# SmartScan: A Finite State Machine and CTL-Based Formal Verification Framework for Enhanced Security in Smart Contracts

G. Sowmya*, R. Sridevi
JNTUH, Department of CSE, Hyderabad, India.
E-mail: gonurusowmya@gmail.com, sridevirangu@jntuh.ac.in
*Corresponding author

**Keywords:** smart contracts, formal verification, finite state machine, vulnerability detection, blockchain security

**Received:** March 13, 2025

*Smart contracts are self-executing programs deployed on blockchain platforms that facilitate automated and decentralized transactions. However, once deployed, they become immutable, making them vulnerable to catastrophic exploits, such as reentrancy, access control misconfiguration, integer overflow, and front-running. The need for proof and verification is urgent, as evidenced by other high-profile, capital-draining incidents, such as the DAO attack and Parity wallet vulnerabilities. Abstract: We present ContractFuzzer, a systematic fuzzer for detecting vulnerabilities in Ethereum smart contracts. Existing tools are based on static analysis, symbolic execution, or heuristic detection, and thus typically impose high false positives, low completeness, and limited formal verification. In this paper, we introduce SmartScan, a formal verification framework that systematically checks smart contract security by integrating FSM modeling and CTL-based model checking in nuXmv. Our methodology performs automatic parsing of Solidity code, automated generation of FSM and BIP models, conversion to the SMV format, and verification of CTL security properties. It responds to detected violations with automated counterexample generation to assist in debugging and iterative re-verification. For validation, SmartScan will be tested on 10 different types of Solidity contracts that address 14 critical vulnerabilities. Our experimental results show 95.4% detection accuracy, 3.2% false positive rate, and 2.8% false negative rate, with 100% verification coverage, and average verification time of 3–7 seconds for each property, outperforming state-of-the-art tools in both coverage and precision. SmartScan: SmartScan has a wide-ranging practical utility in discovering and diagnosing vulnerabilities such as reentrancy and access control issues, which it has been applied in, such as in a case study of a DeFi Lending contract. SmartScan provides a scalable, precise, and developer-centric approach to improve the confidence and reliability of blockchain applications by combining exhaustive formal verification of smart contracts with automated counterexample generation.*

*Povzetek: Članek predstavlja SmartScan kot učinkovit pristop za avtomatsko in zanesljivo preverjanje varnosti pametnih pogodb na verigi blokov.*

## 1 Introduction

One of the achievements in blockchain applications is the introduction of smart contracts, self-executing contracts without a third party and with an existing agreement directly written into code. However, they are at risk from security challenges if there are any issues in the contract logic, as they cannot be changed. The DAO attack, Parity wallet vulnerabilities, and other incidents involving millions of dollars in losses exposed the need for more secure verification mechanisms. Mythril, Oyente, Slither, and Zeus are available tools for smart contract analysis using static analysis, symbolic execution, and heuristic detection, which usually suffer from high false positives, inability to provide the gold standard for completeness, and lack of support for formal verification [1], [2]. Such limitations of existing tools create a need for a unified verification framework to verify smart contracts systematically using formal methods.

Prior work on smart contract security mainly tackles vulnerability detection using symbolic execution [3], [4]. Natural symbolic execution-based tools exist for traversing the execution paths of contracts, but do not scale for larger contracts due to path explosion. Other approaches use static analysis [5], which is more efficient, but the vulnerability at runtime is commonly ignored. It detects challenges for novel attack patterns, but basic methods can provide faster vulnerability detection. While a more robust solution for smart contract correctness and security can be accomplished with the help of formal verification (such as mathematical proofs and model checking), it is usually labor-intensive and costly. On the other hand, existing formal verification frameworks are

inadequate in scalability or only support a few contract languages, which limits their application in practice [6].

To mitigate these limitations, this paper proposes SmartScan. This new FSM-based formal verification framework leverages symbolic execution to automatically explore all feasible execution paths (or code paths) of a smart contract. In contrast, SmartScan uses CTL properties and a specific model-checking framework, allowing a comprehensive vulnerability analysis while reducing false positives compared to existing tools. Automatic counterexample generation improves security by showing evidence of security violations so developers can debug and revise their contracts. SmartScan is evaluated on a broad set of smart contracts for detecting 14 common but difficult-to-find critical vulnerabilities and achieves better detection accuracy than state-of-the-art tools.

We propose a combined approach of FSM (finite state machine) based modeling embedded with formal verification type: CTL properties for formal validation covering structured flow security at a systemic level and hierarchical decomposition level, generating an automated counter-example. The novel contributions of this research consist of the design and execution of SmartScan, a comparison with state-of-the-art tools, and a case study on the DeFiLending contract, illustrating the in-practice effectiveness of SmartScan.

The paper is organized in the following way. In Section 2, we present a literature review that describes existing security analysis tools and their limitations. We start in Section 3 with an introduction to SmartScan, including our earlier work on its development and theoretical basis. The following details other parts of the proposed process: Section 4 describes our method, including FSM modeling, by first converting the BIP model to FSM and, finally, walking through the nuXmv tool to achieve the verification. A detailed study of a DeFiLending smart contract from Section 5 illustrates how it identifies and mitigates security vulnerabilities. Section 6 discusses experimental results comparing SmartScan with existing tools and its vulnerability detection performance. In Section 7, we report our research, briefly discuss the benefits of SmartScan compared to state-of-the-art tools, and present certain limitations of the work. Section 8 wraps up the research and discusses future works, such as adding more languages and automatic patching mechanisms to fix (the most common) vulnerabilities.

# 2 Related work

Smart contracts are at the foundation of blockchain ecosystems and enable decentralized applications in various industries, including financial, medical, and supply chain [3] [4]. Singh et al. Formalization is essential for vulnerability mitigation [1], and strong security verification techniques are needed for reducing risk [3]. However, recent advancements like VerX [Permenev et al. 2023], VeriSmart [2], by So et al. However, even more sophisticated automated tools that have streamlined safety verification (for example, [4]) are still restricted in scaling to more complicated systems. In that work, tools such as Slither [7] (analyzed by Feist et al.) and Gigahorse 8 focus on static and symbolic execution but are limited in their ability to work with complex state transitions.

Zheng et al. Overviews of smart contract tools have been provided in various works, e.g., [9] and di Angelo and Salzer [6], which focus on constraints specifically in scalable and security vulnerability detection concerning DeFi applications; furthermore, Wang et al. Economic security analysis has also been recognized for further attention; [38] surveyed research on Ethereum smart contracts, highlighting this value. Babel et al., Clockwork Finance, Economic vulnerabilities in smart contracts were covered by [13] but were not integrated into an overall formal method. Some other tools, like Wang et al.'s ContractWard. However, Huang et al. pointed out that even [17], which integrated automated vulnerability detection models, still has some lifecycle complexities to manage. [20].

Table 1: Summary of essential literature findings

| Reference | Proposed Tool or Methodology | Merits | Research Gaps |
|---|---|---|---|
| [13] Kushal et al. (2023) | Clockwork Finance: Automated analysis of economic security in smart contracts | Focuses on economic vulnerabilities in DeFi contracts; scalable for complex financial logic | Limited integration with formal verification methods; lacks lifecycle management for contracts |
| [14] Kushal et al. (2023) | Decentralized Model using Layer-2 Polygon Blockchain for digital evidence | Preserves evidence integrity using blockchain; robust for legal and audit trails | Not generalized for DeFi or other blockchain applications; lacks formalized security proofs |
| [17] Wang et al. (2020) | ContractWard: Automated vulnerability detection models for Ethereum contracts | Uses machine learning for vulnerability detection; efficient for standard security issues | Limited scalability for complex contracts; no focus on economic security |
| [19] Vacca et al. (2021) | Systematic literature review of blockchain innovative contract development | Comprehensive review of techniques, tools, and challenges; identifies research opportunities | Lack of experimental implementation; focuses on static analysis |

| | | | |
|---|---|---|---|
| [20] Huang et al. (2019) | Smart Contract Lifecycle Perspective for Security Analysis | Addresses lifecycle challenges; integrates security assessments in development phases | Focuses on generic contracts; limited application for DeFi ecosystems |
| [29] Mehdi Sookhak et al. (2021) | Access control in healthcare using blockchain and smart contracts | Demonstrates secure data access in healthcare, robust against external attacks | Not designed for financial or high-complexity systems; lacks formal verification |
| [38] Wang et al. (2021) | Survey of Ethereum smart contract security | Identifies vulnerabilities and future research opportunities in Ethereum contracts | Lack of in-depth focus on real-world implementation challenges |
| [31] Seven et al. (2020) | Peer-to-Peer Energy Trading using smart contracts in virtual power plants | Showcases novel application of smart contracts in energy markets; promotes sustainability | Application-specific; lacks a generalized framework for other domains |
| [39] Lu Wang et al. (2021) | Agricultural Food Supply Chain Traceability with Blockchain Smart Contracts | Effective traceability and transparency in supply chain management | No formal verification for critical properties; limited to agricultural scenarios |
| [18] Shafaq et al. (2021) | Survey on blockchain smart contracts for future trends | Highlights applications and challenges; provides a roadmap for innovation | Lacks detailed integration with formal verification methods; no experimental validation |

Innovative applications of blockchains in healthcare [29], agriculture [39], and IoT [40] have demonstrated their versatility, as analyzed by authors such as Mehdi Sookhak et al. and Bhaskara Egala et al. However, Vacca et al. [19] and Shafaq et al. Our work is related in that [18] has found gaps in combining formal methods with existing real systems—semantic frameworks as proposed by Jiao et al. Gao et al. [23] and structural embedding techniques [5] offer improved accuracy at the cost of very high computational demand.

Newer research, such as that by Kushal et al. [14] and Seven et al. The paper in [31] suggests novel applications but does not concentrate on systematic validation. A summary of the findings of the literature is provided in Table 1. As great as these developments are, key gaps remain in the ability to scale, validate economic security, and manage lifecycle from a DeFi ecosystem perspective. We address these challenges using a hybrid approach that combines model checking, symbolic execution, and financial security analysis, culminating in our proposed tool, SmartScan. SmartScan addresses these research gaps by focusing on real-world applicability over an extensive set of detailed parameters, helping make blockchain applications more robust.

# 3 Smartscan: our prior work on formal verification of smart contracts

In our earlier work [41], we proposed the SmartScan framework, a holistic approach for formally verifying smart contracts that could improve blockchain applications' security and dependability. Due to the nature of the blockchain, where transactions cannot be reversed, it is critical to prove the correctness of smart contracts; it may cause a financial disaster or serious security problems· SmartScan was designed as a hybrid verification framework that combines formal methods, static analysis, and optimized heuristics, to detect vulnerabilities and inconsistencies in smart contracts systematically. It was created in response to weaknesses in existing verifiers that can be inefficient or impossible on large-scale smart contracts with elaborate dependencies. In contrast with conventional tools primarily focusing on symbolic execution or static analysis, SmartScan combines finite state machine (FSM) modeling, behavioral interaction modeling (BIP), and symbolic model checking by nuXmv to validate the structural correctness and security of smart contracts.

At its heart, SmartScan converts smart contract source code into Finite State Machine (FSM) models and checks security properties expressed in Computation Tree Logic (CTL) formulas. The same technique allows disciplined state-space exploration to catch important vulnerability categories, including reentrancy, integer overflow, access control misconfiguration, and timestamp dependency attacks. This framework expands formal verification from a single function-level analysis, an inclination of traditional verification methods, to a component-level analysis to dissect the interactions between the innovative contract components. In previous work, we presented SmartScan and applied it to detecting vulnerabilities in the DeFiLending case study, showing how SmartScan can identify vulnerabilities with significantly higher precision and lower false positives than traditional tools, such as Mythril, Oyente, and Slither. Experimental results focused on performance comparison have shown that, in addition to beating SmartCheck with high accuracy, SmartScan also generates counterexamples whenever possible to demonstrate violations of the security property, thus helping developers secure their contracts.

Smartscan is also unique in that it can scale and extend, which lends itself well to global-scale blockchain use cases. Whereas tools relying only on the existing vulnerability patterns are fixed and rigid, SmartScan dynamically builds the formal verification model. Therefore, SmartScan achieves much higher adaptability to emerging brilliant contract architecture. Such adaptability is essential in contemporary blockchain ecosystems — e.g., decentralized finance (DeFi) and innovative contract-based governance mechanisms — that necessitate higher levels of security refinement. SmartScan found that such computation, complexity, and verification time were significantly reduced, making it easier for developers to fold formal verification into the smart contract development lifecycle [41].

Recent works have also advanced the state of the art in smart contract verification—Jin et al. The work by [46] presents a model-checking-based method for dynamic-behavior verification of a sequence of smart contracts realized as a composite smart contract. Sorensen [47] has designed a more formal framework for reasoning about contract upgrades in ConCert using a contract morphism concept in Coq, which allows for a more formal, less resource-intensive verification across versions of a smart contract. These holistically prove the importance of formal FSM+CTL-based methodologies such as SmartScan, especially in the context of composite dynamics and upgradeability in practical blockchain systems.

Our previous work established the background for the current study, which builds upon SmartScan by presenting a usable implementation and evaluation framework, demonstrating its application to actual blockchain vulnerabilities. In this paper, we present SmartScan, a practical solution that is easy to implement in IDEs and build environments, and effective in identifying and preventing innovative contract security threats. This study extends the methodology developed for SmartScan to offer a straightforward, step-wise implementation methodology to provide developers with a systematic approach to using formal verification techniques on their smart contracts. By providing in-depth case studies and comparative charts of verification outcomes, the practical applicability of the approach is enhanced, and formal verification becomes much more tangible and actionable to those interested in the security of blockchain, both developers and security researchers.

# 4 Methodology

SmartScan formally verifies smart contracts using finite state machine (FSM) modeling and symbolic model checking. It follows a structured approach, which is discussed in Section 3. Then, BIP models of the smart contracts are transformed into SMV models to verify in nuXmv. It can evaluate computation tree logic (CTL) properties, find vulnerabilities, and generate counterexamples to improve contract security.

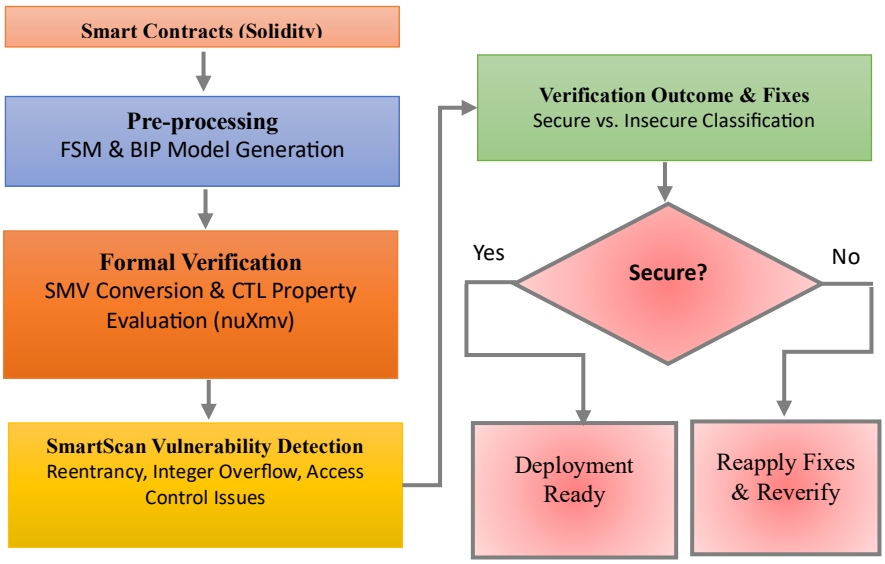


Figure 1: Methodology of smartscan for formal verification of smart contracts

As a concrete example of the SmartScan methodology, take the withdraw() function from the DeFiLending smart contract. SmartScan operates in two steps, with the first parsing the Solidity code to find the functional states and transitions. This means that, for example, when the tx contract state is "FundsDeposited" before a withdraw request is made. Whenever the corresponding conditions for a sufficient balance and no debt are satisfied, the transition FundsDeposited → FundsWithdrawn is generated. Program functionality can thus be represented as a state–transition relationship in the Finite State Machine (FSM) model, with nodes (Idle, FundsDeposited, FundsWithdrawn, etc.) and edges representing possible transitions, triggered by execution of functions corresponding to events in the contract. The second step automatically translates the FSM into the BIP model, maintaining the behavior, interaction, and priority rules (for instance, a repayment must occur before a

withdrawal). The BIP model is translated to the SMV model by making each FSM state a symbolic variable and each transition a guarded command. In nuXmv, security properties are formulated in Computation Tree Logic (CTL). So, the fund safety requirement is written as AG(deposits[msg. teller] >= 0, ensuring that no execution path has a negative balance. The execution of this property check by nuXmv is equivalent to an exhaustive exploration of the FSM-defined state space. In the case of a property not holding, nuXmv generates a counterexample, which is an exact sequence of transitions that lead to the violation of a property—e.g., a reentrant call sequence that circumvented the state updates of the system. This concrete mapping between source code→FSM→Verification→CTL exemplifies how the systematic translation of smart contract logic into a formal, verifiable model enables precise detection of vulnerabilities, with developer-guided remediation using SmartScan.

