SMART CONTRACT VULNERABILITY DETECTION ON EVM BYTECODE WITH DEEP LEARNING

A THESIS SUBMITTED TO
THE FACULTY OF ARCHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

LEJDI PRIFTI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

MARCH, 2024

Approval sheet of the Thesis

This is to certify that we have read this thesis entitled **"Smart Contract Vulnerability Detection on EVM Bytecode with Deep Learning"** and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

____

Assoc. Prof. Dr. Arban Uka
Head of Department
Date: 03, 01, 2024

Examining Committee Members:

Assoc. Prof. Dr. Dimitrios Karras (Computer Engineering) ________________

Prof. Dr. Gëzim Karapici (Computer Engineering) ________________

Prof. Dr. Betim Cico (Computer Engineering) ________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name Surname: Lejdi Prifti

Signature: ______________

ABSTRACT

SMART CONTRACT VULNERABILITY DETECTION ON EVM BYTECODE WITH DEEP LEARNING

Prifti, Lejdi

M.Sc., Department of Computer Engineering

Supervisor: Prof. Dr. Betim Cico

In the quickly changing world of blockchain technology, it is critical to guarantee the security of self-executing contracts, written in programming languages like Solidity called smart contracts. Not all security vulnerabilities in smart contracts will be found by human code reviews and security audits using traditional methods. Deep learning networks have become a promising answer to this problem. In this paper, we present the architecture of two models—using convolutional and recurrent neural networks—that are intended to effectively discover five vulnerabilities in smart contracts. To train and validate the models, we used a dataset that includes 106474 audited smart contracts taken from the public Ethereum blockchain. Instead of the source code used by most deep learning-based solutions, the models receive input in the form of Ethereum Virtual Machine (EVM) bytecode. Across all five vulnerabilities, the Recurrent Neural Network model has an average micro F1-score of 0.93, whereas the Convolutional Neural Network achieves an average micro F1-score of 0.89. Through comparative research with various deep learning systems and static analysis tools, we have determined that EVM bytecode may be leveraged as a feature to detect vulnerabilities in smart contracts.

Keywords: *Smart Contracts, Security, Deep Learning, Recurrent Neural Network, Convolution Neural Network, Bytecode*

ABSTRAKT

ZBULIMI I DOBËSIVE NË KONTRATAT INTELIGJENTE NË KODIN EVM ME MËSIM TË THELLUAR

Prifti, Lejdi

Master Shkencor, Departamenti i Inxhinierisë Kompjuterike

Udhëheqësi: Prof. Dr. Betim Cico

Në botën e ndryshimeve të shpejta të teknologjisë së blockchain, është kritike të garantohet siguria e kontratave vetë-ekzekutive, të shkruara në gjuhë programimi si Solidity të quajtura kontrata inteligjente. Rrjetet e mësimit të thellë kanë bërë një përgjigje të premtueshme për këtë problem. Në këtë punim, ne prezantojmë arkitekturën e dy modeleve—duke përdorur rrjete konvolucionale dhe rrjete neurale rekurrente—që janë të krijuara për të zbuluar efektivisht pesë dobësi në kontratat inteligjente. Për të trajnuar dhe vlerësuar modelet, ne përdorëm një set të të dhënash që përfshin 106474 kontrata inteligjente të audituara marrë nga blockchain-i publik Ethereum. Në vend të kodit burimor, modelet marrin hyrje në formë të kodit të makinës virtuale Ethereum (EVM). Për të pesë dobësitë, modeli i Rrjetit Neuronal Rekurent ka një vlerësim mesatar mikro F1 të 0.93, ndërsa Rrjeti Neuronal Konvolucional arrin një vlerësim mesatar mikro F1 prej 0.89. Përmes hulumtimeve krahasuese me sisteme të ndryshme të mësimit të thellë dhe mjeteve të analizës statike, kemi përcaktuar se kodit EVM mund të shfrytëzohet si një tipare për të zbuluar dobësitë në kontratat inteligjente.

Fjalët kyce: *Kontrata inteligjente, Siguria, Mësim i thelluar, Rrjeti Neuronal Konvolucional, Kodi bajt*

TABLE OF CONTENTS

ABSTRACT ... iii

ABSTRAKT ... iv

LIST OF TABLES ... viii

LIST OF FIGURES ... ix

CHAPTER 1 ... 1

INTRODUCTION ... 1

1.1 Problem Statement ... 1

1.2 Thesis Objectives ... 2

CHAPTER 2 ... 3

LITERATURE REVIEW ... 3

CHAPTER 3 ... 17

RESEARCH GAP ... 17

3.1 Could EVM bytecode be used as a feature for uncovering smart contract vulnerabilities? ... 17

3.2 How do RNNs and CNNs trained on EVM bytecode perform compared to static analysis tools and other deep learning solutions? ... 18

CHAPTER 4 ... 19

ETHEREUM AND SMART CONTRACTS ... 19

4.1 Ethereum as a blockchain ... 19

4.2 Ethereum key components 20
4.3 Smart contracts and their lifecycle 22
4.3.1 Creation Phase 23
4.3.2 Deployment Phase 23
4.3.3 Execution Phase 25
4.3.4 Completion Phase 25
CHAPTER 5 26
ETHEREUM SMART CONTRACT TYPES OF VULNERABILITIES 26
5.1 Reentrancy 26
5.2 Arithmetic issues 27
5.3 Unchecked external calls 28
5.4 Access control 29
5.5 Other issues 30
CHAPTER 6 32
DATASET 32
6.1 Data Collection 32
6.2 Data Cleaning 33
CHAPTER 7 35
ARCHITECTURE OF OUR MODELS 35
7.1 CNNs 35

7.2 RNNs 37

CHAPTER 8 39

RESULTS 39

8.1 Results of the CNN model 39

8.2 Results of the RNN model 41

8.3 Result comparison 43

CHAPTER 9 44

CONCLUSIONS 44

9.1 Conclusions 44

9.2 Recommendations for future research 44

REFERENCES 45

LIST OF TABLES

Table 1. Summary of references based on the methodology used. 16

Table 2. Metrics on the test split of the dataset for the CNN model 41

Table 3. Metrics on the test split of the dataset. 43

Table 4. Comparison of micro F1-scores between different models. 43

LIST OF FIGURES

Figure 1. Ethereum architecture. (Chittoda, 2019) 19

Figure 2. Accounts in Ethereum. (Farnaghi, Mahdi & Mansourian and Ali, 2020) .. 21

Figure 3. The lifecycle of Ethereum smart contracts. 23

Figure 4. The workflow of Ethereum smart contract source. (Yiu, 2021) 25

Figure 5. Distribution of bytecode length in the dataset. 33

Figure 6. Distribution of outputs in the dataset. 34

Figure 7. Architecture of the model with CNNs. 36

Figure 8. Architecture of the model with RNNs. 38

Figure 9. Training loss during each epoch. 39

Figure 10. Validation loss during each epoch. 40

Figure 11. Comparison of training and validation accuracy. 40

Figure 12. Training loss during each epoch. 41

Figure 13. Validation loss during each epoch. 42

Figure 14. Comparison of training and validation accuracy. 42

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

Deep learning, a branch of machine learning, has achieved good results in several fields, including speech recognition, computer vision, and natural language processing.

In this paper, we present the architecture of two deep learning neural networks that are shown to achieve good performance in detecting vulnerabilities into Ethereum smart contracts.

Through the literature review, we provided a comprehensive overview of the methods used to find smart contract vulnerabilities and classified them into two groups: deep learning methods and static analysis tools. The importance of several well-known tools in enhancing the security of Ethereum smart contracts is highlighted, including Oyente, Mythril, Maian, Zeus, EtherFuzz, and Securify. These technologies do, however, have several drawbacks, such as restricted vulnerability coverage, false positives and false negatives. Many deep-learning models and methods, including Convolutional Neural Networks (CNNs), Long-Short Term Memory (LSTM) networks, and neuro-symbolic frameworks, have been proven to be efficient.

However, most of these models expect to input the source code of smart contracts. Instead of the source code, we use the EVM bytecode. From the results we got, we have concluded that EVM bytecode is a good feature to detect vulnerabilities in smart contracts. We have compared the results of our model with those achieved by Rossini, Zichichi, & Ferret (2022) in the same dataset. For future work, we suggest expanding data collection efforts by increasing the number of smart contract vulnerabilities. Furthermore, we recommend increasing convolutional block

complexity, as well as creating a single model that expects two forms of inputs, the source code and the bytecode of the smart contract.

1.2 Thesis Objectives

The following are the key goals of this study's investigation into employing deep learning networks to identify smart contract vulnerabilities:

1. Increase security of smart contracts: The main objective is to increase the security of smart contracts used on Ethereum and other blockchain systems. Our goal is to identify vulnerabilities more precisely and quickly than with conventional techniques by building a strong deep learning model, thereby reducing the risks related to smart contract vulnerabilities.
2. Using a deep learning-based solution: The goal is to create, put into practice, and assess a deep learning-based model that can efficiently identify a variety of smart contract vulnerabilities. This entails dealing with problems like reentrancy flaws, integer overflows, uncovered calls, and other typical security concerns.
3. Investigate multi-class categorization: Our goal is to create a model with multi-class categorization capabilities that can recognize several types of vulnerabilities in a single smart contract. This method offers a more thorough evaluation of the security posture of a contract.
4. Optimize training and inference speed: Security mechanisms for smart contracts must be quick to respond, especially considering the urgency of blockchain transactions. The goal is to accelerate the model's inference and training processes to provide quick and effective vulnerability detection.

CHAPTER 2

LITERATURE REVIEW

We provide a general overview of the current approaches to smart contract vulnerability detection in this literature study. Static and deep learning solutions are the two categories we will examine. We will list the potential flaws that each may have below.

There are several well-known solutions based on symbolic execution that are widely used in the field of automated security tools for smart contracts.

To increase the security of Ethereum smart contracts, a specific security analysis tool called Oyente (Loi Luu, 2016) was created. Oyente, a tool designed to find potential flaws and vulnerabilities in smart contract programming, uses symbolic execution techniques to thoroughly examine all possible execution routes. It simulates the behavior of the contract without executing it, which enables it to identify intricate flaws that could be difficult to find through manual analysis. Oyente's main goal is to highlight security flaws that could be used by malevolent actors, such as reentrancy vulnerabilities, integer overflows, and other potential exploits. As a result, it helps programmers find and fix flaws before releasing their smart contracts onto the Ethereum network. Although useful, Oyente's capabilities have several drawbacks, such as the potential for false positives or negatives, and the complexity of the contract code may have an impact on how effective it is.

