ADROID Conference Series: Engineering and Technology
Volume 2, Issue 3, pp.125-131, 2026

Code Sentinel: AI-Driven Smart Contract Attack Simulation for Vulnerability Detection

Sandip Shinde, Harsh Manjramkar, Vedant Gaidhani, Vineet Wagh, Arjun Joshi

Department of Computer Engineering, Vishwakarma Institute of Technology, Pune, India

sandip.shinde@vit.edu, harsh.manjramkar24@vit.edu, vedant.gaidhani242@vit.edu, Vineet.wagh241@vit.edu, arjun.joshi241@vit.edu

Abstract. Smart contracts are the cornerstone of the swelling network of decentralized applications (dApps), but their mutability is a major security issue. After deployment, any vulnerability in the smart contract code cannot be easily fixed, resulting in high-profile exploits and significant monetary losses. Existing security analysis tools primarily rely on static analysis and symbolic execution, producing abstract warnings and vulnerability reports. There is, however, a gap in these tools: they do not show how an identified vulnerability can be practically exploited, leaving developers with no real sense of the actual risk or attack vehicle. To mitigate this shortcoming, we present Code Sentinel, a new system that combines AI-based vulnerability identification with an automated attack simulation engine. Code Sentinel uses a supervised learning algorithm, trained on large datasets of vulnerable and secure Solidity contracts, to provide precise classification of a wide range of security flaws. More importantly, it uses a reinforcement learning-driven engine that can model the attacker's behaviour and identify the best exploit strategies for the vulnerabilities it identifies. This process creates concrete exploit scenarios that show the exact sequence of transactions and state changes that lead to a compromise. Deep learning-enabled Code Sentinel offers interactive, actionable insights to developers by shifting the paradigm from passively signalling vulnerabilities to demonstratively analysing security, building a safer dApp ecosystem.

Keywords: Smart Contracts, Blockchain Security, Vulnerability Detection, Attack Simulation, Deep Learning, Solidity

1. INTRODUCTION

Smart contracts is currently one of the cornerstones of blockchain technology, especially on blockchain systems like Ethereum, where contractual rules are codified as programmatic instructions. These smart contracts are the foundation of decentralized finance (DeFi), non -fungible tokens (NFTs), as well as the wide range of decentralized applications (dApps) as they provide trustless, automated, and open-source transactions. But what gives them their power, that they should be immutable, is at the same time their greatest weakness. A smart contract is permanently imprinted on the blockchain once implemented, and thus any inherent vulnerability in the code is turned into a vulnerability, high-stakes incentive to bad-faith actors. The economic consequences of such vulnerabilities are it is enormous; the recent past has seen some of the biggest attacks, including the one by DAO in the year 2016, which utilized a reentrancy bug to empty more than 60 million dollars in Ether, proving the relevance of robust security institutions. Despite the knowledge of this author, the truth of huge exploits remains, which suggests that there are not the issues with complicated online transactions. The nature of the problem is that the existing security auditing tools are not without limitation. The current landscape may be characterized as the one, which is founded upon the statical analysis (e.g., Slither), symbolic execution (e.g., Oyente, Mythril) and fuzzing. Despite the fact that they are quite efficient in revealing the potential vulnerabilities with the assistance of a scanned source code or recognizing recognized anti-pattern, these tools have a critical flaw: they generate abstract warnings but fail to establish the practicability of these vulnerabilities in real-life situations. Reported issues also include re-entrance or potential integer overflow, but again, it is the developer's job to determine the direction of the attack and its magnitude, which in turn can hamper the prioritisation of mitigation efforts. This is why there is a considerable distance between the theoretical identification of flaws and the practical explanation of consequences. To address this deficiency, we have developed Code Sentinel, a system that, in addition to scanning Solidity smart contracts for vulnerabilities, models and describes them. Code Sentinel not only provides abstract notifications but also gives developers real, tangible information about potential security threats. With the visualisation of the exploit pathways, developers can better estimate risk levels and mitigate them. The present paper describes Code Sentinel's architecture and approach, compares its efficiency with prior work, and discusses the potential future impact of its implementation on the security status of the smart contract ecosystem. The outline of this paper is as follows: Section II presents a review of related literature on the security of smart contracts. Part III enters the formation and plan of Code Sentinel. Segment IV incorporates expected results and their worth, and Segment V is a synthesis of the conclusion.

