Journal of Theoretical and Applied Information Technology
15th February 2025. Vol.103. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

SMARTSCAN: A COMPREHENSIVE FRAMEWORK FOR EFFICIENT AND OPTIMIZED FORMAL VERIFICATION OF COMPLEX BLOCKCHAIN SMART CONTRACTS

G.SOWMYA[1] , DR.R.SRIDEVI[2]
[1]Research Scholor, JNTUH, Department of CSE, Hyd, India
[2]Professor. JNTUH, Department of CSE, Hyd, Country
E-mail: gonurusowmya@gmail.com, srideviranguu@jntuh.ac.in

ABSTRACT

Smart contracts are gaining popularity as blockchain technology, and its uses continue to develop rapidly. Smart contracts are necessary to enforce real-time contracts in Blockchain systems. It is essential to thoroughly verify both simple and complex smart contracts since inconsistencies could lead to issues like the inability to deliver the required services. Existing tools like SmartCheck can automatically verify the correctness of smart contracts. However, a more complete solution that ensures the accuracy of smart contracts and considers security concerns is needed. In this study, we propose SmartScan, an efficient and optimal framework for formally verifying complex blockchain smart contracts. SmartScan uses a hybrid methodology that includes formal approaches, optimized heuristics, static analysis, and an optimized verification process to find weaknesses and inconsistencies in smart contracts. SmartScan's architecture aims to be robust to complex relationships and many interconnected parts. SmartScan streamlines the verification process by lowering its temporal and computational complexity. More importantly, it can validate the smart contracts of large-scale blockchain applications due to its scalable architecture. It manages the life cycle of smart contracts and can be extended to interface with other apps. Several algorithms in SmartScan aid in realizing the underlying architecture so that its primary functions can be accomplished. In terms of computer complexity, temporal complexity, and the ability to detect flaws and irregularities in simple and sophisticated smart contracts, SmartScan outperforms many currently used approaches, per an objective analysis using the DeFiLending case study. As a result, SmartScan is more than just a tool; it is a scalable and effective solution that can be incorporated into already-existing applications that deal with the life cycle of smart contracts and blockchain application development.

Keywords - *Smart Contracts Verification, Blockchain Technology, Blockchain Applications, Formal Verification, Smart Contact Life Cycle*

1. INTRODUCTION

Blockchain technology is a decentralized digital ledger that enables multiple people to preserve records safely and publicly without needing a single authority. Every block in the chain has a cryptographic link to the block before it, which also has a list of transactions, ensuring the data's integrity and immutability. Organizations like logistics management and Bitcoin that require security, transparency, and trust benefit significantly from blockchain technology. A document with terms encoded into computer code that can be executed independently is called a contract with intelligence. Blockchain technology is used in electronic contracts to manage and implement contracts autonomously based on predefined criteria. Smart contracts enhance transaction efficiency and save money, as they do not need intermediaries. They are indispensable to complex systems such as decentralized finance (DeFi), governance protocols, and virtual asset management.

Official verification is an algebraic technique that can prove the accuracy of algorithms and systems. To ensure they are resistant to a wide range of possible risks. Smart contracts must be long-term secure and resilient against failure, error, or misuse and must meet regulatory compliance requirements. Smart contracts have various applications, especially in the banking sector. It is also crucial as any fault or vulnerability may lead to millions of dollars in losses. As they move autonomously, there is more

willingness to trust the smart contracts when their accuracy is validated. Proving readiness to follow laws and regulations through official documentation might encourage using blockchain technologies more often. Problems can be identified quickly before being implemented once an agreement is published, avoiding costly errors through official verification.

Multiple established methods exist for validating smart contracts, each with advantages and disadvantages. Model testing examines the space of a system to determine things such as safety and liveness. While it is practical, model verification for complex contracts may be resource-intensive. On the other hand, one can use a theorem-type approach to prove that a contract is legal in its terms. It requires a lot of effort and talent, but it provides a lot of assurance. Symbolic code execution is used in place of physical inputs in operation symbolic to enable the investigation of numerous execution paths. Despite its strength, it might not be able to withstand state explosions. Two programs, Mythril and Slither, use static analysis to find issues and flaws in smart contracts without ever having to execute them. Because of their quickness, they could miss some contextual mistakes. Formal approaches have become more accessible for developers to apply because of tools like VeriSol and KEVM, which provide automated frameworks for comparing smart contracts to specifications. Employing appropriate and ideal formal verification approaches is necessary because blockchain and smart contract technologies are widely used. By guaranteeing the precision and safety of smart contracts, blockchain applications can gain more dependability and credibility, paving the way for this innovative technology's more comprehensive implementation and development.

As part of this study, we present a thorough SmartScan framework for the successful and fast formal verification of intricate blockchain smart contracts. A hybrid tool called SmartScan uses formal methods, updated verification processes, improved heuristics, and static analysis to find flaws and inconsistencies in smart contracts. The primary objective of SmartScan's design is to ensure that it can tolerate intricate connections and interactions between various parts. It boosts efficiency by speeding up the verification process in terms of time and calculation. The scalability of SmartScan allows it to validate smart contracts used in large-scale blockchain applications. The framework can be utilized by apps that manage the smart contract lifecycle due to its adaptability and simplicity of usage. Many algorithms are built into SmartScan to support its primary features and simplify the underlying processes. An investigation of the DeFiLending scenario shows that SmartScan outperforms several existing approaches in terms of time complexity, computational complexity, and its capacity to detect bugs and inconsistencies in simple and sophisticated smart contracts. As a result, SmartScan is more than just a tool; it is a scalable and effective solution that can be incorporated into already-existing blockchain development and smart contract lifecycle applications. This paper's remaining sections are organized as follows: The literature on formal verification techniques is reviewed in Section 2. Section 3 presents the necessary preliminaries. Section 4 introduces the proposed framework. Section 5 discusses the case study. Section 6 presents the experimental results, while Section 7 outlines the study's limitations and provides directions for future research.

2. RELATED WORK

This section reviews prior works on smart contract formal verification tools. Singh et al. [1] allowed using smart contracts to enable decentralized apps; however, security issues still exist. Vulnerabilities are addressed by formal approaches such as theorem proofing. An analysis stresses cooperative efforts for increased smart contract security while highlighting methods, languages, and tools Permenev et al. [2] regarding real-world custom requirements; VERX is an automated verification demonstrating the functional features of Ethereum smart contracts. It uses three strategies, showing that it is helpful in real life. Liu et al. [3] investigated the security verification of blockchain smart contracts, fixing flaws and accuracy. Future research shows potential when formal approaches are the focus. So et al. [4] arithmetic safety in Ethereum smart contracts is guaranteed by VERISMART, an accurate verifier. It reduces manual inspections and false alerts by outperforming current analyzers. Gao et al. [5] Word embedding is used automatically to discover bugs and clones, ensuring the dependability of Solidity smart contracts. It can identify 90% of clones and detect bugs effectively after being evaluated on 22,000 contracts.

Angelo and Salzer et al. [6] decentralized apps are powered by smart contracts, which openly manage asset trades. Open-source development promotes confidence. Reusing parts

from reliable repositories such as ConsenSys and the Ethereum Foundation is advised. Five tools inspire. Feist et al. [7] SlithIR is used by Slither, an open-source static analysis framework for Ethereum smart contracts, to find bugs, optimize code, and comprehend them efficiently. Grech et al. [8], with accuracy, comprehensiveness, and scalability for analysis, Gigahorse, a toolchain, decompiles Ethereum smart contracts into high-level 3-address code. Zheng et al. [9] state that smart contracts transform sectors by automating contracts on blockchains. Issues with bugs, security, and bridging gaps across disciplines of knowledge exist despite the promise. Issues with security, language barriers, performance, and few resources beset Zou et al. [10] Blockchain-driven smart contract development. Further study and advancements are required.