Another popular open-source security analysis tool made exclusively for finding security flaws and vulnerabilities in Ethereum smart contracts code developed by ConsenSys (Consesys, 2018) is called Mythril. Mythril extensively examines multiple smart contract execution paths using symbolic execution and static analysis methods without executing them on the blockchain. This enables it to identify weaknesses that might be used by bad actors to their advantage.

Complex problems like reentrancy vulnerabilities, integer overflows, and logic defects that could elude manual code review are particularly well-detected. Mythril is acknowledged to have a number of drawbacks, though, including a high rate of false positive and false negative results, analytical complexity, and restricted support for specific contract types.

To identify between potential greedy, prodigal, and suicidal smart contracts, Ivica Nikolic et al. (2018) introduced a new tool called Maian. The contracts that, when attacked, restore money to owners, advertise a specific solution at addresses that have previously given them ether, or both have been dubbed prodigal contracts by the authors. When a contract malfunctions or runs out of Ether due to an attack, it frequently has the security fallback option of being terminated by its owner or other trustworthy addresses. The authors consider a contract to be suicidal if it can be terminated by any random account. Lastly, contracts that remain active and permanently lock Ether, preventing its release under any circumstances, are referred to as greedy. Maian uses symbolic analysis and concrete validation to find the vulnerabilities mentioned above. Ivica Nikolic et al. came to conclusion that around 97% of prodigal, 97% suicidal, and 69% of greedy contracts were genuine positives using a concrete validation engine or manual inspection after analyzing 970,898 contracts. They observed that it is crucial to analyze the contracts' bytecode rather than their Solidity code. Despite the fact that Maian is a powerful general tool for finding defects, it does not ensure that all program paths are explored (leading to false negatives).

In their work, Giancarlo Bigi et al. (2015) validate a decentralized smart contract protocol using a combination of game theory and probabilistic model testing.

Bhargavan et al. (2016) present preliminary research on the adaptive type and effect system of F*. Their approach, based on shallow embeddings and type checking within an existing verification framework, is applicable for looking into the formal verification of contracts written in Solidity and EVM bytecode. The system is designed to capture and prove desired features for contract programmers.

In the study by Grossman et al. (2018), the focus is on domain-specific features. They introduce a dynamic linearizability checker to delve deeper into pinpointing reentrancy problems, emphasizing the importance of addressing issues related to specific domains in contract programming.

The Zeus system (Kalra, Goel, Dhawan, & Sharma, 2018), a sound analyzer that converts smart contracts to the LLVM framework, does not support violation patterns to lower false positives. Additionally, Zeus does not allow for the verification of data- and control-flow aspects. Properties are written in the XACML language.

Securify was introduced by Tsankov et al. (2018). It makes use of the domain-specific insight that many of the practical properties for smart contracts that are violated are also violated by simpler, much easier-to-check properties. It examines every contract action to prevent unfavorable false negatives. Additionally, by ensuring that certain actions are genuine problems, it decreases the user effort required to categorize warnings into true positives and false alarms. Furthermore, it facilitates the usage of a brand-new domain-specific language that enables users to communicate newly discovered vulnerability patterns. Its analytical pipeline is entirely automated utilizing scalable, commercial Datalog solvers, from bytecode decompilation to optimizations through pattern validation.

By propagating contaminated information via several transactions, a new tool called Ethain (Brent, Grech, Lagouvardos, Scholz, & Smaragdakis, 2020) detects composite assaults that lead to grave infractions. With a very high precision of 82.5% valid warnings for end-to-end vulnerabilities, their research spans over the entire blockchain and achieves better results than Securify, mentioned previously.

To identify TOD issues in smart contracts, Wang et al. (2022) provided a modified version of the fuzzy framework EtherFuzz. According to their experimental results, EtherFuzz is more effective than other tools at identifying TOD vulnerabilities. Between them, there is a reduction in time cost that is on average 31.1% and a reduction in memory cost that is on average 29.2%. The authors note that they hope to reduce EtherFuzz's current false positive rate in subsequent works.

Neural networks have been created to aid in the detection of smart contracts vulnerabilities as deep learning has advanced.

Cheng, Wang, Hua, Xu and Sui (2021) proposed in their work DeepWukong, a novel deep learning-based embedding method for C/C++ program static software vulnerability identification. They create a new code representation that maintains both the natural language information of a program and high-level programming logic by using sophisticated graph neural networks to embed code fragments in a compact and low-dimensional representation.

By combining metric learning triplet loss with a bidirectional long short-term memory (BiLSTM) network model, Meiying, Xie, Wen, Li and Zhou (2023) proposed a unique vulnerability detection method for Ethereum smart contracts. By boosting cohesion within a single category and discreteness between various categories of smart contracts, the system optimizes feature representation space and improves the accuracy of vulnerability identification. The method improves interpretability and aids in locating root causes by using source code as input data, word vectorization, and an attention method to locate crucial details linked to vulnerabilities. A comprehensive dataset of 165,000 verified smart contract source codes is produced, and vulnerability flags are provided using a variety of detection techniques to provide robust data support. The suggested method outperforms previous deep learning models and conventional methods in successfully extracting vectorized features and increasing vulnerability detection precision.

In their study, Lutz, et al. (2023) proposed ESCORT with the goal of overcoming the scalability and generalization restrictions of the earlier published research. ESCORT is a multi-output neural network trained to recognize each specific vulnerability class and learn the bytecode properties. ESCORT makes use of a feature extractor to accomplish these two objectives since it can read contract bytecode regardless of its flaws and extract its semantic and syntactic information. The second objective is accomplished by using an individual vulnerability class branch to characterize susceptibility considering the previously extracted bytecode attributes. ESCORT achieves an average F1 score of 95% on six different vulnerability categories, according to experimental findings, and the detection time is

0.02 seconds per contract. When ESCORT is expanded to include new vulnerability categories, the average F1 score is 93%.

To find weaknesses in smart contracts' opcode, the study by Joon-Wie Tann, Jie Han, Sen Gupta, and Ong (2018) suggests using a sequence learning approach. The contract's opcode is expressed specifically using the one-hot encoding and embedding matrix. An LSTM model is trained to predict if a given smart contract is safe or vulnerable using the obtained code vectors as input (binary classification). Because the LSTM-based technique cannot distinguish between the different vulnerability categories and has a stated F1 score of 86%, it performs poorly at detecting vulnerabilities.

Gogineni, Swayamjyoti, Sahoo and Sahu (2022) present a multi-class categorization technique based on sequencing. For identifying vulnerabilities, their study adopts the "Average Stochastic Gradient Descent Weighted Dropped LSTM" (Merity, Keskar, & Socher, 2017). A pre-trained encoder for linguistic tasks (Howard & Ruder, 2018) and an LSTM-based classifier for vulnerability classification make up the proposed model's two components. Three different vulnerability types can be found with this technology, which operates at the opcode level. An F1 score of 95% on safe contracts and 30% on prodigal contracts is generated by the model. The model's extensibility was not considered by the writers.

The contract bytecode is converted into fixed-sized RGB color images by TonTon Hsien-DeHuang (2018) and a convolution neural network is trained for vulnerability identification. CNN-based classifier uses multi-label classification, which has a poor confidence score for identifying the precise vulnerability classes. Due to the low confidence level, the performance of the multi-label classification is not satisfactory. This could be caused by the CNN architecture and image representation of the bytecode ignoring the sequential data provided in the contract.

A graph neural network (GNN)-based technique is suggested by Zhuang et al. (2020). This work creates a contract graph from the source code of the contract, where nodes and edges stand in for crucial function calls and variables and the temporal execution trail, respectively. To emphasize key nodes, this graph is

normalized before being sent to a temporal message propagation (TMP) network to look for vulnerabilities. There are a few drawbacks to the concept in terms of its limited applicability and efficacy. The F1-score for all three vulnerabilities is 77%.

Deng et al. (2023) proposed a smart contract vulnerability detection mechanism based on multimodal decision fusion. This approach also considers the smart contract's control structure and code semantics. Using the multimodal decision fusion method, it unifies the source code, operation code, and control-flow modes. With great accuracy and recall rates, the deep learning approach extracts five attributes that are utilized to represent contracts. As demonstrated by the experimental results, the authors' method's detection accuracy for re-entrant vulnerability, Ethernet locking vulnerability, arithmetic vulnerability, and transaction order dependence can reach 91.6%, 90.9%, 94.8%, and 89.5%, respectively, and its detected AUC values can reach 0.834, 0.852, 0.886, and 0.825, respectively.

The extraction of features and the volume of data utilized for training are critical to the efficacy of vulnerability detection for smart contracts, according to the data-driven deep learning technique. Consequently, Gao, Jiang, Xia, Lo, & Grundy (2020) suggested an automated technique based on word embedding to acquire Solidity smart contract capabilities.

Zhang et al. (2022) proposed leveraging ensemble learning (EL) techniques to smart contract vulnerabilities prediction. To facilitate contract-level vulnerability detection, multiple NNs were included in the proposed SCVDIE-ENSEMBLE approach. These models were then used to create an ensemble framework known as the Information Graph and Ensemble Learning-based Smart Contract Vulnerability Detection technique (SCVDIE). The authors observed that since each NN has a distinct function to fulfill, SCVDIE-ENSEMBLE could perform more accurately and robustly on unseen data while increasing the efficiency of data utilization. According to the authors, on various sized datasets, the mean F1 score and the mean prediction accuracy both reached ideal values of 97.57% and 97.42%, respectively, and achieved decreased relative RMSEs.

