A Comparison of Machine Learning Techniques for Ethereum Smart Contract Vulnerability Detection

Matteo Rizzo[1,*], Dalila Ressi[1,*], Andrea Gasparetto[1] and Sabina Rossi[1]

[1]*Ca' Foscari University, Venice Italy*

Abstract

Vulnerability detection is particularly relevant in smart contracts, where modifying the code after deployment is impossible. Machine learning solutions provide greater efficiency than static analyzers in speed and detection. This study evaluates various classic machine-learning techniques and state-of-the-art neural networks for training a vulnerability detector. We analyze the largest and most reliably labelled dataset of smart contracts currently available, experimenting with six data representations of smart contracts and a multimodal approach. Our experiments show that both deep and traditional machine learning methods excel in different scenarios. Notably, eXtreme Gradient Boosting achieved an F1-score of 0.91 with the multimodal approach, which suggests its potential for more robust classification. At the same time, the results underscore the need for larger datasets to showcase the full potential of the evaluated methods.

Keywords

Machine Learning, Multimodal, Vulnerability Detection, Smart Contracts

1. Introduction

Smart contracts (SCs) are executable programs stored on a blockchain, an immutable distributed ledger that records data in linked blocks. Among the various blockchains supporting SCs [1], Ethereum is the most popular, as it was the first to introduce them as its primary feature. SCs span multiple applications, including decentralized finance, auctions, supply chain management, and real state transactions [2]. Public blockchains like Ethereum do not allow changes to the content of blocks. This protects the SCs from tampering but also hinders the programmers from fixing any bugs that may be discovered later in their code.

A relevant example is the "DAO Hack" Ethereum experienced in 2016, where a *reentrancy*[1] vulnerability caused the loss of $50 million worth of ether (ETH) and led to the hard fork of the blockchain. Identifying and securing potential vulnerabilities in the code beforehand is critically important, especially before deploying such contracts. This is a well-known problem in academia and industry, where standard solutions are based on formal verification and techniques such as static analysis. For this reason, several tools and frameworks have been designed to detect vulnerabilities in SCs, including Oyente [3], Mythril [4], Vandal[5], Securify [6], Smartcheck[7], and others [8]. Unfortunately, each of these solutions targets only a few specific security properties [9] and cannot be generalized to a broad spectrum of vulnerabilities [10]. Moreover, static analysis-based frameworks usually need much time to run and, in most cases, are not maintained over time [11].

Machine Learning (ML) and Artificial Intelligence (AI) can offer a valid alternative to tackle this problem [2]. Specifically, ML techniques like Support Vector Machines (SVM), Logistic Regression (LR), K-Nearest Neighbors (KNN), and ensemble methods like Random Forests (RF), Gradient Boosting (GB), and extreme gradient boosting (XGB) have been previously exploited to train vulnerability detectors on

OVERLAY 2024, 6th International Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, November 28–29, 2024, Bolzano, Italy

*Corresponding author.

matteo.rizzo@unive.it (M. Rizzo); dalila.ressi@unive.it (D. Ressi); andrea.gasparetto@unive.it (A. Gasparetto); sabina.rossi@unive.it (S. Rossi)

0000-0002-9723-211X (M. Rizzo); 0000-0001-5291-5438 (D. Ressi); 0000-0003-4986-0442 (A. Gasparetto); 0000-0002-1189-4439 (S. Rossi)

[1]https://www.immunebytes.com/blog/reentrancy-attack/

Ethereum SCs [12, 13]. On the other hand, ML and, in particular, Deep Learning (DL) models such as Long Short-Term Memory networks (LSTM) [14, 15], transformers [16] and Graph Neural Networks (GNN) [17] have been more recently investigated for this task. Despite the many proposed ML/AI-based vulnerability detectors, numerous challenges remain unresolved, making it difficult to compare different solutions [17]. Among these issues, the most notable are the lack of a large benchmark dataset, the unclear definition of vulnerabilities and ambiguous taxonomies [18], and the unreliable labelling process. In particular, a commonly adopted solution involves using various static analyzers to label the source code of Solidity-written SCs, which are either collected from Etherscan [19] or obtained from the large dataset called Smartbugs Wild [20]. Unfortunately, multiple studies report significant inconsistencies when using this approach due to the high number of false positives reported by various tools [21].

