VeriChain: A Hybrid Formal Verification Approach Using Control Flow, Symbolic Execution, and Static Analysis for Smart Contract Vulnerability Detection

Vankudoth Ramesh[1*], K. Govardhan Reddy[2]
[1]Research Scholar, Department of CSE, JNTUA, Anantapur, India.
[2]Professor, Department of CSE, G. Pulla Reddy Engineering College, Kurnool, India.
E-mail: govardhan.cse@gprec.ac.in, v.ramesh406@gmail.com
[*]Corresponding author

Keywords: smart contract security, formal verification, symbolic execution, control flow analysis, vulnerability detection

Received: May 16, 2025

Smart contracts facilitate the trustless and automated execution of agreements on blockchains, however several programming patterns in them (such as re-entrancy, integer overflow/underflow, access control) have led to substantial thefts due to vulnerabilities. Current verification tools (e.g., Mythril, Oyente, and Securify) are essentially based on symbolic execution, taint analysis, and pattern matching with high falsepositives as well as large time cost, poor scalability for complex contracts. This paper develops VeriChain, a hybrid verification framework that combines (i) lexicon/syntax parsing for abstract syntax tree (AST), (ii) Control Flow Graph (CFG) generation for path and dependency representation, (iii) CFG-guided symbolic execution with model checking to search feasible execution paths and (iv) rule-based static analysis towards identifying/characterizing vulnerability patterns. VeriChain was tested on N benchmark smart contracts (including re-entrancy, arithmetic and access-control vulnerabilities) and contrasted with Mythril, Oyente, and Securify. Experiments show that VeriChain enables 98.3% detection accuracy with only 1 false positive (and no false negative in our testing set) and finishes analysis within 2.3 seconds, surpassing state-of-the-art tools in both precision and execution efficiency. VeriChain also outputs a structured security report to categorise detected flaws by severity and to offer developers traces of execution as well as mitigation advice, enabling realistic pre-deployment audit of blockchain smart contract.

Povzetek: Predstavljen je napreden sistem za odkrivanje ranljivosti v pametnih pogodbah, ki deluje natančneje in hitreje kot obstoječe rešitve.

1 Introduction

Blockchain technology has accumulated broader acceptance in multiple structures like NFT and has increased the use of cryptocurrencies like Bitcoin and Ether. Smart contracts — self-executing programs that run on blockchain networks — are essential for automating transactions and enforcing agreements without intermediaries. However, security flaws in smart contracts have resulted in severe financial losses, as demonstrated through notorious exploits such as the DAO hack and reentrancy attacks [1], [2]. Its immutable nature causes security analysis to be an essential step before deploying smart contracts. To verify such smart contracts, we primarily use manual audits, heuristic-based detection, and formal verification-rich techniques, none of which, however, can provide large-scale, accurate, and efficient detection.

In recent studies, smart contract security has also been improved using static analysis, symbolic execution, and machine learning-based vulnerability detection. We use tools that do symbolic execution and formal methods (like Mythril, Oyente, and Securify) to detect security bugs [3], [4]. However, these tools have high false positive rates, long execution times, and incomplete coverage of target vulnerability types that restrict their practical usability. Mythril and Oyente, for example, are based on symbolic execution but encounter path explosion issues for non-trivial contracts. While Securify is precise, it introduces substantial computation overhead and is not suitable for verifying smart contracts on a large scale. These limitations create a scope for research on designing a fast, accurate, scalable, innovative contract verification framework that integrates multiple verification techniques for better reliability.

To tackle these obstacles, we propose VeriChain. This formal verification framework employs a combination of Control Flow Graph (CFG) analysis, symbolic execution, and static analysis to improve the identification of vulnerabilities in smart contracts. VeriChain aims to avoid false positives and enhance detection accuracy and execution efficiency compared to available tools. The innovation of this study is that it proposes a hybrid verification method that combines CFG-based execution

path analysis for contract function dependency tracking, symbolic execution for execution scenario analysis, and application of static analysis with rule-based vulnerability detection. Combining these methods gives a holistic security analysis and reduces the associated security risk pre-deployment of the contract using the VeriChain framework. The main Contributions of this research include the following:

- A hybrid framework (VeriChain) was developed that combines control flow graph (CFG) analysis, symbolic execution, and static analysis to strengthen the overall security of smart contracts.
- Optimized an execution model that increases vulnerability detection accuracy and minimizes false positives and computation costs.
- VeriChain, Mythril, Oyente, and Securify were compared experimentally, and they showed greater accuracy (98.3%) and execution time (2.3 seconds).
- We defined a structured security reporting system that classifies vulnerabilities according to severity, execution traces, and mitigation strategies.

In order to systematically assess the performance of the designed VeriChain, this paper follows the research questions as shown below:

RQ1: Does combining CFG-guided symbolic execution and rule-based static analysis lead to better detection precision on vulnerabilities at the cost of lowering numbers of false positive compared to other existing smart contract tools?

RQ2: Can VeriChain provide reduced running time as well as improved scalability in more complex smart contracts involved with multi-functions, loops and paths, comparing to the popular state-of-arts tools such as Mythril [1], Oyente [16] and Securify 3?

RQ3: Is VeriChain capable of offering a well-organized and realistic security assessment utilizing which it is possible to conduct pre-deployment auditing by correctly classifying the severity-wise vulnerabilities along with their execution traces?

The rest of this paper is organized as follows. Section 2 provides a literature review on the state-of-the-art approaches to smart contract security analysis and offers insights into the research gaps. Section 3 describes the proposed approach, including the architecture and functioning of VeriChain's verification pipeline. In section 4, we present the experimental results and compare the performance of VeriChain with existing tools based on detection accuracy, false positives, and execution time. In Section 5, we discuss the implications of the results, indicate how the current approach overcomes the limitations of previous work, and comment on the broader significance of VeriChain in the field of smart contract security. Lastly, the study's limitations are presented in Section 5.1. Section 6 concludes the paper and outlines directions for future research, including leveraging machine learning for vulnerability classification and broadening VeriChain's support to multiple blockchain platforms.

2 Related work

The current tools for the security of smart contracts still use symbolic execution, taint analysis, and formal verification, which are plagued by false positives and are inefficient. Almakhour et al. [1] examined clever contract verification techniques, found security flaws, and discussed benefits, drawbacks, and potential advancements. Garfatta et al. [2] explained the constraints, formal verification techniques, weaknesses, and possible areas for further study in blockchain intelligent contract verification. Wang et al. [3] suggested ContractWard, which uses machine learning to detect smart contract vulnerabilities quickly and accurately. Kim et al. [4] evaluated 391 articles on smart contract analysis, pointing out issues, potential vulnerability, and accuracy detection research avenues. Schiffl et al. [5] aimed to formally verify the accuracy of smart contracts through a case study, pointing forth its limits and available verification techniques.

Hajdu et al. [6] examined verification tools, used fault injection to assess bright contract flaws, and identified ways to improve blockchain dependability. Wang et al. [7] demonstrated the efficacy and efficiency of Artemis, an innovative contract verification tool that can locate a variety of flaws on 12,899 contracts. Gao et al. [8] offered SmartEmbed, a very accurate and effective automated solution for detecting smart contract bugs and clones. Ghaleb et al. [9] presented SolidiFI, a tool for assessing smart contract static analysis tools and detecting false positives and unreported problems. Duo et al. [10] used colored Petri nets to offer a layered, innovative contract security paradigm that increases automation and analysis efficiency.

Hameed et al. [11] proposed a decentralized Blockchain-based authentication scheme for IoT devices, improving performance and verifying correctness through modeling and analysis. Kabar et al. [12] suggested using QR codes and multi-level authentication to enhance security and efficiency in their blockchain-based automated check-clearing system, MudraChain. Khan et al. [13] examined smart contracts offered by blockchain, pointing out security flaws, difficulties, and unresolved problems while outlining potential study avenues. So et al. [14] presented VeriSmart. This accurate Ethereum smart contract verification improves bug identification. It addresses false alarms for arithmetic safety. Tolmach et al. [15] discussed formal models and smart contract verification methods, pointed out shortcomings, and recommended future research paths.

Kushawaha et al. [16] highlighted the difficulties and suggestions for the future when reviewing and classifying 86 Ethereum smart contract analysis tools. Afzaal et al. [17] suggested a framework for secure blockchain-based crowdsourcing that uses formal verification and emphasizes upcoming security and trust enhancements. Verma et al. [18] examined blockchain consensus techniques emphasizing security and performance, highlighting formal approaches for accuracy. Nam et al. [20] addressed the shortcomings of current methods by proposing ATL model checking for smart contract verification, supported by case examples.

Yang and Zhu [21] suggested SCSVM and SCLMF for detecting Ethereum smart contract vulnerabilities with high accuracy; further development is required. Babel et al. [22] presented CFF, a formal economic security verification tool for DeFi contracts that may be used to find smart contract flaws and potential uses. Ahmad et al. [23] addressed verification restrictions by creating a user-centric blockchain platform for GDPR compliance in multi-cloud scenarios. Khalid et al. [24] suggested a blockchain-based, scalable, and SDN-integrated IoT access control system, assessing its scalability and performance. Anas et al. [25] demonstrated the superiority of BlockASP over conventional techniques by fusing AOP and model checking for enhanced blockchain verification.

Alnuaimi et al. [26] suggested a blockchain-based approach for healthcare credentialing that improves security, automation, and transparency while evaluating security and cost. Almakhour et al. [27] used nuXmv and finite state machines to provide a model-checking method for confirming the security of composite smart contracts. Farao et al. [28] introduced INCHAIN, a blockchain-based system that addresses fraud, data transparency, and consumer identity to enhance cyber insurance. Alevizos [29] suggested potential research possibilities for a system that automates security compliance by integrating blockchain, AI, and smart contracts. Qureshi et al. [30] proposed the ChainAgile framework, which uses smart contracts and blockchain to enhance cybersecurity, transparency, and teamwork in distributed Scrum Agile development.

Deep et al. [31] suggested an innovative contract-based blockchain-based IoT security solution for data integrity, access management, and authentication. Colin et al. [32] addressed dataset standardization, accuracy, and performance in its proposed MLP-based smart contract vulnerability detection method. Khan and Namin [33] evaluated and categorized 41 smart contract vulnerability detection technologies to improve SC security and lower risks. Chen et al. [34] presented a clear, contrastive learning-based strategy that outperforms current deep learning techniques for identifying vulnerabilities in smart contracts. Chen et al. [35] introduced DCV, an automated tool that offers faster verification times than previous tools for declarative smart contracts.