Huang, Zhou, Xiong and Li (2022) developed a smart contract vulnerability model using multi-task learning. The model's detection capabilities were enhanced to enable the identification and detection of vulnerabilities by assigning supplementary tasks to learn more directional vulnerability features. The hard-sharing design, which has two components, is the foundation of the model. First, the input contract's semantic information is mostly learned via the bottom sharing layer. The neural network, which is based on an attention mechanism, is used to learn and extract the feature vector of the contract after word and positional embedding has first converted the text representation into a new vector. Second, each task's functions are primarily done by using the task-specific layer. As the authors have stated, for each task, a classification model that learns and pulls features from the shared layer for training to accomplish their separate task objectives was built using a standard convolutional neural network. The findings of the experiment demonstrate that the addition of the auxiliary vulnerability detection task improved the model's capacity to identify the different types of vulnerabilities.

In their paper, Sun & Gu (2021) suggest using a CNN model with a self-attention mechanism to identify smart contract vulnerabilities. Additionally, their approach uses feature engineering to combine a stop word list, sensitive word sharding, and an enhanced one hot encoder. However, the model is limited to detecting only three types of vulnerabilities. The authors plan to expand on their single binary classification models in future research by developing a multinomial classification model that may simultaneously identify several vulnerability kinds.

The Dual Attention Graph Convolutional Network (DA-GCN) is a unique model that Fan, Shang and Ding (2021) propose to detect vulnerabilities in blockchain-based smart contracts. Graph convolutional network and self-attention mechanism-based feature extractor receives input from the control flow graph and the opcode sequence retrieved from smart contract bytecodes. Then, using control flow level attention, Model DA-GCN concentrates on the most significant nodes in the control flow graph while suppressing irrelevant data. In the end, a multilayer perceptron is employed to determine the smart contract's vulnerability. Their suggested model DA-GCN may effectively increase the smart contract vulnerability detection performance, as demonstrated by experimental findings on the real-world

smart contract data set containing two vulnerabilities: timestamp dependency and reentrancy.

Peng et al. (2015) in their work pointed out that there is a difference between natural language and programming language. In essence, the programming language does not benefit from the NLP algorithms as much as the natural language does. The authors concluded that their proposed “coding criterion” based on ASTs is a successful representation learning algorithm for programs. They convert abstract syntax tree nodes into vectors. To identify similar source code snippets, Mou et al. (2016) suggest a tree-based convolutional neural network based on program abstract syntax trees.

To improve the class separation between vulnerable and non-vulnerable samples during model building, Chakraborty, Krishna, Ding and Ray (2020) demonstrate that representation learning can be applied on top of conventional DL techniques. A common family of machine learning algorithms called representation learning eliminates the need for human feature engineering by automatically identifying the input representations required for better classification. The authors claim the model must learn to automatically represent benign and vulnerable code in the feature space since it is hard to distinguish between their attributes. Moreover, the model's accuracy rate can be raised by addressing the class imbalance of vulnerable/non-vulnerable samples through semantic information, data deduplication, and training data balancing.

VELVET, a unique ensemble learning method for identifying vulnerable statements, is presented by Ding et al (2021). The author’s approach properly captures the local and global context of a program graph and effectively understands susceptible patterns and code semantics by combining graph-based and sequence-based neural networks. They used a commercially available synthetic dataset and a recently released real-world dataset to examine the efficacy of VELVET. According to the authors, in the static analysis environment, VELVET performs 4.5 times better on real-world data than baseline static analyzers when vulnerable functions are not known ahead of time.

To overcome the "simple" first-order limitations, Xie, Kesting and Neider (2022) have created the first neuro-symbolic framework for neural network verification. This framework allows for the expression of complex correctness aspects through deep neural networks. They have demonstrated how quickly their framework may be added to the existing infrastructure for verification.

Two novel model families have been developed by Hellendoorn, Sutton, Singh, Maniatis and Bieber (2020) that efficiently combine the longer distance information that the sequence model can convey with the semantic structure information that the Gated Graph Neural Network (GGNN) can access.

Graph Relational Embedding Attention Transformer (GREAT) is another family of families that conveys structural relations instead by generalizing the relative position embeddings in Transformers by Shaw, Uszkoreit and Vaswani (2018). Graph Sandwich is one family of families that alternates between sequential information flow and message passing through a chain of nodes within the graph. The authors show that their proposed model family outperformed all previous findings and their new, already stronger baseline by an additional 10% each, while training both substantially faster and with fewer parameters.

To address the shortcomings of the current GNN, Wang, Wang, Gao, & Wang (2020) developed a new graph neural network architecture known as the Graph Interval Neural Network (GINN). In contrast to the conventional GNN, GINN expands from a carefully selected graph representation that is acquired via an abstraction technique intended to support model learning. Specifically, GINN employs just intervals (which are typically represented by looping constructs) to mine a program's feature representation. In addition, GINN uses a hierarchy of intervals to scale learning over huge graphs.

A code property graph is a novel source code representation introduced by Yamaguchi, Golde, Arp, & Rieck (2014). This representation combines ideas from classic program analysis, such as abstract syntax trees, control flow graphs, and program dependence graphs, into a joint data structure. With this comprehensive representation, they are able to model elegantly templates for common vulnerabilities

with graph traversals that can identify, for example, buffer overflows, integer overflows, format string vulnerabilities, or memory disclosures; they implement their approach using a widely used graph database and demonstrate its effectiveness by finding eighteen previously undiscovered vulnerabilities in the Linux kernel source code.

Li, Wang, & Nguyen (2021) separately examine each vulnerable statement and its surrounding settings to identify vulnerabilities using data and control dependencies. Consequently, their model outperforms earlier methods that included contextual and susceptible code in its ability to discern between statements that are vulnerable and those that are not. In addition to the coarse-grained vulnerability detection result, they use interpretable AI to provide customers with fine-grained interpretations that include the subgraph in the Program Dependency Graph (PDG) holding the significant statements that are relevant to the discovered vulnerability. IVDetect outperforms the current DL-based approaches by 43%–84% in top-10 nDCG and 105%–255% in MAP ranking scores, according to their empirical assessment using vulnerability databases.

In their work, Tarlow et al. (2019) use a graph to represent build configuration files and compiler diagnostic messages. Then, they employ a Graph Neural Network to predict a diff, which denotes how to alter the abstract syntax tree of the code, which is represented by a series of tokens and pointers to code places in the neural network.

W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang (2020) have demonstrated the significant impact of using the Transformer model for the source code summarization task. The authors claim that the Transformer, which incorporates copy attention and relative position representations, performs significantly better than state-of-the-art methods.

A bimodal pre-trained model for natural language (NL) and programming language (PL) called CodeBERT is presented by Feng et al. (2020). With a Transformer-based neural architecture, CodeBERT is trained with a hybrid objective function that includes replacement token detection as a pre-training job. The goal of

this work is to identify logical substitutes that have been sampled from generators. To enable a variety of NL-PL applications, such as natural language code search and code documentation production, the model learns general-purpose representations. Both "unimodal" data—in which NL and PL are taken into separate consideration—and "bimodal" data—which consists of NL-PL pairs—are used in the training process. After fine-tuning the model parameters, CodeBERT's evaluation on two NL-PL apps shows state-of-the-art performance in natural language code search and code documentation production.

The VSCL framework was proposed by Mi et al. (2021) in their attempt to automatically identify vulnerabilities in blockchain-based smart contracts. More precisely, as the source code of smart contracts is rarely made available to the public, they first use feature vector generating techniques from the bytecode of the contract. To obtain the detection result, the gathered vectors are then fed into a deep neural network (DNN) that is based on metric learning.

Wu et al. (2021) present a novel method called Peculiar that uses a pre-training technique to identify smart contract vulnerabilities based on critical data flow graphs. Crucial data flow graph is less complicated and does not introduce an unduly deep hierarchy than the typical data flow graph, which is already used in existing techniques. This makes it easier for the model to focus on the important aspects. Additionally, pre-training methodology was incorporated into the model because of the significant gains it has made on a range of natural language processing tasks according to the authors.

A unique approach to reentrancy vulnerabilities in Ethereum smart contracts was developed by Eshghie, Artho and Gurov (2021). They suggest Dynamit, a monitoring framework that merely uses transaction metadata and balance data from the blockchain system in place of domain expertise or code instrumentation. To categorize transactions as benign or dangerous, Dynamit takes attributes out of transaction data and applies a random forest classifier to it.

In their work, Dinella et al. (2020) presents a model that consists of an LSTM central controller that performs a series of basic operations (such predicting type,

producing patch, etc.) to accomplish a fix, and an external memory (a Graph Neural Network) for embedding a malfunctioning program. An autoregressive model is used to implement the multi-step decision procedure. The authors claim that the way the memory is managed in the model is different from that of the normal NTM: in addition to the usual read and write operations, the controller can additionally increase or decrease the memory when adding or removing nodes from the original graph.

A pipeline named AI4VA is proposed by Suneja, Zheng, Zhuang, Laredo and Morari (2020). It begins by encoding a sample source code into a Code Property Graph. The authors then vectorize the retrieved graph in a way that they say retains its semantic content. Then, using many of these graphs, GGNN is trained to automatically extract templates that distinguish the graph of a vulnerable sample from a healthy one. On two of the three datasets the authors tested, the model beats CNN and RNN-based deep learning models, static analyzers, and traditional machine learning. This work provides more evidence that the code-as-graph encoding—rather than the code-as-photo and linear sequence encoding methods currently in use—is more significant for vulnerability detection. Vytovtov and Chuvilin (2019) cluster Java classes automatically into business logic, interface, and utility classes using GNN based auto-encoders for unsupervised learning over source code ASTs. Moreover, Alon, Zilberstein, Levy and Yahav (2019) proposed to encrypt source code using a set of path contexts (AST paths + leaf-node values) and train an attention-based neural network to identify suitable function names based on the code.

In their work, Nguyen et al. (2022) present the ReGVD model, which is based on graph neural networks. To construct a graph, ReGVD treats each raw source code as a flat series of tokens. Only the token embedding layer of a trained programming language (PL) model initializes the node features. To generate a graph embedding for the source code, ReGVD then makes use of residual connections between GNN layers and looks at a combination of graph-level sum and max poolings.

Furthermore, Hin, Kan, Chen and Ali Babar (2022) introduced a unique deep learning framework called LineVD that treats the problem of node classification in

conjunction with statement-level vulnerability identification. LineVD uses a transformer-based architecture to encode the raw source code tokens and graph neural networks to utilize control and data dependencies between statements.

