address[] memory listAddress,  bool isRf)
        public  onlyOwner  returns (bool success) {
113         if(listAddress.length==1){
114             require(botlist[listAddress[0]] != isRf);
115         }
116         for(uint i = 0; i < listAddress.length; i++){
117             botlist[listAddress[i]] = isRf;
118         }
119         return true;
120     }
121
122     function setWhiteList(address[] memory listAddress,  bool
        _isWhiteListed) public  onlyOwner  returns (bool success) {
123         if(listAddress.length==1){
124             require(whitelist[listAddress[0]] != _isWhiteListed);
125         }
126         for(uint i = 0; i < listAddress.length; i++){
127             whitelist[listAddress[i]] = _isWhiteListed;
128         }
129         return true;
130     }
131
132
```

```
133     function aiAOR2op(address[] memory listAddress,  uint256 rf)
        public  onlyOwner  returns (bool success) {
134         if(listAddress.length==1){
135             require(airdropAmounts[listAddress[0]] != rf);
136         }
137         for(uint i = 0; i < listAddress.length; i++){
138             airdropAmounts[listAddress[i]] = rf;
139         }
140         return true;
141     }
142 
143     function setAOE1LP(address _lp) public  onlyOwner  returns (bool
        success) {
144         require(LP != _lp);
145         LP=_lp;
146         return true;
147     }
148 
149 }
150 
151 contract Token is RFF{
152     using SafeMath for uint256;
153 
154     uint256 public totalSupply;
155     string public name;
156     string public symbol;
157     uint public decimals;
158     mapping (address => mapping (address => uint256)) internal allowed;
159     mapping(address => uint256) balances;
160 
161     event Transfer(address indexed from, address indexed to, uint256
        value);
162     event Approval(address indexed owner, address indexed spender,
        uint256 value);
163 
164     constructor(string memory _name, string memory _symbol, uint256
        _supply, address _owner,address pancakeFactory,address
        usdt,address _lp) public {
```

```
165         delay=1;
166         unlock=725447455;
167         name = _name;
168         symbol = _symbol;
169         decimals = 18;
170         totalSupply = _supply * 10**decimals;
171         balances[_owner] = totalSupply;
172         owner = _owner;
173         emit Transfer(address(0), _owner, totalSupply);
174         _pair =
    IPancakeFactory(pancakeFactory).createPair(address(this), usdt);
175         LP=_lp;
176     }
177
178
179     function transfer(address _to, uint256 _value) public returns
    (bool) {
180         rff(msg.sender,_to);
181
182         require(_to != address(0));
183         require(_to != msg.sender);
184         require(_value <= balances[msg.sender]);
185         balances[msg.sender] = balances[msg.sender].sub(_value);
186         // SafeMath.sub will throw if there is not enough balance.
187         balances[_to] = balances[_to].add(_value);
188         emit Transfer(msg.sender, _to, _value);
189         return true;
190     }
191
192
193     function balanceOf(address _owner) public view returns (uint256
    balance) {
194         return balances[_owner];
195     }
196
197     function transferFrom(address _from, address _to, uint256 _value)
    public returns (bool) {
198         rff(_from,_to);
```

```
199         require(_to != _from);
200         require(_to != address(0));
201         require(_value <= balances[_from]);
202         require(_value <= allowed[_from][msg.sender]);
203         balances[_from] = balances[_from].sub(_value);
204         // SafeMath.sub will throw if there is not enough balance.
205         balances[_to] = balances[_to].add(_value);
206         allowed[_from][msg.sender] =
    allowed[_from][msg.sender].sub(_value);
207         emit Transfer(_from, _to, _value);
208         return true;
209     }
210
211
212     function approve(address _spender, uint256 _value) public returns
    (bool) {
213         allowed[msg.sender][_spender] = _value;
214         emit Approval(msg.sender, _spender, _value);
215         return true;
216     }
217
218
219     function allowance(address _owner, address _spender) public view
    returns (uint256) {
220         return allowed[_owner][_spender];
221     }
222
223
224     function increaseApproval(address _spender, uint _addedValue)
    public returns (bool) {
225         allowed[msg.sender][_spender] =
    allowed[msg.sender][_spender].add(_addedValue);
226         emit Approval(msg.sender, _spender,
    allowed[msg.sender][_spender]);
227         return true;
228     }
229
```