Pinna et al. [11] thoroughly examine more than 10,000 Ethereum Smart Contracts, looking into their features, transactions, positions in the development community, and source code attributes. Results show power-law distributions, developing software, and a wide range of uses not limited to crypto values. Ante [12] examined 20,188 references and 468 publications on smart contracts, identifying six study strands that address legal, social, technological, and economic issues. The promise and multidisciplinary nature of smart contracts are highlighted, along with challenges and prospects for further study. Babel et al. [13] for decentralized finance (DeFi) smart contracts, Clockwork Finance Framework (CFF) is a flexible formal verification tool that guarantees both attack exhaustiveness and contract completeness. Without pre-programmed attack techniques, CFF creates models for well-known DeFi protocols and finds that real-world transactions have an average monthly estimated extractable value (EV) of $56 million. Rana et al. [14] utilized smart contracts to propose a decentralized paradigm on the Layer 2 Polygon blockchain; data security, integrity, and transparency are guaranteed. Superior dependability is demonstrated via simulations, and scalability issues are resolved. Yamashita et al. [15] discussed the hazards of Hyperledger Fabric smart contracts, especially those created in Go. The study presents a novel static analytic technique to identify new risks and emphasizes the necessity of regular surveys and updates.

Kemmoe et al. [16] examined state-of-the-art blockchain-based smart contract technologies, classifying developments in social applications, contract architecture, access control, and cryptography. It points out flaws, emphasizes recent advancements, and makes recommendations for future lines of inquiry. Wang et al. [17] suggested using machine learning to quickly identify Ethereum smart contract vulnerabilities, with an average detection time of 4 seconds and over 96% prediction accuracy—subsequent research endeavors to optimize efficiency and identify new weaknesses. Khan et al. [18] influence on banking and the rise of trustless environments with smart contracts are revolutionary. A survey that addresses obstacles and unresolved issues examines technical, use, and future trends. Vacca et al. [19], a literature survey on blockchain software engineering, identify problems with testing, security, metrics, and smart contract creation. Open challenges are noted for more research on a variety of subjects. Huang et al. [20] investigated the issues with security in smart contracts running on decentralized blockchains. It examines salient characteristics, susceptibilities, and remedies across the software life cycle and makes recommendations for future study avenues.

Rouhani and Deters [21] examined the development of blockchain technology, emphasizing the use of smart contracts for purposes other than cryptocurrency. Research on security, performance, and decentralized applications is categorized under it. Peng et al. [22] thoroughly examine the research prospects and security concerns of using smart contracts in Internet of Things applications, emphasizing possible attacks, programming problems, and vulnerabilities. The survey aims to direct future research on this developing topic. Jiao et al. [23] aimed to increase security in implementing Ethereum smart contracts by introducing formal Solidity semantics that enhance source code verification. Hewa et al. [24] examined how blockchain technology and smart contracts may revolutionize various applications, emphasizing the present advantages and the possibilities for the future. Kushwaha et al. [25] methodically examined Ethereum smart contract security, including flaws, assaults, defenses, and potential future study areas.

Kirli et al. [26] examined 13 initiatives, 178 articles, and blockchain-enabled smart contracts in the energy industry. The report addresses issues and provides solutions emphasizing distributed control, energy, and flexibility trading. Hamledari et al. [27] presented a payment system that ensures speed and accuracy in autonomous payments in building projects by

utilizing blockchain smart contracts and robotic reality capture. After being successfully applied to actual projects, the technique eliminates inefficiencies in the payment workflow and opens the door for broader adoption by fusing on-chain and off-chain reality. Vangala et al. [28] offered a blockchain-based intelligent farming system that guarantees data integrity, transparency, and anonymity. Formal analysis and simulation verify that the suggested authentication strategy delivers greater security and functionality. Sookhak et al. [29] examined the security, taxonomy, and problems of blockchain-based access control for Electronic Health Records (EHRs) in the medical field. Saini et al. [30] presented a safe framework in a blockchain-based innovative healthcare system for exchanging Electronic Medical Records (EMRs). For effective access control, the system uses blockchain technology and encryption.

Seven et al. [31] offered a peer-to-peer energy trading platform built on the Ethereum blockchain for a virtual power plant. It concentrates on the financial side, using smart contracts to establish an auction for safe and affordable transactions within a virtual private network. On the Ethereum Ropsten Test Network, actual data is used to validate the suggested design. Hu et al. [32] addressed vulnerabilities and fraud by implementing a transaction-based approach for Ethereum smart contract categorization and detection. The technique demonstrates exemplary performance in recognizing abnormalities and differentiating across contract types using LSTM. Sanchez [33] executed traces are compared to specifications in the formal methods field of runtime verification, often used in software. This article emphasizes overcoming obstacles by outlining issues in several disciplines. Liu et al. [34] suggested that the EV power selling paradigm for V2G networks is based on blockchain technology and uses reverse auctions and smart contracts. Simulation studies confirm its efficacy. Li et al. [35] suggested using a hierarchical architecture to manage a distributed energy system with various end users and renewable sources. It uses blockchain technology and smart contracts to run safe and effective operations; a case study conducted in Singapore demonstrates the potential outcomes. Future developments will focus on scalability and resolving obstacles to the acceptance of blockchain technology.

Xiong et al. [36] addressed issues with traditional data trading and offered a blockchain alternative using machine learning and smart contracts. It secures data owner rights, removes reliance on other parties, and expedites Ethereum payments. The research, which is available on GitHub, examines the smart contract's conception, execution, security analysis, and performance assessment. Upcoming projects will improve smart contracts to stop post-sale data reselling. Alkadi et al. [37] examined a Deep Blockchain Framework (DBF) focusing on cloud privacy protection for cooperative intrusion detection. Using Ethereum for a privacy-based blockchain and Bidirectional Long Short-Term Memory (BiLSTM) for intrusion detection, DBF performs better than its peers. It offers a safe method of decision assistance for prompt data movement. Future research aims to assess the usefulness and scalability of various real-world datasets. Wang et al. [38] became more popular, especially with Ethereum's smart contracts. This study examines the security of Ethereum smart contracts from 2015 to 2019, pointing out flaws and suggesting areas for further research to improve security. Wang et al. [39] fixed problems with agricultural food supply chains, guaranteeing transparency and traceability. Reliability and security are increased by process tracking and information sharing, which are improved by a consortium and smart contract-based architecture. When used at Shanwei Lvfengyuan Modern Agricultural Development Co., Ltd., the framework facilitates product tracking using QR codes and disintermediation. Egala et al. [40] suggested use blockchain, DDSS, and hybrid computing to create a decentralized IoMT healthcare system that is efficient, secure, and private. From the literature, it was observed that there is a need for developing better formal verification tools for checking blockchain technology smart contracts.

3. PRELIMINARIES

3.1 Formal Verification of Smart Contracts

Formal methods provide a powerful technology for the correctness verification of smart contracts. The use of formal methods to verify smart contracts has been widely recognized, and significant results have been achieved in practice. The Ethereum community has also turned to formal methods to solve the demands for high-assurance contracts. Besides, using formal methods to validate smart contracts can provide a rigorous mathematical model for verifying smart contracts. By analyzing the

model, we can more easily discover logic errors or other new vulnerabilities. The formal verification of smart contracts tends to have an excellent potential for development in the future.