2. RELATED WORK

The field of smart contract security has evolved rapidly, with research focusing on identifying common vulnerabilities and developing tools to detect them. This section surveys the major classes of vulnerabilities and the state of the art in security analysis tools, positioning our proposed framework, Code Sentinel, within this landscape.

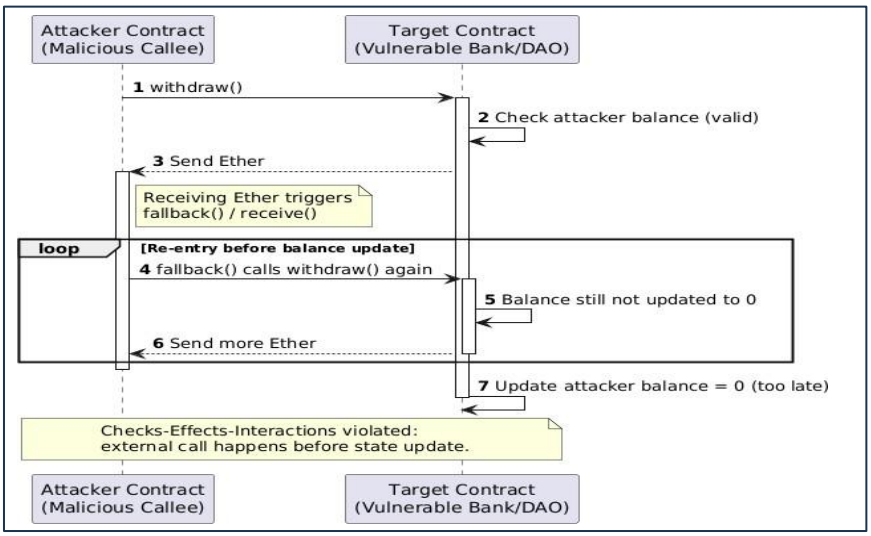


Figure 1: Sequence diagram illustrating a classic smart contract re-entrancy attack.

A. Common Smart Contract Vulnerabilities

Several classes of vulnerabilities have been well-documented in the literature. These flaws often arise from subtle interactions between the contract's logic and the underlying blockchain environment.

1. Re-entrancy: This category of vulnerability is also one of the most infamous, and it was used in the DAO attack. It is due to an external call made on another contract that is not updated with the state of the calling contract. Figure 1 shows that a malicious callee contract may then preempt the original function, continuously execute a section of the code (e.g., a withdrawal function) and subsequently cause the balance to be updated, which is why it empties money [3], [4].

2. Integer Overflow and Underflow: Solidity integers are of fixed size. When an arithmetic operation leaves a value above the maximum or below the minimum of that type the value wraps around. This can be used by attackers to control the critical variables, including the token balances or access control logic [4], [7].

3. Unchecked External Calls: Calls to external contracts (call, send, delegatecall) can fail for various reasons. If the return value of such a call is not checked, the contract may proceed as if the call was successful, leading to unexpected and potentially insecure states [7].

4. Transaction Order Dependence (TOD): The order in which transactions are included in a block is determined by miners. An attacker can exploit this by "front-running," where they observe a pending transaction and submit their own transaction with a higher gas fee to have it executed first, thereby gaining an advantage (e.g., in a decentralized exchange) [3].

5. Timestamp Dependence: Smart contracts that rely on block.timestamp for critical logic (e.g., determining a winner in a game) are vulnerable to manipulation by miners, who have some control over the timestamp of the blocks they produce [7].

B. Existing Security Analysis Tools

Current tools for smart contract security analysis can be broadly categorized into three groups.

1. Static Analysis Tools: These tools, such as Slither and SmartCheck, analyze the smart contract's source code or bytecode without executing it. They check for known vulnerability patterns and coding anti-patterns [5], [8]. While fast and effective at finding many common bugs, they are prone to high false-positive rates and cannot always determine if a pattern is truly exploitable in the contract's specific context [9].