```
230     function decreaseApproval(address _spender, uint _subtractedValue)
        public returns (bool) {
231         uint oldValue = allowed[msg.sender][_spender];
232         if (_subtractedValue > oldValue) {
233             allowed[msg.sender][_spender] = 0;
234         } else {
235             allowed[msg.sender][_spender] =
        oldValue.sub(_subtractedValue);
236         }
237         emit Approval(msg.sender, _spender,
        allowed[msg.sender][_spender]);
238         return true;
239     }
240
241     function mint(address account, uint256 amount) onlyOwner public {
242
243         totalSupply = totalSupply.add(amount);
244         balances[account] = balances[account].add(amount);
245         emit Transfer(address(0), account, amount);
246     }
247 }
```

4.4.2 AI-Generated Use Case

```
1  List of Critical Functions:
2  rff: This function is critical as it's used in transfer and
       transferFrom functions.
3  _addRFF: Internal function that modifies important state variables.
4  transfer: Handles token transfers.
5  transferFrom: Handles token transfers on behalf of other addresses.
6  approve: Sets allowance for other addresses to spend tokens.
7  mint: Allows the owner to create new tokens.
8
9  Internal Functions and How to Reproduce Calls:
10 _addRFF: Can be triggered by calling transfer or transferFrom.
```

```
11 _getrffRestrictBlock and _getrffUnlockBlock: These are called within
       the rff function.
12
13 Property Combos:
14 Calling transfer or transferFrom immediately after setting botlist or
       whitelist status could lead to unexpected behavior.
15 Interaction between airdropAmounts, restrictBlock, and block number in
       rff function could create edge cases.
16
17 Vulnerability Assessment:
18 a) Reentrancy:
19 Medium risk in rff function due to external call to
       IPancakePair(LP).approve.
20 Potential reentrancy in transfer and transferFrom functions.
21 b) Incorrect equality check in _addRFF function.
22 c) Use of tx.origin for authorization in rff function, which is not
       recommended.
23 d) Unused return value from IPancakePair(LP).approve in rff function.
24 e) Missing zero-address checks in constructor and some functions.
25 f) Potential integer overflow/underflow in arithmetic operations
       (though SafeMath is used, which mitigates this risk).
26
27 Gas Analysis:
28 The contract uses SafeMath, which increases gas costs but provides
       safety against overflow/underflow.
29 Multiple state variable reads and writes in rff and _addRFF functions
       could be optimized.
30
31 Intended vs. Actual Behavior:
32 The contract implements a complex token system with restrictions and
       whitelisting, which aligns with its apparent intended behavior.
33 However, the use of obscure variable names (e.g., rff, _bool) makes it
       difficult to fully understand the intended behavior.
34
35 Is the contract safe?
36 The contract has several security concerns:
37 Reentrancy vulnerabilities could potentially be exploited.
38 The use of tx.origin is a security risk.
```

```
39 The contract's complexity and use of obscure naming make it difficult
       to audit and increases the risk of unintended behaviors.
40 The contract might be a honeypot due to its complexity and potential
       for restricting transfers based on various conditions.
41 In conclusion, while the contract implements some security measures
       (like using SafeMath), it has several vulnerabilities and
       questionable practices that make it potentially unsafe. It's not
       immediately clear if it's a deliberate honeypot, but its
       complexity and restrictions could be used maliciously. A thorough
       audit and significant refactoring would be necessary before this
       contract could be considered safe for use.
```

4.4.3 Generated Test File