Contributions. We evaluate various ML and DL methods for vulnerability detection in Ethereum SCs, using the largest manually labelled dataset, Consolidated Groundtruth (CGT) [22], which, to our knowledge, has not been used for AI model training before. We expand the dataset by including graph representations and explore a multimodal approach, a novel strategy in this domain. Our experiments indicate that classical ML techniques tend to outperform state-of-the-art neural networks, particularly when applied multimodally. Our code has been made publicly accessible[2].

2. Dataset

Our target dataset is called "Consolidated Groundtruth" (CGT) [23], aggregating 13 benchmark sets of Ethereum SCs from the literature. It consolidates the ground truth by manually verifying the data associated with these contracts. This consolidation significantly reduces inconsistencies, redundancies, and incompleteness in the available data, ultimately improving its reliability and making it a valuable resource for evaluating SC analysis tools. The ground truth corresponds to three types of labels, fine-grained to coarse: *property*, *SWC*, and *DASP*. The property label relates to the specific label used in the original benchmark dataset to identify a weakness. These are mapped to a common taxonomy, namely the SWC (Smart Contract Weakness Classification) registry [3]. In addition, the authors also provide the mapping related to the more general taxonomy introduced in the DASP (Decentralized Application Security Project) Top 10 [4]. For instance, the property "reentrancy" corresponds to SWC number "107" and DASP class number "1".

The CGT dataset consists of 4,859 contracts and 160 different software weaknesses (some examples can be seen in Table 1). The data points are divided into 13 sets from independent sources, each named according to the publication that presented it. The sets include manually labeled datasets from CodeSmells [10], Zeus [24], eThor [25], ContractFuzzer [26], SolidiFI [27], EverEvolvingGame [28], Doublade [29], NPChecker [30], JiuZhou [31], SBcurated [21], SWCregistry, EthRacer [32], and Not-SoSmartC [5]. The unique characteristics of each set, such as class balance, data distribution, and the vulnerabilities considered (with some overlap across sets) are discussed in the original paper [23].

Each set's data is offered as source code, bytecode, and runtime, although not all data points include all three data types. The dataset contains 3871 source codes, 3076 bytecodes, and 3058 runtimes. From now on, we refer to source code, runtime, and bytecode as *modalities*, corresponding to different data representations of a contract. The *source code* is the contract written in Solidity. The *bytecode* is the compiled binary, also known as "contract creation code", which includes metadata and the actual *runtime* code executed by the blockchain. As part of our contribution, we add three new modalities: opcode, Abstract Syntax Trees (ASTs), and Control Flow Graphs (CFGs). The *opcode* (operation code), or *EVM opcode*, represents the part of a machine language instruction that tells the CPU what operation to perform. Here, we represent these instructions in a human-readable format, such as sequences of

[2]https://github.com/matteo-rizzo/smart-contracts-vulnerabilities-ml-detector
[3]https://swcregistry.io
[4]https://dasp.co
[5]https://github.com/crytic/not-so-smart-contracts

operations like PUSH, JUMP, and STORE. *ASTs* are tree-like structures representing the source code's logic, abstracting away syntax details. *CFGs* are graphical representations of a program's control flow, showing the order in which instructions or statements are executed. We generated opcode and ASTs from the source code using the "solc" Solidity compiler, while CFGs were generated using Slither [33].

Property	Definition	DASP	SWC	N. Contracts	Can be detected by
Unchecked External Calls	Do not check the return value of external call functions.	4	104	25	Oyente, Zeus, Contractfuzzer
Dos Under External Influence	Throwing exceptions inside a loop which can be influenced by external users.	5	113	6	Zeus
Strict Balance Equality	Using strict balance quality to determine the execute logic.	10	132	5	
Unmatched Type Assignment	Assigning unmatched type to a value, which can lead to integer overflow.	10	0	22	Zeus
Transaction State Dependency	Using tx.origin to check the permission.	2	115	5	Zeus
Reentrancy	The re-entrancy bugs.	1	107	11	Oyente, Zeus, Contractfuzzer
Hard Code Address	Using hard code address inside smart contracts.	10	0	84	
Block Info Dependency	Using block information-related APIs to determine the execute logic.	6	120	42	Oyente, Zeus, Contractfuzzer
Nested Call	Executing CALL instruction inside an unlimited-length loop.	5	128	13	
Deprecated APIs	Using discarded or unrecommended APIs or instructions.	10	0	217	
Unspecified Compiler Version	Do not fix the smart contract to a specific version	10	103	508	
Misleading Data Location	Do not clarify the reference types of local variables of struct, array or mapping.	10	0	1	
Unused Statement	Creating values which never be used.	10	135	10	
Unmatched ERC-20 standard	Do not follow the ERC-20 standard for ICO contracts.	10	0	45	
Missing Return Statement	A function denote the type of return values but do not return anything.	10	0	231	
Missing Interrupter	Missing backdoor mechanism in order to handle emergencies.	10	0	488	
Missing Reminder	Missing events to notify caller whether some functions are successfully executed.	10	0	27	
Greedy Contract	A contract can receive Ethers but can not withdraw Ethers.	10	997	6	Mayan
High Gas Consumption Function Type	Using inappropriate function type which can increase gas consumption.	10	0	352	
High Gas Consumption Data Type	Using inappropriate data type which can increase gas consumption.	10	0	0	