3.1.1 Program-based Formal Verification

A smart contract's essence is the program executed on the blockchain. Verifying the correctness of the program is a vital part of ensuring the safety and reliability of smart contracts. Successful practical cases and theoretical studies have been conducted on program-based verification of smart contracts. At the 3rd Global Blockchain Summit in 2017, a blockchain formal verification team of the University of Electronic Science and Technology presented the VaaS (Verification as a service) as an EOS formal verification platform. In addition to the EOS blockchain platform, VaaS supports other common platforms such as Ethereum and Fabric. The principle of the VaaS platform is to translate programs written in the Solidity scripting language into the Coq code, thus establishing a standard formal model for smart contracts and then verifying the correctness of the smart contract by verifying the correctness of the Coq language. Similarly, Bhargavan et al. proposed a verification method based on programming language. They translated the Solidity language into an F* language to check if the contract was correct. In most cases, we can only get the binary code running on Ethereum, and we cannot get the source code of the smart contract. So, in the absence of source code, they decompiled the binary files on Ethereum into F* language and analyzed whether at least some of the attributes to be reached by the contract were satisfied. The complete F* language for any smart contract had not yet been implemented because of enormous work. Moreover, verifying whether the F* language translated by Solidity language was consistent with the decompiled binary code was challenging for the same contract. Furthermore, Grishchenko et al. used the F* language proof assistant to successfully validate the official Ethereum test suite.

3.1.2 Behavior-based Formal Verification

Model checking is well adopted in behavior-based verification. It can conveniently model the interaction between the user and the program to verify whether the smart contract can interact with the user reliably and securely. Some good examples of behavior-based formal validation are demonstrated below. Ellul et al. proposed a runtime verification method. It was a novel state-based technique that ensured the violating party provided insurance for correct behavior. They used the finite state machine to model the contracts, and this method had been partially implemented in a proof-of-concept tool, ContractLarva. Their method referred to the methods proposed by Fenech et al. and Gorin et al., which validated the properties of the contract.

3.2 Need for Formal Verification of Smart Contracts

The blockchain application known as DeFiLending exemplifies a decentralized finance platform where users can deposit cryptocurrency, withdraw funds, borrow assets, and repay debts. This system facilitates crucial financial interactions and must uphold high security, accuracy, and reliability standards to avoid financial loss and safeguard user assets. In the contract, users deposit funds into a shared pool, providing collateral for potential borrowing; the platform then allows users to borrow assets proportionate to their deposit value. This ensures the system has enough liquidity to perform loans but maintains stringent collateral requirements of 150% or more of a single loan. As such, a smart contract relies on complicated computations and handles high-value financial assets, so it must use formal verification to ensure that it behaves as desired while not introducing risk.

Correct collateral management is a critical element of lending systems based on smart contracts. Incorrect calculations or modifications on the collateral part could lead users to borrow more than needed, putting the platform's liquidity at risk and raising the possibility of insolvency. Facilitating formal verification would also help to avoid overborrowing, as deposits, collateral, and amounts that can be borrowed should always be valued using the intended logic on your platform. For smart contracts to ensure that consumers can obtain funds deposited into accounts, accurate tracking of deposits and withdrawals is also required. This is to prevent cases when a user withdraws more than he has, which leads to losing funds for other users. Formal verification could help confirm that quantities on the platform are equal to those provided by customers, making payments and withdrawals more accurate and ensuring confidence.

Journal of Theoretical and Applied Information Technology
15th February 2025. Vol.103. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

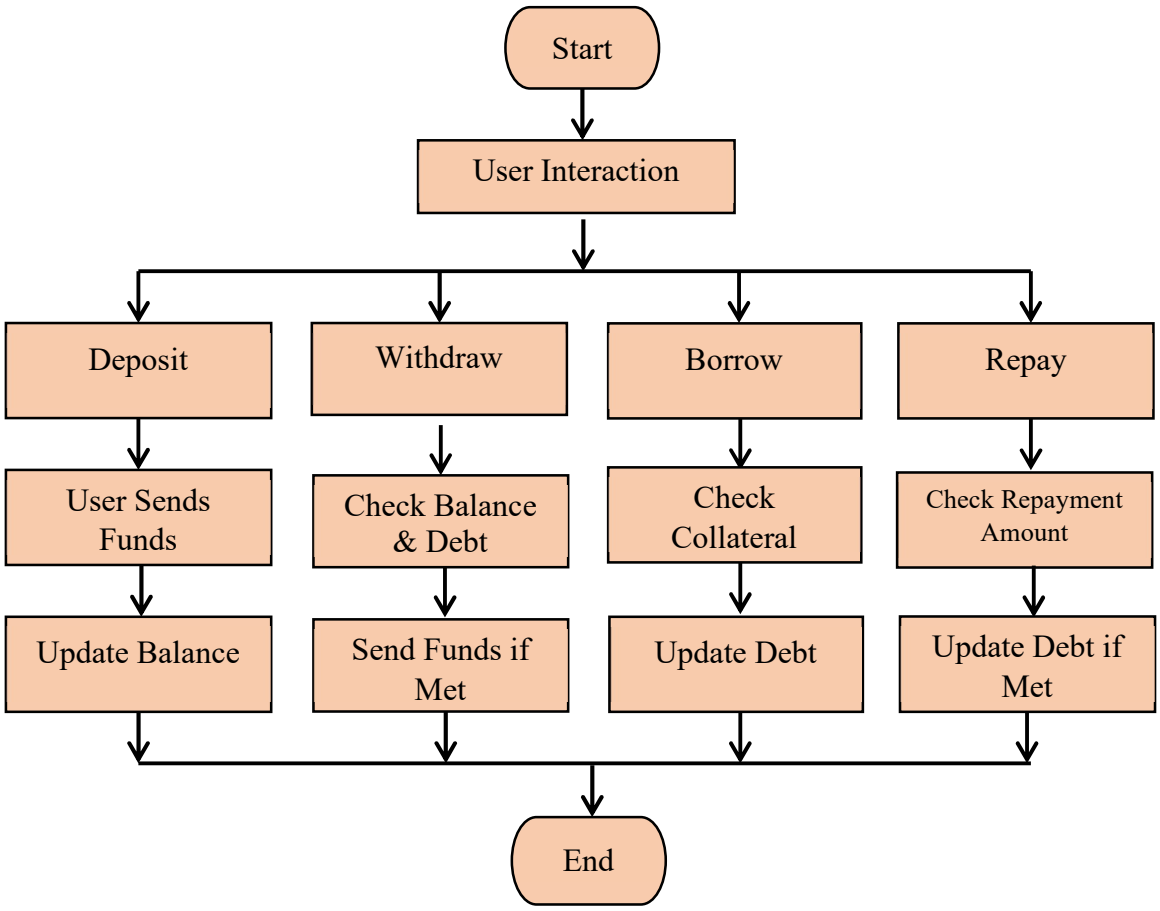

Figure 1: Illustrates the flow of the DefiLending application's smart contract

Protection is essential when creating smart contracts since reentrancy assaults and other flaws could allow dishonest users to keep performing operations like `withdraw` and `borrow,` thus exhausting the contract's cash. An attacker may start a money transfer more than once before the system amount is updated if they attempt to enter a contract again while a transaction is underway. Before any funds are transferred, platforms can expressly verify that operations, such as fund deductions and other state changes, are adequately protected against reentrancy. Furthermore, smart contracts could be impacted by integer overflow and underflow problems, especially in previous Solidity versions.