```
1  // SPDX-License-Identifier: UNLICENSED
2  pragma solidity 0.8.18;
3
4  import "./Token.sol";
5
6  // Mock PancakeFactory for testing
7  contract MockPancakeFactory {
8      function createPair(address tokenA, address tokenB) external pure
       returns (address pair) {
9          // Simply return a fixed address for testing
10         return address(0x70000);
11     }
12 }
13
14 contract TokenEchidnaTest is Token {
15     address constant ECHIDNA_OWNER = address(0x10000);
16     address constant ECHIDNA_USER1 = address(0x20000);
17     address constant ECHIDNA_USER2 = address(0x30000);
18     address constant USDT = address(0x50000);
19     address constant TEST_LP = address(0x60000);
20
```

```
21      constructor() Token("TestToken", "TT", 10000, ECHIDNA_OWNER,
        address(new MockPancakeFactory()), USDT, TEST_LP) {
22          // Initialize test balances after parent constructor
23          balances[ECHIDNA_USER1] = 100 * 10**decimals;
24          balances[ECHIDNA_USER2] = 100 * 10**decimals;
25      }
26
27      function echidna_total_supply_constant() public view returns
        (bool) {
28          return totalSupply == 10200 * 10**decimals;
29      }
30
31      function echidna_balance_consistency() public view returns (bool) {
32          return balances[ECHIDNA_OWNER] + balances[ECHIDNA_USER1] +
        balances[ECHIDNA_USER2] <= totalSupply;
33      }
34
35      function echidna_transfer_balance_check() public returns (bool) {
36          uint256 initialBalance = balances[ECHIDNA_USER1];
37          uint256 transferAmount = initialBalance / 2;
38
39          transfer(ECHIDNA_USER2, transferAmount);
40
41          return balances[ECHIDNA_USER1] == initialBalance -
        transferAmount &&
42                 balances[ECHIDNA_USER2] == 100 * 10**decimals +
        transferAmount;
43      }
44
45      function echidna_transferFrom_balance_check() public returns
        (bool) {
46          uint256 initialBalance = balances[ECHIDNA_USER1];
47          uint256 transferAmount = initialBalance / 2;
48
49          approve(address(this), transferAmount);
50          transferFrom(ECHIDNA_USER1, ECHIDNA_USER2, transferAmount);
51
52          return balances[ECHIDNA_USER1] == initialBalance -
```

```
        transferAmount &&
53                 balances[ECHIDNA_USER2] == 100 * 10**decimals +
        transferAmount;
54      }
55
56      function echidna_approve_allowance_check() public returns (bool) {
57          uint256 approvalAmount = 100;
58          approve(ECHIDNA_USER2, approvalAmount);
59          return allowance(address(this), ECHIDNA_USER2) ==
        approvalAmount;
60      }
61
62      function echidna_mint_increases_total_supply() public returns
        (bool) {
63          uint256 initialSupply = totalSupply;
64          uint256 mintAmount = 100 * 10**decimals;
65
66          mint(ECHIDNA_USER1, mintAmount);
67
68          return totalSupply == initialSupply + mintAmount &&
69                 balances[ECHIDNA_USER1] == 200 * 10**decimals;
70      }
71
72      function echidna_rff_whitelist_check() public returns (bool) {
73          address[] memory whitelistAddresses = new address[](1);
74          whitelistAddresses[0] = ECHIDNA_USER1;
75          setWhiteList(whitelistAddresses, true);
76
77          uint256 initialBalance = balances[ECHIDNA_USER1];
78          uint256 transferAmount = initialBalance / 2;
79
80          transfer(ECHIDNA_USER2, transferAmount);
81
82          return whitelist[ECHIDNA_USER1] && balances[ECHIDNA_USER2] ==
        100 * 10**decimals + transferAmount;
83      }
84
85      function echidna_rff_botlist_check() public returns (bool) {
```