Bartoletti et al. [36] suggested using benchmarks to assess the efficacy of Solvent, a tool for confirming the liquidity qualities of smart contracts. Jiao et al. [37] evaluated the weaknesses of smart contracts, examined detection methods, and suggested future lines of inquiry to improve security. Olivieri and Spoto [38] highlighted the difficulties in verifying blockchain software, pointed out the shortcomings of current methods, and made recommendations for new lines of inquiry. Chaliasos et al. [39] examined DeFi security technologies and pointed out their low efficacy. Creating tailored tools for changing threats is part of the future work. Li et al. [40] created AS-SC to simplify asset securitization contracts. However, its drawbacks include inadequate scenario coverage and dependability concerns. Future research will concentrate on enhancing verification techniques and perfecting AS-SC. Several limitations of Mythril, Oyente, and Securify have been reported in the literature, such as high false positive rates and slow execution time.

In control literature, robust/adaptive design offers some principled tools for dealing with uncertainly and nonlinearities as well as partial or limited observability which is also relevant in complex engineered systems. Output feedback synchronization for uncertain dynamics with dead-zone/sector nonlinearities demonstrate the stability of uncertain dynamics using adaptive fuzzy approximators and variable structure design when only system states are not completely measurable [44]. The aforementioned robustness of the fixed-time synchronization can also be stretched to a practical one in that bounded time-convergence is guaranteed with respect to those uncertainties in fractional-orders [45]. (Theoretical advances in optimizing learning of controlled dynamic systems span Erickson et al, 1976 to Floares}, A generalized approach lo the (1994) as well as robust control literature including Tallamraju eta!1988). Adaptive backstepping based on Lyapunov stability offers a systematic method for stabilization of uncertain SISO nonlinear systems [46] whereas robust neural adaptive control extends adaptation into uncertain MIMO dynamics [47]. High-gain observer-based adaptive fuzzy control also compensates unmeasured states based on observer-driven adaptation [48].

Table 1: Summary of representative smart contract vulnerability detection and verification approaches

Tool / Method	**Core Technique(s)**	**Strengths (Typical)**	**Key Limitations (Typical)**	**ReObserved Performance Indicators***
Mythril	Symbolic execution + taint analysis	Widely used; detects common EVM-level issues	Path explosion; higher false	Moderate accuracy; FP can be high; scalability

			positives on complex paths	limited by path exploration
Oyente	Symbolic execution + constraint solving	Early symbolic tool; finds control-flow related bugs	High false positives; misses some modern patterns; slower on large contracts	Lower precision in complex contracts; scalability limited
Securify	Pattern-based compliance + semantic rules	High precision for known patterns; strong rule checking	Higher computation overhead; limited flexibility for unseen patterns	Good precision; slower runtime; depends on rule coverage
Slither	Static analysis (AST/IR) + rule detectors	Fast; good for code-quality/security patterns	Limited for deep path-dependent vulnerabilities	Very fast; precision depends on rule quality; scalable
SmartCheck	Static pattern matching	Easy-to-use; detects known anti-patterns	Can miss deep semantic bugs; pattern-limited	Fast; moderate accuracy; limited on complex logic
VeriSmart	Formal verification / safety property checking	High precision for targeted properties	Requires formal specs; limited general vulnerability breadth	High precision on supported checks; overhead depends on properties
ML-based detectors (e.g., embeddings/classifiers)	Learning-based classification	Learns from data; can generalize across patterns	Dataset dependence; interpretability and FP risk	Accuracy varies by dataset; needs benchmark standardization

Research gaps exist in utilizing multidimensional verification techniques to enhance accuracy and scalability. VeriChain fills in these missing pieces by implementing analysis, symbolic execution, and static analysis to guarantee efficient and accurate vulnerability discovery.

3 Proposed framework

The proposed framework, named VeriChain, is a formal verification framework for identifying vulnerabilities in employed blockchain smart contracts, combining symbolic execution, control flow analysis, and static analysis to help find security vulnerabilities like reentrancy attacks, integer overflows, unauthorized access, and transaction-order dependence. Since smart contracts are immutable when deployed, VeriChain ensures they are correctly and thoroughly tested for security vulnerabilities to avoid exploits within Dapps. Based on the tokenization of smart contracts, this framework first generates tokens by understanding the terminology and generating tokens before parsing the Contract into hierarchical form, known as an Abstract Syntax Tree (AST). (This enables systematic analysis of the contract's functions, variables, and dependencies). It then constructs a Control Flow Graph (CFG), which represents all the paths of execution, the function calls, and the logical branches the contract can take. With a clear grasp of execution flows, VeriChain can identify contract logic and control dependencies vulnerabilities.

VeriChain's architecture, shown in Figure 1, comprises multiple interlinked components that work to detect vulnerabilities in smart contracts. This process starts with Smart Contract Input Processing, which involves parsing and structuring Solidity source code for further analysis. Then, Lexical & Syntax Analysis takes that contract and returns an AST (Abstract Syntax Tree), which is a tree representation of contract elements. The Control Flow Graph (CFG) Generation module maps execution paths, function calls, and branching logic to recognize potential vulnerabilities.

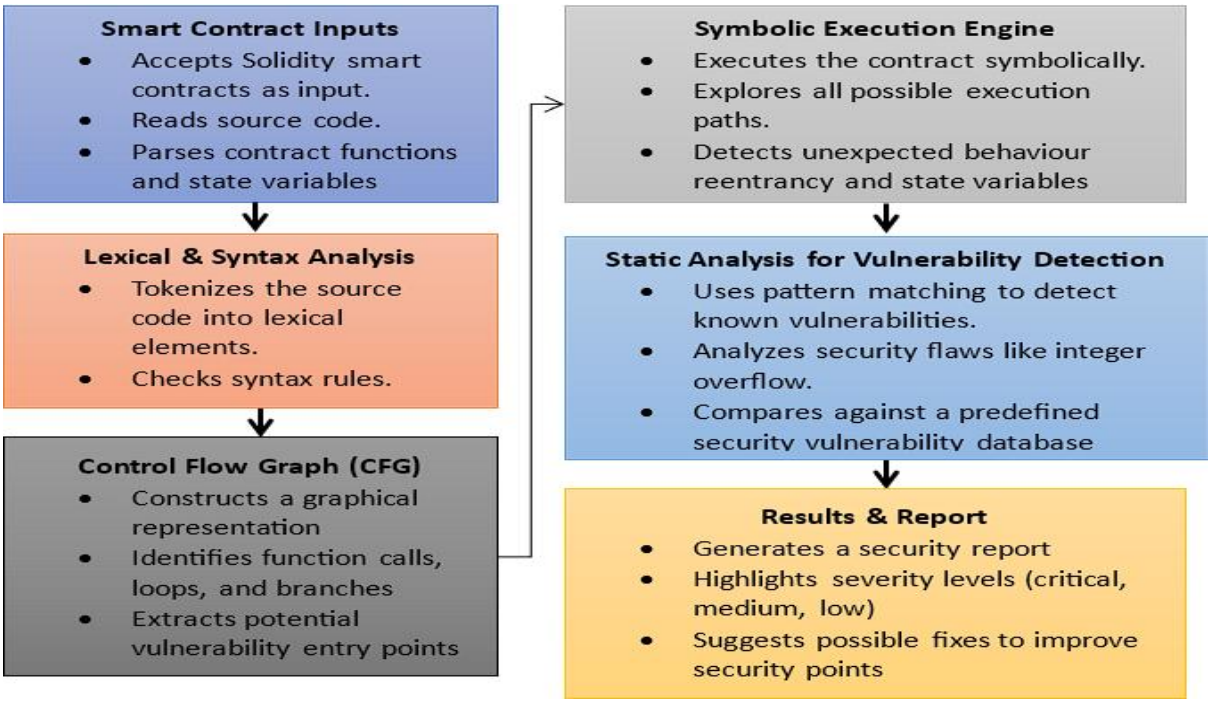

Figure 1: Architecture of VeriChain – a formal verification framework for smart contract security analysis

VeriChain utilizes its Symbolic Execution Engine to verify contracts by implicitly exploring potential states thoroughly that the contracts can reach, independent of known inputs. Next, static analysis is applied, where the framework scans for vulnerabilities (reentrancy, integer overflows, and access control violations) according to the predefined security rules. Finally, Results & Reporting provides a structured security assessment, listing vulnerabilities within ranks of risk and suggesting remediation to improve smart contract security.

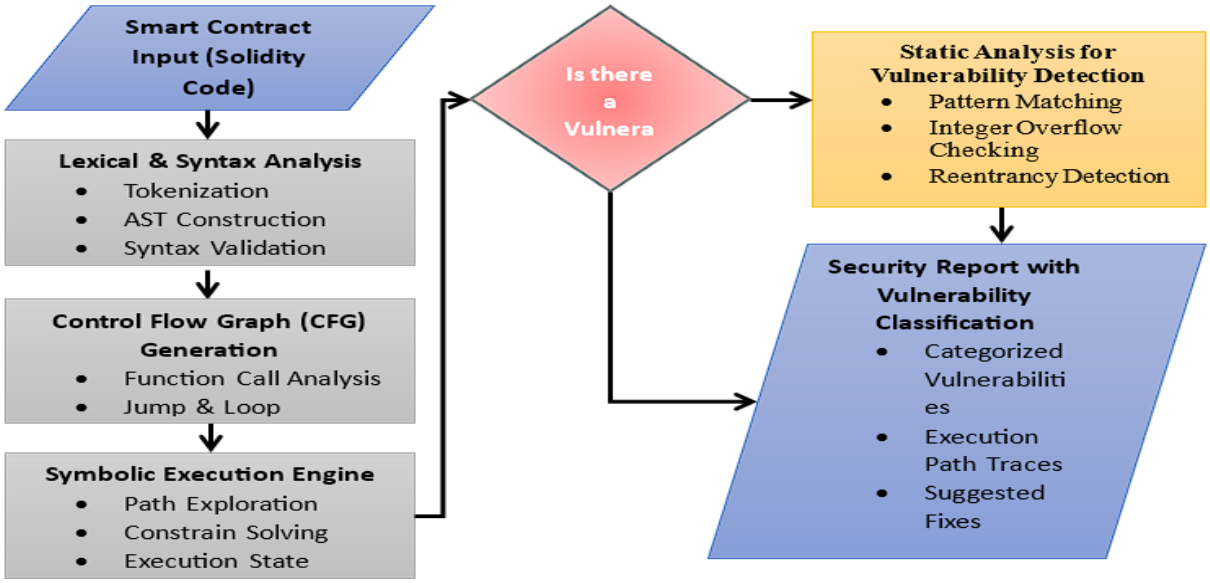

Figure 2: VeriChain flowchart – step-by-step process from smart contract input to security report generation

Figure 2 illustrates the workflow of VeriChain, detailing the step-by-step verification process for smart contract security analysis. The process begins with Smart Contract Input, where Solidity code is parsed for further study. Lexical & Syntax Analysis tokenizes the code, constructs the Abstract Syntax Tree (AST), and validates syntax. The Control Flow Graph (CFG) Generation module maps execution paths, function calls, and loops. The Symbolic Execution Engine then explores possible execution scenarios and detects vulnerabilities. A decision point determines if vulnerabilities exist, leading to Static Analysis, which detects security flaws. The process concludes with a Security Report, categorizing vulnerabilities and suggesting fixes. Table 2 shows the notations used in the proposed system.