Both unwanted consequences may happen when dams overflow or underflow, passing a value above its upper bound and below zero. With things like deposits and debts able to be automatically analyzed, formal verification techniques come in handy for ensuring that big transfers or edge-case inputs cannot cause unforeseen financial disasters by ensuring they are always kept within defined bounds. Last but not least, verifying an authority also ensures compliance with lending regulations (such as needing enough collateral before approving a loan). Besides fulfilling the collateral requirements, our formal verification gives a mathematical assurance that no borrower can draw more than their collateral can bear. This behavior is vital to the long-term sustainability of the platform and ensuring other people can hold their assets there without risk.

Thereon, the agreement must then be completed (completed formally/initially), and after that, the complete pre-conditions to provision and deposit of the loan should be checked. It can also be systematically enforced for plausible adherences. Official inspection models invariant properties (e.g., deposits>=debts) and analyzes a mathematical procedure with each transaction to guarantee that overdraws are avoided, ensuring the contract's validity. By a strict methodology, the integrity of each transaction is guaranteed and protected from threats, which brings customers' trust and safety to the DeFiLending program.

Table 1: Acronyms Used In This Paper

Acronym	Description
BIP	Behavior, Interaction, Priority
FSM	Finite State Machine
SMV	Symbolic Model Verifier
CTL	Computation Tree Logic
nuXmv	Symbolic Model Checker Tool

3. PROPOSED FRAMEWORK

We created a formal method for checking smart contracts. SmartScan technology was developed to evaluate complex blockchain smart contracts for accuracy and security. The source code and contract definition are transformed into an FSM (Finite State Machine), which is a mathematically rigorous model. The framework of an FSM fostering this analysis can uncover surprising behaviors and potential shortcomings by easing the application of model-checking tools. Once the FSM is ready for efficient validation, The framework creates a Behavior, Interaction, Priority (BIP) model. Symbolic model checkers such as nuXmv analyze the state space of a BIP to detect security flaws or violations of predefined properties. The result is a complete verification report containing the evaluation results and any problems identified. SmartScan aims to build trust in inaccessible networks and improve smart contracts' safety, reliability, and value by automating the verification process. So, the architecture is demonstrated in Figure 2 of SmartScan. SmartScan enhances the efficiency of complex blockchain smart contract processes. This paradigm posits that formal definition, verification, and modeling are interconnected processes. A smart agreement's accuracy and security features are described in the Contract Specification, the first document in the Formal Specification Phase. Using user-defined correctness qualities, these specifics can be adjusted to meet the terms of the contract. However, security features address common security issues in smart contracts by leveraging these pre-existing components.

Those qualities are reflected in Computation Tree Logic (CTL) equations, a temporal logic used in model verification that dictates the order in which certain events must occur in a system. The contract can be expressed as a finite state machine (FSM) capturing its behavior regarding states and transitions. Identifying potential weaknesses in the channels over which the contract runs is necessary. We recommend exploring the limitations of this FSM. FSM modeling: SmartScan can detect any vulnerabilities resulting from the nature of smart contracts and ensure they are accurately represented in security and functionality. The Solidity contract source code is first converted into a formal language known as BIP (Behavior, Interaction, Priority) in the Modeling Phase. This allows for a deeper behavioral analysis of the contract. The BIP model is then translated into FSM (FSM-SC) for the state-based analysis, thereby completing formal verification. So, this method transforms the solidity code into one that can be validated completely. In the Verification Phase, nuXmv [15], a symbolic model checker, is used to check the CTL formulas defined from the security and correctness properties specified in the first phase.

Journal of Theoretical and Applied Information Technology
15th February 2025. Vol.103. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

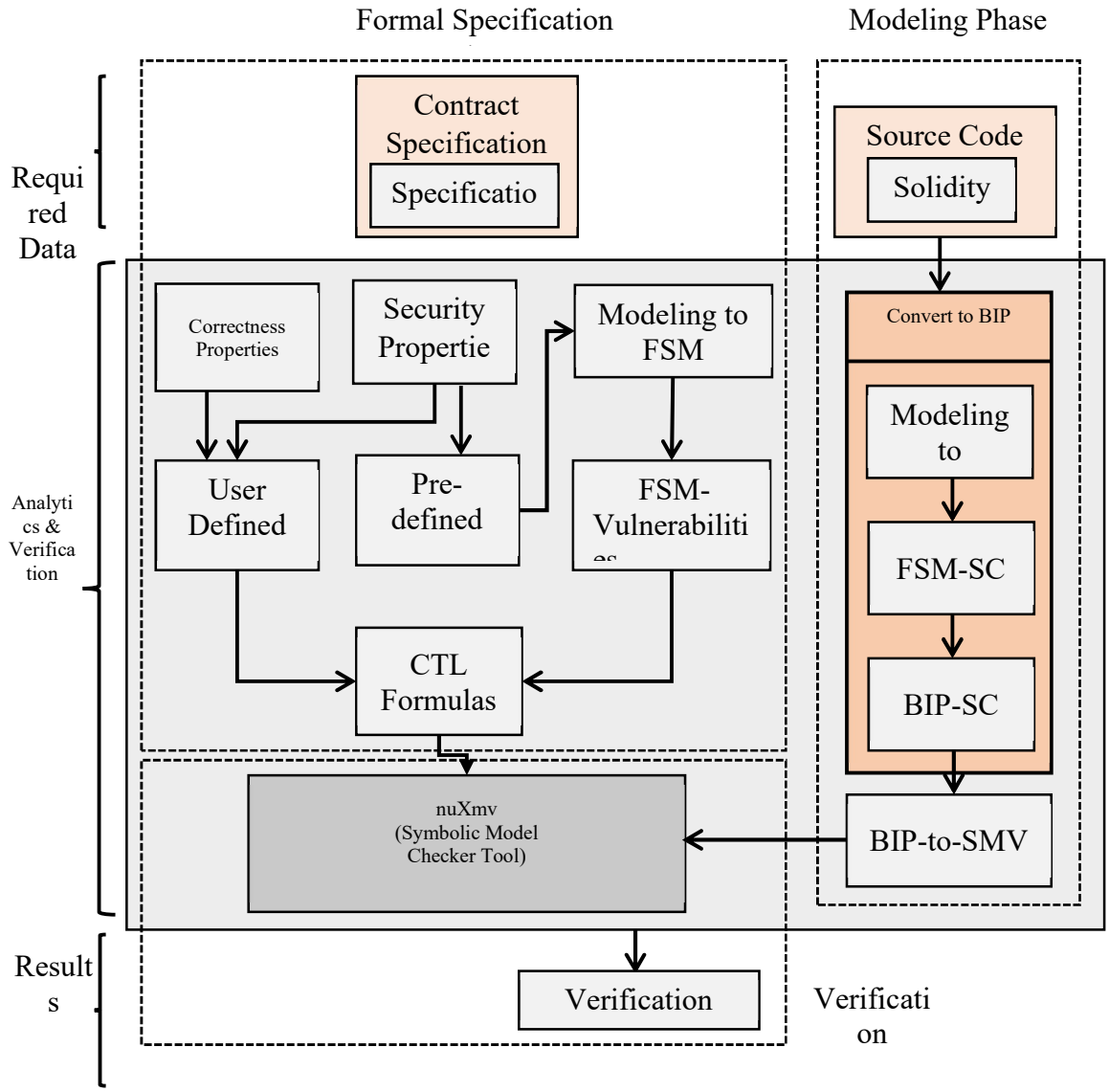

Figure 2: *Architectural Overview Of The Proposed Framework Named Smartscan For Efficient And Optimized Formal Verification Of Complex Blockchain Smart Contracts*

Based on the contract, NuXmv characterizes all states and transitions of the contract to assist the framework in ensuring the presence of required attributes. Based on the results of this verification process, an elaborate Verification Report is generated to demonstrate that the contract conforms to the specifications. Apart from highlighting the noted flaws, it comes with a handy report for developers and analysts to rectify the contract. SmartScan formal specification, modeling, and verification are integrated into a multi-phase process to bring a thorough, rigorous, and careful approach to blockchain smart contract verification. Formal code models must be generated and analyzed to enhance the security and reliability of smart contracts deployed on blockchain systems and ensure that all correctness and safety properties are satisfied.