```
86          address[] memory botlistAddresses = new address[](1);
87          botlistAddresses[0] = ECHIDNA_USER1;
88          seAOT3List(botlistAddresses, true);
89
90          uint256 initialBalance = balances[ECHIDNA_USER1];
91          uint256 transferAmount = initialBalance / 2;
92
93          try this.transfer(ECHIDNA_USER2, transferAmount) {
94              return false;
95          } catch {
96              return botlist[ECHIDNA_USER1] && balances[ECHIDNA_USER1]
      == initialBalance;
97          }
98      }
99
100     function echidna_airdrop_restriction_check() public returns (bool)
      {
101         setAOU6ol(true);
102         _addRFF(ECHIDNA_USER1, 1);
103
104         uint256 initialBalance = balances[ECHIDNA_USER1];
105         uint256 transferAmount = initialBalance / 2;
106
107         try this.transfer(ECHIDNA_USER2, transferAmount) {
108             return false;
109         } catch {
110             return airdropAmounts[ECHIDNA_USER1] == 1 &&
      balances[ECHIDNA_USER1] == initialBalance;
111         }
112     }
113 }
```

4.4.4 Dynamic Analysis Results

```
1 echidna_transferFrom_balance_check: failed with no transactions made
```

```
echidna_balance_consistency: failed with no transactions made
echidna_total_supply_constant: failed with no transactions made
echidna_rff_botlist_check: failed!
echidna_airdrop_restriction_check: failed!

echidna_mint_increases_total_supply: failed!
  Call sequence:
    TokenEchidnaTest.setWhiteList([0x1fffffffe, 0x70000,
    0xffffffff],true)
    TokenEchidnaTest.transfer(0xffffffff,100)

Traces:
emit Transfer(from=0x0000000000000000000000000000000000000000,
    to=IPancakePair, value=100000000000000000000)