Table 1
Vulnerabilities considered by Codesmells [10].

3. Experimental Setup

This study explores the feasibility of using ML to detect vulnerabilities in Solidity SCs automatically. We approach this as a multiclass, multilabel classification task, as a single contract may be affected by multiple vulnerabilities. We evaluated ten different ML and DL models: the pretrained transformer CodeBERT [34], Feed-Forward Neural Network (FNN), LSTM, GNN, SVM, RF, LR, KNN, GB, and XGB. These models were selected based on their strengths and suitability for handling text data, graph data, imbalanced classes, and small datasets. We used pretrained Codebert from Huggingface, specifically model "microsoft/codebert-base". Concerning the tunable neural network architectures, we used three fully connected layers for the FNN, a recurrent layer followed by a fully connected layer for the LSTM, and three graph convolutional layers interleaved by batch normalization for the GNN. All neural models were trained for 50 epochs using the AdamW optimizer with an initial learning rate of 0.001. These were implemented using PyTorch v2.4.1. For SVM, RF, LR, KNN, GB, and XGB we used the default parameters from scikit-learn v1.5.1.

Source codes, bytecodes, runtimes, and opcodes were treated as text, while ASTs and CFGs were treated as graphs. Each modality was processed by a subset of the models we evaluated.

For text modalities, CodeBERT processed raw text, LSTM worked with Stanford's GloVe 100d word embeddings, and the remaining models utilized a Term Frequency-Inverse Document Frequency (TF-IDF) representation with 256 features. TF-IDF is a commonly used method for transforming text data into numerical features based on the importance of terms, which is effective for traditional ML models like SVM, RF, LR, KNN, GB, and XGB. LSTM, on the other hand, uses word embeddings to capture the semantic relationships and context of words, which is crucial for models that rely on sequential and contextual information in text data. Finally, CodeBERT is pre-trained on a large corpus of code, and it operates directly on the raw text, leveraging its programming language understanding to detect vulnerabilities in source code.

For graph modalities, we fed to a GNN, along with SVM, RF, LR, KNN, GB, and XGB, a set of graph features we developed. For ASTs, we captured the general characteristics of each node, such as the node type, its depth in the tree, the number of children, and a complexity measure (calculated as the product of depth and number of children). Additionally, we extracted node-specific features:

			Model									
Modality	**Definition**	**Metric**	GB	KNN	LR	RF	SVM	XGB	CodeBERT	LSTM	FNN	GNN
Bytecode	**Property**	**Precision**	0.82 ± 0.02	0.83 ± 0.02	0.82 ± 0.01	**0.84** ± 0.02	0.82 ± 0.01	0.83 ± 0.02	0.81 ± 0.03	0.82 ± 0.00	0.61 ± 0.01	-
		Recall	**0.80** ± 0.02	**0.80** ± 0.01	0.69 ± 0.01	**0.80** ± 0.02	0.69 ± 0.01	0.79 ± 0.02	0.71 ± 0.03	0.70 ± 0.00	0.77 ± 0.03	-
		F1-score	0.79 ± 0.02	0.79 ± 0.01	0.72 ± 0.01	**0.80** ± 0.02	0.72 ± 0.01	0.79 ± 0.02	0.74 ± 0.00	0.74 ± 0.00	0.66 ± 0.01	-
	SWC	**Precision**	0.87 ± 0.01	0.91 ± 0.01	**0.92** ± 0.00	0.91 ± 0.01	**0.92** ± 0.00	0.90 ± 0.02	0.91 ± 0.00	0.91 ± 0.00	0.78 ± 0.05	-
		Recall	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.84 ± 0.00	0.84 ± 0.00	**0.86** ± 0.01	-
		F1-score	0.83 ± 0.01	0.85 ± 0.01	**0.86** ± 0.00	0.85 ± 0.01	**0.86** ± 0.00	0.85 ± 0.02	**0.86** ± 0.00	**0.86** ± 0.00	0.79 ± 0.03	-
Opcode	**Property**	**Precision**	0.78 ± 0.06	**0.86** ± 0.07	0.26 ± 0.03	0.85 ± 0.07	0.22 ± 0.02	0.84 ± 0.07	0.58 ± 0.20	0.45 ± 0.03	0.37 ± 0.00	-
		Recall	0.76 ± 0.07	0.79 ± 0.06	0.99 ± 0.01	0.81 ± 0.04	**1.00** ± 0.01	0.81 ± 0.04	0.72 ± 0.10	0.72 ± 0.00	0.70 ± 0.00	-
		F1-score	0.75 ± 0.05	0.80 ± 0.06	0.40 ± 0.03	**0.81** ± 0.05	0.36 ± 0.02	**0.81** ± 0.04	0.61 ± 0.17	0.54 ± 0.02	0.48 ± 0.00	-
	SWC	**Precision**	0.76 ± 0.05	**0.88** ± 0.02	**0.88** ± 0.02	**0.88** ± 0.02	**0.88** ± 0.02	0.86 ± 0.05	0.82 ± 0.00	0.30 ± 0.00	0.84 ± 0.03	-
		Recall	0.74 ± 0.07	0.80 ± 0.02	0.80 ± 0.02	0.80 ± 0.02	0.80 ± 0.02	0.78 ± 0.05	0.82 ± 0.00	0.86 ± 0.00	**0.87** ± 0.02	-
		F1-score	0.73 ± 0.06	0.82 ± 0.02	0.82 ± 0.02	0.82 ± 0.02	0.82 ± 0.02	0.80 ± 0.05	0.82 ± 0.00	0.45 ± 0.00	**0.85** ± 0.03	-
Runtime	**Property**	**Precision**	0.82 ± 0.02	**0.84** ± 0.02	0.82 ± 0.01	**0.84** ± 0.02	0.82 ± 0.01	**0.84** ± 0.02	0.80 ± 0.06	0.82 ± 0.00	0.60 ± 0.01	-
		Recall	0.80 ± 0.02	**0.81** ± 0.02	0.69 ± 0.01	**0.81** ± 0.01	0.69 ± 0.01	0.80 ± 0.01	0.70 ± 0.00	0.70 ± 0.00	0.78 ± 0.03	-
		F1-score	0.79 ± 0.02	0.80 ± 0.01	0.72 ± 0.01	**0.81** ± 0.01	0.72 ± 0.01	0.80 ± 0.02	0.73 ± 0.03	0.74 ± 0.00	0.66 ± 0.01	-
	SWC	**Precision**	0.88 ± 0.01	0.91 ± 0.01	**0.92** ± 0.00	0.92 ± 0.01	**0.92** ± 0.00	0.91 ± 0.01	0.91 ± 0.00	0.91 ± 0.00	0.79 ± 0.03	-
		Recall	0.84 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.84 ± 0.01	0.84 ± 0.00	0.84 ± 0.00	**0.86** ± 0.02	-
		F1-score	0.84 ± 0.01	0.85 ± 0.01	**0.86** ± 0.00	**0.86** ± 0.01	**0.86** ± 0.00	**0.86** ± 0.01	**0.86** ± 0.00	**0.86** ± 0.00	0.79 ± 0.02	-
Source	**Property**	**Precision**	0.83 ± 0.02	0.85 ± 0.01	0.83 ± 0.01	**0.86** ± 0.02	0.83 ± 0.01	0.85 ± 0.01	0.80 ± 0.06	0.62 ± 0.00	0.63 ± 0.01	-
