2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops) | 979-8-3315-3281-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ACSACW65225.2024.00037

Automated Vulnerability Detection in Smart Contracts using Control Flow Graphs and Machine Learning

Charles Lohest
ICTEAM/INGI
UCLouvain
Louvain-la-Neuve, Belgium
charles.lohest@uclouvain.be

Samy Bettaieb
ICTEAM/INGI
UCLouvain
Louvain-la-Neuve, Belgium
samy.bettaieb@uclouvain.be

Axel Legay
ICTEAM/INGI
UCLouvain
Louvain-la-Neuve, Belgium
axel.legay@uclouvain.be

***Abstract*—The security of smart contracts, a fundamental component of decentralized applications (dApps) on blockchain platforms, remains a critical concern due to the risk of severe financial losses from vulnerabilities. Traditional detection methods, such as fuzzing and symbolic execution, are effective at finding vulnerabilities but tend to be computationally expensive and time-consuming. This paper introduces a novel approach to vulnerability detection in smart contracts by combining machine learning with control flow graph (CFG) analysis. The proposed method trains machine learning models on optimized control flow graphs (CFGs) generated from pre-labeled smart contracts, enhancing both the speed and accuracy of vulnerability detection. Unlike existing tools, it operates solely on the contract's bytecode, eliminating the need for Solidity source or ABI files and thereby enhancing accessibility and applicability. The proposed system, SmartCFG, demonstrates superior computational efficiency and classification accuracy compared to traditional methods. While primarily designed to assist auditors, our approach provides a robust preliminary step in identifying potentially vulnerable contracts, complementing manual audits and enhancing overall smart contract security assessments.**

***Index Terms*—Smart Contracts, Vulnerability Detection, Control Flow Graphs, Graph-Based Learning**

1. Introduction

Smart contracts have emerged as a critical component of decentralized applications (dApps) on blockchain platforms such as Ethereum, enabling trustless and automated transactions without intermediaries. However, the security of smart contracts remains a significant challenge, as vulnerabilities within these contracts can lead to severe financial losses [28]. Traditional bug detection methods, such as fuzzing and symbolic execution, are widely used to detect vulnerabilities in smart contracts. These methods work by executing transactions step-by-step to explore numerous potential execution paths and identify any weaknesses or bugs [8], [12], [13]. While they can be effective, these techniques can be computationally intensive and time-consuming to uncover a single vulnerability. Figure 1 illustrates the time and resources consumed by different methods [5], [14], [16], [21]. Furthermore some tool also need some high level code

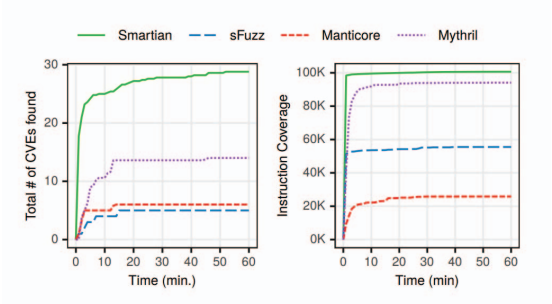

Figure 1. Comparison against state-of-the-art tools

such as Solidity file or complementary files to the bytecode called ABI files that may not be available publicly since the only part stored on the blockchain is the bytecode of the smart contract [19], [26].

To overcome these limitations, we propose a novel approach that combines machine learning and graph representation to enhance the speed and accuracy of vulnerability detection in smart contracts. Our research focuses on training a machine learning model using a dataset of pre-labeled smart contracts, each tagged with known vulnerabilities. Once trained, the model needs only the Control Flow Graph (CFG) of the smart contract under analysis, enabling it to efficiently assess and classify the contract as either vulnerable or secure. This method significantly improves efficiency, as generating a CFG is most of the time computationally inexpensive, typically takes less than a second and needs only the bytecode of the contract, providing a marked advantage over traditional techniques.

However, the primary target audience for this tool is not end-users of smart contracts but professional auditing firms, as noted in [1]. Given the high stakes often involving substantial sums of real money, it is crucial not to rely solely on automated tools, even those with high accuracy rates

Acknowledgments. This research is supported by the Walloon region's CyberExcellence program (grant #2110186)

979-8-3315-3281-9/24/$31.00 ©2024 IEEE
DOI 10.1109/ACSACW65225.2024.00037

(e.g., 99%). The proposed tool is designed to flag contracts that may be vulnerable, serving as a preliminary step in the auditing process. Its primary function is to perform an initial classification and provide an indication of whether manual review is necessary, based on the prediction confidence. Additionally, for contracts classified as vulnerable with high confidence, complementary tools can be used to verify the specific vulnerabilities detected. This tool helps auditors save significant time in the detection phase by analyzing the structural properties of contracts through CFGs, avoiding the need to execute the contracts directly.