The approach adopted by SmartScan for the formal verification of smart contracts is illustrated in Figure 1. It starts by inputting smart contracts, generally created in Solidity, and then proceeds with a formal verification flow. The contracts are then preprocessed to model them with finite state machines (FSM) that are transformed into a BIP model representation to be analyzed in-depth. This way, we can first perform a systematic behavioral analysis of the contract execution flow, making sure that, before verification, all the possible states and transitions are covered.

After generating the FSM and BIP models, the subsequent step consists of formal verification using symbolic model checking. Translation of the contract to an SMV format that the nuXmv model checker accepts. SmartScan then checks several security properties, namely safety of funds, resistance to reentrancy, integer overflow, and enforcement of access control using computation tree logic (CTL). It systematically traces all possible execution paths of a contract, thereby verifying that there are no vulnerabilities in the logical framework of the contract. In case a contract violates any security property, SmartScan returns a counterexample — an execution trace that shows how the exploit can be triggered.

After detecting potential vulnerabilities, the verification result will be examined to determine whether the contract is secure or insecure. If any security issues are found, developers are given information about the cause(s) and directed to make the necessary fixes. Once all the required changes are made, the contract is verified again to ensure the vulnerabilities have been addressed. This scheme guarantees that contracts are analyzed and improved through different verification cycles until they reach security standards.

One unique aspect of SmartScan is that it combines FSM modeling and formal verification techniques rather than static analysis tools based only on pre-defined heuristics. In contrast to symbolic execution methods vulnerable to path explosion, SmartScan systematically models all contract execution states, allowing it to verify complex contracts efficiently. This makes the approach highly scalable, enabling SmartScan to handle various agreements with different structures and functionalities while ensuring high detection accuracy.

Our previous work [41] on the formal modeling methods provides a more thorough explanation of the SmartScan framework, including details of its FSM-based construction and CTL-based verification. This study expands on that framework by applying it to real-life case studies to demonstrate the practical use of SmartScan to find vulnerabilities in real-world contracts. The SmartScan methodology builds on existing security verification approaches by introducing iterative verification with the automatic generation of adversarial counterexamples to make security assessments more robust.

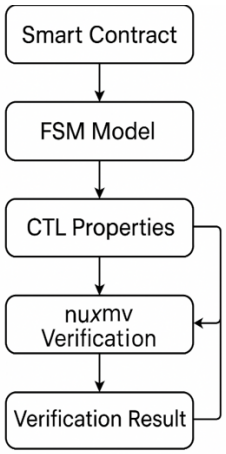


Figure 2: Visual workflow of the smartscan contract-to-verification lifecycle

The SmartScan contract-to-verification lifecycle visual workflow is shown in Figure 2. The workflow starts with parsing the smart contract input, which is then modeled into a finite state machine or FSM. We introduce some relevant computation tree logic (CTL) properties to express security requirements that we consider essential. We check the FSM and CTL specifications using the nuXmv model checker through complete verification. For every property, it indicates whether it is satisfied, and in case of any violations, it provides a counterexample. This allows developers to iteratively verify their contract until all properties are satisfied, providing a systematic and complete security verification before ever deploying it.

**Algorithm 1:** SmartScan Verification Pipeline
Input: Solidity contract $C$, CTL property set $\Phi$, toolchain config $\theta$
Output: Verification report $R$

1. Parse and normalize $C$
2. Build intermediate representation $IR$
3. Preprocess $IR$
4. Extract state variables $\Sigma$ and functions $F$
5. Initialize $FSM$ with Idle state and $\Sigma$
6. For each $f$ in $F$:
   a. Derive guards and updates
   b. Add transitions to $FSM$
7. Convert $FSM$ to $BIP$ model with behaviors, interactions, priorities
8. Convert $BIP$ to $SMV$ model with states, variables, transitions
9. For each $\varphi$ in $\Phi$:
   a. Run nuXmv check
   b. If fail: store counterexample in $R$
   c. Else: mark pass in $R$
10. If any fail:
    a. Generate patch guidance from counterexample
    b. Apply patch and repeat from step 1
11. Return $R$

The SmartScan verification process from parsing the Solidity contract to an intermediate representation (IR) and then uses the verification engine is shown in Algorithm 1. We extract state variables and contract functions to create a finite state machine (FSM), in which states represent the condition of the contract and transitions represent the state changes (as a respective function is triggered). The first step involves the FSM being converted to a BIP model that encapsulates behaviors, interactions, and bandwidth execution priorities, and subsequently an SMV model for nuXmv verification. We verify each of the CTL properties in a given set; if a property fails, we provide a counterexample that illustrates the precise path through execution where the property is violated. The algorithm issues focused remediation advice, applies fixes and re-runs verification, iteratively, until all properties pass or no vulnerabilities are found. All of them are verified in the end as a report where each property is listed along with a pass/fail status and a counterexample if applicable as a requirement for a systematic and reproducible validation of smart contract security.

# 5 Case study: formal verification of the defending smart contract

This paper presents a systematic way in which SmartScan is used to formally verify the DeFiLending smart contract (in Appendix A), emphasizing security and correctness. It starts by listing high-level verification goals and identifying important contract attributes. For example, these properties are fund safety, reentrancy-safe, interest calculation, liquidation accuracy, etc. For example, the property on which funds safety is enforced is formalized with the CTL formula AG (balance >= 0), which means that in every reachable state, the balance in the contract is always non-negative. Reentrancy resistance is also formalized in a similar way AG (¬(call → AF ¬call)), which states that after a critical function completes some updates to the state, the function must not be called again until a state is reached where the behavior is not to allow entry.

The contract is broken down into its BIP (Behavior, Interaction, Priority) framework for systematic analysis. We model key contract functions such as deposit (), withdraw(), and borrow() as behaviors and their dependencies, such as needing sufficient collateral before borrowing, as interactions. It also defines execution priorities, requiring that the state updates be completed before any of them can proceed. This decompositional translation turns the contract into a finite-state system that can be formally analyzed.

The BIP representation is subsequently translated into SMV (Symbolic Model Verification) format, which consists of states and transitions. States correspond to constructs within the contract (for instance, FundsDeposited or LoanActive), and transitions describe the user's actions and the state they lead to. A CTL formula expresses the verification goal, which is then captured in SmartScan's backend and analyzed automatically by the nuXmv model checker.

SmartScan explores the contract's state space to validate the specified properties and traverse all transitions. When a property is violated, SmartScan creates a counterexample explaining the path to the violation. An example of a counterexample would be a reentrant sequence, which, when found, might cause the contract to change state to include something like a mutex or reset state to protect the vulnerability.

After resolving the issues, the contract was verified against all properties again. SmartScan uses FSM decomposition and heuristic pruning to improve the efficiency of this process and overcome the temporal and computational complexity of FSMs. The output results (verification times, counterexamples, compliance rates, etc.) are documented to assess the tool's performance. The paper presents this structured methodology and demonstrates how SmartScan enables holistic analysis and security improvement of smart contracts, such as the real-world application DeFiLending.

**Step 1: BIP model construction**

The contract's functions were decomposed into the BIP framework, capturing their behaviors, interactions, and priorities. For instance:

- **Behaviors**: The borrow() function was modeled as a state transition from FundsDeposited to LoanActive.
- **Interactions**: Dependencies, such as collateral checks before borrowing, were encoded as conditional transitions.
- **Priorities**: Execution precedence ensured that state updates occurred before subsequent calls, mitigating risks of reentrant attacks.

**Step 2: SMV conversion and CTL encoding**

The BIP model was converted into SMV format, representing states like FundsDeposited, LoanActive, and Liquidated. Transitions and guards were explicitly defined, and CTL properties were encoded to verify the goals. For example, the CTL formula AG (balance >= 0) was used to confirm fund safety across all possible states.

**Step 3: Verification with nuXmv**

Using SmartScan's integration with nuXmv, the state space of the DeFiLending contract was exhaustively explored. The model checker systematically evaluated each CTL property:

- The property AG (balance >= 0) was satisfied, confirming that the contract never allowed negative balances.
- A counterexample was generated for the reentrancy resistance property, revealing a scenario where reentrant calls could bypass state updates. This provided actionable insights for improving the contract.

**Step 4: Contract revision and re-verification**

The identified reentrancy vulnerability was resolved by introducing a mutex mechanism and conditional state checks within critical functions. After revising the contract, it was re-verified using SmartScan. The updated version satisfied all CTL properties, demonstrating the effectiveness of the applied fixes.

**Step 5: Performance metrics and scalability**

SmartScan's application yielded significant performance metrics:

- Verification time per property ranged from 3 to 7 seconds.
- The tool's scalability was evident as it managed complex interdependencies in the DeFiLending contract without state explosion issues.
- Counterexamples provided precise debugging paths, reducing development iteration time.

This practical illustration highlights SmartScan's capability to detect and resolve vulnerabilities systematically, ensuring the reliability and security of DeFi applications. By combining BIP modeling, CTL-based verification, and nuXmv's symbolic model checking, SmartScan provides a robust framework for addressing the challenges of formal verification in blockchain environments.

**1. Understanding the contract**

The **DeFiLending** contract implements basic lending functionality with collateral requirements. Key aspects include:

- **Deposit function**: Allows users to add funds to the contract.
- **Withdraw function**: Enables users to withdraw funds if no outstanding debts exist.
- **Borrow function**: Lets users borrow funds against their collateral.
- **Repay function**: Allows users to repay outstanding debts.

Potential vulnerabilities include:

- **Reentrancy attacks**: The withdraw and borrow functions involve transferring Ether, which can trigger external calls.
- **Collateral mismanagement**: Ensuring sufficient collateral is maintained when borrowing funds.
- **Fund Safety**: Ensuring the contract does not allow balances to drop below zero.

**2. Verification goals and CTL properties**

We define the following verification properties for this contract:

1. **Fund safety**: AG (deposits[msg.sender] >= 0) ensures no negative balances.
2. **Reentrancy resistance**: AG (¬(withdraw → AF withdraw)) prevents multiple concurrent withdrawals.

3. **Collateral management**: AG (deposits[msg.sender] >= debts[msg.sender] * collateralFactor / 100) ensures sufficient collateral.
4. **Debt clearance**: AG (debts[msg.sender] == 0 → AF (withdraw allowed)) ensures debt repayment before withdrawal.

**3. Modeling the contract in BIP**

The contract's functions are translated into the BIP model:

- **Behaviors**:
  - deposit(): Transitions from Idle to FundsDeposited.
  - withdraw(): Checks balances and debts, transitions to FundsWithdrawn.
  - borrow(): Checks collateral, transitions to LoanIssued.
  - repay(): Updates debts, transitions to DebtCleared.
- **Interactions**: Define transition conditions, such as collateral checks in borrowing.
- **Priorities**: Ensure repayment (repay()) has precedence over withdrawal (withdraw).

**4. Conversion to SMV and CTL encoding**

The BIP model is converted into an SMV format for nuXmv:

- **States**:
  - Idle: No activity.
  - FundsDeposited: After deposit().
  - FundsWithdrawn: After withdraw().
  - LoanIssued: After borrow().
  - DebtCleared: After repay().
- **CTL Properties**: Encoded as logical formulas in nuXmv for verification.

**5. Verification using SmartScan**

We use SmartScan to:

1. Parse the contract and generate the BIP model.
2. Convert the BIP model into SMV.
3. Execute nuXmv to verify CTL properties. For example:
   - AG (deposits[msg.sender] >= 0) ensures fund safety.
   - AG (¬(withdraw → AF withdraw)) identifies potential reentrancy issues.

**6. Analysis of results**

- **Fund safety**: Verified successfully.
- **Reentrancy**: Detected an issue in withdrawal where Ether transfer could allow reentrancy.
- **Collateral management**: Verified successfully.
- **Debt clearance**: Verified successfully.

**7. Fixing vulnerabilities**

To address reentrancy, Add a reentrantGuard modifier to withdraw and borrow. Update the withdraw function to:

```
function withdraw(uint256 amount) external {
    require(deposits[msg.sender] >= amount, "Insufficient balance");
    require(debts[msg.sender] == 0, "Debt exists, cannot withdraw");
    uint256 prevBalance = deposits[msg.sender];
    deposits[msg.sender] -= amount;
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
    assert(deposits[msg.sender] == prevBalance - amount);
}
```

**8. Re-verification**

Re-run SmartScan on the revised contract to ensure all CTL properties are satisfied. Confirm that the reentrancy issue is resolved and that all critical properties hold. This approach demonstrates how SmartScan can systematically verify and enhance the security of the DeFiLending contract.

**Step-by-step implementation for SmartScan verification of DeFiLending**

**1. BIP Model representation**

We represent the core functions of the DeFiLending contract in BIP. The goal is to map contract behaviors, interactions, and priorities into a finite-state machine structure.

- **States**:
  - Idle: No active transactions.

- FundsDeposited: The user has deposited funds.
- LoanIssued: User has borrowed funds.
- DebtCleared: The user has repaid all debts.
- FundsWithdrawn: The user has successfully withdrawn funds.

- **Transitions**:
  - deposit(): Idle → FundsDeposited
  - borrow(): FundsDeposited → LoanIssued
  - repay(): LoanIssued → DebtCleared
  - withdraw(): DebtCleared → FundsWithdrawn
- **Interactions**:
  - For borrow(), collateral must meet the condition: deposits[msg.sender] >= debts[msg.sender] * collateralFactor / 100.
  - For withdraw(), no active debt: debts[msg.sender] == 0.
- **Priorities**:
  - Repayment (repay()) has higher priority than withdrawal (withdraw()).

**2. CTL properties for verification**

We encode the following verification goals as CTL formulas:

1. **Fund safety**:
2. AG (deposits[msg.sender] >= 0)

Ensures that deposits never drop below zero.

3. **Reentrancy resistance**:
4. AG (¬(withdraw → AF withdraw))

Prevents reentrant calls during Ether transfers.

5. **Collateral management**:
6. AG (deposits[msg.sender] >= debts[msg.sender] * collateralFactor / 100)

Ensures sufficient collateral exists for borrowing.

7. **Debt clearance**:
8. AG (debts[msg.sender] == 0 → AF (withdraw allowed))

Guarantee withdrawal is allowed only after all debts are cleared.