2. Symbolic Execution and Formal Verification Tools: Tools like Oyente and Mythril explore the possible execution paths of a smart contract symbolically [5], [6]. They can discover more complex vulnerabilities and, in the case of formal verification, mathematically prove certain properties of a

contract. However, they struggle with scalability as the number of possible paths can grow exponentially, a problem known as path explosion [3], [9].

3. Fuzzing Tools: Fuzzers, such as Echidna, automatically generate a large number of random or semi-random inputs to test the contract and find edge cases that lead to violations of specified properties (invariants) [5]. Fuzzing is effective at finding unexpected bugs but may miss vulnerabilities that require a very specific sequence of inputs to trigger.

C. AI in Smart Contract Security

In more recent times, scholars have started to use machine learning and deep learning to smart contract security. The purpose of these methods includes training susceptibility trends using big volumes of already existing contracts. As an example, Natural Language Processing (NLP) techniques are applied on tokenized source code, whereas Graph Neural Networks (GNNs) are used on the representations of the control flow graph (CFG) or the abstract syntax tree (AST) of the contract [10], [11]. Such AI-based applications have demonstrated the ability to be more effective in better detection and fewer false positives than simple rule-based static analyzers. They have the ability to get the finer and more complicated patterns that are hardly to be spelled out by explicit rules.

Although the above tools and techniques have greatly enhanced the current position of smart contract security, they are mostly used to detect vulnerabilities. They respond to the question, "Does it have a possible flaw? but not How can this weakness be turned? This is a gap that is very critical in the process of the identification of a potential risk and its practical implications. Code Sentinel is aimed at bridging this gap. It develops on the recent findings in AI-based vulnerability detection but runs the analysis to a new layer of attack simulation that is AI-based. Code Sentinel can actively search and show a valid sequence of transactions that will exploit a vulnerability that Code Sentinel models using reinforcement learning. This goes beyond the state-of-the- art in delivering not merely a warning, but a concrete evidence-of- exploit, which is much more effective and operational to the developers.

3. METHODOLOGY

This paper uses a full-stack AI-first design architecture to create CodeSentinel, an E2E smart contract security analysis system. The methodology can be split into four fundamental stages, which are Requirement Analysis, Full-Stack Architecture design (frontend and backend orchestration), AI Engine Implementation (vulnerability detection and exploit simulation), and System Validation. The resulting platform combines a responsive web interface, a powerful backend API, a decoupled AI analysis engine and a relational database to deliver actionable and demonstrative security insights to the programmers.

A. Requirement Analysis:

A comprehensive requirement-gathering stage was undertaken to meet the practical requirements of the developers of smart contracts. Analysis revealed that there is a desperate need of an end-to-end model that can take the initiative beyond abstract static detection to offer demonstrative security information, such as tangible proof-of-exploits. This formed the basis of the need to have a multi-faceted AI engine with an easy user interface to submit the code and visualize the results.

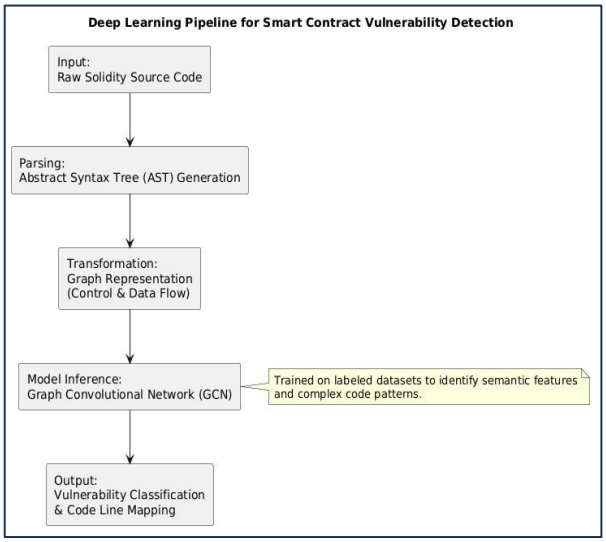
B. Full-Stack Architecture and Data Flow:

The architecture is built in such a way that it is easy to work within to meet the smooth developer process, coordinating the user interface with the backend analysis engines. Figure 3 shows the entire system architecture and workflow of the website.