By relying solely on bytecode, our approach, SmartCFG, extends beyond a single blockchain platform, supporting broader adoption across ecosystems and providing auditing firms, developers, and other stakeholders with a flexible and scalable security tool. Additionally, SmartCFG's adaptability to bytecode alone lowers the entry barrier, allowing it to be leveraged in contexts where source code access is limited, enhancing accessibility across the blockchain landscape.

While our method offers a rapid and scalable way to detect potential vulnerabilities, it is intended to complement, rather than replace, traditional audit practices. The flagged contracts should be manually reviewed and validated to ensure their security, given the critical nature of the financial transactions they govern. This cautious approach balances the need for speed with the imperative of accuracy and reliability, ensuring that the final verification of smart contracts is thorough and trustworthy.

Furthermore, existing methods for generating Control Flow Graphs (CFGs) often face significant limitations due to the large number of distinct operations, with approximately 140 opcodes in the Ethereum Virtual Machine (EVM). This high level of granularity complicates the application of machine learning techniques, as the resulting graphs are often overly complex and less meaningful for automated analysis. To overcome these challenges, we propose a novel approach for CFG generation that includes several targeted optimizations. These optimizations aim to create CFGs that are more comprehensive, context-aware, and effectively structured for machine learning integration. By reducing the complexity and enhancing the interpretability of the graphs, our method significantly improves the capability of machine learning models to accurately classify smart contracts as either vulnerable or safe.

Moreover, our approach introduces specific design considerations that align CFG construction more closely with the needs of machine learning workflows, thereby enabling more robust and reliable vulnerability detection. The efficacy of this improved CFG generation process will be thoroughly evaluated through a comparative analysis of the performance of machine learning models using graphs generated by our tool, SmartCFG, versus those produced by existing solutions like EtherSolve [6].

In Section 3, we introduce the Ethereum Virtual Machine (EVM), its opcodes, and graph-based learning methods crucial to our study. Section 4 presents our novel approach for generating optimized Control Flow Graphs (CFGs) from smart contract opcodes. We detail the techniques used to construct CFGs and their relevance in understanding the execution flow and logic of smart contracts. Finally, in Section 5, we examine a range of machine learning techniques that make use of CFGs, including Graph Neural Networks (GNNs), an SVM model utilising graph kernels, and the gSpan algorithm, to detect and predict potential vulnerabilities in smart contracts. We also compare the performance of models trained using CFGs generated by our tool, SmartCFG, against existing solutions like EtherSolve, and discusses the experimental results evaluating the efficiency of our method in detecting vulnerabilities. The results demonstrate how our enhancements contribute to better classification accuracy and reduced computational overhead.

The research further discusses how these models can be trained and optimized using datasets of labeled smart contracts, thereby providing a robust framework for vulnerability detection. By focusing on CFG-based analysis, our approach enables a more efficient initial assessment of smart contracts' security, identifying contracts that require more in-depth scrutiny.

2. Related works

Several tools currently exist for detecting vulnerabilities in smart contracts. However, as discussed in Section 1, many of these tools require access to the source code, necessitating the provision of a Solidity file or an ABI (Application Binary Interface) file, which provides additional instructions about the contract obtained during compilation. For instance, tools like Smartian [5] require both the bytecode and the ABI to function effectively, while Mythril [21] achieves significantly better performance when the solidity code is provided instead of the bytecode. Additionally, there are static analysis tools, such as SmartCheck [25] or Slither [8], that rely on pattern matching within Solidity code to identify vulnerabilities [4].

These dependencies impose notable limitations, particularly given that only the bytecode is always publicly available on the blockchain. While certain websites may offer the Solidity source code or ABI file [2], it is more advantageous to detect vulnerabilities directly from the bytecode alone, as proposed by the method in this paper. Moreover, many tools are optimized primarily for Ethereum, reducing their effectiveness on other platforms like Binance Smart Chain or Polkadot. By operating solely on bytecode, SmartCFG offers greater flexibility, making it adaptable across different blockchain environments. This bytecode-only approach removes dependencies on source code and expands the applicability of SmartCFG, enabling it to be more widely adopted across diverse blockchain platforms.

Furthermore, traditional detection tools tend to focus on identifying specific types of bugs and are optimized for finding particular issues, such as reentrancy vulnerabilities [17] or integer overflow/underflow errors [3]. This specialization limits their detection scope. In addition, our proposed tool leverages a dataset containing 20 different families of bugs exported from slither audit [15], enabling it

to identify whether a contract potentially has a vulnerability belonging to any of these categories. As a result, our tool offers a broader detection range across a more diverse set of contracts.