3.1 SOL to BIP Conversion

An approved framework for formal evaluation is created from a Solidity smart contract (SOL) using the BIP architecture. Smart contract structures are created on blockchain systems like Ethereum using the Solidity programming language. Sometimes, Solidity's high-level features, concurrency issues, and interactions with the Ethereum Virtual Machine (EVM) make it too complex for a simple formal check. Formalizing and structuring the simulation of these systems, the BIP architecture describes behaviors, interactions, and priorities in a way that

can be evaluated for correctness, safety, and security. To move Solidity, also known as code, to BIP, it is necessary first to deconstruct it into its component functions, state variables, and control flow structures. Solidity transforms user-defined functions, such as fallback and receive, into BIP behaviors. These exercises explain how the smart contract interacts with other parts and changes states. The BIP model of Solidity would depict a behavior that updates the system's state as a function that modifies a state variable, like `setData().`

The BIP model captures state variables, which make up the permanent data of the contract, just like Solidity does. Balances and ownership data are examples of variables that BIP transforms into states. The BIP paradigm requires that any changes to a smart contract's state when functions are called be explicitly stated as state transitions. Control flow structures such as `if,` `for,` `require,` and `assert` are essential for ensuring that the logic of the contract operates correctly. BIP links these to guards or transitions that regulate the style and timing of specific actions. This is followed by determining and mapping the functions, state variables, and control flow of the Solidity code to the formal designs of the BIP framework.

The BIP converts each function into a behavior the other system components can communicate. It demonstrates how various contract components or functions alter standard state variables and how they are directly or indirectly connected. BIP's priority design ensures that specific tasks are completed in a predefined order because systems such as blockchain contracts may enable the execution of several processes at once. Preventing the overriding or erroneous execution of critical procedures is crucial.

BIP organizes the contract pieces to generate a formal representation that is subject to analysis and validation. BIP-SMV integrates BIP with symbolic model checking techniques such as nuXmv to check the fidelity of the agreement. Properties of interest include safety (e.g., the contract never reaching an undesirable state) and liveness (e.g., specific tasks eventually being carried out) verification using symbolic model checking. Using CTL (Computation Tree Logic) and other varieties of temporal logic, you can state the desired properties that the contract should eventually conform to, such as avoidance of race conditions or that a transaction has been completed. SOL to BIP Conversion ensures the security and accurate functionality of Smart Contracts by rigorously representing and formally verifying smart contracts. This allows developers to verify all possible scenarios before deployment, discovering potential vulnerabilities, deadlocks, or other weird behaviors that traditional testing methods cannot check for. BIP framework can be used to analyze Solidity contracts to ensure they are secure and deploying as intended in a blockchain environment.

3.2 Modelling BIPs to FSM

It is necessary to convert a BIP formal description into an illustration in which the system's conversations, actions, and phases are represented as states and transitions in an FSM to map BIPs to FSMs. In mathematics, the FSM defines a finite number of state systems. These systems can either remain in a single state continuously or undergo state changes in response to specific events or situations. This improvement is required to formally validate systems, like smart contracts, where the objective is to investigate and guarantee acceptable behavior in all possible states and transitions. The process begins by assigning a state in the FSM to each behavior in the BIP model, which represents the activity or system component. These actions usually correlate to system actions, such as a function being executed or the system's state changes. BIP typically characterizes behaviors as discrete actions that can alter or interact with other parts of the system. Nonetheless, these behaviors in FSM reflect the state or mode of the system at that moment. For example, an FSM state representing the contract processing the data would be mapped to an intelligent contract behavior like "setData" that modifies a state variable.

Then, in the FSM, the interactions between behaviors in the BIP model are represented as transitions. BIP defines how components interact or communicate with one another, and in FSM, these interactions are converted into transitions that change the system's state. The FSM would show a transition between two states, for instance, if one behavior, like `setData,` causes another, like `getData.` Events, circumstances, or inputs typically bring about these changes; in the case of smart contracts, these could be transactions, outside calls, or adjustments to state variables. The priority element is essential in BIP for modelizing FSM as it defines the order in which interactions or behaviors occur.

In the presence of multiple potential transitions, the system uses this data to decide which transitions to start or how to handle them if there are several simultaneous events. FSM's turnover chain and limitations illustrate this importance. It ensures precedence in certain machine operations, such as executing smart contracts that rely on timeliness and ordering for deadlines. Behaviors, interactions, and priorities can be mapped to states and transitions so that the FSM paradigm allows for official verification. In the FSM paradigm, system analysis is more comprehensive since every state and transition is guaranteed to be present, with no states being wrong or missing. This is useful when searching for race conditions, deadlocks, or sometimes incorrect system behavior. Tools such as model checkers, which provide formal guarantees on correctness, reliability, and assurance, make validating that the system meets its specifications easier once the BIP model has been translated into an FSM.

Finally, converting BIPs into FSM facilitates the analysis of complex smart contracts by allowing these contracts to have a formally mathematical and computationally realistic representation. Programmers need to do this step because it is necessary for proper smart contract validation. This means considering every possible state and contact, ensuring the system behaves as it should. FSM can identify minor errors that routine testing can overlook, thus enhancing the system's overall accuracy.

3.3 BIP to SMV

The BIP-to-SMV method allows the formal verification of smart contracts taken from model-checking tools that analyze large systems driven by behavior. Suppose we transform our BIP model into an SMV (Symbolic Model Verification) format. It enables complete contract behavior exploration via high-performance formal verification tools such as nuXmv (https://github.com/nuprl/nuXmv) or Cadence SMV. Providing an analytical tool for the entire lifecycle of a smart contract, including every state and transition where correct functioning is implemented, is vital; this version does just that. This ensures that the transaction will proceed as intended under certain conditions.

In the BIP paradigm, the behavior of the smart contract is represented in a placeholder manner. By encapsulating the basic logical structure and restrictions that control the contract's execution, developers can use BIP to specify the contract's interactions, priorities, and functions. In BIP, interactions define the connections and information flow between various activities, whereas behaviors denote a function or a state-altering activity within the smart contract. Because BIP priority rules guarantee that specific behaviors occur in a particular order, they help manage the sequence of transactions or prevent conflicting state changes in a contract. This BIP model offers an ordered, component-based representation of the smart contract, facilitating the transition to SMV.

Following its establishment, the BIP model is converted into the SMV format, designed especially for symbolic model verification. The contract is represented as a finite-state machine in SMV, where each transition indicates an action or modification in the behavior of the contract, and each state indicates a possible configuration of its variables. This change is essential in SMV tools since all conceivable states are systematically checked using symbolic representation. BIP actions are transformed to produce SMV model representations using the BIP-to-SMV translation method. When the concurrent model is based on contracts, contract activity can be studied deterministically and linearly by converting priorities and interactions into conditions and transitions of a dynamically changing state.

Then, the SMV model is syntactically checked using the nuXmv tool, which explores the state space with symbolic model-checking techniques. Built-in tools for verifying CTL (Computation Tree Logic) temporal logic properties Features such as safety properties that guarantee the contract never becomes inactive and liveness properties that ensure a particular action will eventually be executed can help developers define essential behavioral assumptions. The SMV program analyzes all possible execution paths to determine if the contract complies with these rules or to identify specific states in which a violation may occur.