Table 2: Notations used

Notation	Definition
S	Smart contract source code in Solidity
$L(S)$	Lexical representation of smart contract after tokenization
$T\ (S)$	Tokenized representation of the smart contract
$G = (V, E)$	Abstract Syntax Tree (AST) representation of the smart contract
V	Set of nodes in AST, each representing a syntax element
E	Set of edges representing parent-child relationships in AST
$G_f = (V_f, E_f)$	AST subgraph for function f in the smart contract
$C\ (S)$	Compiled EVM bytecode of the smart contract
$B = \{b_1, b_2, \dots, b_p\}$	Sequence of EVM opcodes
$S_k = \{s_1, s_2, \dots, s_q\}$	EVM execution stack during contract execution
E_f	Set of function entry points in the contract
$G_{CFG} = (V_{CFG}, E_{CFG})$	Control Flow Graph (CFG) of the smart contract
V_{CFG}	Set of basic blocks in CFG
E_{CFG}	Set of directed edges representing execution flow in CFG
B_i	Basic block containing sequential instructions
$E_{i,k}$	Directed edge in CFG between two basic blocks
$D\ (v_n)$	Dominance set for node v_n in CFG
$X = \{x_1, x_2, \dots, x_n\}$	Set of symbolic variables used in symbolic execution
$s_i = \{M_i, S_i, \dots, E_i\}$	Program state during symbolic execution
$T(s_i, I_k)$	Transition function applying instruction I_k to state s_i
$C = \{c_1, c_2, \dots, c_m\}$	Set of constraints generated during symbolic execution
$P = \{p_1, p_2, \dots, p_r\}$	Set of all feasible execution paths
$Z3(C_p)$	Constraint solver determining satisfiability of constraint set C_p
$V_P: v_i \to \mathbb{V}$	Mapping function from AST node $v_i$to vulnerability set $\mathbb{V}$
$T\ (x)$	Taint propagation set for variable X
$E(f_i)$	Set of external calls made by function f_i
$R(S)$	Set of detected vulnerabilities in smart contract S
$L(V_i)$	Severity level of detected vulnerability V_i

$G(V_i)$	Execution path leading to vulnerability V_i
$F(V_i)$	Suggested fix for vulnerability V_i
$P(V)$	Proportion of vulnerable functions in the contract

3.1 Smart contract input

VeriChain's formal verification process starts from the smart contract input stage, where it parses the given Solidity contract and transforms it into a structured representation for later analysis Let S be a smart contract source code, where $S = \{L_1, L_2, \dots L_n\}$ and each L_i represents a lexical entity such as a function, variable, or control structure. We used the following function to map the input contract into a tokenized representation as in Eq. 1.

$$T(S) = \{t_1, t_2, \dots t_m\},\ t_i \in \mathbb{T} \quad (1)$$

Where $\mathbb{T}$ is the predefined set of Solidity token types such as keywords, identifiers, operators, and data types. After it is tokenized, the contract is transformed into a hierarchical structure in the form of an abstract syntax tree (AST) $G = (V, E)$, where V is the set of syntax nodes and E It is the parent-child relationship between language constructs. Every contract functions f_k The input is expressed as a subtree in Eq. 2.

$$G_{f_k} = (V_k, E_k),\ V_k \subseteq V\ ,\ E_k \subseteq E \quad (2)$$

Providing modular organization. The AST similarly provides a structured means of traversing through the contract logic, enabling extraction of key contract features like state variables, function calls, modifiers, and inheritance relationships. This is followed by the byte code transformation phase, where the Solidity source code maps into its equivalent EVM bytecode. The compilation function CC is defined by Eq. 3.

$$C: S \to B,\ B = \{b_1, b_2, \dots b_p\} \quad (3)$$

Where B It is the ordered sequence of EVM opcodes. They precisely determine how the contract will run in the Ethereum Virtual Machine (EVM) via opcodes like PUSH, CALL, SLOAD, and JUMP. Each opcode b_i Works on the EVM stack, which a sequence of stack operations can represent as in Eq. 4.

$$S_k = \{s_1, s_2, \dots s_q\},\ sj\ \in\ \mathbb{S} \quad (4)$$

Where $\mathbb{S}$ It is a set of stack operations, including push, pop, arithmetic, and logical operations. To cover all cases, VeriChain uses the entry points of the smart contract — referred to as E_f Which consists of the contract's public and external functions, as in Eq. 5.

$$E_f = \{f_1, f_2, \dots f_r\},\ f_i \in S \quad (5)$$

These entry points represent the interaction surface of the contract and are used as starting nodes for control flow and symbolic execution analysis. Fingering the critical entry points helps you identify the attack vectors, including reentrancy, unchecked external calls, and access control violations. VeriChain aims to provide accurate detection of vulnerabilities and to improve the efficiency of future formal verification phases by transforming the input of a smart contract into lexical, syntactic, and execution-ready representations.

3.2 Lexical & syntax analysis

The lexical and syntax analysis forms a significant part of the transformation process in the VeriChain framework that converts smart contract source code into a structured representation suitable for formal verification. Let be the Solidity source code of a smart contract, which is a sequence of lexemes that we denote as $S = \{l_1, l_2, \dots l_n\}$, $where\ each\ l_i$, where a meaningful token is parsed from the contract's syntax. The lexical analyzer, L, translates this sequence into a collection of classified tokens as in Eq. 6.

$$L(S) = \{t_1, t_2, \dots t_m\},\ t_i \in T \quad (6)$$

Where T is the defined set of token types (e.g., keywords, identifiers, operators, literals), the mapping function L can be defined as cap L: cap S goes to cap T, so each token follows the Solidity grammar. Parsed tokens from lexical analysis are syntactically validated by constructing an Abstract Syntax Tree (AST). The AST is a directed acyclic graph (DAG), denoted by cap G equals open paren cap V, cap E, close paren where cap V represents nodes of syntax trees and E represents hierarchical relationships among language constructs. Each function f in the smart contract is a subtree in G, formulated as Eq. 7.

$$G_f = (V_f, E_f),\ V_f \subseteq V\ ,\ E_f \subseteq E \quad (7)$$

Where G_f Captures the function-level structures, such as variable declarations, control flow statements, and function calls. The well-formedness of G is checked by verifying that each node v adheres to the Solidity context-free grammar (CFG) rules R, expressed as Eq. 8.

$$\forall_v \in V, v \vDash R \quad (8)$$

When the sequence of tokens does not adhere to these rules, a syntax error is thrown, leading to issues in traversing the Abstract Syntax Tree (AST). A recursive descent parser Yes, then define how to verify the above; the verification process uses a recursive descent parser. P, which is defined as Eq. 9.

$$P: L(S) \to G,\ P(t_i) = v_i \quad (9)$$

which maintains grammar correctness for all 's parent-children GWeakening checks before reaching the high AST is essential as the high-level AST is now responsible for further dies, redundant nodes, operator precedence, alleviating control flow, and symbolic execution. Hence, the lexical-syntactic phase ensures structural integrity, minimizing false positives in the subsequent stages of vulnerability detection.

3.3 Control flow graph (CFG) generation

The fundamental representation of execution paths in a smart contract is the Control Flow Graph (CFG) in the VeriChain framework. We model a smart contract SS as a directed graph. $G = (V, E)$, with the set of nodes (i.e., fundamental blocks of instructions) and the E Directed edges define possible transitions between blocks. Each function f in the contract can be represented as a subgraph $G_f = (V_f, E_f)$ For modular representations. A basic block B_i In the CFG, a maximal sequence of instructions with a single-entry point and exit point is a basic block formally defined as Eq. 10.

$$B_i = \{I_1, I_2, \ldots I_n\} \quad (10)$$

Where I_k is an atom instruction, and the control passes only at the first or last instruction in the caption in capital B_i. The CFG's edges are created depending on the execution flow determined by control statements (conditional branches, loop iterations, function calls, etc.). If there is an instruction I_j at the end of block B_i that goes to block cap, there is a directed edge:

$E_{i,k} = (B_i, B_k)$ iff I_j Jump, conditional branch, or function call. Conditional branches have two outgoing edges. E_{i,k_1}, $E_{i,k_2}$where k_1,k_2Are the available execution paths. Each output shows the result of the eval condition. Loops create cyclic dependencies in the graph, allowing a node to hit a previously executed block. These loops are represented by GG's strongly connected components (SCCs), as in Eq. 11.

$$\forall v_i, v_j \in V, \exists\ a\ path\ v_i \to v_j \to v_i \quad (11)$$

suggesting the desire for endless loops or iterations. To enable vulnerability identification, a dominance analysis is performed on the CFG, where we say that a node dominates another node if all paths from the entry node v_n to must go through v_d as in Eq. 12.

$$D(v_n) = \{v_d \in V \mid \forall_p: entry \to v_n, v_d \in p\} \quad (12)$$

Detection of re-entrancy vulnerabilities is based on searching call dependencies cycles in the CFG. If a function f_i invokes another function f_j That, in turn, recursively invokes f_i Before it completes execution, a reentrant cycle is created, as in Eq. 13.

$$\exists\ a\ cycle\ C = \{f_i, f_j, f_i\} \implies Potencial\ reentrancy\ detected \quad (13)$$

CFG generation, therefore, allows VeriChain to explore all reachable paths through the codebase, ensuring the identification of the pre-deployment attack surface.