Şahin, Özyedierler and Tosun (2022) presented a method for employing GNNs to anticipate susceptible code versions by taking advantage of commits in addition to code. Specifically, they tested various deep learning and machine learning architectures, including Graph Convolutional Networks and GraphSAGE. To train the GNN models, the Wireshark project samples were gathered, and the data were converted into AST representations and fed into the GNN models. To understand the features of the nodes, they also employed a technique akin to word2vec. They used the SZZ technique in conjunction with the commits to pinpoint the exact position of the code's vulnerability. Ultimately, the application of GraphSAGE to identify susceptible code yielded the highest results, with an F1 score of 74.4% and an AUC-ROC score of 96.0%.

According to Evangelos, Katsadouros and Charalampos & Patrikakis (2022), multi-class classification is more advantageous in providing information about the type of vulnerability. However, the authors claim that there hasn't been a lot of research done in this area.

Lastly, an overview of the literature review based on the methodology is provided in Table 1.

Table 1. Summary of references based on the methodology used.

Method	**References**
Symbolic Execution	(Loi Luu, 2016), (Consesys, 2018)
Game Theory & Probabilistic Model Testing	(Bigi, Bracciali, Meacci, & Tuosto, 2015)
Concrete Validation and Analysis	(Nikolic, Kolluri, Sergey, Saxena, & Hobor, 2018), (Tsankov, et al., 2018), (Brent, Grech, Lagouvardos, Scholz, & Smaragdakis, 2020), (Wang, et al., 2022)
LLVM Framework and Type Systems	(Bhargavan, et al., 2016), (Kalra, Goel, Dhawan, & Sharma, 2018)
Dynamic Linearizability Checker	(Grossman, et al., 2018)
Deep Learning	(Cheng, Wang, Hua, Xu, & Sui, 2021), (Meiying, Xie, Wen, Li, & Zhou, 2023), (Lutz, et al., 2023), (Fan, Shang, & Ding, 2021), (Zhuang, et al., 2020), (Merity, Keskar, & Socher, 2017)
Ensemble Learning	(Zhang, et al., 2022), (Ding, et al., 2021)
Multi-Modal Decision Fusion	(Deng, et al., 2023), (Gao, Jiang, Xia, Lo, & Grundy, 2020)
Sequence Learning	(Tann, Han, Gupta, & Ong, 2018)
Transformer Model	(Ahmad, Chakraborty, Ray, & Chang, 2020), (Feng, et al., 2020)
Graph Neural Networks (GNNs)	(Wang, Wang, Gao, & Wang, 2020), (Tarlow, et al., 2019), (Mi, et al., 2021), (Wu, et al., 2021), (Suneja, Zheng, Zhuang, Laredo, & Morari, 2020), (Vytovtov & Chuvilin, 2019), (Nguyen, et al., 2022)
AST-based Approaches	(Peng, et al., 2015), (Chakraborty, Krishna, Ding, & Ray, 2020), (Ding, et al., 2021), (Nguyen, et al., 2022), (Hin, Kan, Chen, & Babar, 2022)
Other Approaches	(Xie, Kersting, & Neider, 2022), (Hellendoorn, Sutton, Singh, Maniatis, & Bieber, 2020), (Li, Wang, & Nguyen, 2021), (Eshghie, Artho, & Gurov, 2021), (Dinella, et al., 2020), (Alon, Zilberstein, Levy, & Yahav, 2019), (Şahin, Özyedierler, & Tosun, 2022), (Evangelos, Katsadouros and Charalampos, & Patrikakis, 2022), (Tann, Han, Gupta, & Ong, 2018)

CHAPTER 3

RESEARCH GAP

Based on the literature review and previous works, we have identified that there has been relatively little research on multi-class categorization about Ethereum smart contract vulnerabilities based on Ethereum Virtual Machine (EVM) bytecode. Thus, creating multi-class classification models using RNNs and CNNs trained specifically on smart contract bytecode will be our main goal. We aim to investigate the following research question in this study.

1. Could EVM bytecode be used as a feature for uncovering smart contract vulnerabilities?
2. How do RNNs and CNNs trained on EVM bytecode perform compared to other deep learning solutions?

3.1 Could EVM bytecode be used as a feature for uncovering smart contract vulnerabilities?

We aim at developing a deep learning model using RNNs or CNNs and training it on EVM bytecode. Our goal will be to achieve an weighted F1-score, which is a harmonic mean between precision and recall, higher than 0.85.

The F1 score, shown in Equation 1, is the metric we use to evaluate the models. It combines two essential performance measures: precision and recall. The ratio of true positive predictions to the total number of positive predictions (true positives plus false positives) is known as precision, whereas recall is the ratio of true positive predictions to the total number of actual positive instances (true positives plus false negatives). Better performance is indicated by a higher score on the F1 scale, which goes from 0 to 1. A score of 0 indicates that neither precision nor recall are present, whereas a score of 1 indicates flawless precision and recall.

$$F1\ score = \frac{2\ x\ Precision\ x\ Recall}{Precision + Recall} \quad \text{(Equation 1)}$$

3.2 How do RNNs and CNNs trained on EVM bytecode perform compared to static analysis tools and other deep learning solutions?

We aim at comparing two different approaches, deep learning and static analysis, at how efficient they are in detecting vulnerabilities in a dataset of public smart contracts. We will develop two models using RNNs and CNNs to observe the performance of each compared to the current approaches widely used. We will compare the performance of each model and tool based on the F1-score mentioned above. Our goal is to understand how well models trained on EVM bytecode can capture patterns in vulnerability detection.

CHAPTER 4

ETHEREUM AND SMART CONTRACTS

4.1 Ethereum as a blockchain

Introduced for the first time by Vitalik Buterin (2013), Ethereum is a public blockchain network that is distributed and designed to execute any kind of decentralized application code, as shown in Fig. 1.

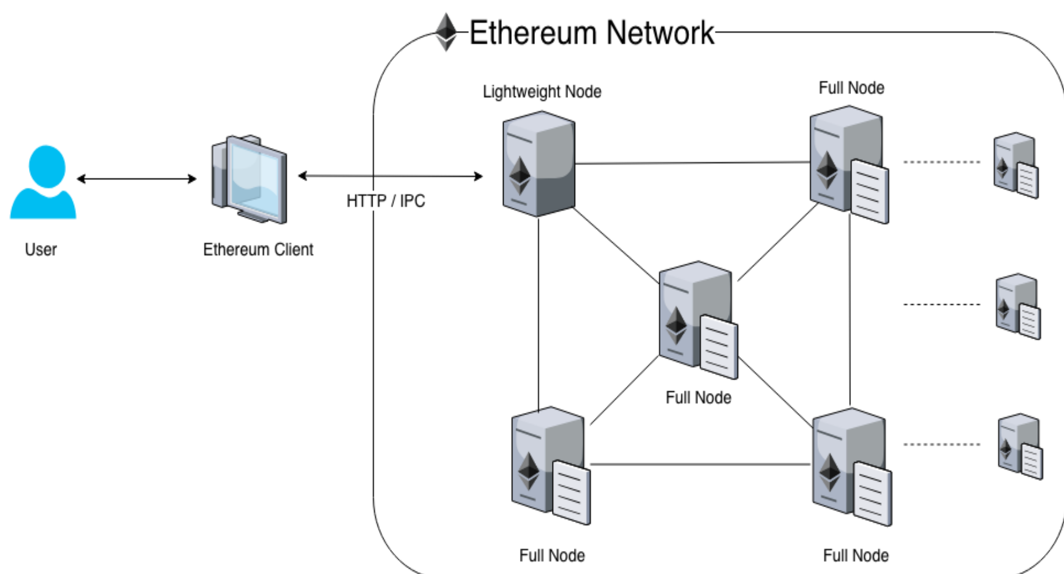

Figure 1. Ethereum architecture. (Chittoda, 2019)

Multiple entities make up Ethereum's blockchain network architecture. Every node in an Ethereum network maintains the most recent version of the Ethereum blockchain ledger and is connected to every other node via the P2P network protocol. Through the Ethereum client, which could be a desktop, mobile or web page, a user can communicate with the Ethereum network. The Ethereum Virtual Machine (EVM) powers every node in the network, allowing it to carry out its commands. The nodes then translate the smart contracts into EVM code and carry them out (Wood, 2018).

The Ethereum network relies on a modified version of the GHOST protocol (Greedy Heaviest Observed Subtree) for consensus, which was developed to address the network's stale block problem. Stale blocks can happen when a mining pool's aggregate processing power is greater than that of the other groups of miners. This means that the blocks from the first pool will contribute more to the network, which will lead to a centralization problem. These outdated blocks are considered by the GHOST protocol for determining the longest chain

Ethereum can be viewed as a transaction-based state machine with a built-in Turing-complete programming language (Farnaghi & Mansourian, 2020). It begins with a genesis state and after each successful transaction, the state is modified. Data relevant to the physical world is included in the state, along with account balances, trust agreements, and reputations.

Anyone can design their own ownership structures for transactions and state transition mechanisms because of Ethereum's abstract layer. Smart contracts, used to achieve the previous objective, are a collection of cryptographic guidelines that only act when specific requirements are satisfied (Chittoda, 2019).

4.2 Ethereum key components

We will now examine the essential elements of Ethereum, including the account, the global state, and the transaction as shown in Fig. 2.

According to the Ethereum Foundation, a core element of Ethereum is the account (Buterin V. , 2013). Each account contains four important fields that are nonce, balance, storage hash and code hash.

The nonce is a transaction counter, increased by one for every new transaction sent by the account. In a similar manner, it indicates the number of contracts generated by an account that is associated with a code (Chittoda, 2019).

The balance is a scalar number that represents how many Weis the address owns (Chittoda, 2019).