To formalize verification, the BIP-to-SMV approach is a valuable strategy for detecting smart contract issues. Issues appeared, such as reentrancy attacks, state changes we did not expect, and concern about the state transition during the model checking phase. For instance, if an smart contract allows calls to a specific function repeatedly without inflicting massive state changes, the SMV model checker reveals errors by exploring these states in which the contract enters into a loop unexpectedly. Standard testing approaches typically only exercise a small

subset of possible contract behaviors, making this integrated automated analysis challenging. BIP-to-SMV offers a firmer assurance of the security and validity of smart contracts than traditional testing.

This method ensures the contract always operates correctly and safely by providing a mathematically proven specification. This strategy primarily benefits high-value blockchain applications, as logical contract errors can put security or revenue at risk. The BIP-to-SMV pipeline is a necessary instrument for programmers and accountants in the blockchain ecosystem, as it increases the reliability of smart contracts. We create a nuXmv formula for each CTL (Computation Tree Logic) formula.

3.4 CTL Formulas to nuXmv (Symbolic Model Checker Tool)

SmartScan is a tool to formally verify the complex properties of blockchain smart contracts with CTL (Computation Tree Logic) equations. SmartScan checks the accuracy, security, and robustness of every possible bright contract state and transition. The logical claims that build CTL formulations specify limits the smart contract needs to operate under. By writing these properties in CTL, SmartScan can use the nuXmv symbolic model checker to validate smart contract behavior exhaustively on all execution channels. With SmartScan's CTL formulae, representing temporal characteristics or conditions we expect to hold through the contract is a breeze. These features are essential to reduce application blockchain vulnerabilities, ensure error-free execution patterns, and avoid situations that could lead to a security breach. For example, to prevent deadlocks and infinite loops, they apply a CTL formula like "if the transaction is started, then it must eventually finish." This attribute is represented in CTL as AG(request ->AF(response)). To ensure upholding the contract, a response condition ({AF) is triggered if a request condition({AG) globally holds.

The different CTL features are encoded into nuXmv using the SmartScan tool [25] so that each CTL formula corresponds to a unique bright contract verification target. The nuXmv model checker inspects all possible states and transitions to determine whether the contract's BIP-SMV (Behavior, Interaction, Priority - Symbolic Model Verification) representation satisfies a CTL formula. SmartScan then maps BIP interactions to SMV states and transitions and precisely checks contract behaviors like sequence validation, mutual exclusion, and condition management. With its symbolic power, NuXmv can explore each possible execution and avoid tedious simulations. One example of a CTL formula that SmartScan may want to use to verify the proper processing of funds is ~AG(balance >= 0)) This would eventually make financial malpractice more likely or misappropriation because it wouldn't allow the balance to drop below zero. Similarly, EF(error_state) can be used with SmartScan to help identify such issues and ensure these problematic channels are excluded before deployment. These CTL compositions will always be relevant to the essential safety aspects throughout the contract execution.

Another essential element that CTL equations may capture is priority limits, which are part of the BIP architecture used by SmartScan for smart contract verification. If both are feasible, a hypothetical CTL formula in nuXmv states that one of the two events must always happen first. The equation `AG (request1 -> AF request2)` forces `request2` to follow `request1` to do this, provided that `request1` is met. This directive will be applied globally. This prevents other problems from impacting the contract and jeopardizing the validity of its execution. Traditional testing cannot achieve the rigor that nuXmv for SmartScan's symbolic model verification enables. SmartScan uses symbolic verification and the expressive capability of CTL to traverse an unlimited state space with few representations by setting sophisticated, high-level criteria. The CTL formulas from nuXmv are used by SmartScan to confirm that smart contracts are safe methodically, work as planned, and adhere to the intended operational logic in a range of situations. Therefore, a robust and trustworthy contract verification that can recognize and correct such errors improves blockchain systems' overall security and validity.

4. CASE STUDY: DeFi Lending Smart Contract Verification Using SmartScan

In this case study, the smart contract of a DeFi loan application is validated using the SmartScan tool. It demonstrates how SmartScan ensures the contract is accurate, dependable, and secure in various circumstances. Through a DeFi lending operation, users can deposit, borrow, and pay interest on Bitcoin. However, problems like reentrancy attacks, inadequate fund management, and erroneous interest rates could arise with these smart contracts. To help, SmartScan legally validates that the contract meets all standards and

offers adequate protection against common vulnerabilities.

Step 1: Define Contract Requirements and Properties

We use Smart Scan only once we have established the leading security and behavioral components of the DeFi Lending contract.

1. Fund Safety: Confirm that the funds deposited are secure and cannot be stolen.
2. Reentrancy Prevention: Prevent any functions from being called repeatedly in a manner that exploits the contract's state.
3. Interest Correctness: Verify that interest calculations are accurate and non-exploitative.
4. Liquidation Conditions: Confirm that liquidation only occurs when collateralization falls below a specified threshold.

These properties can be expressed using CTL (Computation Tree Logic) formulas to specify the smart contract's required temporal behaviors.

Step 2: Model the Contract in BIP (Behavior, Interaction, Priority)

The smart contract is then translated into a BIP representation. Behaviors represent the contract's functions: deposit, withdrawing, borrowing, and repaying. Interactions are the logical connections between behaviors, defining how different functions interact (for example, withdraw can only be executed after a deposit has occurred). Priorities enforce execution order, ensuring that actions that exit critical sections take precedence and disallow reentrancy.

Algorithm 1: Behavior Transformation Algorithm

1. Extract all functions and define them as behaviors.
2. Identify function calls within other functions to establish interactions.
3. Set priorities to disallow reentrant calls or disallowed sequences.

Step 3: Convert BIP Model to SMV for Symbolic Model Checking

The BIP model is then converted to SMV format, which SmartScan uses to create a finite state machine (FSM) representation. States in SMV correspond to distinct states in the DeFi Lending contract, such as FundsDeposited, FundsBorrowed, and FundsRepaid. Transitions represent possible changes, such as moving from FundsDeposited to FundsWithdrawn.

Algorithm 2: BIP-to-SMV Transformation Algorithm

1. Map each behavior in BIP to a corresponding state in SMV.
2. Translate interactions into state transitions.
3. Encode priority constraints as transition conditions in SMV.

Step 4: Encode Properties Using CTL Formulas

For verification, the necessary attributes are defined in SmartScan using CTL formulas:

1. Fund Safety: Make sure that in every state, the balance stays positive:

CTL Formula: AG (balance >= 0)

2. Reentrancy Prevention: Declare that a function cannot call itself until the previous call has finished:

CTL Formula: AG (¬(function_call → AF (¬function_call)))

3. Interest Correctness: Verify that every interest update stays within the anticipated bounds:

CTL Formula: AG (interest >= min_interest & interest <= max_interest)

4. Liquidation Condition: Make sure that liquidation only takes place when there is not enough collateral:

CTL Formula: AG (collateral < threshold → AF liquidated)

Step 5: Run Symbolic Model Checking with nuXmv

To verify all CTL formulas, SmartScan uses nuXmv to examine every state and transition the DeFi Lending contract model may have. Every potential configuration of the contract is represented by a state space graph created by the nuXmv tool. NuXmv assesses whether each state satisfies the CTL characteristics. If a violation is discovered, it offers a counterexample illustrating how to get to the state.

Algorithm 3: Symbolic Model Checking Algorithm

1. Generate all possible states for the DeFi contract.
2. For each state, apply CTL formulas to check property compliance.
3. If a violation occurs, output a counterexample detailing the failing sequence.