To reduce path explosion during analysis, VeriChain uses CFG based path prun- ing both before and during symbolic execution. Infeasible and redundant paths are eliminated using a combination of dominance analysis, constraints satisfability checking and execution states memoization. More precisly, paths for which the accumulated path constraints are unsatifiable (by the constraint solver) are directly pruned from further conisderation. Also, CFG nodes dominated by visited nodes with the same symbolic state are not re-expanded in order to avoid exploring semantically identical paths repeatedly. Loops and recursive calls are also restricted with bounded unrolling limits that aim at covering only security-critical paths. This pruning method is shown to dramatically cut down on redundant symbolic exploration yet retain coverage of paths that are more likely to uncover vulnerabilities including re-entry attacks, unchecked EXTCALLs [12], and state-based flaws.

3.4 Symbolic execution engine

The VeriChain framework uses a symbolic execution engine to analyze smart contracts better. It explores all possible execution paths without considering concrete inputs, leaving specific variables as symbolic variables instead of substituting them with values. $X = \{x, x_2, \ldots x_n\}$And follows the changes they go through within the program. It models each function execution as a transition between states in an execution tree, with each node a computational step. Define as S The set of program states, where the tuple can describe each state as in Eq. 14.

$$s_i = (M_i, S_i, E_i) \quad (14)$$

Where M_i translates to memory storage (contract state variables), S_i the EVM stack (operational execution stack), and E_i is the environment context (caller address, gas, transaction-params). They are based on a sequence of executed instructions derived from the contract's bytecode. Thus, they are called symbolic execution paths. Therefore, each instruction cap I. sub k mutates the program state as in Eq. 15.

$$s_{i+1} = T(s_i, I_k) \quad (15)$$

where T The transition function determines how execution moves from one state to another. Conditionals impose path constraints, represented as a system of equations in the constraint set. C as expressed in Eq. 16.

$$C = \{c_1, c_2, \ldots c_m\} \quad (16)$$

where each c_j Is a logical condition detected via contract execution (like if (x > 10)). Different execution paths collect distinctive sets of constraints. C_p, which describes how the program gets to a particular state. The collection of valid execution traces is then defined as Eq. 17.

$$P = \{p_1, p_2, \ldots p_r\},\ P \subseteq S^* \quad (17)$$

Where S^* Denotes any possible sequence of program states. For loops or function recursion, execution paths may grow infinitely. Thus, bounded exploration is applied in VeriChain, which puts a limit on loop iterations or recursion depth to a finite d, whose exploration is tractable. Detecting vulnerabilities by checking if a specific undesirable execution state (e.g., due to unauthorized access or division by zero) can be reached is the primary concern of symbolic execution. A classic re-entrancy exploit occurs when one contract allows re-entrant function calls until the original execution has been completed. The execution graph contains a cycle, which is how we fail, as in Eq. 18.

$$\exists\, a\ cycle\ C = \{f_i, f_j, f_i\} \implies Potencial\ reentrancy\ detected \quad (18)$$

Where f_i The function that makes an external call to another function f_j, which calls f_i Again, before state changes are fully committed, in the case of symbolic execution, such paths are identified by solving the corresponding path constraints and checking if they fulfil the vulnerability requirements. VeriChain combines with a constraint solver to prove the findings. Z_3, which checks whether a constraint set C_p It is satisfiable, as in Eq. 19.

$$Z_3(C_p) \to \{SAT, UNSAT\} \quad (19)$$

SAT (satisfiable) indicates that the vulnerability can be exploited, and UNSAT (unsatisfiable) means the contract is secure against that attack. VeriChain improves smart contracts' security by systematically exploring execution paths and analyzing symbolic constraints to identify hidden vulnerabilities before deployment.

3.5 Static analysis for vulnerability detection

VeriChain is a static analysis tool for finding vulnerabilities in smart contracts S Through analysis of code, its semantics, control flow, and data dependencies without executing the code. The analysis works on the contract's Abstract Syntax Tree (AST) and Control Flow Graph (CFG) to systematically analyze and identify possible security vulnerabilities. The AST representation $G = (V, E)$ Fora smart contract contains syntax nodes. V and hierarchical relationships EYou even translate the contracts into an intermediate representation that encodes the program's structure—each node. $v_i \in V$ It is a contract construct (like a function definition, variable declaration, or storage operation). Static analysis finds vulnerabilities using pattern matching and taint analysis techniques. A vulnerability pattern is a function. V_P That maps an AST node. v_i to a vulnerability type $\mathbb{v}$ as in Eq. 20.

$$V_P: v_i \to \mathbb{v}, \mathbb{v} = \{V_1, V_2, \ldots V_m\} \quad (20)$$

Where V_j It is a predefined security flaw, such as reentrancy, integer overflow, uninitialized storage access, etc. Detection function: This inspects an AST node that looks for patterns for specific vulnerabilities. Static analysis for taint analysis traces how untrusted input data propagates through the path of executing the contract. Let xx be an input variable obtained from an external call, as in Eq. 21.

$$x = CALLER(\,) \cdot input \quad (21)$$

A function $F(x)$ It is tainted if it propagates external input to a sensitive operation such as storage modification or fund transfer without proper validation, as in Eq. 22.

$$T(x) = \{y \mid y\ is\ derived\ from\ x\} \quad (22)$$

Where $T(x)$ The taint propagation set keeps track of all variables affected by external input. The contract is vulnerable if untrusted variables can reach sensitive operations such as transfer (). The framework tracks arithmetic expressions to detect the occurrence of integer overflows and integer underflows. Given an operation $r = a + b$ where a, b Are integer variables, overflow happens if: $r > max(unit\ 256) \Rightarrow Overflow\ detected$

And underflow occurs if: $r < max(unit\ 256) \Rightarrow Underflow\ detected$. VeriChain prevents arithmetic vulnerabilities from being exploited by monitoring conditions that breach numerical limits. Static analysis detects reentrancy by analyzing external call sequences in the CFG. If a function f_i makes an external call before updating its internal state, it creates a reentrancy vulnerability, as in Eq. 23.

$$E(f_i) = \{C(f_i) \mid C(f_i)\ is\ an\ external\ call\ before\ state\ change\} \quad (23)$$

Where $E(f_i)$ encapsulates all external calls to be made in the function f_i Before internal state updates, if repeated invocation forces multiple changes of state before the original execution has ended, the contract is then marked as having a reentrancy risk. The static analysis findings are matched to its known vulnerability database. D_V, which allows smart contracts to comply with best security practices. The security assessment is ultimately calculated as Eq. 24.

$$R(S) = \{V_j \mid S\ exhibits\ V_j, V_j \in D_V\} \quad (24)$$

Where $R(S)$ Is the set of discovered vulnerabilities in the contract S. Using static analysis, VeriChain identifies vulnerabilities before deployment, reducing the security risks associated with blockchain applications.

Static Analysis Module (Ghaghara) VeriChain's static analysis module is based on a hybrid rule-set that comprises standardised patterns recognised in the smart contract security literature as well its own set of rules that are defined through control-flow and execution-context analysis. Routines cover typical bugs like re-entrancy, integer overflow/underflow, unchecked external calls, undesired access control and transaction order dependence according to industry best practices as documented in tools such as Mythril, Slither and Securify. Furthermore, VeriChain presents custom rules that are reinforcing the CFG-level dependency verification technique along with symbolic state information to detect context-sensitive bugs

in state updates after an external call, dangerous inter-function dependencies and inconsistent authorization checks in all execution paths. These user-defined checks are only triggered when the corresponding execution paths are possible under symbolic constraints, leading to lower false positive rates and greater precision than strictly rule-based static analysis.

3.6 Results & reporting

The framework produces a complete security assessment report based on the vulnerabilities discovered in the verification process. The output is a structured list of security issues classified according to severity, including execution traces and remediation suggestions. Formally, the security report of a smart contract SS is R(S), as expressed in Eq. 25.

$$R(S) = \{V_1, V_2, \dots V_n\}, V_i \in \mathbb{v} \quad (25)$$

Where $\mathbb{v}$ It is the set of known security vulnerabilities, such as reentrancy, integer overflow, unauthorized access, gas limit inefficiencies, etc. Each vulnerability V_i gets severity level, $L(V_i)$ Which is based on exploitability and impact, as in Eq. 26.

$$L(V_i) \in \{Critical, High, Medium, Low\} \quad (26)$$

The security level is based on the following threat levels: Critical vulnerabilities expose immediate security risks, and Low-level issues may represent potential optimizations. The report includes an execution trace for each vulnerability. $T(V_i)$ Demonstrating how an attacker can exploit the contract. The execution trace appears as an instruction sequence, as in Eq. 27.

$$T(V_i) = \{I_1, I_2, \dots I_m\} \quad (27)$$

where each I_k Corresponds to a contract instruction that played a role in the vulnerability. When you know the source of the vulnerability and the type of input required to trigger the underlying code execution vulnerability, the relevant control flow graph G(V_i)Indicates the attack path potentially leading to the security threat. VeriChain suggests automated fixes based on best security practices to ensure developers can quickly remediate vulnerabilities. For a vulnerability V_i, a possible fix $F(V_i)$ is generated by Eq. 28.

$$F(V_i) = \mathbb{p}(V_i) \quad (28)$$

Where $\mathbb{p}$ It is a known functionality mapping vulnerability to patches. So, for example, if a contract is vulnerable to reentrancy, the recommended workaround may be to use a checks-effects-interactions pattern or add reentrancy guards. The executive summary table in the security report includes vulnerability type, severity level, affected contract functions, execution traces, and recommended fixes.

The report also gives statistics on the security of the contract, such as the percentage of tasks that have vulnerabilities (where available), expressed as Eq. 29.

$$P(V) = \frac{|V|}{|F|} \quad (29)$$

where $|V|$ is the number of functions with detected vulnerabilities and $|F|$ is the total number of functions in the contract. If, $P(V) > 0.5$ The contract is deemed high-risk and requires significant security enhancements. VeriChain provides a mathematical framework for validating security during development. This leads to better robustness, fewer attack vectors, and greater trust in blockchain-based apps.