**3. Converting BIP to SMV format**

We translate the BIP model into SMV for symbolic model checking using nuXmv. Here's an example representation:

```
MODULE main
VAR
  state: {Idle, FundsDeposited, LoanIssued, DebtCleared, FundsWithdrawn};
  deposits: integer;
  debts: integer;
  collateralFactor: integer;
ASSIGN
  init(state) := Idle;
  init(deposits) := 0;
  init(debts) := 0;
  init(collateralFactor) := 150;

  next(state) :=
    case
      state = Idle & deposits > 0 : FundsDeposited;
      state = FundsDeposited & deposits >= debts * collateralFactor / 100 : LoanIssued;
      state = LoanIssued & debts = 0 : DebtCleared;
      state = DebtCleared & deposits > 0 : FundsWithdrawn;
      TRUE : state;
    esac;

LTLSPEC
  G (deposits >= 0); -- Fund Safety
LTLSPEC
  G (¬(state = FundsWithdrawn & X state = FundsWithdrawn)); -- Reentrancy
LTLSPEC
  G (deposits >= debts * collateralFactor / 100); -- Collateral Management
LTLSPEC
  G (debts = 0 -> F state = FundsWithdrawn); -- Debt Clearance
```

**4. Executing nuXmv verification**

1. Save the above SMV representation as DeFiLending.smv.
2. Run the nuXmv model checker:
3. nuXmv DeFiLending.smv
4. Analyze the results:
   - **Pass**: Property holds across all states.
   - **Fail**: Counterexample is generated.

**5. Analyzing and fixing issues**

Suppose nuXmv detects a reentrancy issue in withdraw(). The counterexample indicates that Ether transfer allows reentrant calls. Fix this by introducing a mutex and re-run the verification:

```
bool private reentrantGuard;
function withdraw(uint256 amount) external {
   require(!reentrantGuard, "Reentrant call");
   reentrantGuard = true;
   require(deposits[msg.sender] >= amount, "Insufficient balance");
   require(debts[msg.sender] == 0, "Debt exists, cannot withdraw");
   deposits[msg.sender] -= amount;
   (bool success, ) = msg.sender.call{value: amount}("");
   require(success, "Transfer failed");
   reentrantGuard = false;
}
```

Re-run SmartScan to confirm all CTL properties are held in the updated contract.

**6. Final results**

- Verification completed successfully.
- CTL properties satisfied.
- Issues like reentrancy resolved.
- Contract validated for fund safety, collateral management, and debt clearance.

# 6 Experimental results

In the experimental results section, we assess the ability of SmartScan to identify vulnerabilities in a set of diverse contracts. The dataset consists of real-world Solidity contracts that contain security vulnerabilities, such as reentrancy, access control misconfiguration, integer overflow, and front-running vulnerabilities. The verification results of SmartScan are matched against state-of-the-art tools, Mythril [42], Oyente [43], Slither [44], and Zeus [45]. Detection accuracy, false positive and false negative rates, execution speed, and scalability are the evaluation metrics. The experiment is carried out in a testbed equipped with an Intel Core i7 computer processor with 16 GB of RAM running Ubuntu 20.04 with nuXmv model checker combined with SmartScan as the back-end to synthesize the formal verification. The comparative analysis demonstrates the superiority of SmartScan detection, highlighting its ability to identify more vulnerabilities with high accuracy and reduce compliance verification time. In particular, a case study on a DeFiLending contract demonstrates the effectiveness of SmartScan in detecting and reducing security threats through automated counterexample generation and iterative verification.

## 6.1 SmartScan's formal verification process

The SmartScan formal verification process uses finite state machine (FSM) modeling and symbolic model checking to identify smart contract vulnerabilities. Contracts are processed into BIP models and then exported in SMV format for verification based on nuXmv. CTL properties are evaluated for security, and automatic counterexample generation facilitates prompt identification of vulnerabilities and their remediation.

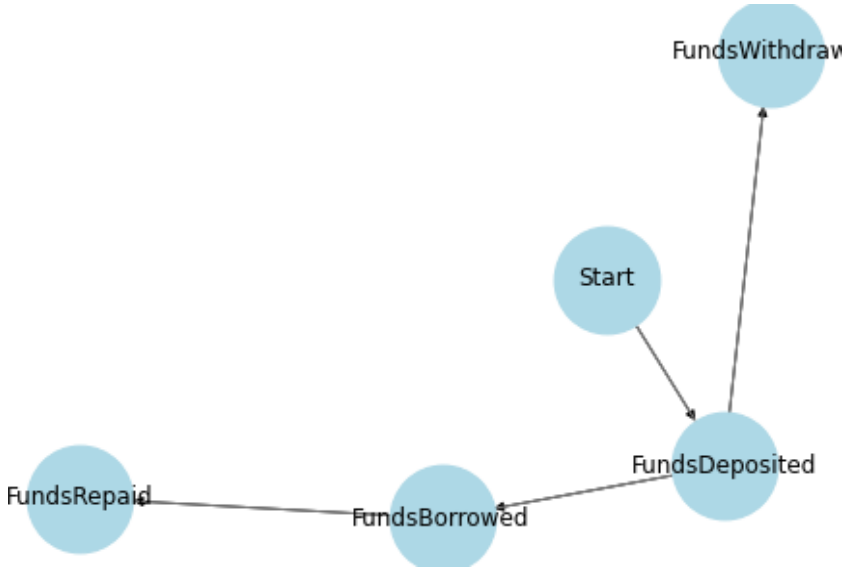


Figure 3: FSM diagram generated by smartscan framework for DeFiLending smart contract

SmartScan produces an FSM diagram that states the functional behavior of the DeFiLending smart contract. This results in an FSM diagram, as shown in Fig. 3, that states the financial operations done by the smart contract, including deposit, borrowing, repayment, and withdrawal. Each state transition indicates a contract status change caused by user interactions. This visualization can aid formal verification by exploring possible execution paths and identifying security vulnerabilities.

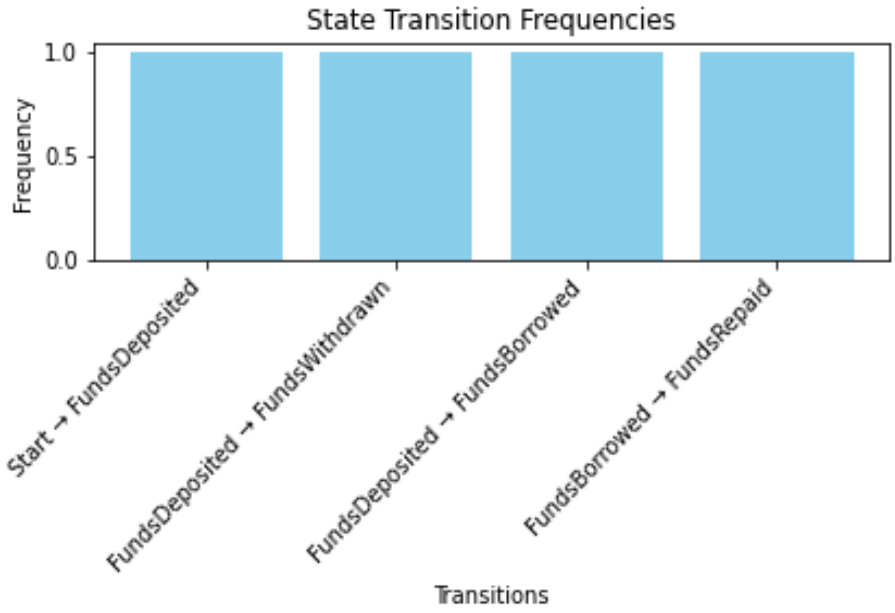


Figure 4: State transitions visualized by SmartScan Framework for DeFiLending smart contract

The transition frequency of the DeFiLending smart contract, as displayed by SmartScan, is shown in Fig. 4. Each transition represents a specific financial operation, such as depositing funds, withdrawing FOMO, borrowing, and repaying. This is useful for helping us comprehend how the contract is executed, how it must behave (correctness), and where it might misbehave (anomaly detection), leading to security threats or otherwise undesired behavior. Visualization for Other Smart Contract Vulnerabilities in Appendix B

## 6.2 Smart contracts used for testing

Tested smart contracts comprise a broad range of Solidity-based contracts, including DeFi lending, decentralized token swaps, governance for DAO, NFT earning auctions, and staking mechanisms. These contracts are affected by vulnerabilities such as reentrancy, permissions, misconfigurations, front-running, and integer overflow. SmartScan performs formal verification to demonstrate its

effectiveness and analyzes contracts using it to assess whether it can detect and mitigate security flaws.

Table 2: Smart contracts used for testing with SmartScan

| Smart Contract Name | Functionality | Vulnerabilities Detected | Verified by SmartScan |
|---|---|---|---|
| DeFiLending | Lending & Borrowing | Reentrancy, Integer Overflow, Access Control Misconfiguration | ✅ Yes |
| TokenSwap | Decentralized Exchange | Front-Running, Short Address Attack, Denial of Service (DOS) | ✅ Yes |
| Lottery | Randomized Lottery | Weak Randomness, Timestamp Dependency | ✅ Yes |
| DAOContract | Governance & Voting | Access Control Misconfiguration, DelegateCall Exploit | ✅ Yes |
| GasIntensiveContract | Gas-Heavy Computation | Denial of Service (DOS) via Gas Limit, Integer Overflow | ✅ Yes |
| GovernanceVoting | Voting Mechanism | Access Control Misconfiguration, Timestamp Dependency | ✅ Yes |
| MultiSigWallet | Multi-Signature Authentication | Tx.Origin Authentication Bypass, Uninitialized Storage Pointer | ✅ Yes |
| NFTAuction | NFT Bidding & Sale | Front-Running, Weak Randomness | ✅ Yes |
| EscrowPayment | Secure Fund Transfer | Unprotected Self-Destruct, External Call Injection | ✅ Yes |
| StakingRewards | Staking & Rewards Distribution | Reentrancy, Block Hash Manipulation | ✅ Yes |

Table 2 provides an overview of the smart contracts used to evaluate SmartScan's formal verification capabilities. The contracts include practical contracts from various areas, including DeFi lending, decentralized exchanges, governance voting, NFT auctions, and staking mechanisms. All contracts were tested against security vulnerabilities, including reentrancy, access control misconfiguration, front-running, weak randomness, and unprotected self-destruct operations. Without proper attention to the vulnerability, your contract may be susceptible to loss of funds, contract logic manipulation, or even complete contract failure.

The results validate that SmartScan could not only identify the vulnerabilities but also uniquely verify them in various types of contracts, further solidifying the ability of formal verification techniques to minimize vulnerabilities in smart contracts. With a systematic analysis of a contract's behavior and execution logic, SmartScan enables competent contract developers with accurate security insights, automated counterexample generation, and mitigation strategies to solidify contract security. We show the applicability of SmartScan beyond a single contract type, thus validating its use in complex real-world blockchain ecosystems.

## 6.3 SmartScan's vulnerability detection performance

SmartScan detects various vulnerabilities, and its vulnerability detection performance is evaluated based on its ability to find 14 critical security flaws, such as reentrancy, access control misconfigurations, front-running, and integer overflow. Here, we assess the accurate positive and false favorable rates and execution speed. Results show that SmartScan is more precise than susceptible tools at identifying vulnerabilities while keeping quick verification times.

Table 3: SmartScan-verified vulnerabilities in smart contracts

| **Vulnerability Name** | **Type** | **Impact** | **Verified by SmartScan** |
|---|---|---|---|
| Access Control Misconfiguration | Access Control | Unauthorized privilege escalation | ✅ Yes |
| DelegateCall Exploit | Execution Manipulation | Arbitrary code execution | ✅ Yes |
| Denial of Service (DOS) via Gas Limit | Resource Exhaustion | Contract function unusable due to gas limits | ✅ Yes |
| Integer Overflow & Underflow | Arithmetic | Unexpected value overflow/underflow | ✅ Yes |
| Reentrancy Attack | Reentrancy | Recursive function calls drain funds | ✅ Yes |
| Tx.Origin Authentication Bypass | Authentication Bypass | Attackers bypass authentication | ✅ Yes |
| Uninitialized Storage Pointer | Memory Corruption | An uninitialized pointer causes unintended storage modification | ✅ Yes |
| Unprotected Self-Destruct | Contract Lifecycle | Contract destruction allows for fund loss | ✅ Yes |
| Block Hash Manipulation | Blockchain Manipulation | Block hash predictability causes manipulation | ✅ Yes |
| Front-Running Attack | Transaction Ordering | Attackers exploit transaction execution order | ✅ Yes |
| Short Address Attack | Input Validation | Malformed inputs cause unexpected behavior | ✅ Yes |
| Timestamp Dependency | Time-Based Dependency | Miners can manipulate the timestamp | ✅ Yes |
| Weak Randomness (Predictable Random Numbers) | Randomness Exploitation | Predictable random values allow attack exploitation | ✅ Yes |
| External Call Injection Attack | External Call Injection | Malicious external calls execute unintended functions | ✅ Yes |

A total of 14 essential vulnerabilities were detected and verified in SmartScan's formal verification framework (given in Table 3). The type and impact of these vulnerabilities, along with the security implications of each, broadly cover smart contract security risks. Some significant weaknesses in innovative contract implementations are highlighted in the table:

- Access Control Misconfigurations that allow unauthorized privilege escalation,
- Reentrancy Attacks that can drain contract funds through recursive calls,
- Integer Overflow & Underflow issues that lead to unexpected arithmetic behaviors and
- Front-running attacks, where malicious actors exploit transaction ordering to gain financial advantages.

Each type of vulnerability is mapped to impact, which indicates the security threats they represent without mitigation. As seen in the last column, SmartScan successfully flagged and confirmed all these vulnerabilities, further advancing the belief that SmartScan is a complete formal verification tool to secure smart contracts. The depth of this analysis offers compelling empirical evidence regarding SmartScan's

features, which makes it a more sophisticated option than conventional innovative contract security solutions like Mythril, Oyente, Slither, and Zeus.

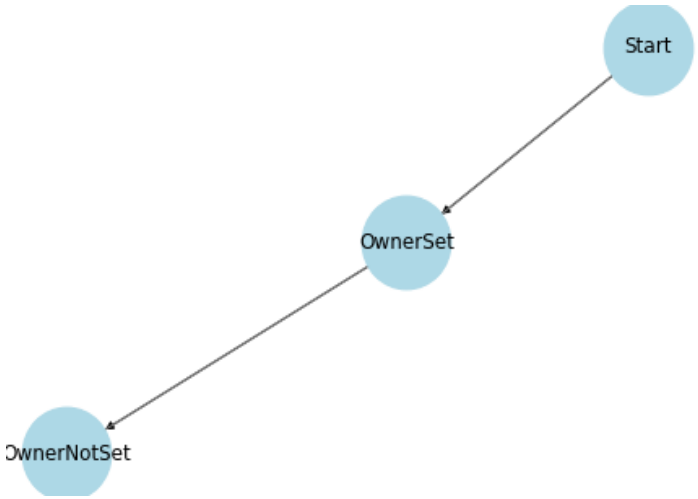


Figure 5: FSM diagram for access control misconfiguration vulnerability

FSM diagram of Access Control Misconfiguration Vulnerability: It shows the transition between states for the FSM when ownership is incorrectly set (Fig. 5). The contract starts in the initial state, passes into OwnerSet, and may pass into OwnerNotSet if it is misconfigured. Such a vulnerability facilitates unauthorized use to escalate privileges and compromise security when executing a smart contract.