		Recall	0.82 ± 0.01	**0.83** ± 0.01	0.77 ± 0.02	**0.83** ± 0.01	0.77 ± 0.01	0.82 ± 0.01	0.70 ± 0.00	0.70 ± 0.00	**0.83** ± 0.03	-
		F1-score	0.81 ± 0.01	**0.82** ± 0.01	0.77 ± 0.01	**0.82** ± 0.01	0.78 ± 0.01	0.81 ± 0.01	0.73 ± 0.03	0.64 ± 0.00	0.70 ± 0.02	-
	SWC	**Precision**	0.88 ± 0.01	0.90 ± 0.01	**0.92** ± 0.00	0.91 ± 0.01	**0.92** ± 0.00	0.90 ± 0.01	0.91 ± 0.00	0.91 ± 0.00	0.79 ± 0.01	-
		Recall	0.83 ± 0.01	0.83 ± 0.02	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	**0.84** ± 0.00	**0.84** ± 0.00	**0.84** ± 0.03	-
		F1-score	0.83 ± 0.01	0.85 ± 0.01	**0.86** ± 0.00	**0.86** ± 0.01	**0.86** ± 0.00	0.85 ± 0.01	**0.86** ± 0.00	**0.86** ± 0.00	0.79 ± 0.01	-
AST	**Property**	**Precision**	0.74 ± 0.02	0.72 ± 0.02	0.69 ± 0.02	**0.76** ± 0.02	0.71 ± 0.02	0.75 ± 0.02	-	-	-	0.21 ± 0.00
		Recall	0.80 ± 0.02	0.78 ± 0.02	0.76 ± 0.02	0.81 ± 0.02	0.78 ± 0.02	0.80 ± 0.02	-	-	-	**1.00** ± 0.00
		F1-score	0.70 ± 0.02	0.71 ± 0.02	0.68 ± 0.02	**0.73** ± 0.02	0.70 ± 0.02	0.72 ± 0.02	-	-	-	0.33 ± 0.00
	SWC	**Precision**	0.74 ± 0.01	0.76 ± 0.06	0.74 ± 0.01	**0.77** ± 0.01	0.74 ± 0.01	**0.77** ± 0.01	-	-	-	0.19 ± 0.00
		Recall	0.74 ± 0.01	0.70 ± 0.09	0.74 ± 0.01	0.78 ± 0.01	0.74 ± 0.01	0.78 ± 0.01	-	-	-	**1.00** ± 0.00
		F1-score	0.64 ± 0.01	0.62 ± 0.02	0.64 ± 0.01	**0.66** ± 0.01	0.64 ± 0.01	**0.66** ± 0.01	-	-	-	0.32 ± 0.00
CFG	**Property**	**Precision**	0.79 ± 0.03	0.81 ± 0.03	0.78 ± 0.03	**0.84** ± 0.03	0.77 ± 0.03	0.81 ± 0.03	-	-	-	0.23 ± 0.00
		Recall	0.77 ± 0.03	0.76 ± 0.02	0.76 ± 0.03	0.77 ± 0.02	0.77 ± 0.03	0.77 ± 0.03	-	-	-	**1.00** ± 0.00
		F1-score	0.75 ± 0.02	0.75 ± 0.02	0.74 ± 0.02	**0.78** ± 0.02	0.74 ± 0.02	0.76 ± 0.02	-	-	-	0.37 ± 0.00
	SWC	**Precision**	0.93 ± 0.02	0.93 ± 0.01	0.96 ± 0.00	**0.97** ± 0.00	0.95 ± 0.00	0.97 ± 0.01	-	-	-	0.23 ± 0.00
		Recall	0.89 ± 0.01	0.89 ± 0.01	0.87 ± 0.00	0.89 ± 0.00	0.88 ± 0.00	0.89 ± 0.01	-	-	-	**1.00** ± 0.00
		F1-score	0.88 ± 0.02	0.89 ± 0.01	0.90 ± 0.00	**0.91** ± 0.00	0.90 ± 0.00	**0.91** ± 0.01	-	-	-	0.35 ± 0.00
Multimodal	**Property**	**Precision**	0.84 ± 0.02	0.84 ± 0.02	0.65 ± 0.02	**0.86** ± 0.02	0.66 ± 0.02	**0.86** ± 0.02	-	-	0.63 ± 0.02	-
		Recall	0.87 ± 0.02	0.82 ± 0.02	**0.89** ± 0.01	0.86 ± 0.01	**0.89** ± 0.01	0.87 ± 0.02	-	-	0.81 ± 0.03	-
		F1-score	0.84 ± 0.02	0.81 ± 0.02	0.73 ± 0.02	0.84 ± 0.01	0.74 ± 0.02	**0.85** ± 0.01	-	-	0.69 ± 0.02	-
	SWC	**Precision**	0.88 ± 0.01	0.91 ± 0.01	**0.92** ± 0.00	0.91 ± 0.01	**0.92** ± 0.00	0.90 ± 0.01	-	-	0.82 ± 0.02	-
		Recall	0.82 ± 0.02	0.84 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	-	-	**0.85** ± 0.00	-
		F1-score	0.83 ± 0.01	**0.86** ± 0.01	**0.86** ± 0.00	**0.86** ± 0.01	**0.86** ± 0.00	0.85 ± 0.01	-	-	0.81 ± 0.01	-