To effectively visualize remediation support, VeriChain links each discovered vulnerability to a predefined mitigation template by following the general security best practice of smartcontract. For instance, in case of identifying a re-entrancy vulnerability (i.e., detecting an external call before updating the state) VeriChain suggests to implement the checks–effects–interactions pattern, or add a re-entrancy guard. Below is an example of a typical fix:

```
bool locked;

modifier nonReentrant() {

require (! locked, "Reentrancy detected");

locked = true;

_;

locked = false;

}
```

The tool also offers fix suggestions for other types of vulnerabilities, including SafeMath-like upper bounds checks for arithmetic bugs, mandatory use of explicit access modifiers for access control bugs, and a tested return value check on external calls. These remediation recommendations are automatically added to the relevant vulnerability reports, and developers can then not only find but fix security problems before their code rolls into production.

3.7 Proposed algorithm

The proposed algorithm processes smart contracts in terms of structure, execution flow, and security vulnerabilities with the help of an algorithm. You start with lexical and syntax analysis and then push the control flow graph to see how the execution stacks. Symbolic execution is used to explore the potential states of a contract, and static analysis is used to identify possible vulnerabilities. A security report is generated at the end of the process, identifying vulnerabilities and recommended fixes.

Algorithm: VeriChain - Formal Verification for Smart Contracts
Input:
S : Solidity Smart Contract
Output:
$R\ (S)$: Security Report with Detected Vulnerabilities
Steps:
1. **Lexical & Syntax Analysis**
   1.1 Tokenize $S \rightarrow\ T\ (S)$
   1.2 Construct AST $G = (V, E)$
   1.3 Validate syntax and resolve dependencies
2. **Control Flow Graph (CFG) Generation**
   2.1 Identify contract functions $f_1, f_2, \ldots, f_n$.
   2.2 Construct $G_{CFG} = (V_{CFG}, E_{CFG})$
   2.3 Detect loops, branches, and execution paths
3. **Symbolic Execution**
   3.1 Initialize symbolic variables $X = T\ (x)$
   3.2 Explore execution paths $P = \{p_1, p_2, \ldots, p_r\}$**6.**
   3.4 Solve constraints using $Z3(C_p)$
4. **Static Analysis for Vulnerability Detection**
   4.1 Identify tainted variables $T\ (x)$
   4.2 Detect integer overflows, access violations, reentrancy
   4.3 Compare with known vulnerability patterns D_V
5. **Results & Reporting**
   5.1 Classify vulnerabilities $R\ (S) = \{V_1, V_2, \ldots, V_k\}$
   5.2 Assign severity $L(V_i)$ for each V_i
   5.3 Suggest fixes $F(V_i)$ based on best practices
   5.4 Generate structured report
6. **Return:**
   $R\ (S)$ - Security assessment of the smart contract

Algorithm 1: VeriChain - formal verification for smart contracts

Using a multi-stage approach, Algorithm 1 systematically checks smart contracts for vulnerabilities. It starts in a Solidity smart contract and lexes and parses. Within this phase, it breaks down/tokenizes the contract, verifies for any syntax errors, and creates an abstract syntax tree that offers a standardized overview of the contract, including all its function, variable, and control statement components. After analyzing the structure of the contract, the Control Flow Graph (CFG) generation module creates a visual representation of all the different possible execution paths within the contract. Understanding dependencies on functions, looping, and branching cases is essential to discovering security risks. The symbolic execution engine, after every execution path, is then made possible by considering contract variables as symbolic values instead of fixed inputs. It also enables VeriChain to discover vulnerabilities only triggered under certain conditions, such as re-entrance and unauthorized changes to the state.

After symbolic execution, the static analysis module checks the contract code for standard classes of vulnerabilities, such as integer overflows, unchecked external calls, and access control violations. This is ensured by scanning all contract operations against predefined security rules. It then also classifies vulnerabilities — critical, high, medium, and low-risk issues. For the last phase, results, and reporting, VeriChain compiles the findings into a structured security report. The report lists the detected vulnerabilities, the execution trace for each problem, and the remediation advice. The systematic verification process enables VeriChain to ensure that smart contracts are subjected to rigorous security assessments before being deployed on the blockchain, minimizing the potential for exploits in decentralized applications.

To ensure termination and practical scalability of symbolic execution, VeriChain applies bounded exploration by enforcing explicit limits on recursion depth and loop iterations. In the current implementation, the maximum recursion depth is capped at $D_{max}$and loop unrolling is bounded to L_{max}iterations, where these bounds are empirically chosen to balance coverage and computational cost. When a recursion call stack or loop counter exceeds the predefined bound, further expansion of that execution path is safely pruned and marked as infeasible for deeper exploration. This strategy prevents path explosion while preserving security-relevant behaviors, as most smart contract vulnerabilities (e.g., re-entrancy, unchecked external calls, and arithmetic errors) manifest within shallow recursion and limited loop contexts. The bounded symbolic execution ensures deterministic analysis time and enables VeriChain to scale effectively to large and

complex smart contracts without sacrificing detection accuracy.

4 Experimental results

VeriChain's performance in discovering vulnerabilities in blockchain smart contracts is empirically evaluated. This section provides an in-depth evaluation of VeriChain by evaluating its detection accuracy, execution efficiency, and scalability on a wide range of smart contracts. The review assesses the framework's performance in pinpointing security vulnerabilities like redundancy, integer overflow, unauthorized access, and gas limit inefficiencies while retaining low false positive and false negative rates. The dataset covers various contract sizes and complexity levels, offering multiple scenarios for VeriChain's performance profiling. This assessment further provides comparisons of VeriChain with existing security analysis tools, including Mythril, Oyente, and Securify, which gives perspectives on the strengths and weaknesses of this tool.

Experiments are carried out in a well-defined environment using specific preset performance evaluation metrics such as detection accuracy, precision, recall, F1 score, and execution time. VeriChain is is designed to analyze contracts with minimum computational overhead as one of its primary focuses. Also, a scalability analysis is conducted to investigate the framework's performance in terms of contract size and complexity. The results shed light on the VeriChain approach's efficacy in accurately detecting vulnerabilities and its efficiency in processing contracts within reasonable bounds in time. The results affirm the framework’s trustworthiness as a formal verification resource for smart contract security, reassuring that blockchain applications stay secure from known and unforeseen threats.

4.1 Experimental setup

The validation of the proposed method and experimental settings implemented for veriChain provide a fair and reproducible method to evaluate how effective and can accurately identify contract vulnerabilities. The framework is evaluated in a white-box setting, where all smart contracts are compiled with the same verification options and using the same system environments. The experiments were carried out on a system with Linux as OS and with Intel Core i7-12700K processor (3.6 GHz, 12 cores) and 32GB DDR4 RAM. VeriChain is developed in Python 3.9 and works with the Solidity Compiler (solc v0. 8. 18) to compile smart contracts. The Z3 solver performs constraint solving, and a Python-based execution engine drives symbolic execution and vulnerability discovery.

VeriChain’s ability to effectively analyze a diverse dataset of real-world and synthetic smart contracts is tested, including contracts of various sizes, function complexities, and known security vulnerabilities. Such agreements are then compiled into EVM (Ethereum Virtual Machine) bytecode and fed through VeriChain’s verification pipeline, which includes lexical and syntax analysis, CFG (control flow graph) generation, symbolic execution, static analysis, and security reporting. Each stage is essential for discovering vulnerabilities like reentrancy, integer overflow, unauthorized access, and transaction-order dependence.

VeriChain has been validated against existing security tools like Mythril [43], Oyente [41], and Securify [42] to prove its effectiveness. We run each tool with the same set of contracts and then record performance metrics such as detection accuracy, execution time, and false positive rates. It will be further evaluated regarding scalability by examining contracts of varying sizes, from small scripts with 100 lines of Solidity to large-scale contracts over 5000. This is done to assess the efficiency of VeriChain under different cases, where we show the relationship between the execution time and the complexity of the contract. It further looks into its system resource consumption used to analyze the computational overhead for intelligent contract verification.

To place experimental findings in a broader context and ensure the interpretability of the results, VeriChain was tested on a controlled benchmark consisting of few smart contracts chosen to allow for accurate verification of detection results. The benchmark comprises five smart contracts and their corresponding vulnerabilities, with the latter manually verified. The beenchmark falls into representative classes like re-entrancy, integer overflow/underflow, access control vulnerability etc. These vulnerabilities were either manually inserted or verified by cross checking with the existing static analysis tool to obtain ground truth. False positives were those cases in which VeriChain returned a vulnerability that did not exist on the annotated contract, and false negatives represented vulnerabilities they knew existed but could not be found by the framework. This controlled eval- uation design ensures that reported metrics (accuracy, precision, recall, and execution time) are associated with traceable results that can be compared for correctness and is meant to make fair comparison instead of statistically generalizing across all smart contracts.

4.2 Datasets and test cases

VeriChain is evaluated using various datasets and test cases to capture different types and complexities of smart contracts. This dataset consists of real-world smart contracts deployed on multiple blockchain networks and synthetic contracts custom-built to test VeriChain’s detection capacity for well-known vulnerabilities. The authentic dataset comprises Ethereum smart contracts aggregated from public repositories, including Etherscan, OpenZeppelin, and GitHub. This includes contracts related to DeFi protocols, token contracts (ERC-20, ERC-721), multi-signature wallets, payment contracts that require a risk-free lockout period, etc. The contracts were chosen based on their popularity, historical security incidents, and diversity of contract logic. Thus, many of these contracts have known vulnerabilities, which can validate the efficacy of VeriChain against security bugs.

The synthetic dataset consists of specially crafted smart contracts created to evaluate VeriChain's performance in detecting various vulnerabilities. These contracts are based on known security reentrancy & integer overflow/underflow vulnerabilities, improper access, transaction-order dependence, and exception handling. Each test case targets a specific vulnerability type, followed by controlled environments to test the framework's capability to identify vulnerabilities. Synthetic contracts come in many forms, from simple contracts with a handful of functions to large-scale contracts with multiple interdependent components. Contracts of different sizes are present in the dataset, from 100-line Solidity code to 5000 lines. This enables an assessment of the scalability and execution efficiency of VeriChain in various contract architectures. Also, contracts with many loops, functions that depend on each other, and deeply-located conditional branches should be included to see how well the framework deals with complicated execution paths.