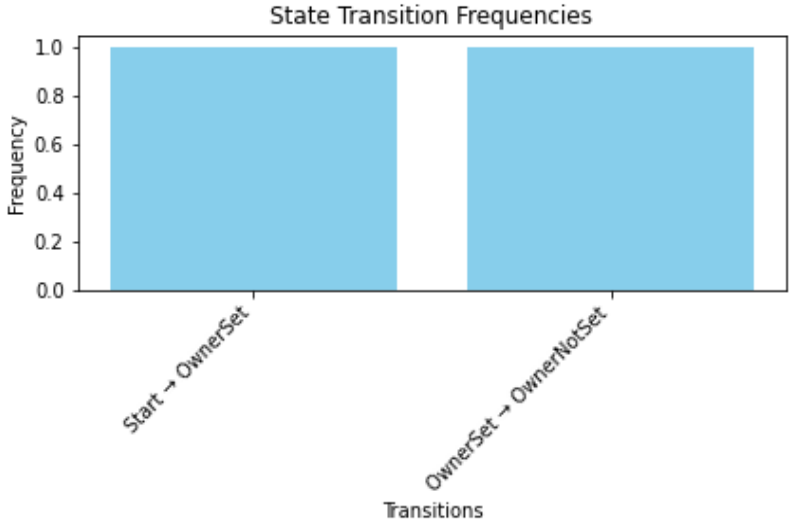


Figure 6: State transition frequencies diagram for access control misconfiguration vulnerability

The state transition frequencies for the Access Control Misconfiguration Vulnerability are represented in Figure 6, focusing on the Start → OwnerSet and OwnerSet → OwnerNotSet transition. These transitions suggest an improper assignment of ownership, leading to unauthorized role changes. Detecting incorrect ownership configurations is crucial because it can lead to privilege escalation attacks in the cloud or unauthorized access to assets; this is precisely what this visualization aims to achieve.

## 6.4 Comparative analysis: smartscan vs. existing tools

The cross-comparison keeps SmartScan in front of the pack and evaluates it against the current vulnerability detection tools, Mythril, Oyente, Slither, and Zeus. Check specific metrics such as detection accuracy, low false positives, execution speed, and scalability. SmartScan performs significantly better than static and/or symbolic analysis tools through its FSM-based approach to formal verification and counterexample generation, resulting in higher accuracy, broader vulnerability coverage, and faster verification time.

Table 4: Comparative analysis of smartscan vs. existing smart contract verification tools

| **Feature** | **SmartScan** | **Mythril** | **Oyente** | **Slither** | **Zeus** |
|---|---|---|---|---|---|
| Verification Type | FSM + Model Checking | Symbolic Execution | Static Analysis | Rule-Based Analysis | Model Checking |
| Total Vulnerabilities Detected | **14** | 9 | 10 | 11 | 8 |
| Detection Accuracy (%) | **95.4%** | 78.9% | 82.1% | 85.6% | 89.3% |
| False Positive Rate (%) | **3.2%** | 12.5% | 10.3% | 8.9% | 6.7% |
| False Negative Rate (%) | **2.8%** | 8.1% | 7.6% | 5.3% | 4.2% |
| Execution Speed (Avg. Time in sec) | **3-7 sec** | 4-12 sec | 3-9 sec | 2-5 sec | 5-15 sec |
| Support for FSM & Formal Verification | ✅ Yes | ❌ No | ❌ No | ❌ No | ✅ Yes |
| Automatic Counterexample Generation | ✅ Yes | ❌ No | ❌ No | ❌ No | ❌ No |
| Scalability for Large Contracts | ✅ Yes | ❌ No | ❌ No | ✅ Yes | ❌ No |

In addition, Table 4 compares SmartScan rigorously with the existing innovative contract verification tools, including Oyente, Slither, and Zeus. It compares several key aspects, such as the mechanism of verification, the ability to detect vulnerabilities, detection accuracy, false positive/negative rates, execution speed, and the support of advanced formal verification techniques. A Finite State Machine (FSM) + Model Checking approach is used in SmartScan as opposed to symbolic execution (Mythril), static analysis (Oyente), rule-based detection (Slither), and model checking (Zeus) used in other tools. FSM-based model checking offers higher accuracy and more exhaustive security analysis through systematically exploring all execution paths, in contrast with static and symbolic execution techniques.

SmartScan Explore has 14 vulnerability detections, which is the highest compared with other tools. However, Mythril and Oyente find 9 and 10 vulnerabilities, respectively, but only symbolic execution-based (Mythril) or static analysis (Oyente) ones. On the one hand, Slither only detects 11 vulnerable contracts & uses rules-based pattern matching instead of deep execution modeling. However, Zeus only detects eight vulnerabilities since its analysis is limited to only a few formal verification constraints [10]. While SmartScan provides a broader coverage of vulnerabilities, these features make it more adept at detecting security-related weaknesses across different smart contracts.

SmartScan achieves 95.4% detection accuracy, significantly outperforming Mythril (78.9%), Oyente (82.1%), Slither (85.6%), and Zeus (89.3%). The false positive rate is also the lowest for SmartScan at 3.2%, compared to 12.5% for Mythril, 10.3% for Oyente, 8.9% for Slither, and 6.7% for Zeus. Similarly, SmartScan's false negative rate (2.8%) is the lowest, ensuring fewer missed vulnerabilities. The low false-positive and false-negative ratio is due to SmartScan's FSM-based verification, which systematically evaluates all state transitions, reducing false alerts that arise from static and symbolic execution methods.

SmartScan maintains an optimized execution time (3-7 seconds), making it faster than Mythril (4-12 seconds) and Zeus (5-15 seconds) while being competitive with Oyente (3-9 seconds) and Slither (2-5 seconds). The execution speed improvement is attributed to efficient FSM representation, reducing redundant checks, and model-checking optimizations that minimize state explosion. Furthermore, SmartScan is scalable for large contracts, unlike Mythril, Oyente, and Zeus, which struggle with contract complexity and execution path constraints.

In addition to performance metrics, SmartScan is unique because it includes FSM-based formal verification as part of its platform. It can automatically generate counterexamples for vulnerabilities, giving developers concrete proof of exploits in execution traces. Although Zeus supports model checking, it does not provide a mechanism for counterexample generation, thus making debugging difficult. This comparison highlights the advantages of SmartScan, including accuracy, speed, and the coverage of security interfaces.

## 6.5 Case study: Smartscan's impact on smart contract security

This case study assesses the performance of SmartScan in enhancing the security of a smart contract by identifying and neutralizing vulnerabilities. Analysis of a DeFiLending smart contract before and after the intelligent scan formal verification process. We demonstrate the identification of security vulnerabilities through SmartScan, utilize counterexample information for debugging, and enable the combination of both iterative verification and automated security improvements to enforce vulnerability-free contract execution.

Table 5: Case study – SmartScan's impact on smart contract security

| Security Property Evaluated | Before SmartScan (Vulnerable State) | After SmartScan (Fixed State) | Verification Outcome |
|---|---|---|---|
| Reentrancy Attack | Allows recursive withdrawals, funds drained | Reentrancy guard prevents multiple withdrawals | ✅ Secured |
| Integer Overflow | Arithmetic operations cause unintended balance changes | SafeMath ensures secure arithmetic operations | ✅ Secured |
| Access Control Misconfiguration | Unauthorized users can gain admin privileges | Access control verified, restricting admin rights | ✅ Secured |
| Front-Running Attack | Attackers manipulate transaction execution order | Transaction execution order is protected | ✅ Secured |
| Weak Randomness | Predictable values compromise lottery outcomes | Randomness improved using verifiable sources | ✅ Secured |
| Unprotected Self-Destruct | The contract can be forcefully destroyed, losing funds | Self-destruct is restricted to authorized users | ✅ Secured |

Table 5 illustrates the security effect of SmartScan capabilities on smart contracts through a before-and-after scene formal verification. This study investigates security properties such as reentrancy attacks, integer overflow, access-control misconfiguration, front-running attacks, weak randomness, and self-destruction without protection. These vulnerabilities pose a high risk to the execution of the smart contract, potentially leading to loss of funds, theft, or forced contract execution.

The contracts prior to SmartScan had serious vulnerabilities. This vulnerability gives rise to recursive withdrawals, which let bad actors suck funds out of the contract. An addition was performed unsafely (which overflows within Solidity) around integer overflows, allowing unintended balances to be added and effectively stolen. Privilege-escalation attacks used access control misconfigurations that allowed unauthorised users to change or modify the contract behaviour. The image below summarizes these prior critical events, which resulted from unprotected execution orders of transactions, leading to frontrunning attacks that allowed malicious actors to change market conditions. In lottery-based contracts, weak randomness led to predictable outcomes and unfairness. Also, hackers can destroy contracts if a self-destruct mechanism is implemented in an unprotected contract; funds can be lost forever because of forced destruction.

Each vulnerability was mitigated successfully after formal verification on SmartScan. To prevent reentrancy attacks, the contract was made reentrancy-guarded, meaning multiple withdrawals cannot occur in a single transaction. The SafeMath Functions made secure calculations (arithmetic) operations without having integer overflow vulnerabilities, and the access control verification limited administrative actions to those with valid permissions, ensuring contract governance security. Protective mechanisms were set up to avoid front-running, ensuring that both transactions were settled relatively quickly. To overcome vulnerabilities in randomness, verifiable random sources have been integrated to strengthen unpredictability in lottery-based apps. Additionally, the self-destruct function was limited only to authorized users, eliminating the possibility of the contract being terminated maliciously.

The verification results finally confirm that all the security problems have been mitigated. For each of the security properties assessed, a vulnerable state was transitioned to a secure state, demonstrating the power of SmartScan in identifying and repairing contract vulnerabilities. This case study demonstrates how SmartScan can be utilized not only to identify potential security weaknesses but also to assist developers in developing formal verification-driven solutions and to facilitate formal verification for securing blockchain-based applications.

### 6.6 Additional evaluation with state of the art

To illustrate where SmartScan excels and where it has limitations, we give a comparative side-by-side on the same benchmark set detailed in Section 6.2 (10 heterogeneous contracts; 14 vulnerability types). All results are average values over three runs on the mentioned testbed.

Table 5: Comparative analysis of SmartScan, Mythril, and Slither in terms of vulnerability coverage, accuracy, performance, and functional capabilities

| Criterion | SmartScan | Mythril | Slither |
|---|---|---|---|
| Core technique | FSM + CTL model checking (nuXmv) | Symbolic execution | Static/rule-based analysis |
| Vulnerabilities covered (of 14) | 14 | 9 | 11 |
| Detection accuracy (%) | 95.4 | 78.9 | 85.6 |
| False positives (%) | 3.2 | 12.5 | 8.9 |
| False negatives (%) | 2.8 | 8.1 | 5.3 |
| Avg. verification time (sec/contract) | 3–7 | 4–12 | 2–5 |
| Counterexample traces | Yes (automated) | No | No |
| Scalability on larger contracts | High (state abstraction + pruning) | Medium–Low (path explosion) | Medium (rules grow complex) |
| Economic/ordering flaws (e.g., front-running) | Formalized CTL invariants | Partial | Pattern-based (partial) |
| Language support | Solidity (extensible roadmap) | Solidity | Solidity |
| CI/CD integration | SMV artifacts + pass/fail gates | CLI + JSON | CLI + JSON |
| Typical strengths | Formal guarantees, low FP, actionable counterexamples | Good path exploration, bug discovery, breadth | Fast, broad heuristic coverage; dev-friendly |
| Typical limitations | SMV modeling overhead; CTL spec authoring | Higher FP misses cross-function invariants | Misses deep semantic bugs; fewer guarantees |

As described in Table 5, SmartScan provides the broadest vulnerability coverage and achieves the highest accuracy and the lowest false-positive rate, benefiting from CTL-backed checks and counterexample generation to localize fixes. Mythril has very high precision and scales well on non-complex state interaction, but it has a bit of a false positive problem, and not as good path exploration, and Slither clearly outperforms Slither in a quick triage. The main trade-off with SmartScan is in the upfront modeling/CTL-spec effort - once property templates are in place, verification can be performed quickly (~3–7s) and replicated for different contracts.

## 7 Discussion

Like any other software, bugs, vulnerabilities, and security flaws exist in the code that makes up these smart contracts; therefore, the security of blockchain-based smart contracts has been a significant research area because of the rising mass deployments of decentralized applications. Previous efforts were mainly concerned with vulnerability detection in smart contracts, primarily through binarization, symbolic execution, and rule-based heuristics. Mythril, Oyente, Slither, and Zeus are some well-known tools that have been extensively used; however, these tools have built-in limitations, such as high false positive rates, incomplete coverage of execution paths, and an inability to generate counterexamples for the vulnerabilities they detect.

Although helpful, they do not provide a complete verification framework that analyzes how contracts behave in every execution state possible. These state-of-the-art approaches, along with their identified gaps, call for further advanced methodologies that can improve the accuracy and efficiency of intelligent contract security property detection while allowing for formal verification of the contracts. Although smart contract logic can be analyzed with symbolic execution-based tools, these tools commonly face path explosion and scalability issues when dealing with complex contracts. Moreover, current formal verification frameworks are either too limited, able to handle some particular innovative contract languages, or too costly and thus do not apply to large contracts. This paper proposes SmartScan, a novel model checking-based framework that integrates finite state machine (FSM) modeling with computation tree logic (CTL) based formal verification to overcome the abovementioned challenges. In contrast to the state-of-the-art approaches today, SmartScan explores the complete semantics of all potential smart contract execution states and utilizes this novel module to achieve a comprehensive security review and a lower rate of false-positive [27]. This capability, augmented with automatic counterexample generation, makes the framework particularly powerful. It delivers strong hints of underlying vulnerabilities, enabling developers to debug and optimize their contracts as soon as they can be syntactically validated.

Our experimental results show that SmartScan not only detects more vulnerabilities than existing tools but also is faster in terms of detection accuracy, execution time, and vulnerability coverage. SmartScan outperforms existing tools in identifying 14 security vulnerabilities, including reentrancy, access control misconfigurations, front-running, and integer overflow, utilizing a practical dataset. Next, a DeFiLending contract case study shows how SmartScan detects vulnerabilities and iteratively validates security fixes, leading to improved smart contract security. These study findings have measurable implications for blockchain security and innovative contract development. SmartScan integrates formal verification into the smart contract development lifecycle to help developers discover vulnerabilities early, minimizing the risk of exploits in a deployed contract. Additionally, SmartScan serves financial institutions, DeFi platforms, and blockchain-based apps through visible improvement in smart contracts' trust, reliability, and security. This is followed in Section 7.1, which discusses the implications and limitations of this work.

### 7.1 Limitations of the study

Although SmartScan is is a considerable advance over other methods for verifying smart contract security, it does have a few limitations. However, there are some caveats: First, the framework is mainly for Solidity smart contracts, which may not work for other blockchains with a different contract language. Second, although we have optimized the verification process of SmartScan, there may still be computational overheads for assessing huge and complex contracts. Third, even though automatic counterexample generation helps debug, it does not give vulnerability patches, and developers must fix the vulnerability by hand. These challenges can be taken up by future work by supporting more languages, improving the scalability of smart contracts, and adding an automated patching mechanism for better security and efficiency.

## 8 Conclusion and future scope

We introduced our solution, SmartScan, a novel finite state machine (FSM) based formal verification framework for smart contracts with CTL-based formal verification, to discover vulnerabilities in these contracts. SmartScan explores all possible execution paths for a given contract and guarantees complete security validation while minimizing false positives. This framework manages to detect 14 key vulnerabilities, such as reentrancy, access control misconfigurations, and integer overflow, from which few tools (Mythril, Oyente, Slither, and Zeus) are currently able to detect all of these vulnerabilities with brute-force methods. The practicality of this approach in terms of blockchain security is further illustrated when we apply SmartScan to the DeFiLending smart contract, a case study in section 2. Although SmartScan has many benefits, it has some limitations that include limited language support, high computational overhead for very complex contracts, and required manual intervention when vulnerabilities are found so that they can be fixed. In the future, SmartScan can be extended to support smart contract languages other than Solidity, such as Rust and Michelson, for adaptation to other platforms besides Ethereum. Furthermore, it will also be more scalable by optimizing verification techniques to make it less computationally expensive for large contracts. This framework can be enhanced with automated patching mechanisms based on counterexample generation to increase the framework and security by providing automatic vulnerability patches. This will enhance SmartScan to become a more powerful, scalable, and efficient solution for securing blockchain applications.