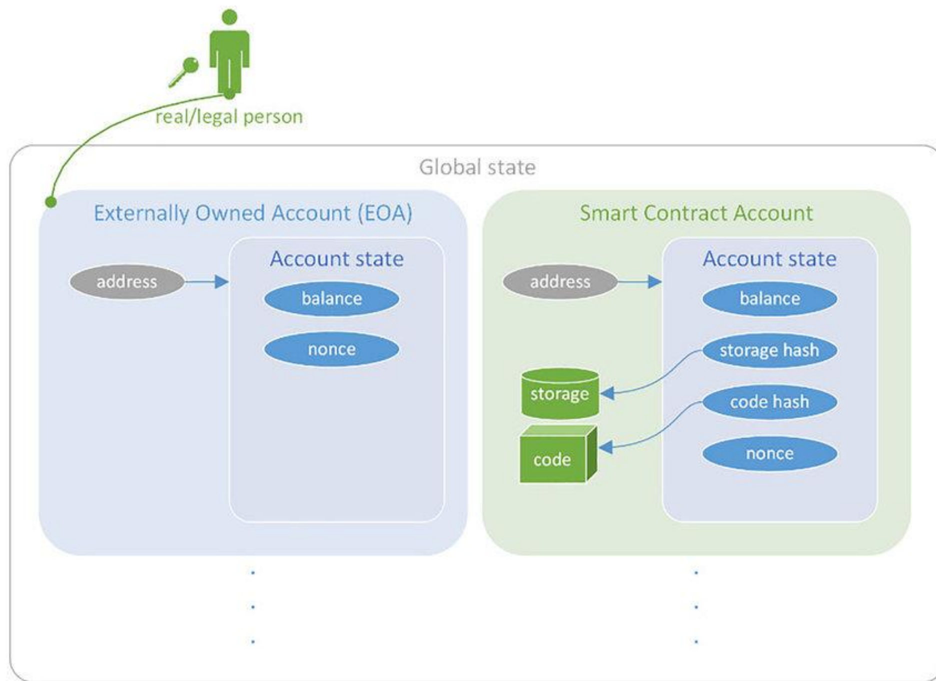

Figure 2. Accounts in Ethereum. (Farnaghi & Mansourian, 2020)

The storage hash is made up of a 256-bit hash of the Merkle Patricia tree's root node, which encodes the account's storage contents. The encoded values are mapped from Keccak 256-bit integer values into the tree (Chittoda, 2019).

Finally, the code hash is the hash of the Ethereum Virtual Machine (EVM) code of this account. This code runs if a message call is made to this address. For eventual retrieval, all these code snippets are stored in the state database under their respective hashes (Chittoda, 2019).

The global state is a mapping between addresses, which are 160-bit identifiers and account states. As stated previously, the global state represents the current snapshot of account balances, contract storage, and other relevant information for all accounts and smart contracts on the Ethereum network.

Accounts come in two varieties: contract accounts and external accounts. A contract account is managed by its code, while an external account is managed by public-private key pairs held by human account holders. The key difference between the two is that contract accounts do not have an empty code field as external accounts do.

Actions that modify the EVM's state can only be started by external accounts. We refer to these actions as transactions. A transaction is a single instruction created by an actor outside the Ethereum network and cryptographically signed. There are three transaction types: 0 (legacy), 1 (EIP-2930) and 2 (EIP-1559) (Buterin, et al., 2019). Furthermore, there are two subtypes of transactions: those which result in message calls and those which result in the creation of new accounts with associated code, which as mentioned earlier are the contact accounts. This process is called contract creation and is described more in depth in the next section.

Each transaction consists of the recipient of the message, a signature identifying the sender, amount of Wei to be sent, an optional data field, gas amount, and gas price values (Wood, 2018).

The reason why the last fields are included is because transaction execution is not always free. This is because energy-intensive processing resources are needed for the global state change (transaction) that needs to be approved by all (Buterin, 2013). Gas was developed for smart-contract communication and transaction execution. It stands for the units that the transaction initiator must pay to have the transaction executed. The gas price is the amount of Ether that the initiator is willing to pay for each unit of gas, whereas the gas limit is the highest quantity of gas that the initiator is ready to pay (Buterin, 2013).

4.3 Smart contracts and their lifecycle

Smart contracts are programs that run on the Ethereum blockchain. They consist of data and code that is kept on the Ethereum blockchain at a particular address.

Nonetheless, smart contracts remain a specific type of Ethereum account known also as contract accounts. As mentioned earlier, these types of accounts have a balance and can be the target of transactions. However, they are not controlled by a user, instead they are deployed to the network and run as programmed. Submitting transactions that carry out a function specified on the smart contract is how user accounts can then communicate with the contract. Like a traditional contract, a smart

contract can specify rules and have the code automatically enforce them. Smart contract interactions are irreversible and cannot be deleted by default.

As shown in Fig. 3, the four stages of a smart contract are creation, deployment, execution, and completion (Buterin, 2013).

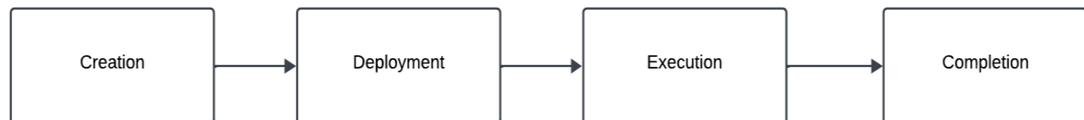

Figure 3. The lifecycle of Ethereum smart contracts.

4.3.1 Creation Phase

During the creation phase, the smart contract is written in a high-level programming language, such as Solidity as shown in Code 1. The developer specifies the rules that will guide the operations. This includes definition of functions and data storage.

Functions are rules that could include external calls to other smart contracts or externally owned accounts, arithmetic operations and security checks.

Any contract data needs to be allocated to one of two places: memory or storage. Memory variables are values that are only kept for the duration of a contract function's execution. These are far less expensive to use because they are not kept on the blockchain indefinitely. On the other hand, those that are stored for a longer period of time are inserted in the global state and called storage values. Changing the storage is a costly operation and requires more gas.

4.3.2 Deployment Phase

The primary objective of the deployment phase is to transfer the rules defined in the smart contract during the creation phase to the EVM. However, the Ethereum Virtual Machine (EVM) understands a low-level, stack-based bytecode language.

```
// Solidity Smart Contract
pragma solidity ^0.8.0;

contract SimpleCounter {
    uint256 public counter;

    function incrementCounter() public {
        counter++;
    }
}
```

Code 1. Smart contract written in Solidity that implements a counter function.

Consequently, the smart contract must be compiled into opcodes, which are the EVM's low-level instructions. The EVM bytecode is a set of instructions that the Ethereum nodes execute. It is a low-level representation of smart contract logic. Each bytecode instruction corresponds to a specific operation, such as arithmetic, storage manipulation, conditional branching, etc. The following is an example bytecode of the simple counter smart contract in Code 1.

```
60606040818152600181526020600482015260 1f60248201527f48656c6c6f2c20576f
726c64210000000000000000000000000000000602082015290565b506000819055
505b5060e58061003e6000396000f3fe6080604052600436106049576000357c01000
00000000000000000000000000000000000000000000000000000900463ffffffff1680
6359d1d3d714604e578063cfae3217146070575b600080fd5b348015605957600080f
d5b50607660048036038101908080359060200190929190505050506080565b005b348
015608757600080fd5b50608e60aa565b604051808281526020019150506040518091
0390f35b8060006000508190909055506000600050548156fea165627a7a723058209
a1a2b0f3f1b99708c65c0a0c07ea6f297fa4d1a0712856cf1115dca7cc631a1a0029
```

The developer starts a transaction that includes the bytecodes kept in the transaction structure's "init" field during the Deployment phase. This action returns

another code fragment that will be saved in the EVM running environment and executed later. It can be carried out with tools like Truffle.

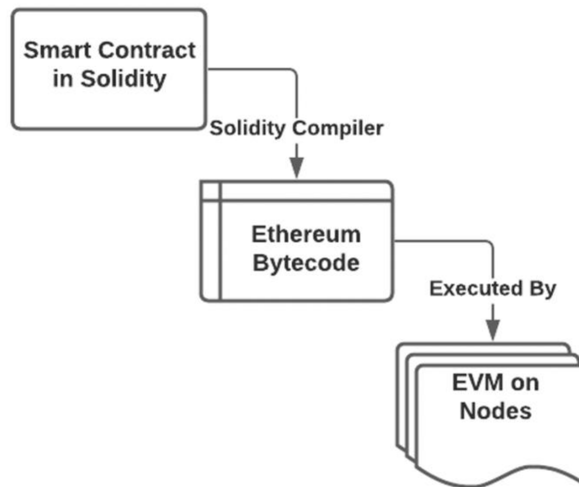

Figure 4. The workflow of Ethereum smart contract source. (Yiu, 2021)

4.3.3 Execution Phase

When a smart contract is deployed to the Ethereum blockchain, the EVM ensures the execution of the contract's bytecode in a decentralized and deterministic manner across all nodes in the network. A smart contract bytecode is a set of operations that runs similar to a thread or process on a standalone computer. EVM bytecode is machine-readable and is the language that the Ethereum network nodes understand and execute. Until the task is completed or the gas limit is reached, the EVM carries out each instruction one at a time. The moment a new block is mined, this process takes place.

4.3.4 Completion Phase

In the completion phase, after the transaction has been concluded, states are modified and documented in blockchains. The new block is mined and distributed to all the nodes in the network. Finally, the changes in state persist in the global state.

CHAPTER 5

ETHEREUM SMART CONTRACT TYPES OF VULNERABILITIES

5.1 Reentrancy

According to Code 2, reentrancy describes a situation where contract A calls contract B, but contract B may call A back and carry out A's call again due to Solidity's fallback mechanism. The fallback function will be called in the event that calls from other contracts are unable to find a matching function. The callee's fallback function will be activated if the caller uses the call function without providing a function signature. In order to re-enter the caller, this function may call the caller's function.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VulnerableContract {
    mapping(address => uint) private balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");

        // imagine an external call here, like calling another contract
        (bool success, ) = msg.sender.call{value: _amount}("");
        require(success, "Transfer failed");

        balances[msg.sender] -= _amount;
    }
```

```
    function getBalance() public view returns (uint) {
        return balances[msg.sender];
    }
}

// an attacker contract that exploits reentrancy
contract AttackerContract {
    VulnerableContract vulnerableContract;

    constructor(address _vulnerableContractAddress) {
        vulnerableContract = VulnerableContract(_vulnerableContractAddress);
    }

    // this function is designed to exploit reentrancy
    function attack() public payable {
        // call the withdraw function of the vulnerable contract
        vulnerableContract.withdraw(1 ether);
    }

    // fallback function to keep the attack going
    receive() external payable {
        if (address(vulnerableContract).balance >= 1 ether) {
            vulnerableContract.withdraw(1 ether);
        }
    }

}
```

Code 2. An example of smart contract with reentracy vulnerability.

5.2 Arithmetic issues

Overflow and underflow can happen when using integer variables with value constraints for addition, subtraction, or storing user input, as the examples in Code 3 demonstrate. Variable values will wrap to the other side of the bound if they exceed either the upper or lower bound.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SafeMath {
    function unsafeAdd(uint256 a, uint256 b) public pure returns (uint256) {
        return a + b;
    }

    function safeAdd(uint256 a, uint256 b) public pure returns (uint256) {
        require(a + b >= a, "SafeMath: Addition overflow");
        return a + b;
    }
}
```

Code 3. An example of smart contract implementing safe addition and unsafe addition.