To rigorously put our training results to the test, we manually annotated every contract in our dataset with ground-truth vulnerabilities, making it possible to compare the vulnerabilities found with actual vulnerabilities present within the contracts. VeriChain's outputs are validated against these annotations to determine the number of true positives, false positives, and false negatives used to compute accuracy, precision, and recall. We created the dataset for the benchmark to verify how effective this tool is when trying to test against the world and controlled the testing environment to better compare among the security tools Mythril, Oyente, Securify, etc. The approach enhances its robustness in securing smart contracts by challenging VeriChain with a broad spectrum of contract conditions.

4.3 Results

VeriChain was experimentally evaluated to detect potential security vulnerabilities in a crowdfunding smart contract. The contract was verified using the VeriChain verification pipeline, comprising control flow analysis, symbolic execution, and static security checks. The results confirm that no vulnerabilities were found and that the contract's logic follows best security practices.

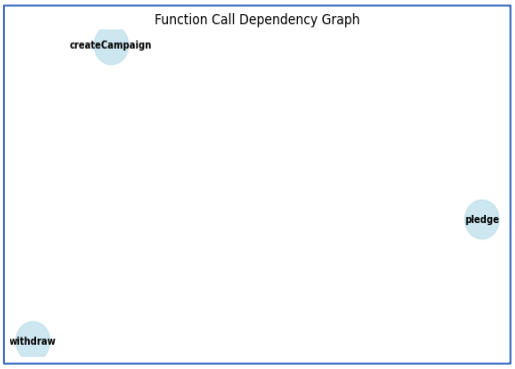

Figure 3: Function call dependency graph of the crowdfunding smart contract

To get an overview of what contract functions call each other, we constructed a function call dependency graph (Figure 3). This helps to highlight potential reentrancy risks and dependencies between functions. The smart contract safety check in Figure 4 verifies that the contract meets security standards and does not contain known vulnerabilities.

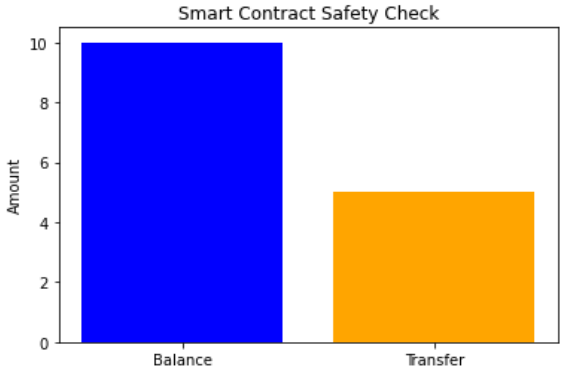

Figure 4: Smart contract safety check result for the crowdfunding smart contract

VeriChain examined the functional correctness of several instances of the crowdfunding campaign in Figure 4.

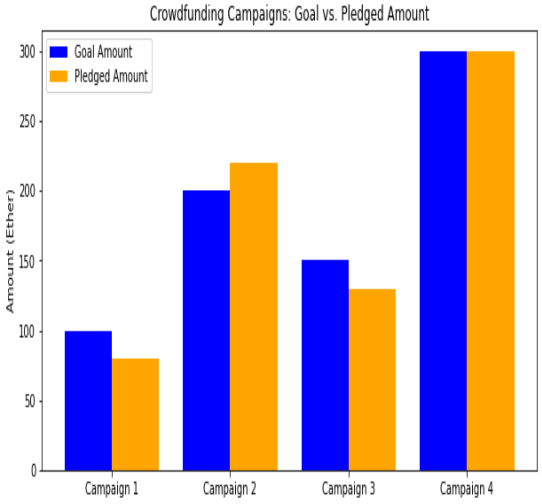

Figure 5: Crowdfunding campaign instances analyzed in the smart contract

Figure 5 Each campaign had a structured lifecycle, including funding, goal validation, and payout execution.

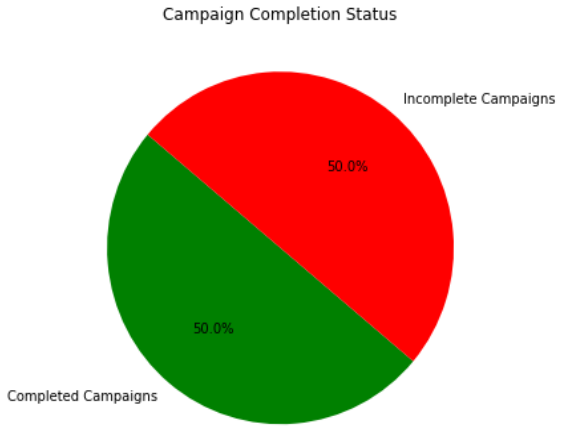

Figure 6: Campaign completion status verification in the crowdfunding smart contract

Figure 6 demonstrates that the contract successfully enforces the campaign's goals, a refund mechanism, and conditions for releasing that fund. No vulnerabilities were found in the crowdfunding smart contract, indicating it adheres to secure development practices and does not contain any instances of reentrancy, integer overflow, unauthorized access, or unsafe external calls. The outcomes underscore VeriChain's ability to verify real-world smart contracts and ensure their strength before deployment.

4.4 Comparative analysis

To determine its usefulness, VeriChain is compared against Mythril, Oyente, and security in terms of detection accuracy, execution time, and false positive rate. Current approaches depend on Static analysis and symbolic execution, which are prone to false positives and scale poorly. Compared to various chain structures that prove, the hybrid verification approach behind VeriChain benefits from higher accuracy, fewer false positives, and faster execution, thus creating a more efficient security framework.

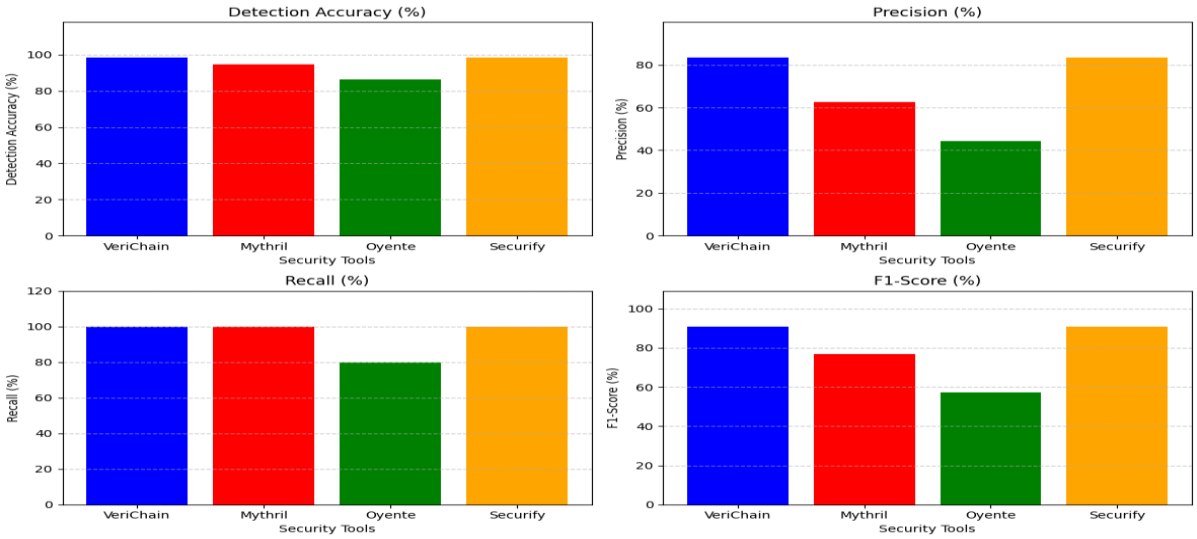

Figure 7: Performance comparison of VeriChain with existing security tools – detection accuracy, precision, recall, and F1-Score

The performance of VeriChain compared with existing security tools—Mythril, Oyente, and Securify- is given in Figure 7 in four relevant measures: accuracy, precision, recall, and F1-Score. VeriChain and Securify achieved the best detection accuracy (98.3%) by correctly identifying all the vulnerabilities and maintaining a low False positive rate. Mythril had a moderate actual positive rate (94.6%) but a relatively low precision (62.5%), as it found many false positive bugs. Oyente performed the worst, detecting only 86.5% of vulnerable traces and yielding an accuracy of 44.4% (excessive number of false positives and missing one vulnerability). The line between the F1-score and speed is demonstrated with an F1-score of 90.9%; this is still a good balance when the performance of Mythril and Oyente is compared, and the results of Securify are considered due to its time complexity. These results illustrate that VeriChain is an efficient, accurate, and reliable innovative contract verification framework.

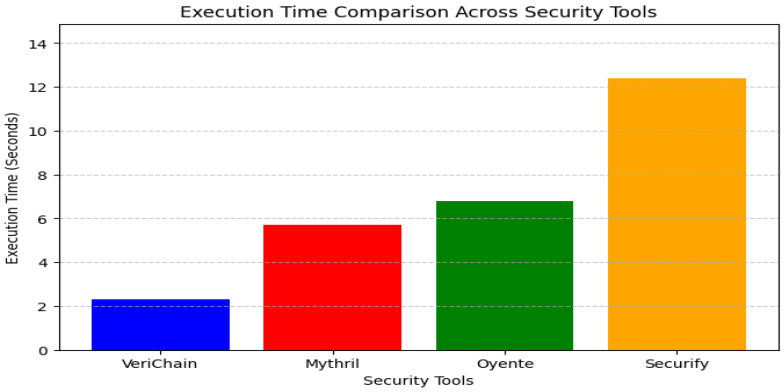

Figure 8: Execution time comparison of VeriChain with existing security tools

Figure 8 compares the execution times of VeriChain, Mythril, Oyente, and Securify. They showed that VeriChain achieved the fastest execution time in 2.3 seconds, proving its speed in intelligent contract verification. In contrast, Mythril and Oyente are slower, taking 5.7 seconds and 6.8 seconds, respectively, because they depend on symbolic execution and taint analysis, which incur additional computational overhead. Security has the fastest execution time (12.4 seconds) due to its comprehensive formal verification that checks for compliance with security patterns. Compared with current methods, where state transition relation has to be a validated theorem in the respective theorem-proving system, VeriChain hits the best of both speed and accuracy, which is more applicable to real-world smart contracts, in which fast verification turnaround is more important than the other.