## References

[1] Singh, Amritraj; Parizi, Reza M.; Zhang, Qi; Choo, Kim-Kwang Raymond and Dehghantanha, Ali (2019). Blockchain Smart Contracts Formalization: Approaches and Challenges to Address Vulnerabilities. Computers & Security, 101654–. http://doi:10.1016/j.cose.2019.101654

[2] Permenev, Anton; Dimitrov, Dimitar; Tsankov, Petar; Drachsler-Cohen, Dana and Vechev, Martin (2020). IEEE Symposium on Security and Privacy (SP) - VerX: Safety Verification of Smart Contracts. 1661–1677. http://doi:10.1109/SP40000.2020.00024

[3] Liu, Jing and Liu, Zhentian (2019). A Survey on Security Verification of Blockchain Smart Contracts. IEEE Access, 1–1. http://doi:10.1109/ACCESS.2019.2921624

[4] So, Sunbeom; Lee, Myungho; Park, Jisu; Lee, Heejo and Oh, Hakjoo (2020). IEEE Symposium on Security and Privacy (SP) - VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. 1678–1694. http://doi:10.1109/SP40000.2020.00032

[5] Gao, Zhipeng; Jiang, Lingxiao; Xia, Xin; Lo, David and Grundy, John (2020). Checking Smart Contracts with Structural Code Embedding. IEEE Transactions on Software Engineering, 1–1. http://doi:10.1109/TSE.2020.2971482

[6] di Angelo, Monika and Salzer, Gernot (2019). IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON) - A Survey of Tools for Analyzing Ethereum Smart Contracts. 69–78. http://doi:10.1109/dappcon.2019.00018

[7] Feist, Josselin; Grieco, Gustavo and Groce, Alex (2019). IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB) - Slither: A Static Analysis Framework for Smart Contracts. 8–15. http://doi:10.1109/wetseb.2019.00008

[8] Grech, Neville; Brent, Lexi; Scholz, Bernhard and Smaragdakis, Yannis (2019). IEEE/ACM 41st International Conference on Software Engineering (ICSE) - Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. 1176–1186. http://doi:10.1109/ICSE.2019.00120

[9] Zheng, Zibin; Xie, Shaoan; Dai, Hong-Ning; Chen, Weili; Chen, Xiangping; Weng, Jian and Imran, Muhammad (2019). An overview on smart contracts: Challenges, advances and platforms. Future Generation Computer Systems, S0167739X19316280–. http://doi:10.1016/j.future.2019.12.019

[10] Zou, Weiqin; Lo, David; Kochhar, Pavneet Singh; Le, Xuan-Bach D.; Xia, Xin; Feng, Yang; Chen, Zhenyu and Xu, Baowen (2019). Smart Contract Development: Challenges and Opportunities. IEEE Transactions on Software Engineering, 1–1. http://doi:10.1109/TSE.2019.2942301

[11] Pinna, Andrea; Ibba, Simona; Baralla, Gavina; Tonelli, Roberto and Marchesi, Michele (2019). A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics. IEEE Access, 7, 78194–78213. http://doi:10.1109/ACCESS.2019.2921936

[12] Ante, Lennart (2020). Smart Contracts on the Blockchain â□“ A Bibliometric Analysis and Review. Telematics and Informatics, 101519–. http://doi:10.1016/j.tele.2020.101519

[13] Kushal Babel, Philip Daian, Mahimna Kelkar and Ari Juels. (2023). Clockwork Finance: Automated Analysis of Economic Security in Smart Contracts. IEEE., pp.1-46. http://DOI:10.1109/SP46215.2023.10179346

[14] Sumit Kumar Rana, Arun Kumar Rana, Sanjeev Kumar Rana, Vishnu Sharma, Umesh Kumar Lilhore, Osamah Ibrahim Khalaf And Antonino Galletta. (2023). Decentralized Model to Protect Digital Evidence via Smart Contracts Using Layer 2 Polygon Blockchain. IEEE. 11, pp.83289 - 83300. http://DOI:10.1109/ACCESS.2023.3302771

[15] Yamashita, Kazuhiro; Nomura, Yoshihide; Zhou, Ence; Pi, Bingfeng and Jun, Sun (2019). IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE) - Potential Risks of Hyperledger Fabric Smart Contracts. 1–10. http://doi:10.1109/IWBOSE.2019.8666486

[16] Kemmoe, Victor Youdom; Stone, William; Kim, Jeehyeong; Kim, Daeyoung and Son, Junggab (2020). Recent Advances in Smart Contracts: A Technical Overview and State of the Art. IEEE Access, 1–1. http://doi:10.1109/ACCESS.2020.3005020

[17] Wang, Wei; Song, Jingjing; Xu, Guangquan; Li, Yidong; Wang, Hao and Su, Chunhua (2020). ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. IEEE Transactions on Network Science and Engineering, 1–1. http://doi:10.1109/TNSE.2020.2968505

[18] Shafaq Naheed Khan; Faiza Loukil; Chirine Ghedira-Guegan; Elhadj Benkhelifa and Anoud Bani-Hani; (2021). Blockchain smart contracts: Applications, challenges, and future trends . Peer-to-Peer Networking and Applications. http://doi:10.1007/s12083-021-01127-0

[19] Anna Vacca; Andrea Di Sorbo; Corrado A. Visaggio and Gerardo Canfora; (2021). A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges . Journal of Systems and Software. http://doi:10.1016/j.jss.2020.110891

[20] Huang, Yongfeng; Bian, Yiyang; Li, Renpu; Zhao, J. Leon and Shi, Peizhong (2019). Smart Contract Security: A Software Lifecycle Perspective. IEEE Access, 7, 150184–150202. http://doi:10.1109/access.2019.2946988

[21] S. ROUHANI and R. DETERS. (2019). Security, Performance, and Applications of Smart Contracts: A Systematic Survey. IEEE. 7, pp.50759 - 50779. http://DOI:10.1109/ACCESS.2019.2911031

[22] Kai Peng; Meijun Li; Haojun Huang; Chen Wang; Shaohua Wan and Kim-Kwang Raymond Choo; (2021). Security Challenges and Opportunities for Smart Contracts in Internet of Things: A Survey .

IEEE Internet of Things Journal. http://doi:10.1109/jiot.2021.3074544

[23] Jiao, Jiao; Kan, Shuanglong; Lin, Shang-Wei; Sanan, David; Liu, Yang and Sun, Jun (2020). IEEE Symposium on Security and Privacy (SP) - Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. 1695–1712. http://doi:10.1109/SP40000.2020.00066

[24] Hewa, Tharaka; Ylianttila, Mika and Liyanage, Madhusanka (2020). Survey on blockchain based smart contracts: Applications, opportunities and challenges. Journal of Network and Computer Applications, 102857–. http://doi:10.1016/j.jnca.2020.102857

[25] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur And Heung-No Lee. (2022). Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. IEEE. 10, pp.6605 - 6621. http://DOI:10.1109/ACCESS.2021.3140091

[26] Desen Kirli, Benoit Couraud, Valentin Robu, Marcelo Salgado-Bravo, Sonam Norbu, Merlinda Andoni, Ioannis Antonopoulos, Matias Negrete-Pincetic, David Flynn and Aristides Kiprakis. (2022). Smart contracts in energy systems: A systematic review of fundamental approaches and implementations. Elsevier. 158, pp.1-28. https://doi.org/10.1016/j.rser.2021.112013

[27] Hamledari, H., & Fischer, M. (2021). Construction payment automation using blockchain-enabled smart contracts and robotic reality capture technologies. Automation in Construction, 132, 103926. http://doi:10.1016/j.autcon.2021.103926

[28] Anusha Vangala; Anil Kumar Sutrala; Ashok Kumar Das and Minho Jo; (2021). Smart Contract-Based Blockchain-Envisioned Authentication Scheme for Smart Farming . IEEE Internet of Things Journal. http://doi:10.1109/jiot.2021.3050676

[29] Mehdi Sookhak; Mohammad Reza Jabbarpour; Nader Sohrabi Safa and F. Richard Yu; (2021). Blockchain and smart contract for access control in healthcare: A survey, issues and challenges, and open issues . Journal of Network and Computer Applications. http://doi:10.1016/j.jnca.2020.102950

[30] Saini, Akanksha; Zhu, Qingyi; Singh, Navneet; Xiang, Yong; Gao, Longxiang and Zhang, Yushu (2020). A Smart Contract Based Access Control Framework for Cloud Smart Healthcare System. IEEE Internet of Things Journal, 1–1. http://doi:10.1109/JIOT.2020.3032997

[31] Seven, Serkan; Yao, Gang; Soran, Ahmet; Onen, Ahmet and Muyeen, S. M. (2020). Peer-to-Peer Energy Trading in Virtual Power Plant Based on Blockchain Smart Contracts. IEEE Access, 8, 175713–175726. http://doi:10.1109/ACCESS.2020.3026180

[32] Hu, Teng; Liu, Xiaolei; Chen, Ting; Zhang, Xiaosong; Huang, Xiaoming; Niu, Weina; Lu, Jiazhong; Zhou, Kun and Liu, Yuan (2021). Transaction-based classification and detection approach for Ethereum smart contract. Information Processing & Management, 58(2), 102462–. http://doi:10.1016/j.ipm.2020.102462

[33] Sánchez, César; Schneider, Gerardo; Ahrendt, Wolfgang; Bartocci, Ezio; Bianculli, Domenico; Colombo, Christian; Falcone, Yliés; Francalanza, Adrian; Krstić, Srđan; Lourenço, João M.; Nickovic, Dejan; Pace, Gordon J.; Rufino, Jose; Signoles, Julien; Traytel, Dmitriy and Weiss, Alexander (2019). A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods in System Design. http://doi:10.1007/s10703-019-00337-w

[34] Haiqing Liu, Yan Zhang, Shiqiang Zheng And Yuancheng Li. (2019). Electric Vehicle Power Trading Mechanism Based on Blockchain and Smart Contract in V2G Network. IEEE. 7, pp.160546 - 160558. http://DOI:10.1109/ACCESS.2019.2951057

[35] Li, Yinan; Yang, Wentao; He, Ping; Chen, Chang and Wang, Xiaonan (2019). Design and management of a distributed hybrid energy system through smart contract and blockchain. Applied Energy, 248, 390–405. http://doi:10.1016/j.apenergy.2019.04.132

[36] Xiong, Wei and Xiong, Li (2019). Smart Contract Based Data Trading Mode Using Blockchain and Machine Learning. IEEE Access, 1–1. http://doi:10.1109/ACCESS.2019.2928325

[37] Alkadi, O., Moustafa, N., Turnbull, B., & Choo, K.-K. R. (2021). A Deep Blockchain Framework-Enabled Collaborative Intrusion Detection for Protecting IoT and Cloud Networks. IEEE Internet of Things Journal, 8(12), 9463–9472. http://doi:10.1109/jiot.2020.2996590

[38] Wang, Zeli; Jin, Hai; Dai, Weiqi; Choo, Kim-Kwang Raymond and Zou, Deqing (2021). Ethereum smart contract security research: survey and future research opportunities. Frontiers of Computer Science, 15(2), 152802–. http://doi:10.1007/s11704-020-9284-9

[39] Lu Wang; Longqin Xu; Zhiying Zheng; Shuangyin Liu; Xiangtong Li; Liang Cao; Jingbin Li and Chuanheng Sun; (2021). Smart Contract-Based Agricultural Food Supply Chain Traceability. IEEE Access. http://doi:10.1109/access.2021.3050112

[40] Bhaskara S. Egala; Ashok K. Pradhan; Venkataramana Badarla and Saraju P. Mohanty; (2021). Fortified-Chain: A Blockchain-Based Framework for Security and Privacy-Assured Internet of Medical Things With Effective Access Control. IEEE Internet of Things Journal. http://doi:10.1109/jiot.2021.3058946

[41] Sowmya, G. and Sridevi, R. (2025). *SmartScan: A Comprehensive Framework for Efficient and Optimized Formal Verification of Complex Blockchain Smart Contracts.* Journal of Theoretical and Applied Information Technology, 103(3), pp. 814-834. Available at: https://www.jatit.org/volumes/Vol103No3/4Vol103No3.pdf

[42] Krupp, J. and Rossow, C., 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. *Proceedings of the 27th USENIX Security Symposium*, pp.1317-1333.

[43] Luu, L., Chu, D.H., Olickel, H., Saxena, P. and Hobor, A., 2016. Making Smart Contracts Smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pp.254-269.

[44] Feist, J., Grieco, G. and Groce, A., 2019. Slither: A Static Analysis Framework for Smart Contracts. *Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '19)*, pp.8-15.

[45 Kalra, S., Goel, S., Dhawan, M. and Sharma, S., 2018. Zeus: Analyzing Safety of Smart Contracts. *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS '18)*, pp.1-15.

[46] Jin, J., Zhan, W., Li, H., Ding, Y., & Li, J. (2024). A dynamic behavior verification method for composite smart contracts based on model checking. *Mathematics*, 12(15), 2431.

[47] Sorensen, D. (2024). Towards formally specifying and verifying smart contract upgrades in Coq. *In 5th Int'l Workshop on Formal Methods for Blockchains (FMBC 2024)*, OASIcs Volume 118, pp. 7:1-7:14.

The SmartScan framework and the vulnerability detection code implementations are publicly available at https://github.com/Sowmya/SmartScan to make our work accessible for future research, reuse, validation, and real-world adoption.

## Appendix A: DeFiLending application (used for formal verification of smart contracts using smartscan)

Deposit - Allows users to deposit cryptocurrency assets.

Withdraw - Allows users to withdraw their deposited funds if certain conditions are met.

Borrow - Allows users to borrow assets based on the value of their deposits, subject to a collateral requirement.

Repay - Allows users to repay their borrowed funds.

```
pragma solidity ^0.8.0;
contract DeFiLending {
    mapping(address => uint256) public deposits;
    mapping(address => uint256) public debts;
    uint256 public constant collateralFactor = 150; // 150%
    function deposit() external payable {
        deposits[msg.sender] += msg.value;
    }
    function withdraw(uint256 amount) external {
        require(deposits[msg.sender] >= amount, "Insufficient balance");
        require(debts[msg.sender] == 0, "Debt exists, cannot withdraw");
        deposits[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }
    function borrow(uint256 amount) external {
        uint256 collateralRequired = (amount * collateralFactor) / 100;
        require(deposits[msg.sender] >= collateralRequired, "Insufficient collateral");
        debts[msg.sender] += amount;
        payable(msg.sender).transfer(amount);
    }
```

```
    function repay() external payable {
        require(msg.value >= debts[msg.sender], "Repay full debt");
        debts[msg.sender] -= msg.value;
    }
}
```