Additionally, once the model is trained, our approach provides faster performance because it can detect vulnerabilities almost instantaneously, even for test sets comprising hundreds of contracts, while maintaining high reliability, as demonstrated in Section 5. This broader coverage and improved efficiency mark a significant advancement over existing solutions, allowing quicker and more comprehensive vulnerability detection in smart contracts.

3. Background

This section outlines key concepts relevant to our study. We start with an overview of the Ethereum Virtual Machine (EVM) and its role in executing smart contracts. Next, we discuss opcodes, which are the fundamental operations within the EVM. We then cover graph-based methods used in analysis and learning, including the Weisfeiler-Lehman kernel, gSpan algorithm, and Graph Neural Networks (GNNs). Finally, we define the performance metrics used to evaluate the models in detecting smart contract vulnerabilities.

3.1. Ethereum Virtual Machine (EVM)

A smart contract is written in a high-level language, typically Solidity, and then compiled into bytecode before being deployed on the Ethereum blockchain. This bytecode is stored on the blockchain, ensuring the contract's immutability and transparency.

The Ethereum Virtual Machine (EVM) interprets and executes the bytecode, acting as a decentralized global computer. When a transaction interacts with a smart contract, the EVM processes the bytecode across all network nodes, ensuring consistent and deterministic execution.

The functionality of the EVM extends beyond mere execution; it also facilitates the reverse engineering of smart contracts. By accessing the binary code of smart contracts stored on the blockchain, it is possible to decompile this code to obtain a sequence of instructions, known as opcodes. These opcodes are low-level representations of the contract's operations and provide a granular view of its computational logic. Analyzing these opcodes allows researchers and developers to trace the execution flow of the contract, understand its behavior, and identify potential vulnerabilities or optimizations.

3.2. Opcodes

As previously mentioned, smart contracts on the Ethereum blockchain are stored as binary code, which can be decompiled using the Ethereum Virtual Machine (EVM) and represented as a series of operations known as opcodes. These opcodes provide a low-level representation of the contract's logic and are crucial for understanding the execution flow of the contract. Each opcode is structured as in Table 1.

TABLE 1. MAIN COMPONENTS OF AN OPCODE

Component	Description
Address	The location of the operation within the contract's memory
Instruction	The specific operation to be performed
Arguments	Additional data required for the operation

Opcodes are fundamental to the functioning of the EVM, as they dictate the specific instructions that the EVM must execute. These instructions can have various effects on the EVM's internal components, including the stack, virtual memory, and virtual storage. The stack is a last-in, first-out (LIFO) data structure used by the Ethereum Virtual Machine (EVM) to temporarily store information during contract execution. It holds critical data elements such as destination addresses for jump instructions, intermediate computation results, and addresses of dependent contracts, allowing the EVM to manage the flow of execution efficiently. In contrast, the virtual memory serves as a temporary storage area for dynamic data that arises during contract execution, such as function arguments, local variables, and intermediate data structures. Together, the stack and virtual memory work in tandem to facilitate the contract's execution, with the stack managing immediate operational data and the virtual memory accommodating more complex and variable-sized data requirements. Virtual storage, on the other hand, is a more permanent data structure that retains information between transactions.

The execution of opcodes can manipulate these components in various ways, such as pushing or popping values on the stack, reading from or writing to virtual memory, and modifying virtual storage. Understanding the behavior of opcodes and their impact on these internal components is essential for analyzing smart contracts, as it allows for a detailed tracing of how contracts operate at a fundamental level. This insight is valuable for various purposes, including security auditing, performance optimization, and reverse engineering of smart contracts to detect potential vulnerabilities or understand their logic.

3.3. Learning on graphs

Notation.. A graph $G = (V, E)$ is defined where V is a set of nodes and E is a set of directed edges, such that $\{\{u, v\} \subseteq V\}$ describes the edge from u to v. A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

Weisfeiler-Lehman (WL) kernel. The WL graph kernel [22] measures the similarity between graphs by iteratively refining node labels based on their neighborhoods. Its straightforward approach and efficiency in representing local structural features have made it a popular choice for graph classification tasks. The kernel is derived from the Weisfeiler-Lehman graph isomorphism test [27]. In this study, we refer to the SVM model using the WL kernel as WL.

gSpan. The graph-based Substructure pattern mining (gSpan) algorithm [31] is a graph mining algorithm designed to discover frequent subgraphs within a graph dataset, eliminating the need for the expensive candidate generation process. By efficiently navigating the graph space, gSpan uncovers substructures that recur across various graphs.