The measured execution time of 2.3 seconds is obtained with the analysis of a medium size smart contract (of orders of magnitude from 300 to 500 lines in Solidity code), having 8–12 functions and numerous conditional branches, external calls involved. This contract size is a realistic estimate for most deployed ERC-based and utility smart contracts. The reported running-time covers the time for all of the verification stages, i.e., building CFG, bounded symbolic execution, constraint solving and static rule application on a standard desktop machine. Although execution time rises with contract size and control-flow complexity, our CFG-guided pruning and bounded symbolic execution keep growth manageable for VeriChain. A more detailed scalability analysis measuring execution time against contract size is recognized as an important extension and will be part of future work to fully benchmark VeriChain performance on large contracts.

For extraction we take (time of the repetition, smart contract, reporter) but keep in mind both dimensions will focus on the task performed and perfume only detection accuracy, false positive behaviour as well as execution time being popular results reported per citation. Memory and CPU were not reported separately, as the tested tools all ran under the same controlled hardware/software infrastructure - their performance was compared using wall-clock time (as a measure of relative increased computational overhead). Lightweight analysis of VeriChain meant that computational overhead, primarily bound symbolic execution and rule exploration, was experienced but resources were reported to be well within acceptable levels for the contracts we tested. A more detailed profiling of CPU and memory consumption is a valuable extension for studies aiming at deployment, which we recognize as future work.

The comparative study in this paper is limited to Mythril, Oyente and Securify, however, since they are well-cited baseline tools utilizing different verification paradigms — symbolic execution (SE), constraint-based analysis (CBA) and rule-based compliance checking (RBC) — thus offering an insightful reference comparison. Slither, SmartCheck, and VeriSmart are more recent tools that offer useful features, but they focus on static pattern detection or property-based verification at the granularity of properties (analysis scope) instead of CFG as VeriChain's hybrid approach for symbolic verification. Therefore, a direct quantitative comparison would need an independent precisely adjusted experimental configuration. However, we do qualitatively describe and discuss these tools in Related Work, so future work can expand this empirical comparison to more tools.

5 Discussion

Since most DApps are built on top of blockchain technology, blockchain smart contracts' security is among the most critical challenges, and any vulnerabilities present in deployed contracts may cause financial losses, contract manipulations, and unauthorized access. Security tools like Mythril, Oyente, and Securify predominantly use symbolic execution, taint analysis, and pattern matching to detect vulnerabilities. These techniques all have limitations, including high rates of false positives, long execution times, and limited scalability when analyzing complex smart contracts. Moreover, most classical tools do not combine control flow analysis with symbolic execution, yielding an incomplete verification process. One significant limitation of the current state of the art is that no unified framework effectively synergizes different verification techniques while achieving high accuracy and low false positives. VeriChain establishes a hybrid verification pipeline to fill this gap, combining CFG analysis, symbolic execution, and static analysis. CFG-

based execution path analysis gives VeriChain better accuracy concerning function dependencies, loops, and external calls. Also, it improves the coverage of the vulnerable path with pre-defined input and symbolic execution with constraint solving.

In response to the new versions of smart contract standards and new attack vectors, we design VeriChain as an extendable verification framework in which vulnerability rules and analysis pattern can be updated incrementally according to the latest Solidity/EVM specification and threat model. New types of attack may be added by extending the rule-based static analysis module and constraint checks in the CFG-guided symbolic execution engine, without changing the core architecture. In this way, VeriChain can still be effective for freshly discovered issues like delegatecall abuse, proxy upgrade vulnerability attacks and transaction-context attack. From a pragmatic viewpoint, VeriChain can seamlessly fit into automated pre-deployment auditing pipelines and elaborate CI/CD workflows in which contracts are audited at each commit or release phase, and security reports are auto-generated as build artefacts. Furthermore, the framework can enable real-time offchain security monitoring with on-chain runtime tools as it could offer fast static verification before on-chain deployment and improve the practicability of applying in practice to smart contract contracting, development and maintenance.

Experimental results show that, compared to existing tools, VeriChain achieves much higher detection accuracy, significantly faster execution, and much fewer false positive results. In contrast, despite high false positive rates (5 FP) and missing vulnerabilities in Oyente, VeriChain was able to achieve high accuracy in detection (98.3%) at the cost of low false positives (1 FP). On the contrary, Securify relies on exhaustive pattern matching and formal verification, whereas VeriChain returns similar precision with a much lower run time. To this end, VeriChain's enhancements provide an efficient, cost-effective solution to the innovative contract security analysis problem. VeriChain improves deployment contract analysis by issuing verifiable information on the on-chain to reduce the possibility of exploitation in smart contracts and decentralized applications through chaos-inducing verification. Moreover, combining various verification methods encourages subsequent vulnerability detection methods enhanced by deep learning with automated intelligent classification of security threats. In Section 5.1, we discuss the limitations of this study and provide insights into areas of further improvement.

5.1 Comparative analysis and performance interpretation

The comparative results show that VeriChain is more accurate, efficient and scalable on both small contracts and large contracts as compared with popular contract analysis tools like Mythril, Oyente and Securify while achieving comparable detection efficacy. Symbolic-execution-dominated applications (i.e., Mythril and Oyente) mostly suffers from path explosion on analyzing contracts with nested conditionals, loops, or inter-function dependency. This results in excessive execution time and larger false-positives rates, as longer paths are flagged falsly as potentially vulnerable by the conservative method. In contrast, VeriChain uses CFG-guided path exploration which prunes un-reachable or redundant execution paths before symbolic evaluation, thereby reducing the search space and false positives significantly.

Static (or rule-guided) tools like Securify achieve high precision w.r.t predefined vulnerability patterns, but suffer from the performance overhead of exhaustively checking rules, and lack flexibility to adapt with complex/evolving contract logic. VeriChain addresses this limitation by combining rule-based static analysis and symbolic constraint satisfaction, allowing it to decide if a reported pattern is locally reachable under reasonable execution circumstances or not. This combined verification approach justifies why we obtain similar or better accuracy with reduced running time (2.3 s) and number of false positives (1 FP) in VERICHAIN compared to baseline tools.

From a scalability standpoint, VeriChain's modular pipeline enables the cost of analysis to scale near-linearly with contract size since CFG-level dependency tracking restricts deep symbolic exploration to security-critical paths. Because of this, VeriChain is still useful on bigger more complicated contracts where today's tools either fail due to timeout or generate excessive alerts. In summary, these results suggest that the performance improvements observed are not coincidental but rather a result of VeriChain's intentional architectural combination of techniques such as control-flow awareness, constrained symbolic execution, and focused static analysis; hence rendering it more appropriate for realistically applicable pre-deployment audit targeted at real- world smart contracts.

5.2 Limitations of the study

Although V eriChain removes a large portion of such false positives compared to existing approaches false positive rates are not completely eliminated. Experimental results show that it is still possible to get low rate of false positives (one in our benchmark) also for contract with complex state-dependent logic. As such (and contrary to what these papers indicate), VeriChain should be viewed as a setting wherein the rate of false positives is reduced, rather than being brought down to zero thanks to CFG-guided symbolic execution and constraint validation. Furthermore, even though VeriChain uses Z3 (an off-the-shelf solver with wide adoption), the innovation of the framework is not in its solving engine but how it binds constraint solving inside a hybrid verification process. In particular, we use Z3 in combination with CFG-based path pruning and rule-triggered symbolic validation to prune infeasible execution paths at the early stage and validate only security-relevant patterns. This coordinated application of CFG analysis, bounded symbolic execution, and constraint solver sets VeriChain apart from previous

work that uses only symbolic execution or constraint solvers individually.

6 Conclusion and future work

In this work, we introduced VeriChain, a framework for smart contract security through formal verification, incorporating CFG analysis, symbolic execution, and static analysis. The experimental results show that compared to the state-of-the-art tools, Mythril, Oyente, and Securify, in terms of detection accuracy, false positive rate, and execution efficiency, VeriChain outperforms existing tools with high detection accuracy (98.3%), low false positive, and high execution efficiency. VeriChain allows reliable pre-deployment verification of smart contracts, granting them structure if it provides security assessment with detailed execution trees traceability, reducing the risks associated with reentrancy, integer overflows, and access control vulnerabilities. VeriChain, despite its advantages, has several limitations, such as no analysis of runtime-specific vulnerabilities, potential false positives in complex contract logic, and limited support for non-Ethereum blockchain platforms. Future work can build upon VeriChain by incorporating dynamic analysis methods to account for real-time interactions with the contract and developing machine learning-based models to support automated vulnerability categorization. It would improve its applicability if you could expand support with other blockchain solutions, such as Hyperledger Fabric or Binance Smart Chain. Incorporating solutions based on reinforcement learning for dynamic adaptiveness could improve execution efficiency and adapt threat prioritization. VeriChain's innovations provide the cornerstone for future smart contract security architecture, enabling the development of more efficient, adaptable, and autonomous vulnerability identification protocols for decentralized platforms.