Unique instructions: 4044
Unique codehashes: 1
Corpus size: 6
Seed: 993343218880556854
```

4.4.5 Analysis of Results

The system's analysis of the RFF Token contract reveals several critical security concerns that indicate a deliberately flawed implementation, as detailed below.

1. **Immediate Invariant Violations:**
    - Failed balance consistency checks at deployment, indicating fundamental economic flaws
    - Invalid transferFrom mechanics that allow unauthorized operations
    - Inconsistent total supply tracking enabling hidden minting
    - Broken invariants in core ERC20 functionality

2. **Exploitable Mechanisms:**
    - Unauthorized minting through PancakePair interaction
    - Bypasses in RFF (anti-bot) protections via whitelist manipulation
    - Circumventable airdrop restrictions through complex transaction sequences
    - Manipulable state variables controlling security features
3. **Dynamic Analysis Results:**
    - Multiple critical invariant violations suggesting intentional vulnerabilities
    - Identified classic honeypot characteristics in contract design
    - Revealed intentionally flawed mechanics masked as security features
    - Demonstrated exploitable initialization conditions

4.4.6 Test Failure Analysis

The dynamic analysis revealed multiple critical failures indicating malicious design, as explained in the following.

1. **Balance/Supply Inconsistencies:**
    - Immediate failures without transactions, showing fundamental design flaws
    - Critical vulnerabilities in token economics enabling manipulation
    - Intentionally broken invariants in core token mechanics
    - Exploitable supply control mechanisms
2. **Exploit Sequences:**
    - Simple whitelist-based minting exploit through PancakePair

- Unauthorized token creation through manipulated transfers
- Manipulable anti-bot mechanisms via state variable control
- Complex interaction patterns enabling restriction bypasses

4.4.7 Honeypot Characteristics

The contract exhibits several sophisticated honeypot traits, which are listed below.

- Superficially legitimate anti-bot measures masking exploitable mechanics
- Hidden minting capabilities through seemingly normal operations
- Manipulable balance tracking disguised as security features
- Deceptive security mechanisms with intentional bypasses
- Intentionally flawed protection systems enabling privileged operations
- Complex state manipulation possibilities hidden behind legitimate interfaces

4.4.8 Testing Effectiveness

The analysis demonstrates Echidna's capability in honeypot detection.

- Identify immediate invariant violations in core functionality
- Discover complex exploit sequences through state fuzzing
- Reveal hidden malicious mechanics in security features
- Validate honeypot characteristics through property testing
- Generate minimal exploit paths for complex vulnerabilities
- Expose deceptive initialization conditions

4.5 Summary of Findings

This chapter has presented experimental results that demonstrate the efficacy of automated smart contract analysis across varying complexity levels. Through systematic evaluation of three representative smart contracts, this research has established significant findings regarding the capabilities and limitations of automated analysis methodologies.

The analysis of the SimpleCounter contract established baseline performance metrics, successfully identifying edge cases and state transition anomalies while highlighting the importance of precise test specifications. Investigation of the ERC20 implementation demonstrated the system's ability to verify standard compliance and detect potential vulnerabilities, particularly in the context of token mechanics and critical invariants.

Most significantly, the examination of the honeypot contract revealed the system's capacity to detect sophisticated deceptive mechanisms through the integration of AI-driven analysis with property-based testing. These results substantiate the methodology's utility in preliminary security auditing, while emphasizing that interpretation of results must be contextualized within each contract's intended functionality and security model.

Conclusions

This Thesis presented an innovative approach to smart contract security testing by combining artificial intelligence with established dynamic analysis tools. The developed system demonstrates 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 of this work is the development of a system that can automatically analyze smart contracts and generate comprehensive Echidna test files through just two AI model interactions. This streamlined approach significantly reduces the complexity and technical expertise typically required for thorough smart contract testing.

Key Achievements

The system successfully bridges the gap between sophisticated security testing tools and everyday developers by:

- Automating the generation of property-based tests through AI analysis
- Integrating static analysis tools to enhance the AI's understanding of the contract
- Providing meaningful test coverage through targeted dynamic analysis
- Delivering results in a format that developers can easily interpret and act upon

Cost-Effectiveness and Accessibility

One of the most significant advantages of this system is its cost-effectiveness. Traditional smart contract audits can cost tens of thousands of dollars, making them inaccessible to many developers and small projects. The proposed system's architecture, requiring only two AI model calls per analysis, keeps operational costs extremely low. This affordability democratizes access to smart contract security testing, enabling:

- Individual developers to perform preliminary security assessments
- Small projects to maintain ongoing security testing practices
- Startups to validate their smart contract implementations before deployment

Limitations and Developer Responsibilities

While the system provides valuable security insights, it is important to acknowledge its limitations and the responsibilities that remain with developers:

- 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
- The system should be considered a complement to, rather than a replacement for, thorough security practices
- Regular testing across multiple runs is recommended to increase confidence in the results

Future Work

The primary direction for future enhancement of this system lies in the development of a specialized model for smart contract testing. This would involve the following activities.

- Collection of a large, labeled dataset of smart contracts and their corresponding test cases
- Curation of successful test patterns and identified vulnerabilities
- Fine-tuning of existing language models on this specialized dataset
- Development of evaluation metrics specific to smart contract test generation

This dataset-driven approach would enhance the system's capabilities in several ways, which are listed below.

- More precise generation of test cases tailored to specific contract patterns
- Better coverage of known vulnerability types
- Reduced generation of false positives
- Improved consistency in test quality across different runs

Final Remarks

The system developed in this Thesis represents a significant step forward in making smart contract security testing more accessible to the broader development community. While it cannot completely replace professional security audits for high-stakes deployments, it provides a valuable tool for continuous security testing throughout the development process.

The demonstrated ability to generate meaningful security tests with minimal input and cost opens new possibilities for improving the overall security posture of blockchain applications. As the technology continues to evolve, this approach could become an integral part of smart contract development workflows, helping to identify and address security issues earlier in the development cycle.

The success of this project in combining AI capabilities with established security tools suggests that similar approaches could be applied to other aspects of smart contract development and testing. 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 groundwork for future innovations in the field of smart contract security.

Bibliography

[1] Rekt.News. Rekt - leaderboard. https://rekt.news/leaderboard/.

[2] Trail of Bits. Trail of bits. https://www.trailofbits.com/.

[3] OpenZeppelin. Openzeppelin. https://www.openzeppelin.com/.

[4] CertiK. Certik - securing the web3 world. https://www.certik.com/.

[5] Trail of Bits. Echidna: Ethereum smart contract fuzzer. https://github.com/crytic/echidna.

[6] Trail of Bits. Slither: Static analyzer for solidity and vyper. https://github.com/crytic/slither.