Graph Neural Networks. GNNs [20] are used to learn a representation vector for each graph, which is then used to predict the graph's label. GNNs iteratively update each node's embedding by aggregating information from its neighbours. At each step, a node's embedding is refined by the combination of its previous state with the aggregated embeddings of its neighbours. After several iterations, a readout function compiles the final node embeddings into a graph-level representation, which is used for classification. There exists a variety of GNN architectures [9]–[11], [29]

Graph Isomorphism Network. GIN is a type of GNN architecture shown to be as powerful as the Weisfeiler-Lehman graph isomorphism test [27], [29]. Using GINs, we can implement an architecture inspired by Jumping Knowledge Networks [30]. With this approach, the final representation of the graph is computed using the node embeddings over multiple iterations. These are then concatenated to form a comprehensive representation, rather than using a specific layer. Our GIN model employs this strategy.

Performance metrics. We evaluate our different models using different metrics, calculated from four key indicators:

- *True Positives (TP)*: Vulnerable smart contracts correctly identified as vulnerable.
- *False Positives (FP)*: Non-vulnerable smart contracts incorrectly identified as vulnerable.
- *True Negatives (TN)*: Non-vulnerable smart contracts correctly identified as non-vulnerable.
- *False Negatives (FN)*: Vulnerable smart contracts incorrectly identified as non-vulnerable.

The primary metrics used for evaluating our models include:

- *Accuracy*: The proportion of correctly classified smart contracts (both vulnerable and non-vulnerable). $\frac{TP+TN}{TP+TN+FP+FN}$
- *Precision*: The proportion of correctly identified vulnerable smart contracts. Low precision indicates that many non-vulnerable contracts are incorrectly classified as vulnerable. $\frac{TP}{TP+FP}$
- *Recall*: The proportion of actual vulnerable smart contracts correctly identified by the model. Low recall suggests that many vulnerable contracts are missed. $\frac{TP}{TP+FN}$
- *F1-Score*: The harmonic mean of precision and recall, which balances the two metrics and is particularly useful for imbalanced datasets. $\frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$

4. SmartCFG

The source code of smart contracts can be retrieved from the blockchain and is composed of two parts: the creation bytecode and the runtime bytecode. In this study, we focus on the runtime bytecode, as it encapsulates the specific instructions that govern the contract's behavior during execution. The runtime bytecode provides a comprehensive view of the contract's operational logic and its interactions with the Ethereum Virtual Machine (EVM).

4.1. Basic Block construction

After decompilation, a list of opcodes is obtained, which can be categorized into several distinct types to simplify the process of node labeling. This categorization significantly reduces the number of unique operations represented in each node's content, as it consolidates a broad range of individual opcodes from 140 different opcodes into only six distinct categories. This reduction not only optimises the creation of the graph but also enhances the efficiency of machine learning algorithms applied to it, as this simplified categorisation makes it easier to identify patterns and makes the analysis more manageable. The six distinct categories are the following:

- **Arithmetic**: Operations related to mathematical computations.
- **Block**: Instructions that interact with block-specific information.
- **Stack**: Operations that manipulate the stack, a last-in, first-out (LIFO) data structure used during execution.
- **System**: Instructions that handle low-level system functions.
- **Environmental**: Operations that interact with the external environment, such as retrieving data from the blockchain.
- **Control Flow**: Instructions that direct the execution flow of the contract.

At this stage, our primary focus is on the *Control Flow* category. This category includes the JUMP and JUMPI instructions, which are fundamental for managing the execution path of a smart contract and for constructing its control flow graph. A JUMP instruction must always target an address containing a JUMPDEST instruction, which serves as a valid destination for control flow redirection.

While both JUMP and JUMPI perform similar functions, there is a critical difference: JUMPI is a conditional jump, creating two potential execution paths—one leading to the address most recently obtained from the stack, and another continuing sequentially to the next instruction. In contrast, the JUMP instruction is an unconditional jump that directs execution flow directly to the address retrieved from the stack.

To facilitate the construction of a coherent execution path, we divide the instructions into distinct segments known as "Basic Blocks." Each basic block begins with a JUMPDEST instruction and typically ends with a JUMP or JUMPI instruction. However, exceptions may occur in cases where a JUMPDEST appears before a JUMP instruction; in such instances, a new basic block is still created to maintain the integrity of the control flow analysis.

4.2. Graph creation

After constructing the basic blocks, we proceed to build the control flow graph (CFG) by tracing the flow of execution between these blocks. In the CFG, each node represents a basic block, and the edges indicate possible transitions between these blocks. The destination of the next basic block is typically stored in the last instruction executed, which is often a JUMP or JUMPI instruction.