Table 2
Model performance across modalities and labelling for CodeSmells (highest score per modality in bold).

for ContractDefinition nodes, we extracted contract type and base contracts; for FunctionDefinition nodes, we captured function name, visibility, state mutability, and control structures (e.g., loops and conditionals); for VariableDeclaration nodes, we gathered data on visibility, storage location, and mutability. We also handled Mapping and BinaryOperation nodes. Similarly, each node in the CFG was characterized by general and CFG-specific features. General features included a block identifier, a list of opcode values, the number of opcodes in the block, and the number of successors (outgoing edges) to describe the control flow. CFG-specific features captured the control flow type, loop-related, arithmetic operations, and branching features, indicating the node's presence and type of loops or branches.

We propose a first multimodal investigation to explore the potential of leveraging multiple modalities simultaneously. As a case study, we processed source code and bytecode. For this early-stage exploration, we employed a simple approach by concatenating the TF-IDF representations of the individual modalities. Thus, we test the multimodal approach on the models that leverage TF-IDF. This approach is scalable to more multimodal combinations and could be refined in future work by exploring different ways of combining representations. We evaluated the models using standard metrics for multiclass and multilabel tasks, including precision, recall, and F1-score. These metrics were selected for their ability to provide a comprehensive view of model performance, especially in the context of imbalanced datasets. To ensure a more reliable analysis, we performed a 5-fold cross-validation and reported our results in terms of average and variance across the folds.

4. Results and Insights

Preliminary experiments exhibited poor performance, primarily due to the presence of many outliers, making it challenging to learn effectively from the entire dataset. We initially struggled to achieve satisfactory metrics because of the multilabel nature of the classification problem and the highly imbalanced class distribution. To address this issue, we analyzed each subset of the dataset independently. However, we acknowledge that the CGT dataset is relatively small for effectively training an ML algorithm, especially when focusing on individual subsets. These challenges rendered learning on some of the sets impractical and resulted in outcomes barely surpassing random guessing for others. We report our extensive results in the supplementary material linked in the code repository. The most noteworthy results were obtained on the largest set available within CGT, "CodeSmells," discussed in this section. These results are particularly significant given the decent amount of available data and balance, influencing the outcomes of experiments on other sets.

The CodeSmells dataset comprises 551 source codes, 551 bytecodes, and 504 runtimes. It allows for generating 103 opcodes, 287 ASTs, and 470 CFGs [6]. It is characterized by 20 "Property" labels mapped to 10 more general items in the SWC taxonomy (see Table 1 for details). Since DASP is more general than SWC, we report results only for the "Property" (fine-grained) and SWC (coarse) labels in Table 2. The evaluation reveals that, for most modalities and models, the metrics associated with the SWC labelling are significantly higher than those related to the "Property" labelling. This indicates that a coarser ground truth enables better learning outcomes, as models find handling more fine-grained labelling of vulnerabilities challenging. Under the "Property" labelling, ensemble models such as RF, GB, and XGB consistently emerge as the top performers across various modalities. Their consistently strong performance underscores the robustness and adaptability to different feature representations, making them a reliable choice for fine-grained detection. Conversely, DL models demonstrate remarkable performance for most modalities' under the SWC labelling. They exhibit a steep increase in all metrics compared to the finer "Property" labelling, highlighting their ability to capture broader patterns when the labelling is less granular.

Regarding the graph modalities—AST and CFG—the GNN achieves perfect recall, successfully capturing all positive instances. However, this comes at the expense of very low precision, leading to subpar F1-scores. The GNN's aggressive approach in predicting positives results in numerous false positives, making it less suitable for tasks that require a balanced classification approach. Overall, performance on AST data is the most modest, even with the coarser SWC labelling. In contrast, CFGs yield more promising results, indicating that graph representations hold potential for this task, but probably require a more complex GNN architecture to achieve better results.

When comparing results across modalities, similar performance trends emerge. This consistency suggests that each modality contains sufficient information for effective classification. The choice of modality can thus be tailored based on the task's specific requirements, balancing the data's richness with the complexity of interpretation. Analyzing multimodal data reveals that incorporating various data types significantly enhances the performance of all models, especially for fine-grained "Property" labelling. This improvement is particularly noticeable in ensemble models like RF and XGB, which experience substantial increases in F1-scores by integrating diverse features. This demonstrates the potential of multimodal integration in improving classification.