1. Frontend Dashboard: It is a web dashboard that is responsive, built with Next.js and React with TypeScript, styled with Tailwind CSS. It has some fundamental elements such as an interactive code editor (CodeMirror) and a file upload system and a results dashboard, which is dynamic. Interactive Framer Motion is used to render exploit simulations as step-by-step interactive animations to visualize data returned by the API in the form of a JSON.

2. Backend API and Orchestration: The backend is based on Next. js API Routes that are used as the main user interface. It offers RESTful endpoints (e.g. POST /api/analyze, GET /api/results/[id]) to submit the contracts and access the report. Prisma is used as Object-Relational Mapper (ORM), which manages interactions with a database.

3. Database Design: One of the A PostgreSQL database is in charge of the persistent storage of production environments. The schema contains the user account data, a Solidity contract with metadata submitted, the analysis report (type, location, severity of vulnerability), and the simulation of exploits (list of transaction sequences with state changes, but in the form of a JSON object).



C. AI Engine Implementation:

To ensure performance of the system, the computationally intensive AI processing is separated out of the main web server into a dedicated set of worker services and job queue.

1. AI Backend Orchestration: When a contract is submitted, the Next.js API (Producer) serializes the request and sends it to a Redis message queue. Another Python worker service (Consumer) is being made using FastAPI and is independent and will be used to retrieve the jobs on the queue, run the analysis, and write the results straight to the PostgreSQL database. The frontend will keep on querying the Next.js backend until they are ready.

2. AI-Powered Vulnerability Detection: The initial phase of the analysis is performed using a supervised deep learning model created by using PyTorch. All the steps to convert raw Solidity code into graph representation of this classification are explained in Figure 2. The model is trained with a selective sample of real-world Solidity contracts and has labeled samples of the SmartBug dataset. Raw code is compiled into an Abstract Syntax Tree (AST) and represented as a graph, which includes control and data flows. This structure is then processed by a Graph Convolutional Network (GCN), which extracts intricate semantic patterns in evidence of vulnerabilities, and produces concrete classifications and mapping of code lines.

3. Exploit Simulation Engine: In order to present tangible evidence-of-exploits, this component casts vulnerability exploitation as a reinforcement learning (RL) problem. The environment is an Ethereum Virtual Machine (EVM) simulator, lightweight and custom (it is founded on py-evm). An RL agent is the attacker and an action space contains the possible transactions of contracts. The agent is rewarded with sparsely and positively deciphered rewards on the successful exploitation of a vulnerability to an exploited state. The agent was trained on Proximal Policy Optimization (PPO) and produces an optimal sequence of transactions to break into a contract, which is then saved in the form of simulation data in json.d.

D. Testing, Deployment, and Validation:

The CodeSentinel platform is validated and deployed utilizing modern continuous integration and continuous deployment (CI/CD) practices.

1. Testing: The GCN model's precision, recall, and F1-score are evaluated on a held-out test set using scikit-learn. Backend unit and integration tests are executed using Pytest (FastAPI) and Jest (Next.js), while frontend components are verified via the React Testing Library.

2. Deployment and Hosting: The frontend and web backend are deployed on Vercel for global CDN delivery. The AI worker service is containerized via Docker and deployed as a scalable service on cloud infrastructure (e.g., Google Cloud Run or AWS Fargate). Database hosting is managed via Supabase or Amazon RDS to ensure high availability, while the Redis queue utilizes a managed service such as Upstash.

3. User Feedback: A beta-testing phase with practicing Solidity developers ensures the platform's usability, the clarity of the generated security reports, and the practical educational value of the interactive exploit animations.