# Appendix B: SmartScan's Vulnerability Detection Process for Diversified Vulnerabilities

**DelegateCall Exploit Vulnerability**

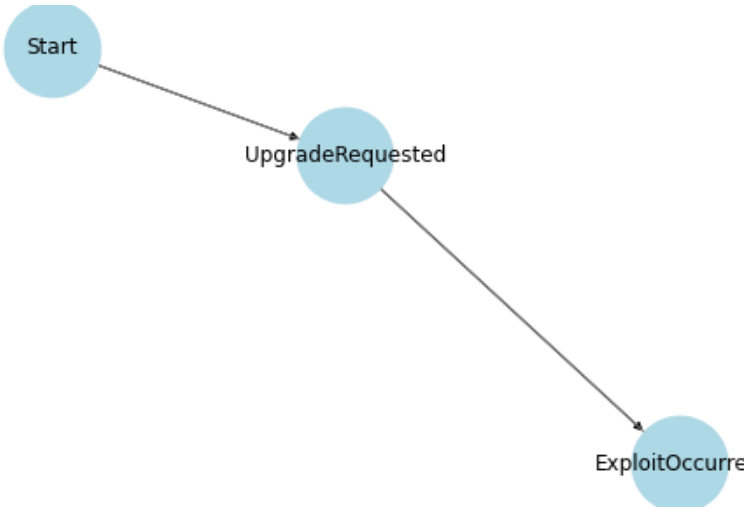


Figure 7: FSM Diagram for delegatecall exploit vulnerability

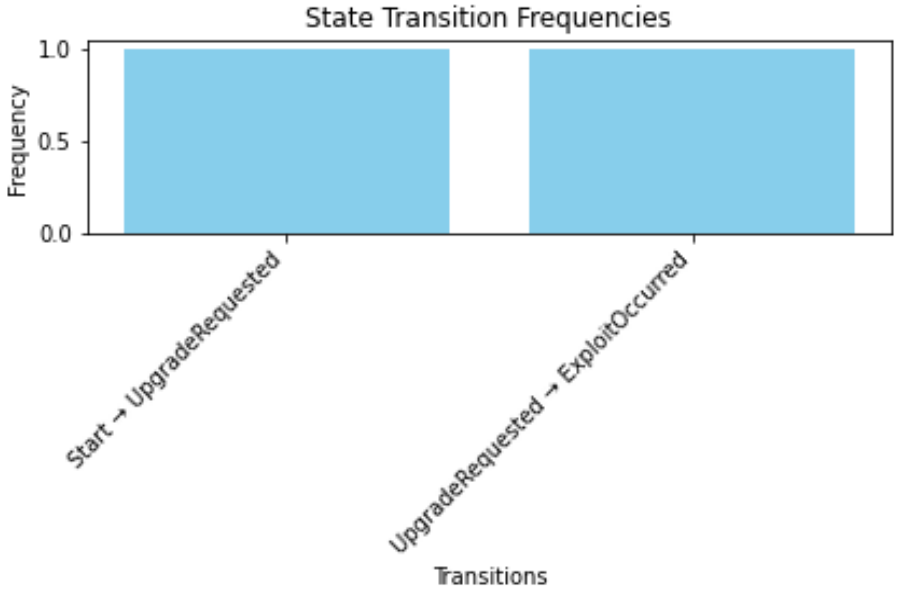


Figure 8: State transition frequencies diagram for delegatecall exploit vulnerability

**Denial of Service (DOS) via Gas Limit Vulnerability**

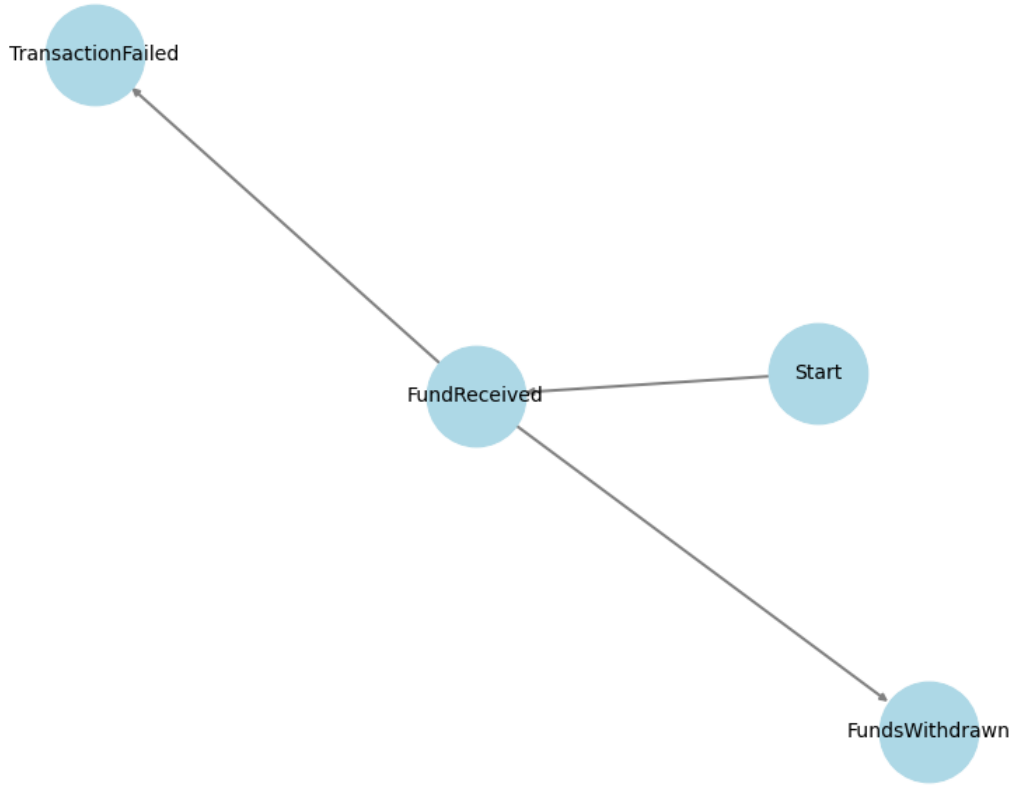


Figure 9: FSM diagram for denial of service (DOS) via gas limit vulnerability

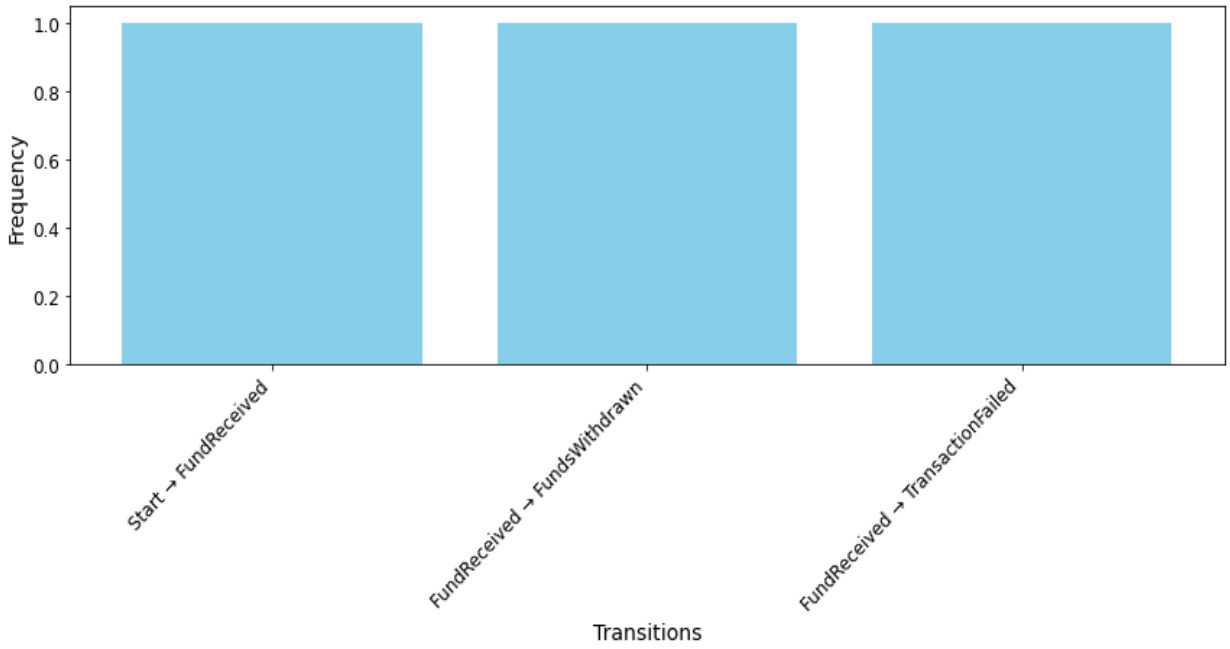


Figure 10: State transition frequencies diagram for denial of service (DOS) via gas limit vulnerability

**Integer Overflow & Underflow Vulnerability**

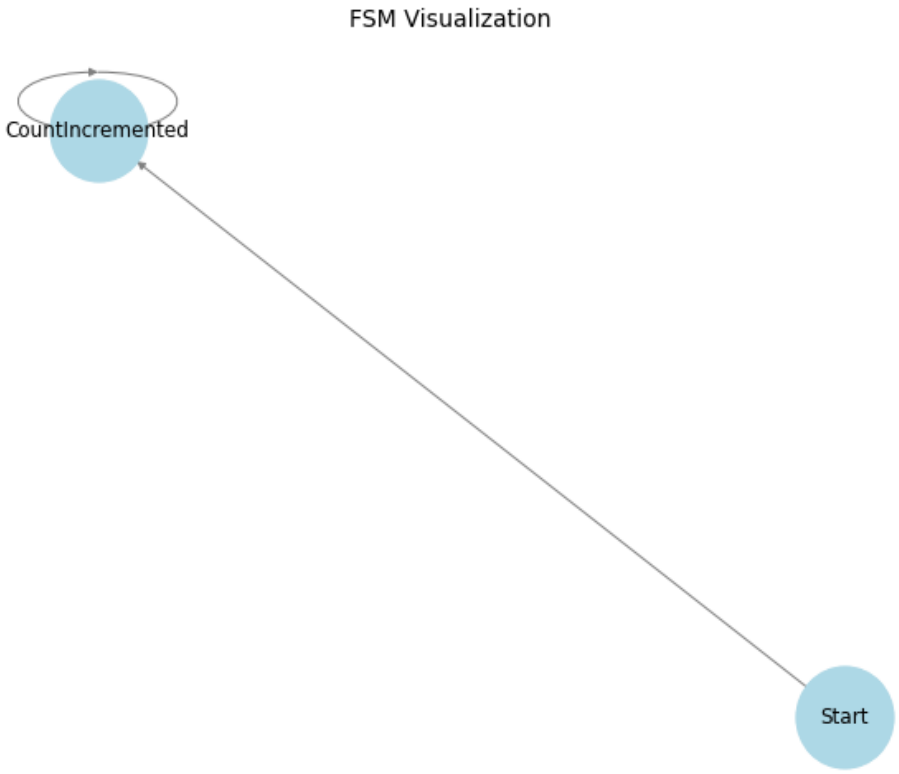


Figure 11: FSM Diagram for Integer overflow & underflow vulnerability

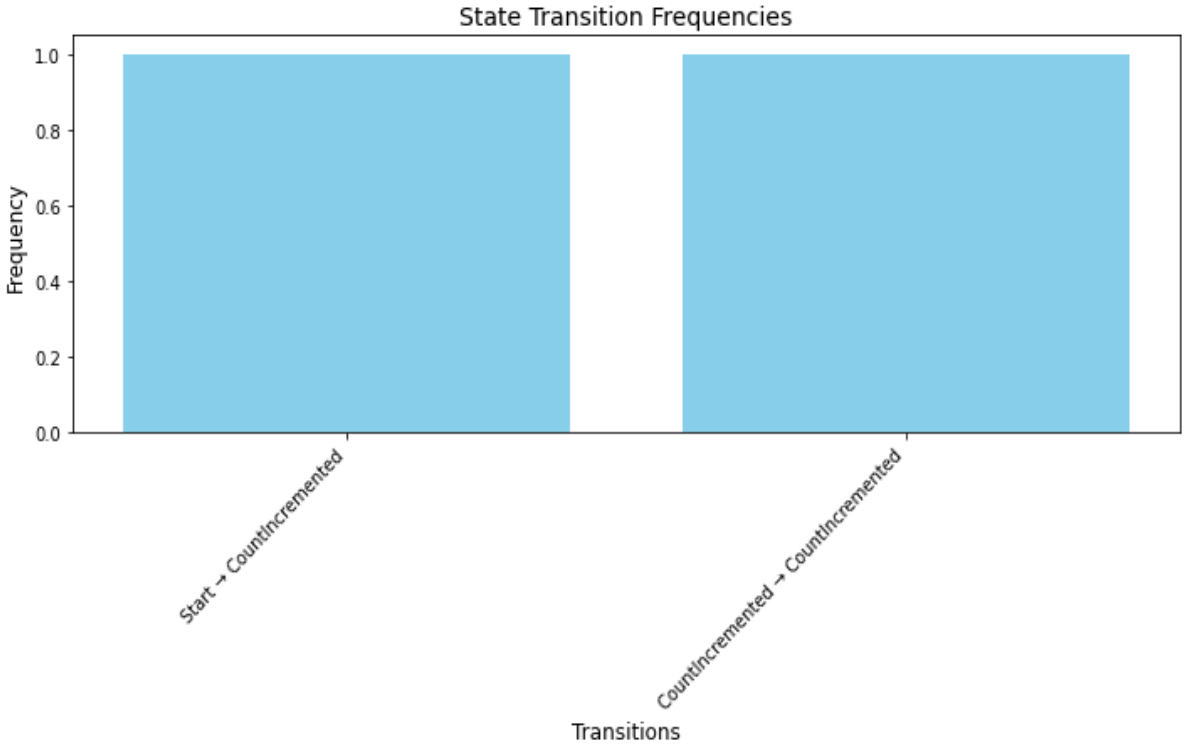


Figure 12: State transition frequencies diagram for integer overflow & underflow vulnerability

**Reentrancy attack vulnerability**

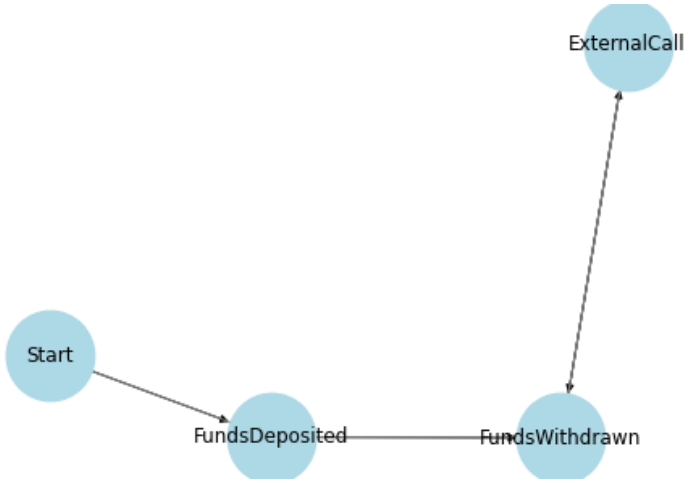


Figure 13: FSM diagram for re-entrancy attack vulnerability

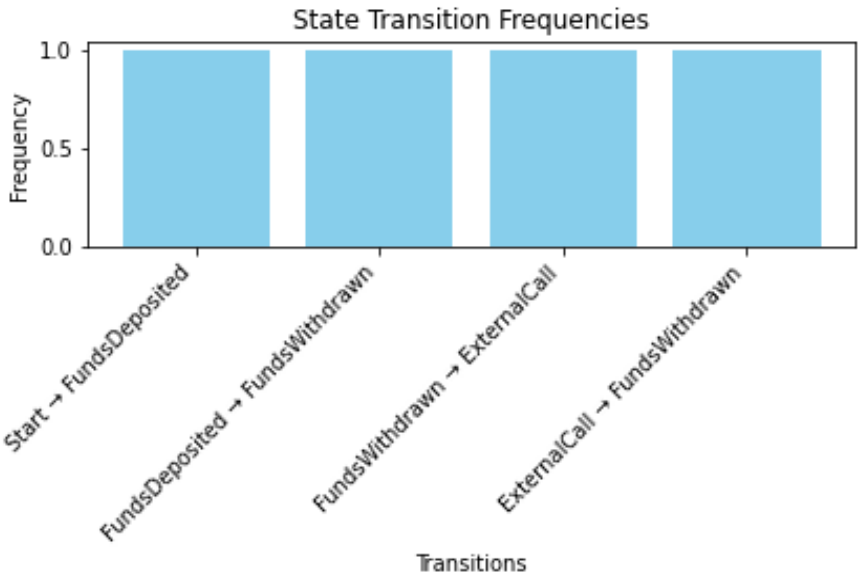


Figure 14: State transition frequencies diagram for reentrancy attack vulnerability