In most cases, the address of the next basic block is determined by a PUSH instruction that precedes a jump command, pushing the target address onto the stack. When this occurs, the address of the subsequent block can be easily identified by examining the second-to-last instruction within the current block. However, there are instances where the jump destination is more difficult to resolve, leading to the creation of what we refer to as ”orphan blocks.” These blocks lack a clearly defined destination address, complicating the graph construction process.

To accurately resolve these complexities, our implementation employs a symbolic stack that mimics the behavior of the Ethereum Virtual Machine (EVM). This symbolic stack performs operations such as PUSH, POP, and SWAP, allowing us to emulate the execution of EVM instructions. Through this mechanism, we can accurately retrieve destination addresses for each PUSH operation. After the symbolic stack is managed correctly, we can simply pop the stack to obtain the jump address. In the case of a JUMPI instruction, we also generate an edge to the next basic block, representing both possible execution paths.

Additionally, we implement symbolic representations of storage and memory to more precisely simulate the behavior of the control flow graph.

To prevent infinite loops and minimize redundant paths within the graph, we store all edges in a dictionary structure, with a default maximum repetition limit set to 20. This approach effectively constrains the graph's complexity while ensuring that all potential execution paths are adequately explored.

5. Experimental results

Our experimental results indicate that SmartCFG achieves notable efficiency improvements, demonstrating value in both small-scale contract analysis and large-scale contract audits. This efficiency is particularly beneficial to auditors who need to screen numerous contracts quickly. By performing rapid preliminary assessments, SmartCFG can help auditors prioritize high-risk contracts for further investigation. Additionally, SmartCFG's bytecode-only method shows promise in cross-platform adaptability, making it well-suited for diverse auditing scenarios and stakeholder needs. This adaptability makes SmartCFG a valuable tool in environments where access to full source code is not always feasible, enhancing its usability across a broader range of real-world conditions.

We first describe the dataset used for both training and testing, followed by a detailed explanation of the experimental setup. We then assess the speed performance of our approach before moving on to a comparative analysis of various models, such as GIN, gSpan, and WL kernels, applied with two different graph construction strategies. Finally, we highlight the advantages of our top-performing model, particularly in comparison to traditional methods and a deep learning-based approach.

5.1. Dataset selection

The dataset utilized in this study comprises a collection of smart contracts identified as vulnerable, sourced from the Slither audit framework [8]. This audit provides several thousand distinct contracts, each associated with various families of vulnerabilities. To ensure a comprehensive dataset, we supplement this collection with contracts deemed ”safe,” which are obtained from Google BigQuery [7]. By combining these sources, we construct a substantial dataset comprising approximately equal numbers of vulnerable and safe contracts, thereby ensuring a balanced representation of each category. This balance is crucial for training and evaluating machine learning models, as it allows for a fair assessment of the model's ability to classify both types of contracts accurately.

To optimize performance, we select a subset of this dataset for training and testing our models. It is crucial to maintain distinct training and testing datasets to prevent overfitting and to ensure the generalizability of the models. Additionally, we guarantee that the dataset is balanced by maintaining an equal distribution of vulnerable and safe contracts in both the training and test sets. This balance ensures that both classes are sufficiently represented, allowing the model to learn effectively from each category and be fairly evaluated on its ability to classify both types of contracts.

Each contract in the dataset is labeled to indicate its vulnerability status. Vulnerable contracts are assigned a label of 1, while safe contracts are labeled as 0. This labeling facilitates supervised learning by clearly distinguishing between vulnerable and non-vulnerable contracts, allowing the models to learn and predict with greater accuracy.

This carefully curated and balanced dataset enables robust training and evaluation of machine learning models, enhancing their ability to accurately classify smart contracts based on their vulnerability status. The approach ensures that the models are trained on a representative sample of both vulnerable and safe contracts, thereby improving their performance and reliability in real-world applications.

We perform our evaluation using 3 sizes for the train set: 500, 1000 and 4000 samples and a single test set of 200 samples.

5.2. Experimental setup

We evaluate the models and our graph building strategy on a machine with Intel Core processor CPUs (13th Gen Intel® Core™ i7-1355U × 12) and 16GB RAM, running Ubuntu 22.04.4 LTS.

We compare the models using different train set sizes.

We evaluate each ML algorithm: GIN, gSpan and WL (Weisfeiler-Lehman kernel), on two graph construction strategies: where each node is an instruction and where each node is a basic block.