Overall, the findings highlight the importance of selecting appropriate models based on the data modality and the granularity of labelling. Ensemble models emerge as the most consistently strong performers across all modalities for fine-grained labelling. In contrast, DL models are the top performers with coarse SWC labelling, leveraging their capacity to model broad patterns effectively. In fine-grained labelling scenarios, multimodal data generally enhances model performance significantly, suggesting that integrating diverse features is a promising strategy. In future work, we plan to explore various approaches to address class imbalance, including data augmentation [35] and other upsampling

[6]Generating opcodes, ASTs, and CFGs required compilation, but many CGT dataset source codes encountered fatal errors, preventing the generation of these modalities. The errors depended on the 'solc' version, the operation (opcodes or ASTs), and Slither's internal processing, reducing the available data for the added modalities.

techniques. Additionally, we will investigate different GNN architectures to achieve improved results.

5. Conclusion

Contrary to formal verification methods, ML cannot guarantee the correctness of the detection. However, a learning algorithm's flexibility allows it to overcome limits such as detection speed, the number of vulnerabilities considered, and different versions of the same vulnerability. Moreover, it can provide a good solution to mitigate the high false positive rates reported by static analyzers. Our results open new avenues for researching multimodal handling in ML for vulnerability detection. Future work will focus on fine-tuning these models, exploring their explainability [36], and investigating hybrid approaches, particularly in multimodal settings. Additionally, enhancing dataset balance and exploring advanced techniques for imbalanced classification, like data augmentation, is crucial to improving detection accuracy for less common vulnerabilities. To address the limitations identified in this study, we plan to collect a new data set that mitigates the issues of existing ones, providing a more robust foundation for training and evaluating models.

Acknowledgments

This paper was funded by Veneto Agricoltura within the scope of the project "Guaranteeing the continuity of the agrifood chain: the digitization of wholesale markets". This study was carried out within the PE0000014 - Security and Rights in the CyberSpace (SERICS) and received funding from the European Union Next-GenerationEU - National Recovery and Resilience Plan (NRRP) – MISSION 4 COMPONENT 2, INVESTIMENT 1.3 – CUP N. H73C22000890001. This work has been also partially supported by the Project PRIN 2020 - CUP N. 20202FCJMH "NiRvAna - Noninterference and Reversibility Analysis in Private Blockchains". This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

References

[1] M. Bartoletti, L. Benetollo, M. Bugliesi, S. Crafa, G. Dal Sasso, R. Pettinau, A. Pinna, M. Piras, S. Rossi, S. Salis, et al., Smart contract languages: a comparative analysis, Future Generation Computer Systems (2024) 107563.
[2] D. Ressi, R. Romanello, C. Piazza, S. Rossi, Ai-enhanced blockchain technology: A review of advancements and opportunities, Journal of Network and Computer Applications (2024).
[3] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 254–269.
[4] Mythril project, https://github.com/ConsenSys/mythril, 2019. [Online].
[5] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, B. Scholz, Vandal: A scalable security analysis framework for smart contracts, arXiv preprint arXiv:1809.03981 (2018).
[6] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, M. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 67–82.
[7] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, Y. Alexandrov, Smartcheck: Static analysis of ethereum smart contracts, in: Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain, 2018, pp. 9–16.
[8] F. R. Vidal, N. Ivaki, N. Laranjeiro, Vulnerability detection techniques for smart contracts: A systematic literature review, Journal of Systems and Software (2024) 112160.
[9] M. Bartoletti, A. Ferrando, E. Lipparini, V. Malvone, Solvent: liquidity verification of smart contracts, arXiv preprint arXiv:2404.17864 (2024).
[10] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, T. Chen, Defining smart contract defects on ethereum, IEEE Transactions on Software Engineering 48 (2020) 327–345.

[11] M. Di Angelo, T. Durieux, J. F. Ferreira, G. Salzer, Evolution of automated weakness detection in ethereum bytecode: a comprehensive study, Empirical Software Engineering 29 (2024) 41.

[12] J.-W. Liao, T.-T. Tsai, C.-K. He, C.-W. Tien, Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing, in: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), IEEE, 2019, pp. 458–465.

[13] J. Song, H. He, Z. Lv, C. Su, G. Xu, W. Wang, An efficient vulnerability detection model for ethereum smart contracts, in: Network and System Security: 13th International Conference, NSS 2019, Sapporo, Japan, December 15–18, 2019, Proceedings 13, Springer, 2019, pp. 433–442.