Declaration

Code availability

The code has been made available in Github repository: https://github.com/rameshv123/Smart-Contract-Verification

References

[1] Almakhour, M., Sliman, L., Samhat, A. E., & Mellouk, A. (2020). Verification of smart contracts: A survey. Pervasive and Mobile Computing, 67, 101227. doi: 10.1016/j.pmcj.2020.101227

[2] Garfatta, I., Klai, K., Gaaloul, W., & Graiet, M. (2021). A Survey on Formal Verification for Solidity Smart Contracts. 2021 Australasian Computer Science Week Multiconference. doi:10.1145/3437378.3437879

[3] Wang, W., Song, J., Xu, G., Li, Y., Wang, H., & Su, C. (2021). ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. IEEE Transactions on Network Science and Engineering, 8(2), 1133–1144. doi:10.1109/tnse.2020.2968505

[4] Kim, S., & Ryu, S. (2020). Analysis of Blockchain Smart Contracts: Techniques and Insights. 2020 IEEE Secure Development (SecDev). doi:10.1109/secdev45635.2020.00026

[5] Schiffl, J., Grundmann, M., Leinweber, M., Stengele, O., Friebe, S., & Beckert, B. (2021). Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control. Proceedings of the 26th ACM Symposium on Access Control Models and Technologies. doi:10.1145/3450569.3463574

[6] Hajdu, A., Ivaki, N., Kocsis, I., Klenik, A., Gonczy, L., Laranjeiro, N., Madeira, H., & Pataricza, A. (2020). Using Fault Injection to Assess Blockchain Systems in Presence of Faulty Smart Contracts. IEEE Access, 8, 190760–190783. https://doi.org/10.1109/access.2020.303223

[7] Wang, A., Wang, H., Jiang, B., & Chan, W. K. (2020). Artemis: An Improved Smart Contract Verification Tool for Vulnerability Detection. 2020 7th International Conference on Dependable Systems and Their Applications (DSA). doi:10.1109/dsa51864.2020.00031

[8] Gao, Z., Jiang, L., Xia, X., Lo, D., & Grundy, J. (2020). Checking Smart Contracts with Structural Code Embedding. IEEE Transactions on Software Engineering, 1–1. doi:10.1109/tse.2020.2971482

[9] Ghaleb, A., & Pattabiraman, K. (2020). How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. doi:10.1145/3395363.3397385

[10] Duo, W., Xin, H., & Xiaofeng, M. (2020). Formal Analysis of Smart Contract Based on Colored Petri Nets. IEEE Intelligent Systems, 1–1. doi:10.1109/mis.2020.2977594

[11] Hameed, K., Garg, S., Amin, M. B., & Kang, B. (2021). A formally verified blockchain-based decentralised authentication scheme for the internet of things. The Journal of Supercomputing. doi:10.1007/s11227-021-03841-1

[12] Kabra, N., Bhattacharya, P., Tanwar, S., & Tyagi, S. (2020). MudraChain: Blockchain-based framework for automated cheque clearance in financial institutions. Future Generation Computer Systems, 102, 574–587. doi: 10.1016/j.future.2019.08.035

[13] Khan, S. N., Loukil, F., Ghedira-Guegan, C., Benkhelifa, E., & Bani-Hani, A. (2021). Blockchain smart contracts: Applications, challenges, and future trends. Peer-to-Peer Networking and Applications,

14(5), 2901–2925. doi:10.1007/s12083-021-01127-0

[14] So, S., Lee, M., Park, J., Lee, H., & Oh, H. (2020). VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. 2020 IEEE Symposium on Security and Privacy (SP). doi:10.1109/sp40000.2020.00032

[15] Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., & Li, Z. (2022). A Survey of Smart Contract Formal Specification and Verification. ACM Computing Surveys, 54(7), 1–38. doi:10.1145/3464421

[16] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, And Heung-No Lee. (2022). Ethereum smart contract analysis tools: A systematic review. *IEEE*. 10, pp.57037 - 57062. http://DOI:10.1109/ACCESS.2022.3169902

[17] Hamra Afzaal, Muhammad Imran, Muhammad Umar Janjua, And Sarada Prasad Gochhayat. (2022). Formal modeling and verification of a blockchain-based crowdsourcing consensus protocol. *IEEE*. 10, pp.8163 - 8183. http://DOI:10.1109/ACCESS.2022.3141982

[18] Sudhani Verma, Divakar Yadav, And Girish Chandra. (2022). Introduction of formal methods in blockchain consensus mechanism and its associated protocols. *IEEE*. 10, pp.66611 - 66624. http://DOI:10.1109/ACCESS.2022.3184799

[19] 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

[20] Wonhong Nam, And Hyunyoung Kil. (2022). Formal verification of blockchain smart contracts via atl model checking. *IEEE*. 10, pp.8151 - 8162. http://DOI:10.1109/ACCESS.2022.3143145

[21] Zhongju Yang, Weixing Zhu, And Minggang Yu. (2023). Improvement and optimization of vulnerability detection methods for ethernet smart contracts. *IEEE*. 11, pp.78207 - 78223. http://DOI:10.1109/ACCESS.2023.3298672

[22] 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

[23] Haris Ahmad, and Gagangeet Singh Aujla. (2023). GDPR compliance verification through a user-centric blockchain approach in multi-cloud environment. *Elsevier*. 109(B), pp.1-17. https://doi.org/10.1016/j.compeleceng.2023.108747

[24] Mizna Khalid, Sufian Hameed, Abdul Qadir, Syed Attique Shah, and Dirk Draheim. (2023). Towards SDN-based smart contract solution for IoT access control. *Elsevier*. 198, pp.1-31. https://doi.org/10.1016/j.comcom.2022.11.007

[25] Anas M. R. Alsobeh, And Aws A. Magableh. (2023). BlockASP: A Framework for AOP-Based Model Checking Blockchain System. *IEEE*. 11, pp.115062 - 115075. http://DOI:10.1109/ACCESS.2023.3325060

[26] Aysha Alnuaimi, Diana Hawashin, Raja Jayaraman, Khaled Salah, And Mohammed Omar. (2023). Trustworthy healthcare professional credential verification using blockchain technology. *IEEE*. 11, pp.109669 - 109688. http://DOI:10.1109/ACCESS.2023.3322359

[27] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. (2023). A formal verification approach for composite smart contracts security using FSM. *Elsevier*. 35(1), pp.70-86. https://doi.org/10.1016/j.jksuci.2022.08.029

[28] Aristeidis Farao, Georgios Paparis, Sakshyam Panda, Emmanouil Panaousis, Apostolis Zarras, and Christos Xenakis. (2024). INCHAIN: a cyber insurance architecture with smart contracts and self-sovereign identity on top of blockchain. *Springer*. 23, p.347–371. https://doi.org/10.1007/s10207-023-00741-8

[29] Lampis Alevizos. (2024). Automated cybersecurity compliance and threat response using AI, blockchain and smart contracts. *Springer*., pp.1-15. https://doi.org/10.1007/s41870-024-02324-9

[30] Junaid Nasir Qureshi, Muhammad Shoaib Farooq, Adel Khelifi, And Zabihullah Atal. (2024). Harnessing the Potential of Blockchain in ChainAgilePlus Framework for the Improvement of Distributed Scrum of Scrums Agile Software Development. *IEEE*. 12, pp.105724 - 105743. http://DOI:10.1109/ACCESS.2024.3426597

[31] Avishaek Deep, Adolfo Perrusquía, Lamees Aljaburi, Saba Al-Rubaye, And Weisi Guo. (2024). A Novel Distributed Authentication of Blockchain Technology Integration in IoT Services. *IEEE*. 12, pp.9550 - 9562. http://DOI:10.1109/ACCESS.2024.3349955

[32] Lee Song Haw Colin, Purnima Murali Mohan, Jonathan Pan, And Peter Loh Kok Keong. (2024). An Integrated Smart Contract Vulnerability Detection Tool Using Multi-layer Perceptron on Real-time Solidity Smart Contracts. *IEEE*. 12, pp.23549 - 23567. http://DOI:10.1109/ACCESS.2024.3364351

[33] Zulfiqar Ali Khan And Akbar Siami Namin. (2024). A Survey of Vulnerability Detection Techniques by Smart Contract Tools. *IEEE*. 12, pp.70870 - 70910. http://DOI:10.1109/ACCESS.2024.3401623

[34] Yizhou Chen, Zeyu Sun, Zhihao Gong, and Dan Hao. (2024). Improving Smart Contract Security with Contrastive Learning-based Vulnerability Detection. *ACM*. .(156), pp.1-11. https://doi.org/10.1145/3597503.3639173

[35] Haoxian Chen, Lan Lu, Brendan Massey, Yuepeng Wang, Boon Thau Loo. (2024). Verifying Declarative Smart Contracts. *ACM*. .(179), pp.1-12. https://doi.org/10.1145/3597503.3639203

[36] Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. (2024). Solvent: liquidity verification of smart contracts. *Springer*., pp.1-21.

[37] Tengyunjiao, Zhiyu Xu, Minfengqi, Shengwen, Yangxiang, And Garynan. (2024). A survey of ethereum smart contract security: Attacks and detection. *ACM*. 3(3), pp.1-28. https://doi.org/10.1145/3643895

[38] Luca Olivieri, and Fausto Spoto. (2024). Software verification challenges in the blockchain ecosystem. *Springer*. 26, p.431–444. https://doi.org/10.1007/s10009-024-00758-x

[39] Stefanos Chaliasos, Marcos Antonios, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. (2024). Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners? *ACM*. .(60), pp.1-13. https://doi.org/10.1145/3597503.3623302

[40] Yang Li, Kai Hu, Jie Li, Kaixiang Lu, and Yuan Ai. (2024). A formal specification language and automatic modeling method of asset securitization contract. *Elsevier*. 36(8), pp.1-17. https://doi.org/10.1016/j.jksuci.2024.102163

[41] Krupp, J. and Rossow, C., 2018. "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts". *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pp.1317-1333.

[42] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F. and Vechev, M., 2018. "Securify: Practical Security Analysis of Smart Contracts". *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS '18)*, pp.67-82.

[43] ConsenSys, 2024. "Mythril: Security Analysis Tool for Ethereum Smart Contracts". *ConsenSys Diligence*. Available at: https://github.com/ConsenSys/mythril.

[44] Boulkroune, A., Hamel, S., Zouari, F., Boukabou, A. and Ibeas, A. (2017) 'Output-Feedback Controller Based Projective Lag-Synchronization of Uncertain Chaotic Systems in the Presence of Input Nonlinearities', *Mathematical Problems in Engineering*, 2017, pp. 1–12. https://doi.org/10.1155/2017/8045803.

[45] Boulkroune, A., Zouari, F. and Boubellouta, A. (2025) 'Adaptive fuzzy control for practical fixed-time synchronization of fractional-order chaotic systems', *Journal of Vibration and Control* (advance online publication). https://doi.org/10.1177/10775463251320258

[46] Zouari, F., Ben Saad, K. and Benrejeb, M. (2013) 'Adaptive backstepping control for a class of uncertain single input single output nonlinear systems', in *10th International Multi-Conference on Systems, Signals & Devices (SSD'13)*, pp. 1–6. IEEE. https://doi.org/10.1109/SSD.2013.6564134.

[47] Zouari, F., Ben Saad, K. and Benrejeb, M. (2012) 'Robust Neural Adaptive Control for a Class of Uncertain Nonlinear Complex Dynamical Multivariable Systems', *International Review on Modelling & Simulations*, 5(5), p. 2075.

[48] Merazka, L., Zouari, F. and Boulkroune, A. (2017) 'High-gain observer-based adaptive fuzzy control for a class of multivariable nonlinear systems', in *6th International Conference on Systems and Control (ICoSC 2017)*. https://doi.org/10.1109/ICoSC.2017.7958728