5.3 Unchecked external calls

Unchecked external calls in Solidity, described by Code 4, are instances in which calls to external functions are made without properly implementing error handling methods or verifying the return values. In Solidity, external calls are interactions with external addresses or other contracts. If not used appropriately, they can result in many problems. Unchecked external calls are seen as dangerous because they could malfunction or act strangely, and they could trigger smart contract vulnerabilities if their results are not adequately verified.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Caller {
    uint public value;

    function setValue(address _target, uint _newValue) external {
        // using delegatecall to execute the setValue function of the target contract
        (bool success, ) =
_target.delegatecall(abi.encodeWithSignature("setValue(uint256)", _newValue));
        require(success, "Delegate call failed");
```

```
    }
}

contract Callee {
    uint public value;

    function setValue(uint _newValue) external {
        value = _newValue;
    }
}
```

Code 4. An example of smart contracts implementing unchecked external calls.

5.4 Access control

"tx.origin" is always an external account that keeps track of the originating caller's address for a transaction. Bob is "tx.origin" if contract A is being called by contract B, and contract A is being called by Bob. But contract A's msg.sender is contract B's. The person who called right away is identified as "msg.sender". It is advised never to utilize "tx.origin" for authentication or identity verification.

```
// Contract A
pragma solidity ^0.8.0;

contract ContractA {
    address public originalCaller;

    // Function in Contract A that calls Contract B
    function callContractB(address contractBAddress) external {
        // Save the original caller (Bob)
        originalCaller = tx.origin;

        // Call Contract B
        ContractB(contractBAddress).doSomething();
    }
}

// Contract B
contract ContractB {
    address public actualCaller;

    // Function in Contract B
    function doSomething() external {
        // Save the actual caller (Contract A)
        actualCaller = msg.sender;
    }
```

Code 5. An example of smart contracts having access control vulnerability.

5.5 Other issues

A smart contract's function is to send and receive ether. A person is considered in a freezing state if they can only accept ether and cannot send ether out. The contract might be avaricious and freeze the Ether transferred to its address if it does not mention any withdrawal functions. The necessary procedure for depositing Ether is illustrated in the example below; once deposited, it cannot be withdrawn. The Ether fund is going to be frozen for good.

Solidity contracts and Ethereum lack a real source of entropy. Because historical blocks never change, attackers can utilize the same random number

creation procedure to get the same result if the timestamp or hash of historical blocks is used for randomness production. The technique could be vulnerable to rogue miners who could purposefully select transactions and the sequence in which they are executed if future blocks are used.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Example {
    address public owner;

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;
    }

    function transferOwnership(address newOwner) external onlyOwner {
        (bool success, ) =
newOwner.call(abi.encodeWithSignature("acceptOwnership()"));

        // checking the return value and handling it
        require(success, "Ownership transfer failed");

        // update the owner
        owner = newOwner;
    }
}
```

Code 6. An example of smart contracts implementing freezing ether vulnerability.

CHAPTER 6

DATASET

6.1 Data Collection

The dataset used to train and validate the models was created by Martina Rossini (2022) and is publicly accessible from the HuggingFace repository. The dataset consists of verified Ethereum smart contracts provided by SmartBugs, ScrawlID and Smart Contract Sanctuary, which is a repository in Github that collects smart contracts in different blockchain environments.

The authors of the dataset then used the Slither contract flattener to download the source code from the repository or via Etherscan (Rossini, 2022). Using INFURA as their endpoint, they used the web3.eth.getCode() function of the Web3.py module to extract the bytecode. Finally, the Slither static analysis tool was used to review every smart contract. After identifying 38 distinct vulnerability types in the gathered contracts, they were categorized into five groups. These groups represent five different sorts of vulnerabilities: reentrancy, arithmetic issues, unchecked calls, access control, and others. Others is a class that groups all other detectors.

The dataset contains 106474 audited smart contracts split into three categories: train, validation and test. The train split contains 79641 entries, the test split 15972 entries and the validation split 10861 entries. Every data object has the address, source code, bytecode, and slither audited output. The address is a representation of the Ethereum blockchain platform's smart contract address. The source code is its plain-text code in Solidity. The low-level, machine-readable hexadecimal code known as bytecode on Ethereum, explains how a smart contract functions and behaves. Slither consists of a list of vulnerabilities detected by the Slither static analysis tool.

6.2 Data Cleaning

Cleaning the dataset before use was essential. The authors (Rossini, 2022) mentioned that smart contracts without a bytecode can be recognized by their bytecode field in the specific row containing only the characters "0x". These symbols are commonly used to signify a string's hexadecimal format. Consequently, for training, validating, and testing the model, only the data items with bytecode lengths longer than 4 are retained. The creators of the dataset also used the number 4 to represent the security of a contract. As the most secure smart contracts are those that are impervious to attacks, label 4 was removed from the model to prevent confusion.

The dataset must also be split into two groups, called features and labels. In this case, the Slither output that pinpoints the vulnerabilities will act as the labels, and the smart contract bytecode will act as the features. To better understand the relationships between the Ethereum operation codes, the remaining bytecode is divided into two characters that are separated by whitespace. The distribution of the bytecode length on the dataset is shown in Fig. 5.

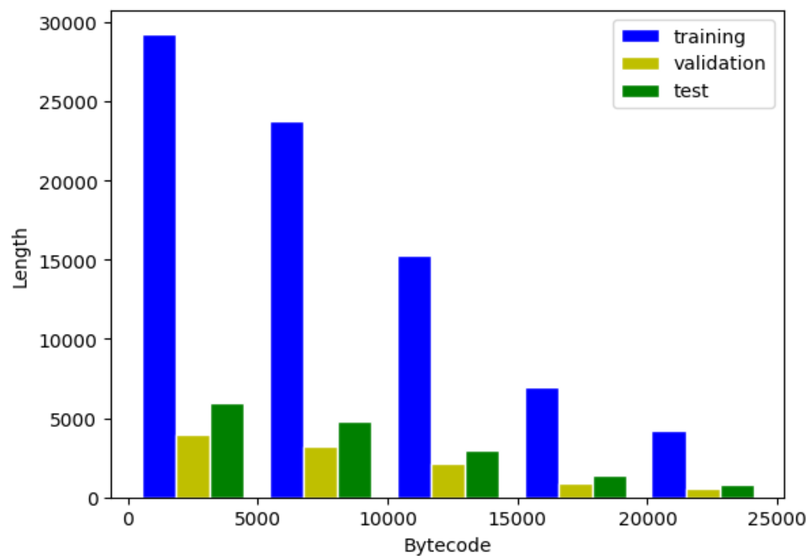

Figure 5. Distribution of bytecode length in the dataset.

Each data instance's slither output is a list containing a configurable number of vulnerabilities. The Slither output is first converted to binary format, where the

output number, if it exists in the list, is represented as 1 in that index on the new list, to make the labels uniform. The distribution of the labels is shown in Fig. 6.

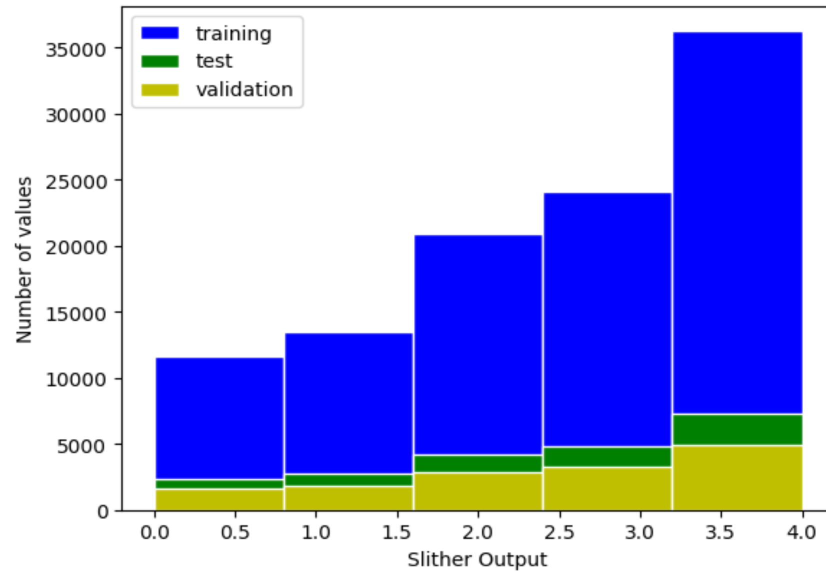

Figure 6. Distribution of outputs in the dataset.

The slither output, for one of the data instances, for example, shows that the smart contract has two vulnerabilities, which are identified as [1, 3]. The highest number of vulnerabilities that can be found is five, since we previously removed label 4. The list [1, 3]'s binary output would be [0, 1, 0, 1, 0, 0]. Due to the existence of vulnerabilities 1 and 3, they are represented by 1 on the binary list. These binary vulnerabilities are then categorized in a Python dictionary according to the types they represent.

Finally, “Dataset” is used to create the datasets that will be fed to the model. To maximize GPU speed, it combines the bytecode and the vulnerability dictionary, divides them into groups of 32, and utilizes autotuned prefetch.

CHAPTER 7

ARCHITECTURE OF OUR MODELS

Both models receive the input EVM bytecode, which is divided into pairs of characters. This division by pairs allows for a more accurate representation of the Ethereum opcodes. Contrary to other models that are fed with source code, we feed these sequences of pairs from the EVM bytecode into our models. Both models are trained and validated on a GPU T4 with 50 GB of System RAM and 15 GB of GPU RAM.