[14] L. Zhang, W. Chen, W. Wang, Z. Jin, C. Zhao, Z. Cai, H. Chen, Cbgru: A detection method of smart contract vulnerability based on a hybrid model, Sensors 22 (2022) 3577.

[15] J. Cai, B. Li, J. Zhang, X. Sun, B. Chen, Combine sliced joint graph with graph neural networks for smart contract vulnerability detection, Journal of Systems and Software 195 (2023) 111550.

[16] V. K. Jain, M. Tripathi, An integrated deep learning model for ethereum smart contract vulnerability detection, International Journal of Information Security (2023) 1–19.

[17] D. Ressi, A. Spanò, L. Benetollo, C. Piazza, M. Bugliesi, S. Rossi, Vulnerability detection in ethereum smart contracts via machine learning: A qualitative analysis, 2024. URL: https://arxiv.org/abs/2407.18639. arXiv:2407.18639.

[18] F. R. Vidal, N. Ivaki, N. Laranjeiro, Openscv: an open hierarchical taxonomy for smart contract vulnerabilities, Empirical Software Engineering 29 (2024) 101.

[19] Etherscan website, Accessed: 24-08-2024. URL: https://etherscan.io/.

[20] Smartbugs wild dataset, Accessed: 24-08-2024. URL: https://github.com/smartbugs/smartbugs-wild.

[21] J. F. Ferreira, P. Cruz, T. Durieux, R. Abreu, Smartbugs: A framework to analyze solidity smart contracts, in: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, 2020, pp. 1349–1352.

[22] M. di Angelo, G. Salzer, Consolidation of ground truth sets for weakness detection in smart contracts, in: Financial Cryptography and Data Security. FC 2023 International Workshops, Springer Nature Switzerland, Cham, 2024, pp. 439–455.

[23] M. Di Angelo, G. Salzer, Consolidation of ground truth sets for weakness detection in smart contracts, in: International Conference on Financial Cryptography and Data Security, Springer, 2023, pp. 439–455.

[24] S. Kalra, S. Goel, M. Dhawan, S. Sharma, Zeus: analyzing safety of smart contracts., in: Ndss, 2018, pp. 1–12.

[25] C. Schneidewind, I. Grishchenko, M. Scherer, M. Maffei, ethor: Practical and provably sound static analysis of ethereum smart contracts, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 621–640.

[26] B. Jiang, Y. Liu, W. K. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, 2018, pp. 259–269.

[27] A. Ghaleb, K. Pattabiraman, How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 415–427.

[28] S. Zhou, M. Möser, Z. Yang, B. Adida, T. Holz, J. Xiang, S. Goldfeder, Y. Cao, M. Plattner, X. Qin, et al., An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2793–2810.

[29] Y. Xue, J. Ye, M. Ma, L. Ma, Y. Li, H. Wang, Y. Lin, T. Peng, Y. Liu, Doublade: unknown vulnerability detection in smart contracts via abstract signature matching and refined detection rules, arXiv e-prints (2019) arXiv–1912.

[30] S. Wang, C. Zhang, Z. Su, Detecting nondeterministic payment bugs in ethereum smart contracts, Proceedings of the ACM on Programming Languages 3 (2019) 1–29.

[31] P. Zhang, F. Xiao, X. Luo, A framework and dataset for bugs in ethereum smart contracts, in: 2020 IEEE international conference on software maintenance and evolution (ICSME), IEEE, 2020, pp. 139–150.

[32] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, P. Saxena, Exploiting the laws of order in smart contracts, in: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, 2019, pp. 363–373.

[33] J. Feist, G. Grieco, 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, 2019, pp. 8–15.

[34] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, others https://etherscan.io/, Codebert: A pre-trained model for programming and natural languages, arXiv preprint arXiv:2002.08155 (2020).

[35] A. Gasparetto, D. Ressi, F. Bergamasco, M. Pistellato, L. Cosmo, M. Boschetti, E. Ursella, A. Albarelli, Cross-dataset data augmentation for convolutional neural networks training, in: 2018 24th International Conference on Pattern Recognition (ICPR), IEEE, 2018, pp. 910–915.

[36] M. Rizzo, A. Veneri, A. Albarelli, C. Lucchese, M. Nobile, C. Conati, A theoretical framework for ai models explainability with application in biomedicine, in: 2023 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), 2023, pp. 1–9. doi:10.1109/CIBCB56990.2023.10264877.