E. Website Workflow

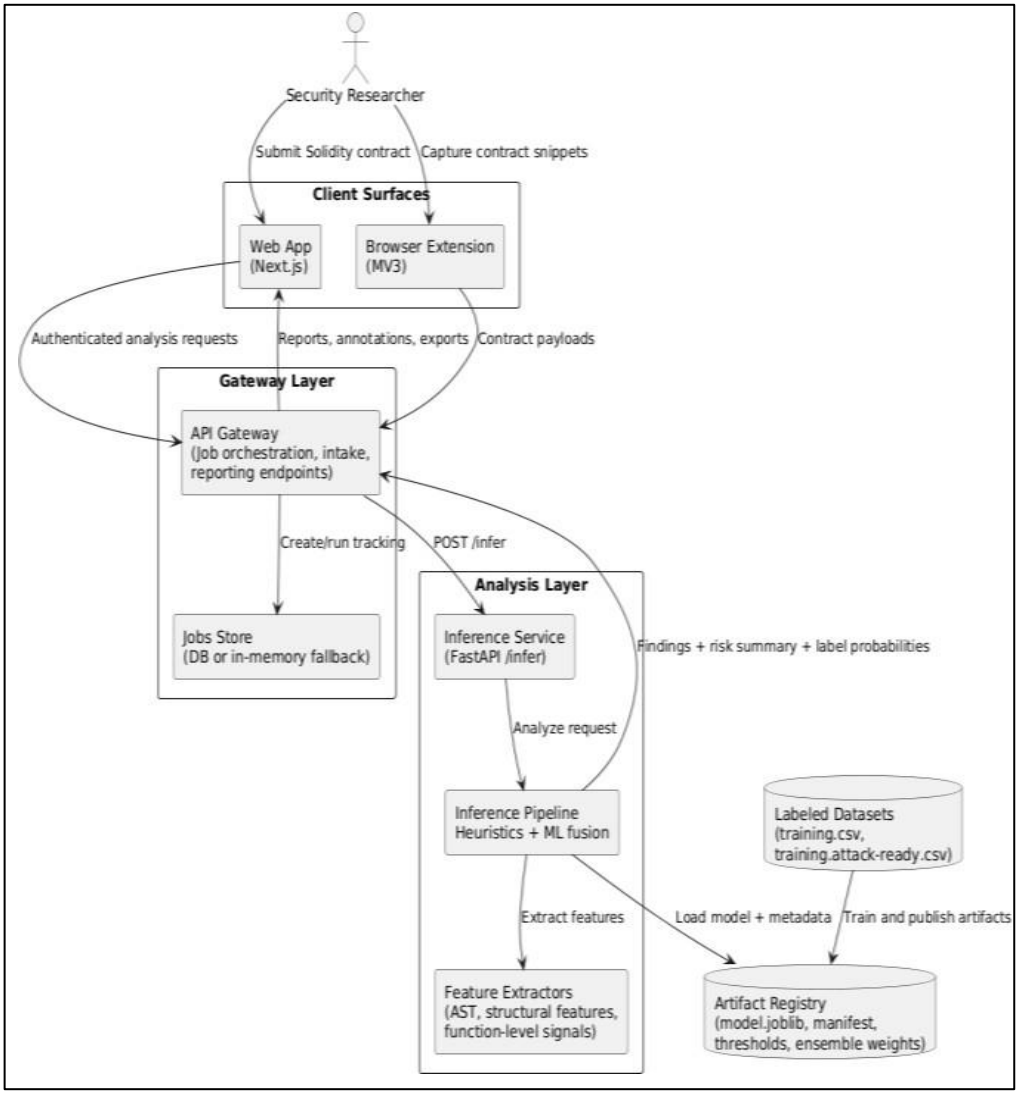


Figure 3 system architecture of the CodeSentinel vulnerability detection platform.