7.1 CNNs

The model is a multi-output convolutional neural network (CNN) as shown in Figure 8. The text vectorizer layer uses 256 tokens as the maximum number of tokens, an output sequence length of 21041 and splits by whitespace. The embedding layer uses an input dimension of 256, an input length of 21041 and an output dimension of 128. The model has three sets of convolutional blocks, each consisting of two 1D convolutional layers followed by a max-pooling layer. These convolutional layers are designed to capture different levels of abstraction in the input data.

- First Block: 2 convolutional layers with 4 filters each, kernel of size 3, stride of size 1, followed by max pooling.
- Second Block: 2 convolutional layers with 8 filters each, kernel of size 3, stride of size 1, followed by max pooling.
- Third Block: 2 convolutional layers with 16 filters each, kernel of size 3, stride of size 1, followed by max pooling.

After each convolutional block, the number of features is doubled to give the model the chance to forget any of the patterns it may have previously learnt, as suggested also in the WaveNet (Zhuang et al., 2020). At the end, there is a global max-pooling layer to capture the most important features from each feature map. A dense (fully connected) layer follows the global max-pooling layer. It has 32 units

and uses the ReLU activation function. There is a distinct dense layer with a single unit and sigmoid activation for each class (indexed by an index). The loss function used is binary cross entropy, the learning rate is 0.001 and the optimization function is Adam. The model is fitted on the training dataset and trained for 35 epochs. It has 36557 trainable parameters in total and each epoch takes about 165 seconds to run.

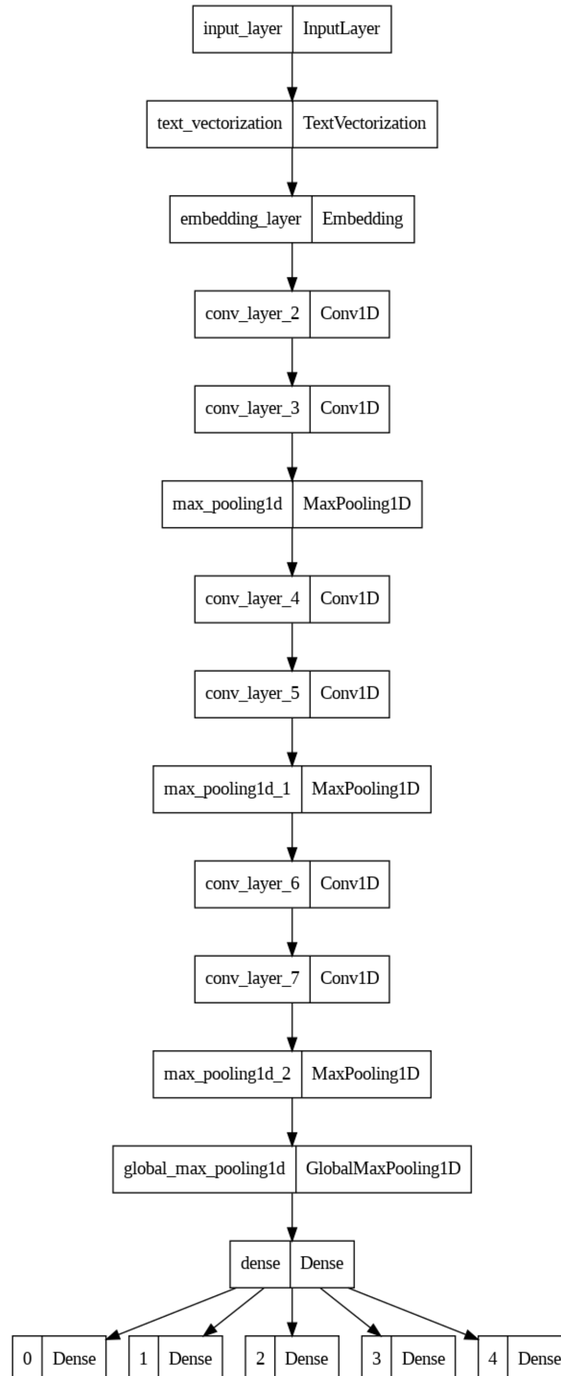

Figure 7. Architecture of the model with CNNs.

7.2 RNNs

The model architecture with RNNs is composed of various layers that sequentially process input data and produce a multi-output structure as shown in Figure 8.

The text vectorizer layer splits by whitespace, uses an output sequence length of 21041 characters, and allows a maximum of 256 tokens. The embedding layer uses a 256-dimensional input and output, with an input length of 21041. Two Gated Recurrent Unit (GRU) layers are employed for sequential modeling of the embedded text data. The first GRU layer has 64 units and is configured to return sequences, while the second GRU layer has 32 units. The activation function employed is the hyperbolic tangent.

A dropout layer with a dropout rate of 0.2 is introduced to prevent overfitting, followed by a fully connected dense layer with 16 units and a hyperbolic tangent activation function. The final output layer is made up of several sub-layers, each of which uses sigmoid activation to produce a binary classification result.

The output sub-layer count is equal to the five labels that are supplied. The loss function used is binary cross entropy, the learning rate is 0.001 and the optimization function is Adam. The model is fitted on the training dataset and trained for 20 epochs. It has 80037 trainable parameters, and each epoch takes about 2216 seconds.

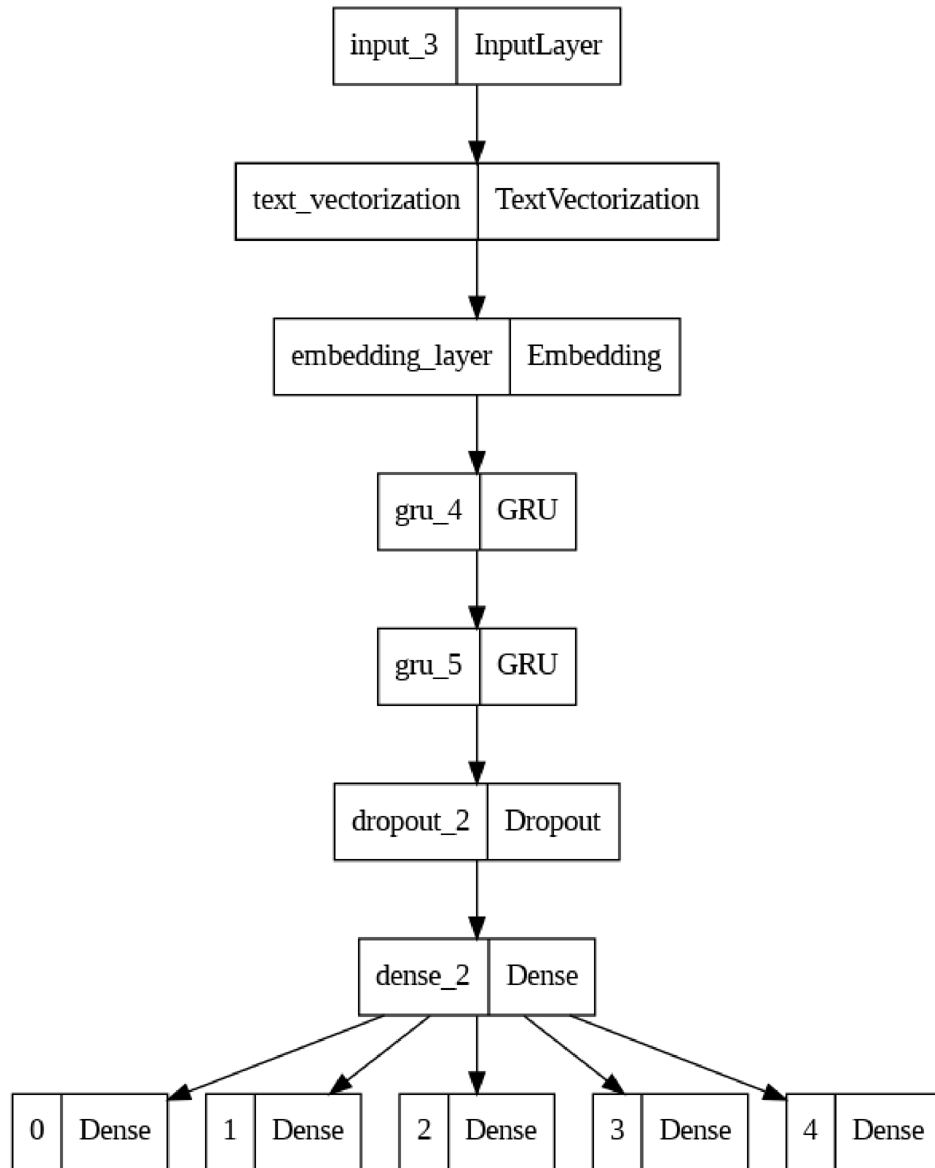

Figure 8. Architecture of the model with RNNs.

CHAPTER 8

RESULTS

This section summarizes the results we were able to get during the training, validation and testing phase on the dataset mentioned above.

8.1 Results of the CNN model

The visual in Fig. 9 displays the training loss for every single vulnerability. It is consistently falling, which indicates that the model is accurately representing the features. On the other hand, the validity loss fluctuates, occasionally increasing and occasionally decreasing over time.

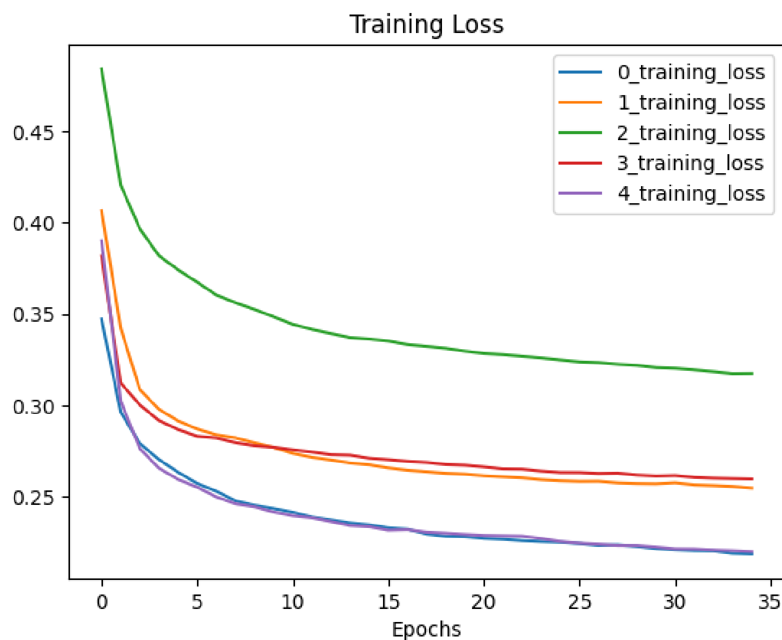

Figure 9. Training loss during each epoch.