5.3. Results and discussion

Speed evaluation:. Figure 1 demonstrates that our approach significantly outperforms other tools, particularly fuzzing and symbolic execution frameworks, in terms of speed. As noted in [4], these heavyweight analysis tools can take hours to detect vulnerabilities. In contrast, as shown in Figure 3, the combined time for training and testing on more than 600 samples never exceeds 50 seconds. More importantly, the time taken for testing alone, once the model is trained, is crucial to consider. We observe that the maximum testing time for 200 contracts is just 6 seconds, which averages to 0.03 seconds per contract.

Additionally, we compare the time required to generate control flow graphs (CFGs) with existing approaches [6]. As depicted in Figure 2, our tool, SmartCFG, takes 56 seconds to generate 500 CFGs, and up to 350 seconds for 4,000 contracts, averaging 0.08 seconds per contract. In comparison, the existing tool, Ethersolve [6], requires 150 seconds to generate 500 CFGs and over 880 seconds for 4,000 graphs, equating to 0.22 seconds per graph. This demonstrates that our method is approximately 2.75 times faster than Ethersolve.

When combining the time required for both CFG generation and testing, we can process around 200 contracts in just 22 seconds, enabling rapid detection of multiple vulnerabilities within a short period equivalent to 0.01 second per contract.

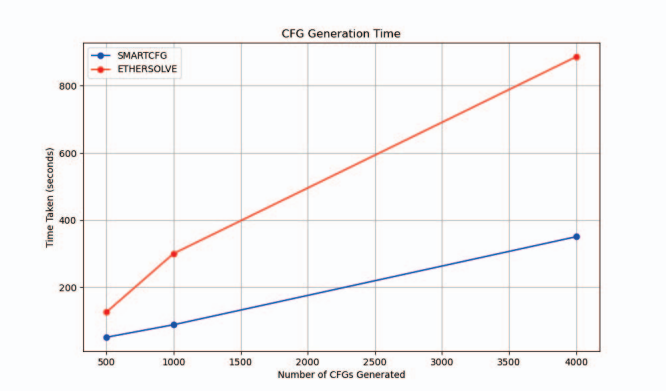

Figure 2. Time for CFG generation

Classification evaluation. $< Model >$ + *block* refers to the algorithm used (gSpan, GIN or WL) by the model with graphs using basic block for the nodes. $< Model >$ refers to the algorithm used by the model using graphs where individual opcodes are the nodes.

Table 2 illustrates the results of the initial comparison between the three ML algorithms using the two graph construction strategies on a train set of 500 samples. The

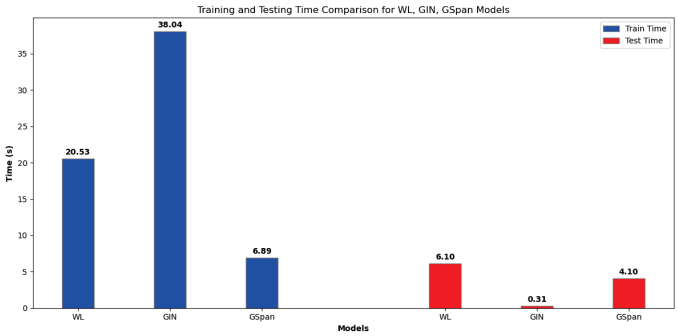

Figure 3. Training and testing time comparison for the different models

utilisation of this modest training set is intended to facilitate the deployment of the gSpan algorithm without unduly penalising it, as increasing the dataset size introduces significant variation in the data. This added variation makes it more difficult to identify subgraphs with ease.

Across models, using basic blocks as nodes in the graph consistently improves performance compared to using individual instructions. This shows that representing smart contracts using basic blocks is more effective for identifying patterns and vulnerabilities, as the complete blocks give more insight on the logic of the contract.

We observe that even though the GIN has slightly better accuracy and a better F1-score than the gSpan model, it has a much lower recall (62.6% compared to 86.2%). In the context of smart contract vulnerability detection, we assume that we favor a higher recall as it indicates less false negatives, or undetected vulnerable contracts.

Nevertheless, the WL model proves to be superior in all metrics, with a recall of 96,36% using basic blocks as nodes. Even without using the block structure and relying solely on individual opcodes for the nodes, the WL kernel achieves its highest recall of 80%. While this demonstrates a reasonable ability to capture vulnerability patterns. However 80% is still not particularly high, and prove that there is a significant difference in performance between nodes that use the block structure and those containing only a single instruction.