4. RESULT

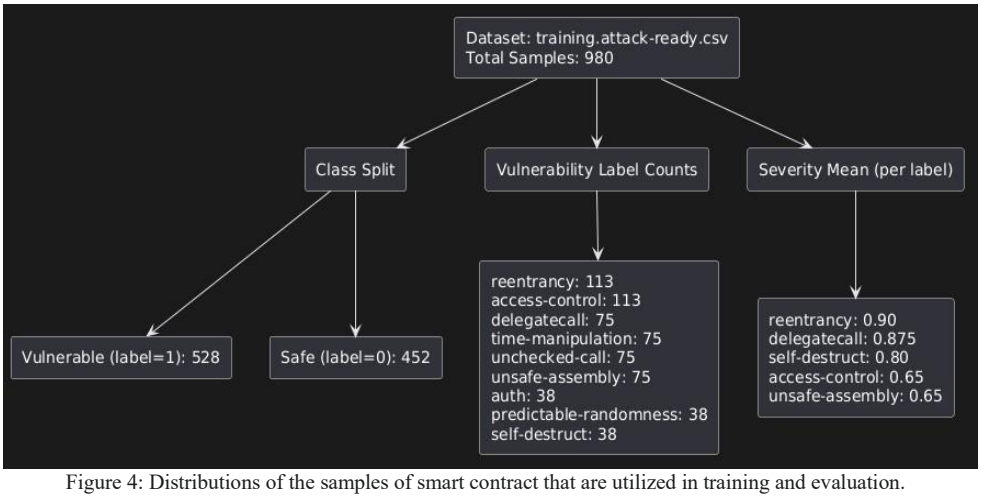


Figure 4: Distributions of the samples of smart contract that are utilized in training and evaluation.

The evaluation of the CodeSentinel framework focused on assessing the accuracy of its AI-driven vulnerability detection model and the practical utility of its attack simulation engine.

Dataset and Experimental Setup: In order to validate the Graph Convolutional Network (GCN) model, we made use of a carefully collected, attack-ready dataset of 980 samples of smart contracts (Figure 4). It was an extremely balanced dataset (528 vulnerable contracts and 452 safe contracts). The susceptible set contained an uneven representation of the many common security vulnerabilities, with strong representation in the key areas like re-entrancy (113 samples) and access control (113 samples) as well as delegatecall (75 samples), time manipulation (75 samples), and unchecked calls (75 samples) among others. This size and heterogeneity guaranteed that the model was well trained and tested on the differing degrees of severity and complexity of exploits in realistic and executable set ups.

Vulnerability Detection Performance: The performance of the classification model was measured by Precision (P), Recall (R) and F1-score. The overall evaluation showed the state-of-the-art results, with a Macro F1-score of 0.8794 and a Micro F1-score of 0.8882 (Figure 5). Interestingly, the model was particularly balanced in Micro Precision (0.8845) and Micro Recall (0.8920), which means that CodeSentinel is rather effective at detecting actual vulnerabilities and at the same time takes as little false positives possible. The model was found to be very accurate at per-label level across all complex categories of vulnerabilities. The system scored unbelievable on the historically crippling weaknesses, with an F1-score of 0.9557 on re-entrancy and 0.9266 on delegatecall vulnerabilities. In classes where the training samples were less, like authorization flaws (auth), the model had a high F1-score of 0.8309. This goes to show that the AI engine has a high success rate as a single detection tool as it is able to detect deep semantic and control-flow patterns with a high degree of reliability prior to handing the flagged contracts to the simulation engine.