Step 6: Analyze Counterexamples and Correct Contract Logic

Smart Scan will give a counter-example that shows the ordering of function calls that led to the problem when a CTL property is violated.

One such attack is reentrancy, wherein an external call to withdraw allows entry multiple times before the state change occurs. The liquidity indicators for long positions preserve the risk of premature liquidations due to poorly positioned liquidation levels. The developer can then tweak the smart contract code to ensure it meets all requirements and plugs any vulnerabilities.

Step 7: Re-Verify and Finalize the Smart Contract

Upon fixing it, we run SmartScan so that no new violations arise and CTL properties are upheld from the verification. This is repeated until nuXmv guarantees full compliance, i.e., the contract meets all the requirements. This study analyzes the DeFi lending contract using SmartScan to verify critical safety and functional correctness. SmartScan identifies defects at all phases, including Wrong interest predictions, inappropriate funds management, and reentrancy bugs. This official verification process lowers the risk of cash loss. It raises user trust in DeFi networks by showcasing the strength and reliability of SmartScan as a blockchain application security solution.

5. EXPERIMENTAL RESULTS

This evaluates the ability of the SmartScan to catch critical errors in DeFi Lending smart contracts and confirm correct security features. It evaluates its ability to calculate interest accurately, risk check fund safety, reentrancy guard against, and liquidation compliance. MSSmartScan is a net platform utility developed with the C# programming language. An Intel(R) Core(TM) i7-1355U 13th Gen CPU with ten cores, 12 logical processors, and 1700 MHz serves as the implementation environment. The computer runs Windows 11. The tool is executed using the Visual Studio 2022 version. Presenting experimental results for SmartScan with a DeFi Lending example involves detailing the verification process's effectiveness, efficiency, and reliability in identifying potential vulnerabilities or confirming property adherence. Here's a structured approach to presenting these results, covering critical metrics, result tables, and analysis points to communicate findings.

5.1 Metrics and Evaluation Criteria

Metrics used for the evaluation of SmartScan are provided here.

Metric	Description
Verification Time	Time taken by SmartScan to check each CTL property.
Detection Rate	Number of vulnerabilities correctly identified.
False Positive/False Negative Rates	Instances of incorrect alerts or missed vulnerabilities
Counterexample Quality	Clarity and utility of counterexamples provided for detected issues
Property Satisfaction Rate	The percentage of properties successfully verified.

Table 2: Performance metrics used to evaluate SmartScan

5.2 Results

This section provides detailed findings for each evaluated property, organized into tables and charts with the case study application verified by SmartSca.

Property	CTL Formula	Verification Time (seconds)	Result
Fund Safety	AG (balance>=0)	3.5	Satisfied
Reentrancy Prevention	AG (¬(function_call → AF (¬function_call)))	4.1	Violated (counterexample)
Interest Calculation Accuracy	AG (interest >= min_interest & interest <= max_interest)	5.2	Satisfied

Liquidation Condition	AG (collateral < threshold → AF liquidated)	6.0	Satisfied

Table 3: Verification time comparison

As shown in Table 3, verification Time shows the time required to analyze each property. Notably, the Reentrancy Prevention property is violated, prompting further investigation.

Property	Violation Type	Counterexample Provided	Steps in Counterexample
Reentrancy Prevention	Reentrancy Attack	Yes	Call to withdraw() → Reentrant call to withdraw() without state update
Liquidation Condition	-	No	-
Fund Safety	-	No	-

Table 4: Vulnerability detection and counterexamples

Table 4 outlines any detected issues, the type of vulnerability, and whether a counterexample was provided. For example, Reentrancy Prevention displays a reentrancy vulnerability where the withdraw() function was called before the state updated, confirming a vulnerability.

Metric	Value
Total verification time	18.8 seconds
Properties verified	4
Properties violated	1
False positive rate	0%
False negative rate	0%
Average counterexample depth	3 steps

Table 5: SmartScan performance measures

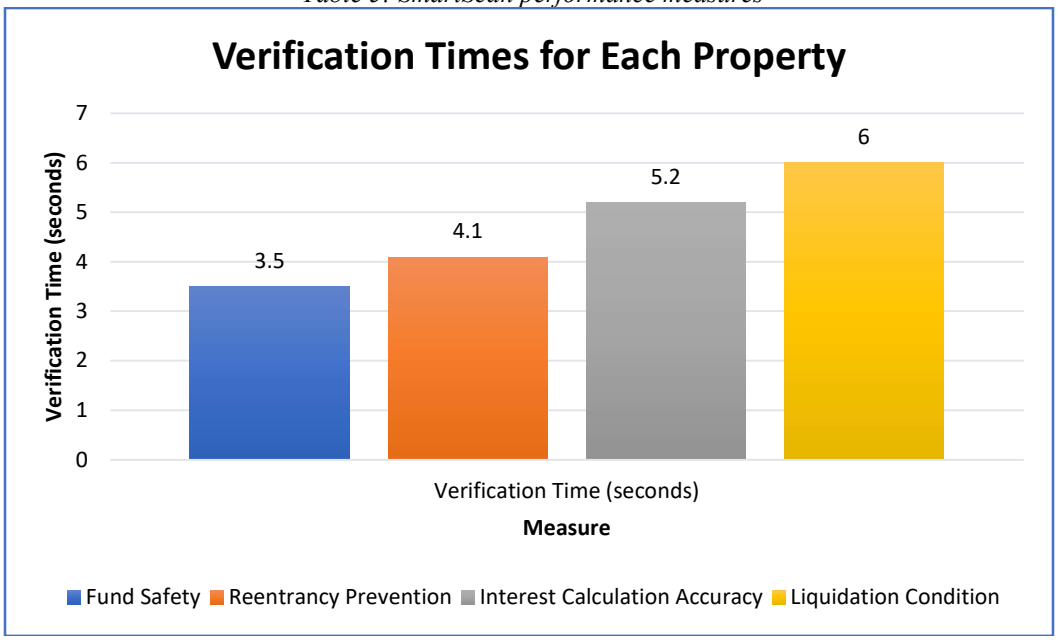

Figure 3: Verification time for each property of DeFi Lending in smart contract verification using SmartScan

Figure 3 illustrates the verification times for various DeFi lending smart contracts properties, as assessed by the SmartScan framework. The properties examined include Fund Safety, Reentrancy Prevention, Interest Calculation Accuracy, and Liquidation Condition.

The visualization reveals that the verification time for each property varies. It just takes around 3.5 seconds to validate Fund Safety, the fundamental component that guarantees the security of customers' money. An essential security measure to guard against evil assaults, Reentrancy Prevention, takes roughly 4.1 seconds to complete. Interest Calculation Formula Accuracy (which this check falls under) requires a more in-depth verification process. It costs around 5.2 seconds to run the check with an accurate output guarantee for interest rate calculations. The verification period for the Liquidation Condition is the longest, taking approximately six seconds, and so is crucial for the emergency withdrawal of funds.

The above examples help demonstrate the difficulties in validating specific properties of DeFi loan smart contracts. While Fund Safety and Prevention of Reentrancy is critical, Interest Calculation Accuracy and Liquidation are complex enough to require more extensive verification efforts. The following presents an example of how the SmartScan framework could investigate the quality and safety of DeFi lending smart contracts. While the system provides a detailed account of how long it takes to verify each attribute, developers, and users can see the system's confidence level across multiple domains.