TABLE 2. PERFORMANCE COMPARISON (TRAIN 500, TEST 200 SAMPLES)

Model	Accuracy	Precision	Recall	F1 Score
GIN	0.5124	0.7412	0.5530	0.4213
GIN + block	0.6529	0.7025	0.6258	0.6027
GSpan	0.4919	0.4658	0.5862	0.5191
GSpan + block	0.6210	0.5618	0.8621	0.6803
WL	0.9008	0.9778	0.8000	0.8800
WL + block	0.9587	0.9464	0.9636	0.9550

Subsequently, the optimal model, designated as *WL + block*, is evaluated in comparison to the WL model on graphs generated by the algorithm of Ethersolve [6], utilising a training set comprising 1,000 samples. In Table 3 we observe that our approach outperforms method proposed

by Ethersolve, which employs over 140 different instructions [6]. We observe a perfect recall of 100% using the Ethersolve graphs, indicating that the model successfully identified all vulnerable smart contracts. However, the low precision (46.77%) shows that a significant number of legitimate contracts were incorrectly classified as vulnerable, highlighting the model's bias toward over-classification of vulnerabilities. This over-classification undermines the tool's usefulness for auditors, as it creates unnecessary work by flagging too many contracts for manual review, reducing the efficiency intended for low-confidence classifications.

Finally, in Table 4, we compare our best model with another tool that utilizes deep learning techniques, specifically LSTM and one-hot encoding, for classifying malicious smart contracts [23], [24]. Both models were evaluated using the same dataset: 4,000 samples for training and 200 samples for testing. The deep learning-based tool reported in their paper [24] an accuracy of 99%, while using our dataset it only reached 88.64%. On the other hand, by increasing the training dataset size we demonstrate that the accuracy and F1-score of our *WL + block* model improve, reaching 97.5% and 97.2% respectively, while keeping the same recall (without missing more vulnerable smart contracts). These results suggest that, on our dataset, our approach outperforms the existing tools that rely on binary code classification. This superior performance is likely due to the fact that the deep learning tool uses only opcode sequences for classification, while our method generates a more comprehensive representation of the contract.

We also considered a commonly observed proportion of safe and vulnerable contracts found in analyses such as [18], where approximately 20% of contracts are identified as malicious. Based on this, we conducted the detection process using a dataset with 20% malicious and 80% safe contracts, reflecting this distribution. This configuration allows us to assess the model's performance in a scenario closer to real-world conditions, grounded in the available audit data. The results obtained are summarized in Table 5.

We observe a high precision (97.06%), which can be attributed to the larger proportion of safe contracts in the dataset. While the precision is excellent, the recall (83.5%) is relatively lower, indicating that the model misses some vulnerable contracts. This is a common challenge in imbalanced datasets, where the model tends to focus more on the majority class (safe contracts). Despite this, the overall accuracy (96%) and F1 score (91.19%) demonstrate that the model maintains good balance in detecting both safe and malicious contracts. To address the lower recall, future work could focus on techniques such as data augmentation or cost-sensitive learning, to ensure the model is better at detecting vulnerable contracts without sacrificing precision.

TABLE 3. PERFORMANCE COMPARISON (TRAIN 1000, TEST 200 SAMPLES)

Model	Accuracy	Precision	Recall	F1 Score
WL + block	0.9256	0.8833	0.9636	0.9217
WL Ethersolve	0.4677	0.4677	1.0000	0.6374

TABLE 4. PERFORMANCE COMPARISON (TRAIN 4000, TEST 200 SAMPLES)

Model	Accuracy	Precision	Recall	F1 Score
WL + block	0.9752	0.9815	0.9636	0.9725
LSTM [23], [24]	0.8864	0.8696	0.9091	0.8889

TABLE 5. PERFORMANCE ON IMBALANCED DATASET (20% VULN, 80% SAFE)

Model	Accuracy	Precision	Recall	F1 Score
WL + block	0.9600	0.9706	0.8350	0.9119

6. Conclusion

In this paper, we introduced SmartCFG, a novel approach to vulnerability detection in smart contracts by leveraging machine learning and optimized Control Flow Graph (CFG) generation. Our method significantly enhances the speed and accuracy of vulnerability detection compared to traditional tools such as fuzzing and symbolic execution, which are often computationally expensive and time-consuming. By operating solely on the contract's bytecode, without requiring Solidity source code or ABI files, our approach improves accessibility and broadens the scope of vulnerability detection.

SmartCFG's reliance on bytecode alone makes it adaptable across various blockchain platforms, expanding its utility beyond Ethereum. This platform-agnostic approach allows SmartCFG to support a range of stakeholders, including auditors, developers, and blockchain users concerned with contract security. By enabling rapid, preliminary vulnerability identification across blockchain environments, SmartCFG can play a critical role as a first-line defense, flagging potentially vulnerable contracts and streamlining the audit process.