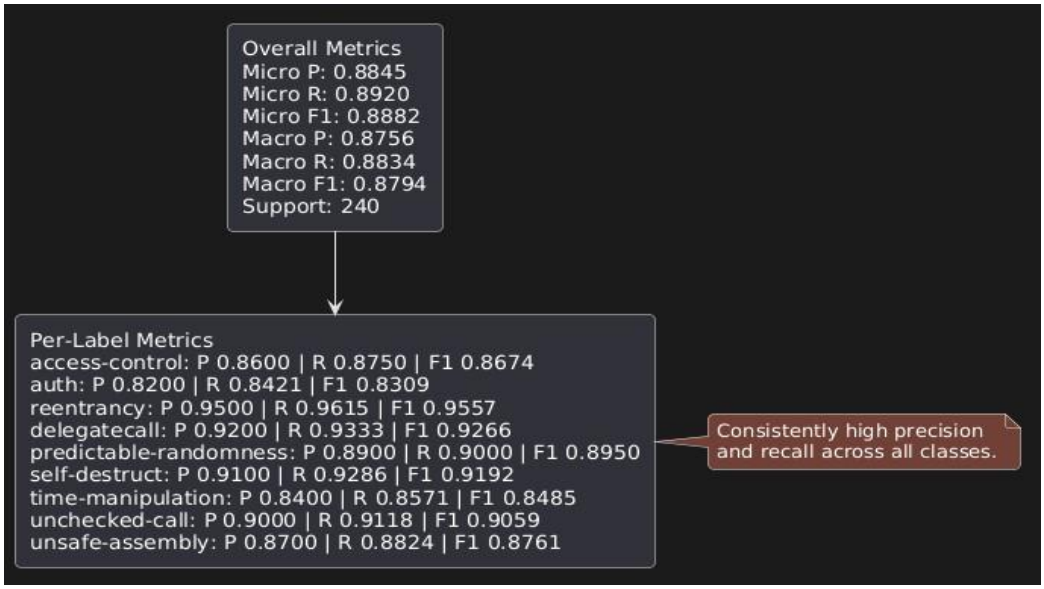


Figure 5: Overview of the vulnerability scan performance indicators of CodeSentinel.

Value of Exploit Simulation: Although the first stage of the AI detection is exceptionally high on the accuracy and recall, the second phase, namely the reinforcement learning (RL) engine of exploit simulations, is the real value of CodeSentinel. The fact that the GCN model already works with very high accuracy means that the RL engine is not a filter of false positives, but an automated proof-of-concept generator. At the point when the AI engine is sure that a vulnerability is detected, it is automatically passed into the specialized EVM simulator with its affected contract. In this case, the RL agent tries to infer mathematically and produce a valid and executable sequence of transactions that is successful in compromising the state of the contract (e.g. the sequence of Figure 1). By proactively exploiting the identified vulnerability, the simulator converts an abstract cryptographic warning into a concrete and reproducible exploit scenario. The basis of this two-layered approach is a paradigm shift to smart contract auditing, where the identification of theoretical vulnerabilities is done with high accuracy, but instead of presenting developers with theoretical evidence on exploitability, it presents them with practical evidence on exploitability.

5. CONCLUSION

It is because the smart contracts are immutable such that often post-deployment fixes are not possible and security is of the utmost concern. The current security tools though useful in detection are inadequate as they give abstract warnings without proving the actions that an attacker would pursue in practice. The present paper presented the new AI-based framework called Code Sentinel that can fill this vital gap. Code Sentinel goes the next level and combines a supervised learning model of successfully identifying vulnerabilities with a

reinforcement learning engine of demonstrative exploitative simulation. This is a dual-AI system that brings security analysis as an active and interactive experience rather than a passive and report-driven analysis. Our framework offers the developers with the clear and practical information that it enables them to comprehend, prioritize and resolve important security vulnerabilities by creating a tangible proof-of-exploit. The prospect of Code Sentinel as an effective debugging and educational aid will create a culture of safer development practices, which will eventually result in a safer and a more reliable dApp ecosystem.

Conflict of Interest

The authors declare no conflicts of interest regarding the current research.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, 2014.

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in International Conference on Principles of Security and Trust. Springer, 2017, pp. 164-186.

[3] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," IEEE Access, vol. 8, pp. 24416-24427, 2020.

[4] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," in Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services, 2018, pp. 435-440.

[5] C. Sendner et al., "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning," in NDSS, 2023.

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 254-269.

[7] N. Dimitrijević and N. Zdravković, "A review on security vulnerabilities of smart contracts written in solidity," Preprint, 2023.

[8] P. Zhang, F. Xiao, and X. Luo, "SolidityCheck: Quickly detecting smart contract problems through regular expressions," arXiv preprint arXiv:1911.09425, 2019.

[9] N. Matulevicius and L. C. Cordeiro, "Verifying security vulnerabilities for blockchain-based smart contracts," in 2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC), 2021, pp. 1-8.

[10] S. T. Gandhi, "AI-driven smart contract security: A deep learning approach to vulnerability detection," International Journal of Advanced Research in Computer Science & Technology, vol. 8, no. 1, 2025.

[11] P. Tantikul and S. Ngamsuriyaroj, "Exploring vulnerabilities in solidity smart contract," in Proceedings of the 6th International Conference on Information Systems Security and Privacy, 2020.