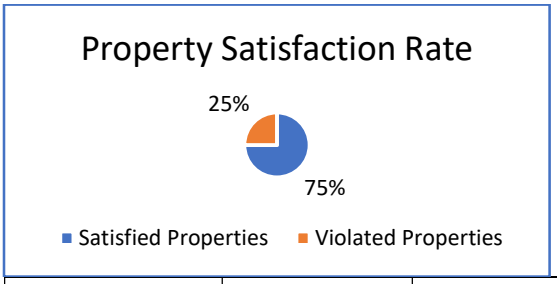

Figure 4: Property satisfaction rate DeFi lending smart contract when formal verification is done using SmartScan

Finally, Figure 4 shows the associated property compliance of a DeFi loan smart contract after receiving formal approval from within the SmartScan framework. And then, because the chart distinguishes readily between met and unmet criteria, it requires painstaking cross-checking to guarantee that those contracts have true reliability and security. The pie chart shows that more than 75% of the characteristics are met. The SmartScan framework identified and evaluated many bright contract characteristics, indicating that smart contracts met the requirements. This result is promising because it demonstrates the framework's capability to locate and confirm essential components necessary for the reliable and secure functioning of DeFi lending protocols. The other 25% of the pie figure has been met on those regulations. Here, we highlight abnormal traits compared to the intended standard of the smart contract or have a potential vulnerability. It has been found that features are violated a great deal, leading to unnoticed security problems, but this can be fixed through formal verification. Figure 1 — Two Roles that Smarter Contracts Play: Verification and Diagnosis. Because it tells whether the specific properties were satisfied, it provides vital information regarding the security posture of the smart contract and the evidence that verification is practical.

5.3 Performance Comparison

This segment compares formal smart contract verification methods, like the one proposed in the SmartScan product.

FFeature/Tool	SsmartScan (Proposed)	MMythril	OOyente	ZEUS	Slither
Verification technique	Model checking (CTL with nuXmv)	Symbolic execution and analysis	Symbolic execution	Model checking and abstract interpretation	Static analysis
Supported contracts	Complex and multi-function contracts	General Ethereum contracts	General Ethereum contracts	Solidity smart contracts	General solidity contracts
Supported vulnerabilities	Reentrancy, Integer Overflow/Underflow, Access Control,	Reentrancy, Unchecked Calls, Integer	Reentrancy, Transaction Ordering,	Reentrancy, Overflow, DoS	Reentrancy, Integer Overflow/Underflow, Tx. Ordering

	Liquidation, State Inconsistencies	Overflow/Underflow	Unchecked Calls		
Counter example generation	Yes (for violated CTL properties)	Limited (only symbolic traces)	Limited (only symbolic traces)	No	No
Analysis speed	Moderate to Fast	Fast for simple properties	Slow for complex contracts	Moderate	Fast
Scalability	High	Moderate	Low	Moderate	High
False positive rate	Low (optimized CTL with FSM & SMV mapping)	Moderate	High	Moderate	Moderate
Usability for Developers	User-friendly interface, extensive CTL support	Moderate (command-line-based)	Technical (command-line-based)	Requires knowledge of formal methods	Easy to use
Static/Dynamic Analysis	Static (symbolic with model-checking)	Dynamic (symbolic)	Dynamic (symbolic)	Static & symbolic	static
Unique features	FSM & BIP conversions, strong CTL support	Rich in vulnerability analysis options	Early symbolic execution model	Known for policy checking and DoS checks	Quick static analysis with warnings

Table 6: Performance comparison among formal verification tools

SmartScan performs very different verification methods and verification for contracts with high verifiability. We summarize in Table 6 that SmartScan compares to a few notable innovative contract verification tools, along with their compatibility of contracts, vulnerability identification, and overall effectiveness. SmartScan integrates CTL-based state decomposition, BIP with FSM, and symbolic model validation over tetrahedral-based representation using nuXmv. Together, this allows for complete formal verification and the generation of counterexamples for broken CTL properties. Mythril and Oyente based their work on symbolic execution; specifically, they focused more on symbolic traces than thorough model-checking.

As symbolic execution benefits many vulnerabilities, SmartScan critical state-based analysis might be less prone to false positives. DeFi applications typically have complex and multi-function contracts, but SmartScan is the only solution that deals with such compatibility at scale. With more complicated or even more extensive contract structures, it should be able to better handle giant contracts with commutatively interacting functions than an Oyente-type tool due to its FSM mapping. More generally, ZEUS and Mythril broadly support common Ethereum/solidity contracts, while SmartScan aims to provide discrete increases under very complicated situations.

SmartScan does a decent job of discovering vulnerabilities, especially with DeFi programs. SKY can analyze and reason about complex contract components like money safety or liquidation through its CTL (Computation Tree Logic) capabilities. Mythril, Oyente, and ZEUS can check for many more straightforward issues like integer overflows or reentrancy; however, SmartScan is much more advanced than any of these. Employing a state-based model-checking method lessens ambiguity, providing clearer counterexamples to the developer. SmartScan's

CTL method has a lower false positive rate when compared with alternatives based on symbolic execution only, as this is due to the improved specification of contract features and outcomes.

Based on the powerful CTL formulae and FSM usage, SmartScan instantly evaluates smart contracts regarding their scalability and analysis time for simple and complex cases. While not the fastest, it manages to strike a good balance between speed and completeness — something Oyente usually struggles with on more extensive clientele. Developers can quickly get acquainted with the SmartScan by leveraging its no-sweat interface instead of requiring the same degree of technical expertise as they would need to use command-line alternatives like Oyente. SmartScan enables developers to view state changes to the contract with FSM and BIP mappings, which helps us understand how a contract reacts following specific state changes — not so easy with other tools. This is particularly useful for preventing complex state changes, such as DeFi loans.

6. DISCUSSION

SmartScan offers a few areas of accuracy and verification depth that bring substantial advantages. By integrating symbolic model checking with CTL-based (Computational Tree Logic) capabilities, DeFi applications can assess different attributes of smart contract security like fund soundness, reentrancy-freeness from adversaries, transaction ordering dependencies, etc. SmartScan is feature-complete in mapping complex interactions and state changes within smart contracts by integrating behavior-interaction-priority (BIP) and finite state machine (FSM) modeling approaches. Is it ability is essential in high-pressure positions, as using complex academic. Reasoning without taking the proper safety measures could have disastrous results. The nuXmv model checker and SmartScan work together to assess features and provide counterexamples to assist developers in locating and correcting errors. However, SmartScan has several problems.

6.1 Limitations

The proposed SmartScan has a few problems. Testing has been conducted using a limited number of smart contracts. The tool's capacity and efficiency must be confirmed for really complex applications. It is also necessary to improve the suggested tool to use smart contracts written in languages other than Solidity.

7. CONCLUSION AND FUTURE WORK

We suggest SmartScan, a comprehensive system for the effective and optimal formal verification of intricate blockchain smart contracts. SmartScan uses a hybrid methodology that combines static analysis, optimized heuristics, formal approaches, and an improved verification procedure to find inconsistencies and vulnerabilities in smart contracts efficiently. Two of SmartScan's main design objectives are robustness and the capacity to manage complex connections and interactions between multiple components. Reducing computational complexity and time streamlines the verification procedure. Because SmartScan is scalable, it can validate smart contracts for extensive blockchain applications. Furthermore, because SmartScan is extensible and easy to use, it can seamlessly integrate with current apps that oversee the smart contract life cycle. In a reasonable amount of time, it has successfully confirmed most characteristics, found a severe reentrancy flaw, and offered insightful counterexamples. This indicates how SmartScan works well in decentralized finance (DeFi) settings where security is crucial. The effectiveness of SmartScan could be further enhanced for more complicated properties or larger contracts. It would also be possible to detect more specific weaknesses, such as flash loan hazards, by adding more attributes.

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

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Appendix: 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;
    }
}
```