Through extensive experimentation, we demonstrated that using basic blocks as nodes in CFGs provides more meaningful insights into smart contracts' logic, leading to improved detection performance across various machine learning models. The *WL + block* model, in particular, showed the best results in terms of recall and F1-score, highlighting its effectiveness in identifying vulnerable contracts. Furthermore, our method outperformed existing solutions such as Ethersolve by offering faster CFG generation and classification, making it highly efficient for large-scale audits.

While SmartCFG offers robust initial classification, it is designed to complement, rather than replace, manual audits. By flagging potentially vulnerable contracts and leveraging prediction confidence, our tool assists auditors in prioritizing contracts for further review, thus streamlining the auditing process without compromising accuracy.

Overall, SmartCFG represents a significant advancement in automated smart contract auditing, providing a fast, reliable, and scalable solution for vulnerability detection. Its platform-agnostic approach and adaptability to various auditing scenarios reinforce its value in the broader blockchain

ecosystem, enhancing security practices without sacrificing efficiency.

References

[1] "Code4rena: Smart contract audit competitions," https://code4rena.com/, accessed: 2024-09-19.

[2] "Etherscan: Ethereum block explorer," https://etherscan.io/, accessed: 2024-09-19.

[3] 101 Blockchains, "Underflow and overflow vulnerabilities in smart contracts," https://101blockchains.com/underflow-and-overflow-vulnerabilities-in-smart-contracts/, 2023, accessed: 2024-09-24.

[4] S. Bonomi, S. Cappai, and E. Coppa, "Evaluating the vulnerability detection efficacy of smart contracts analysis tools," in *Computer Safety, Reliability, and Security*, A. Ceccarelli, M. Trapp, A. Bondavalli, and F. Bitsch, Eds. Cham: Springer Nature Switzerland, 2024, pp. 200–217.

[5] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *Proceedings of the International Conference on Automated Software Engineering*, 2021.

[6] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode," *CoRR*, vol. abs/2103.09113, 2021. [Online]. Available: https://arxiv.org/abs/2103.09113

[7] A. Day, "Ethereum in bigquery: a public dataset for smart contract analytics," https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics/?hl=en, 2018, accessed: 2024-09-19.

[8] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, May 2019. [Online]. Available: http://dx.doi.org/10.1109/WETSEB.2019.00008

[9] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[10] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[11] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[12] 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 (CCS)*. ACM, 2016, pp. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[13] F. Ma, Z. Xu, M. Ren, Z. Yin, Y. Chen, L. Qiao, B. Gu, H. Li, Y. Jiang, and J. Sun, "Pluto: Exposing vulnerabilities in inter-contract scenarios," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4380–4396, 2021.

[14] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[15] Mwritescode, "Slither audited smart contracts," https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts, 2023, accessed: 2024-09-24.

[16] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[17] OWASP, "Reentrancy attacks," https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC01-reentrancy-attacks.html, 2023, accessed: 2024-09-24.

[18] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," 2020. [Online]. Available: https://arxiv.org/abs/1902.06710

[19] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang, "Smart contract vulnerability detection technique: A survey," *arXiv preprint arXiv:2209.05872*, 2022.

[20] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[21] N. Sharma and S. Sharma, "A survey of mythril, a smart contract security analysis tool for evm bytecode," *Indian Journal of Natural Sciences*, vol. 13, no. 75, pp. 51 003–51 010, December 2022. [Online]. Available: https://www.researchgate.net/publication/366391033

[22] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *JMLR*, vol. 12, no. 77, pp. 2539–2561, 2011.

[23] W. J. Tann, "Safe smart contracts," https://github.com/wesleyjtann/Safe-SmartContracts/tree/master, 2023, accessed: 2024-09-19.

[24] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," *arXiv preprint arXiv:1811.06632*, 2018.

[25] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018, pp. 9–16.

[26] Y. Wang, S. Sheng, and Y. Wang, "A systematic literature review on smart contract vulnerability detection by symbolic execution," in *International Conference on Blockchain and Trustworthy Systems*. Springer, 2024, pp. 226–241.

[27] B. Weisfeiler and A. A. Lehman, "A Reduction of a Graph to a Canonical Form and an Algebra Arising During This Reduction," *Nauchno-Technicheskaya Informatsia*, vol. Ser. 2, no. N9, pp. 12–16, 1968.

[28] Wikipedia contributors, "The DAO (organization)," n.d. [Online]. Available: https://en.wikipedia.org/wiki/The_DAO

[29] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[30] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *International conference on machine learning*. PMLR, 2018, pp. 5453–5462.

[31] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE, 2002, pp. 721–724.