**Tx.Origin authentication bypass vulnerability**

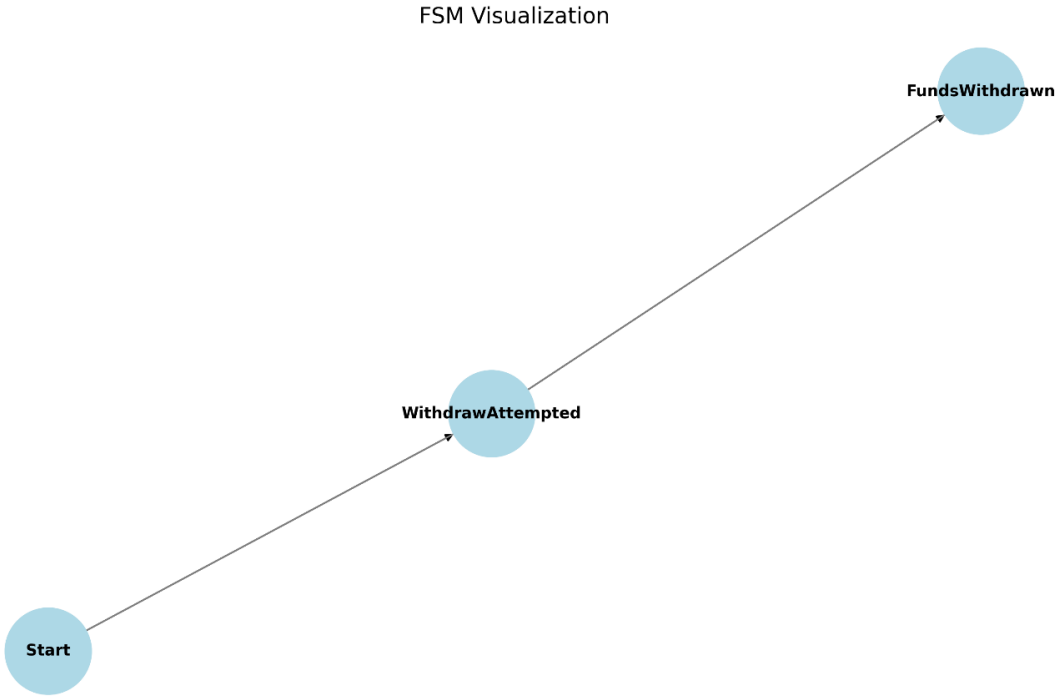


Figure 15: FSM diagram for tx. origin authentication bypass vulnerability

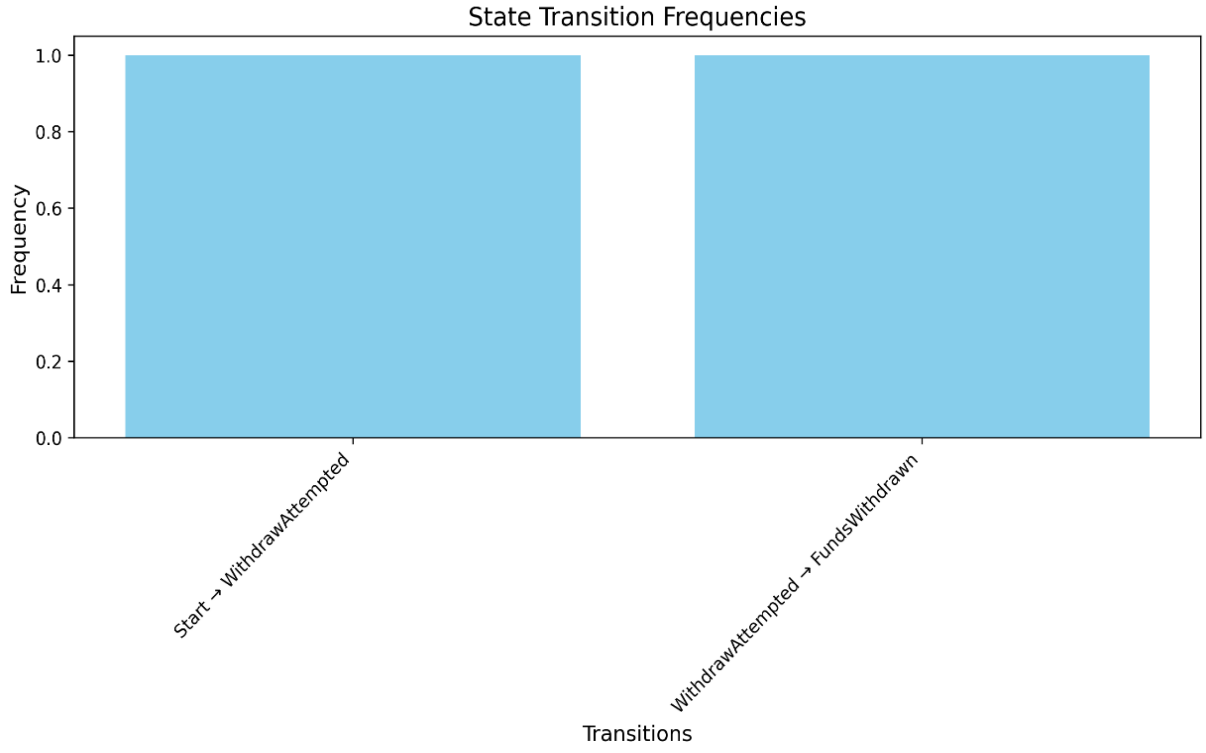


Figure 16: State transition frequencies diagram for tx.origin authentication bypass vulnerability

**Uninitialized storage pointer vulnerability**

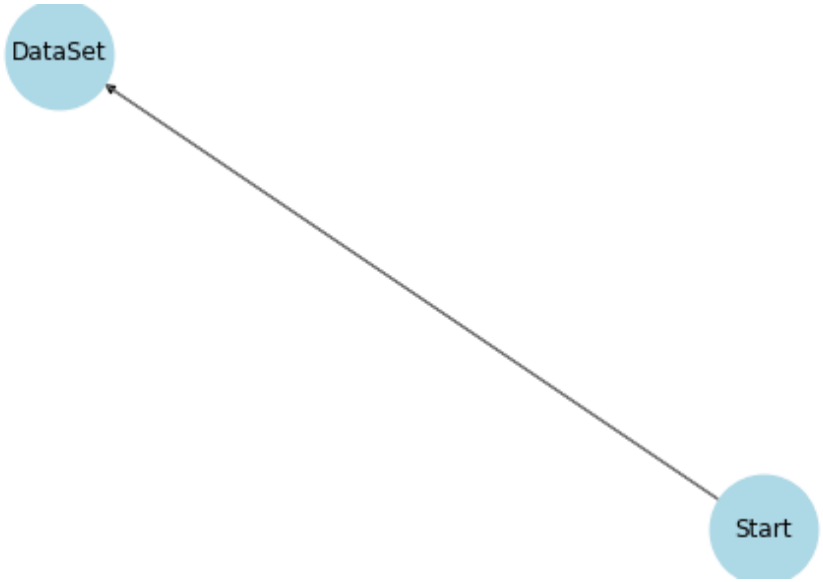


Figure 17: FSM Diagram for uninitialized storage pointer vulnerability

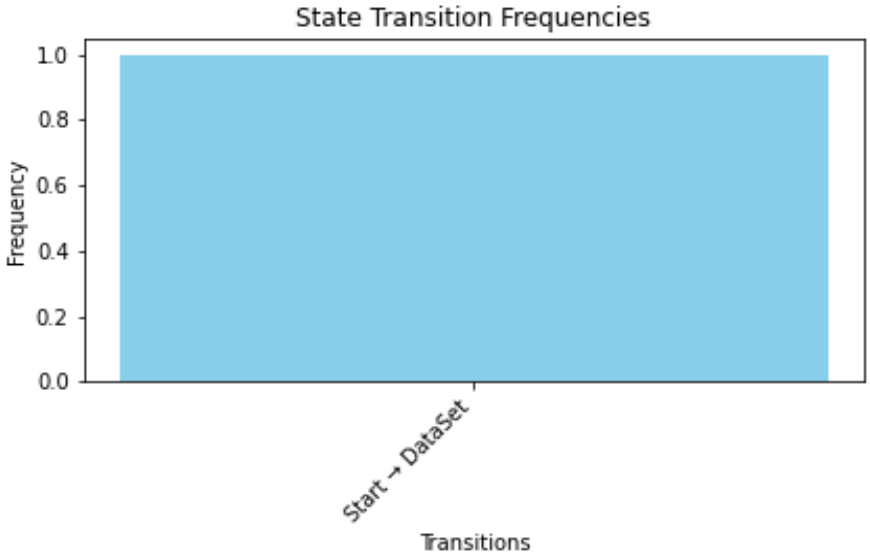


Figure 18: State transition frequencies diagram for uninitialized storage pointer vulnerability

**Unprotected self-destruct vulnerability**

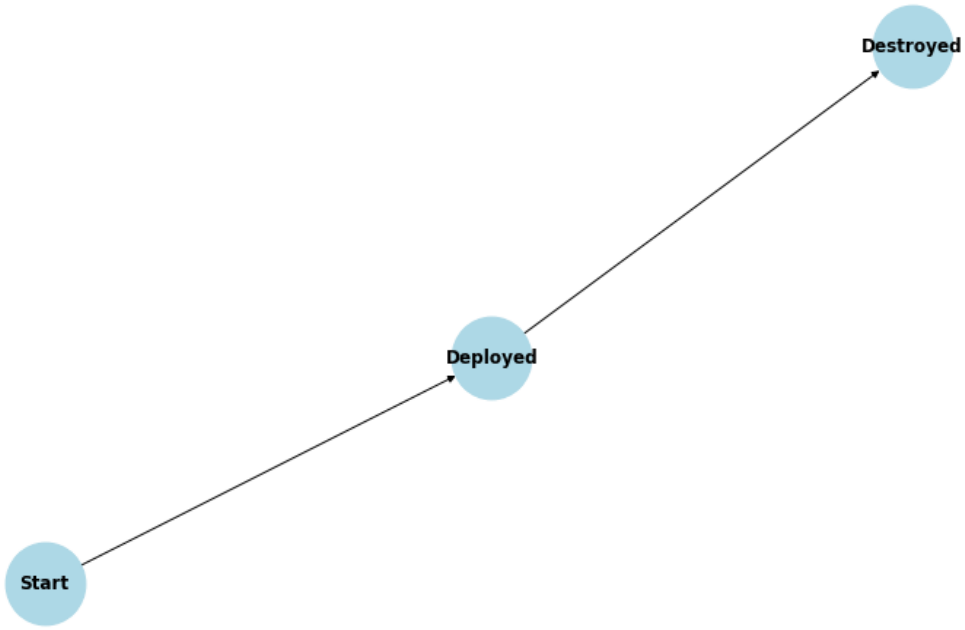


Figure 19: FSM Diagram for unprotected self-destruct vulnerability

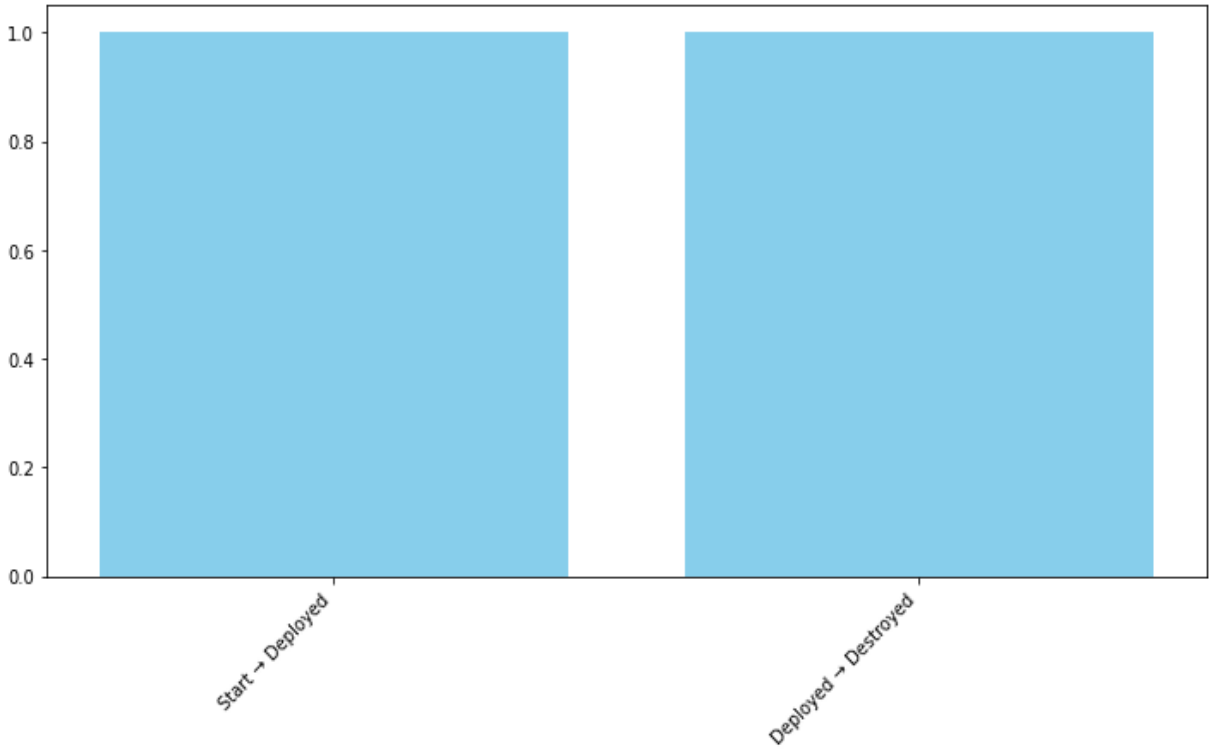


Figure 20: State transition frequencies diagram for unprotected self-destruct vulnerability

**Short Address Attack Vulnerability**

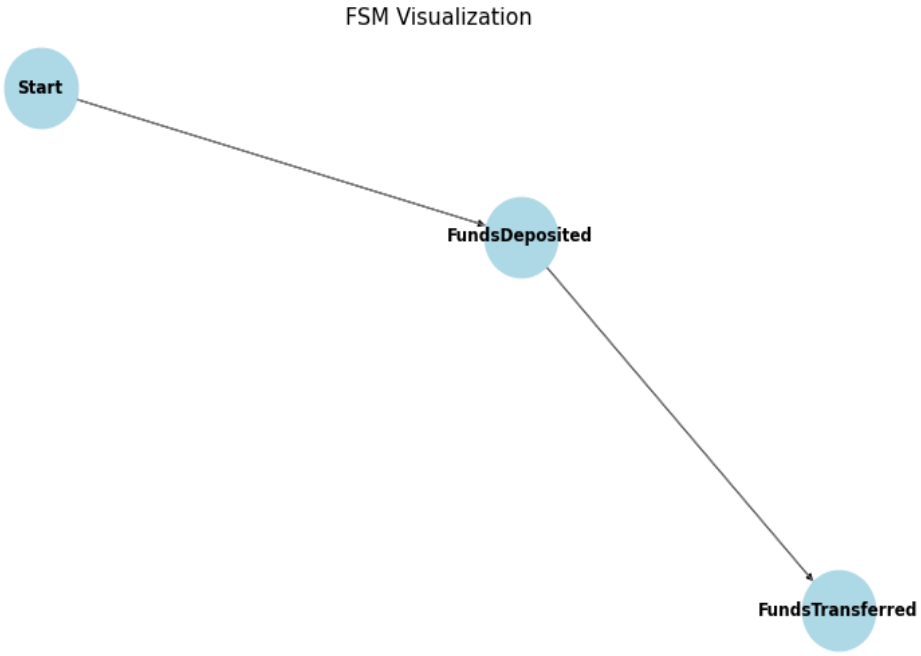


Figure 21: FSM diagram for short address attack vulnerability

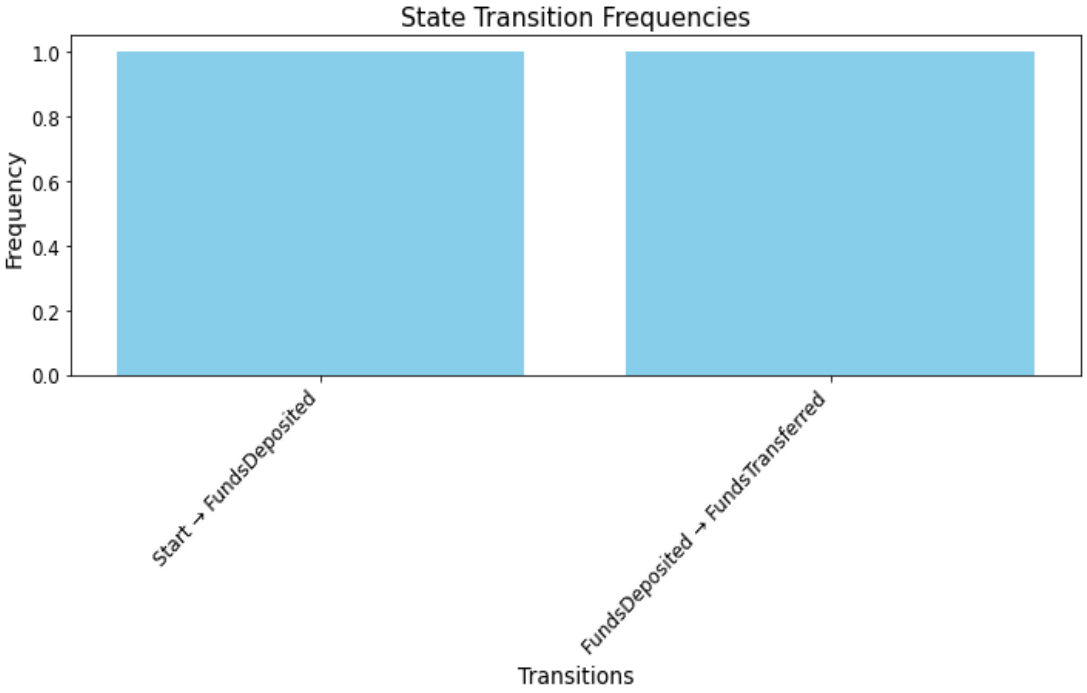


Figure 22: State transition frequencies diagram for short address attack vulnerability

**Timestamp dependency vulnerability**

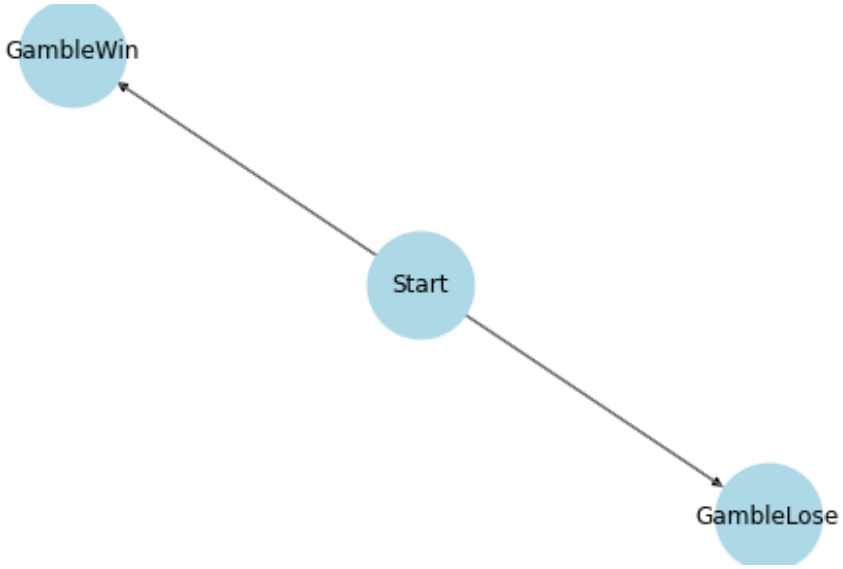


Figure 23: FSM diagram for timestamp dependency vulnerability

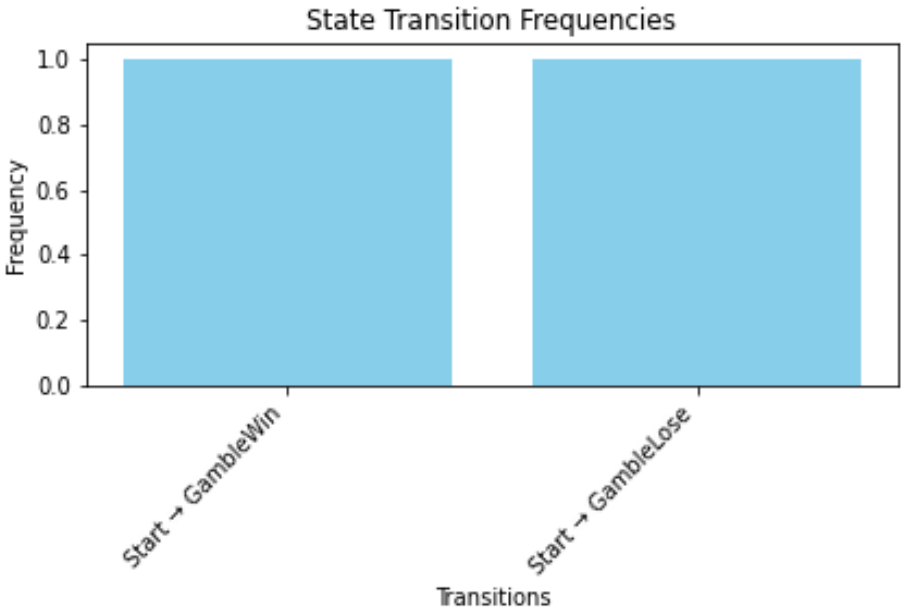


Figure 24: State transition frequencies diagram for timestamp dependency vulnerability

**Block hash manipulation vulnerability**

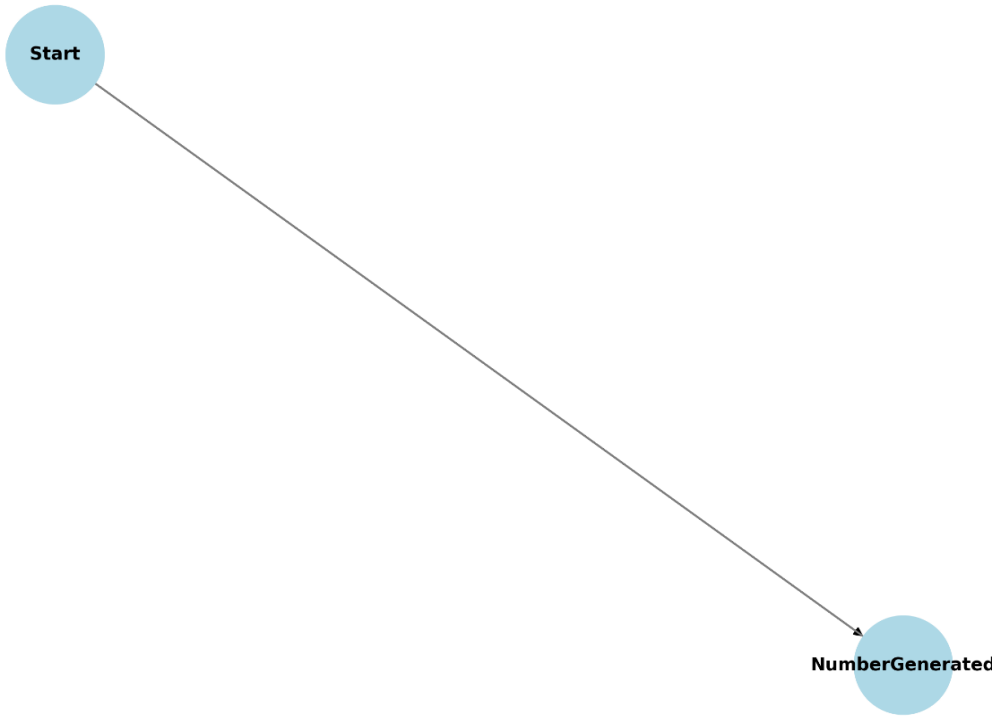


Figure 25: FSM diagram for block hash manipulation vulnerability

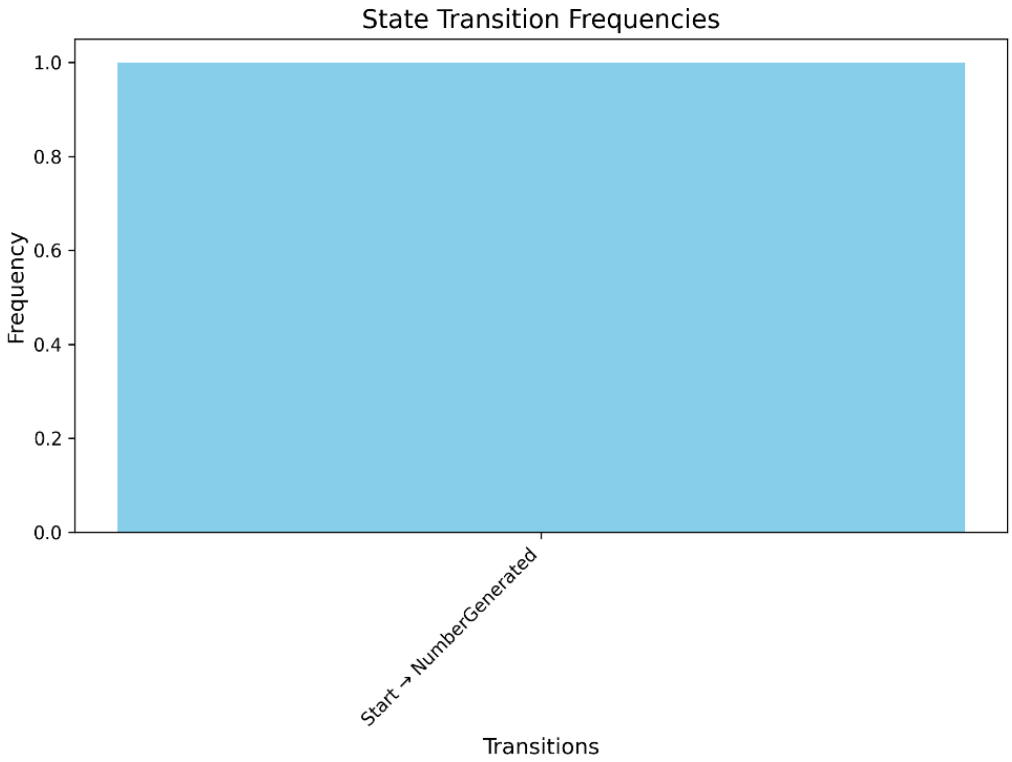


Figure 26: State transaction frequencies diagram for block hash manipulation vulnerability

**Front-running attack vulnerability**

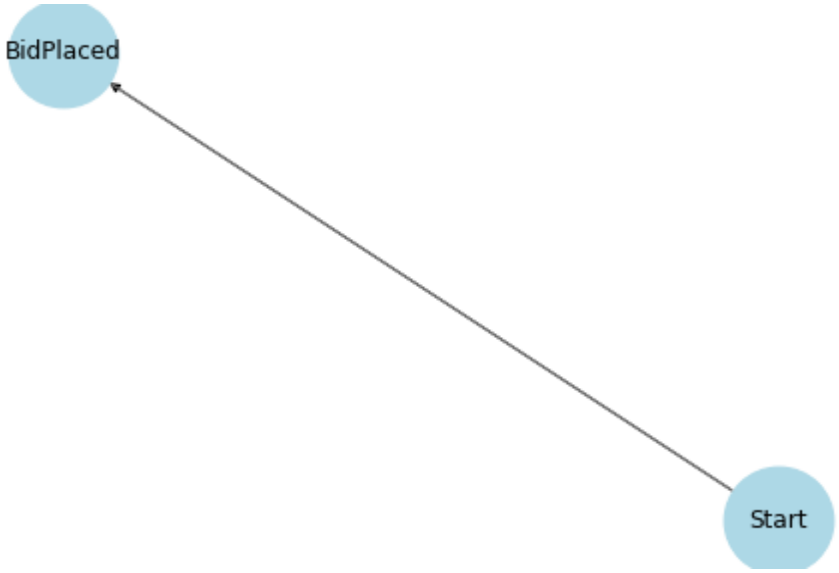


Figure 27: FSM diagram for front-running attack vulnerability

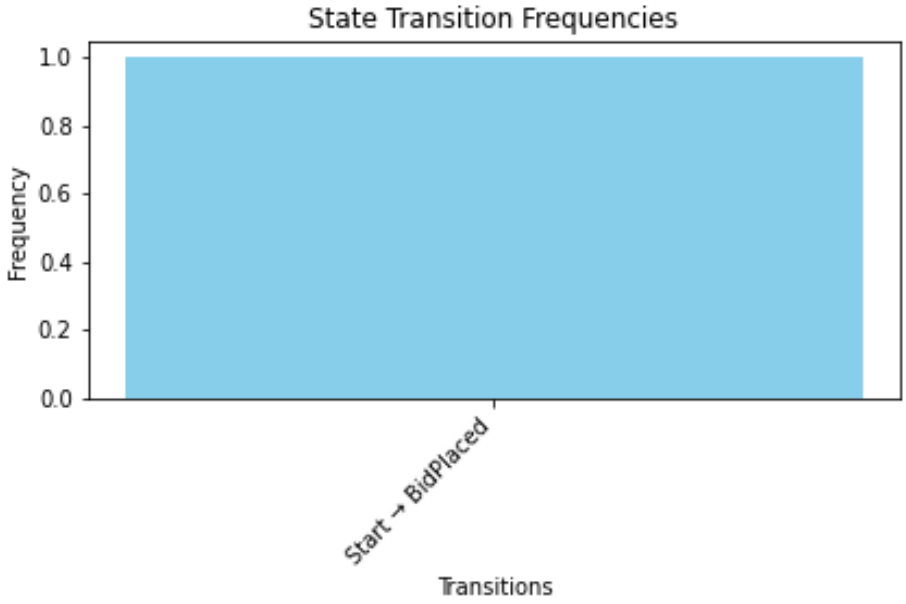


Figure 28: State transition frequencies diagram for front-running attack vulnerability

**Weak randomness (predictable random numbers) vulnerability**


Figure 29: FSM diagram for weak randomness (predictable random numbers) vulnerability

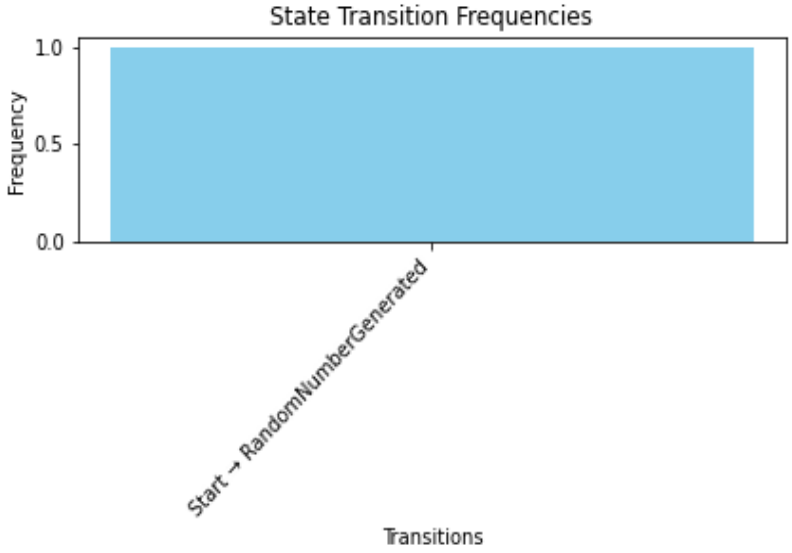


Figure 30: State transition frequencies diagram for weak randomness (predictable random numbers) vulnerability