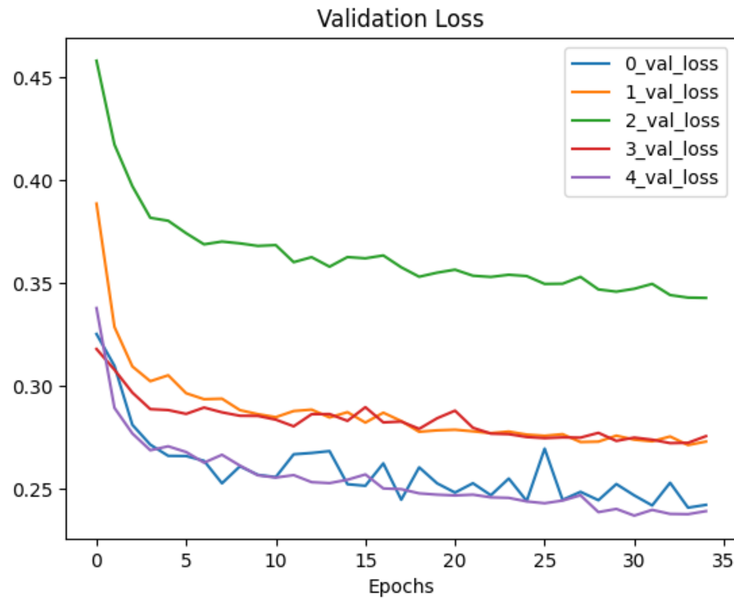

Figure 10. Validation loss during each epoch**.**

Accuracy in training and validation is rising continuously as shown in Figure 11. Accuracy is highest for the first and final flaws, particularly access control and unchecked calls. The "other" class vulnerabilities have the lowest precision, making it more difficult for the model to map these vulnerabilities. This is because there is less data with this kind of vulnerability and there are several bytecode changes that could be indicative of this issue.

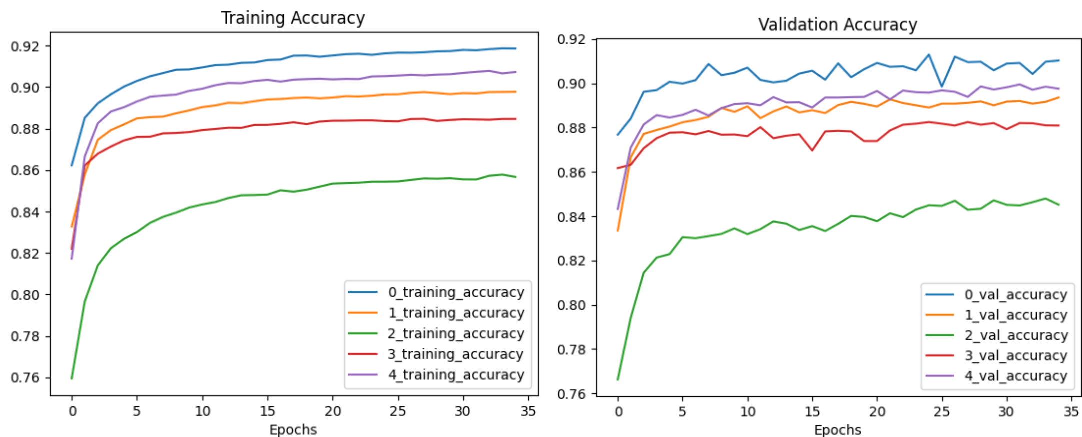

Figure 11. Comparison of training and validation accuracy.

The model was tested on the test dataset and evaluated, as shown in Table 2, on four metrics, accuracy, precision, recall and weighted f1-score.

Table 2. Metrics on the test split of the dataset for the CNN model

	Access control	**Arithmetic Issues**	**Other**	**Reentrancy**	**Unchecked external calls**
Accuracy	90.515671	89.510709	84.730859	88.267069	90.157653
Precision	0.899902	0.891799	0.842886	0.881834	0.901656
Recall	0.905157	0.895107	0.847309	0.882671	0.901577
Weighted F1-score	0.901644	0.881681	0.837517	0.879274	0.901437

8.2 Results of the RNN model

The training loss for each vulnerability is shown in Figure 12. It is steadily declining, proving that the model is capturing the patterns of EVM bytecode. It is worth noting that the validity loss with RNNs, shown in Figure 13, is more consistent over time than CNNs, which experienced fluctuations.

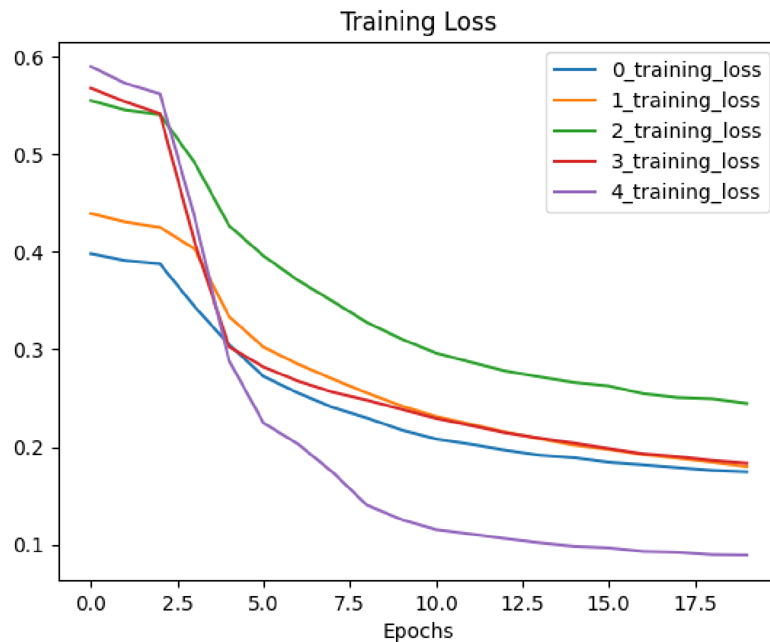

Figure 12. Training loss during each epoch.

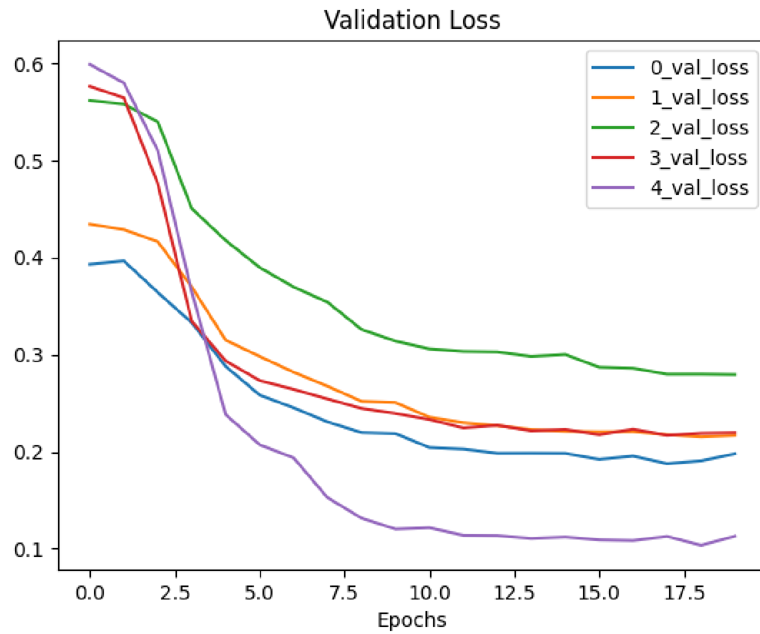

Figure 13. Validation loss during each epoch.

In Figure 7.2.3, training and validation accuracy are both steadily improving. The first and last faults, namely access control and unchecked external calls, have the highest accuracy. The model has a harder time mapping vulnerabilities in the "other" class since they have the lowest precision.

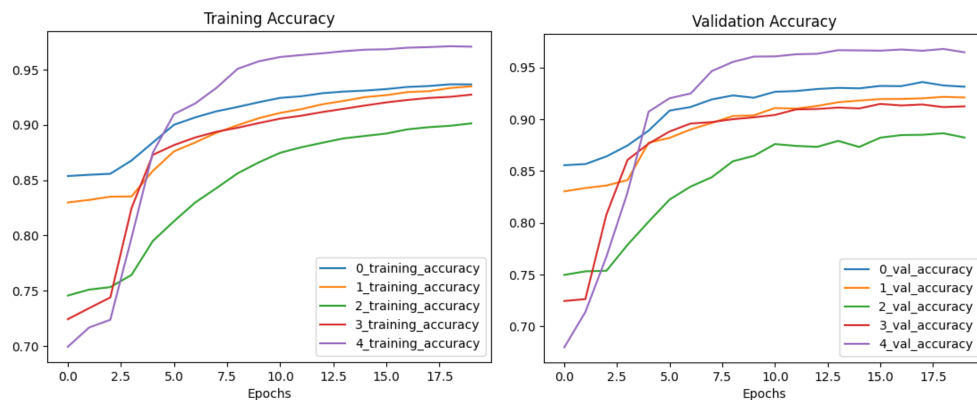

Figure 14. Comparison of training and validation accuracy.

We used the test dataset to evaluate the model on four metrics such as accuracy, precision, recall and f1-score. The results are shown in Table 3.

Table 3. Metrics on the test split of the dataset.

	Access control	Arithmetic Issues	Other	Reentrancy	Unchecked external calls
Accuracy	93.185101	92.337165	88.945418	91.369889	96.815527
Precision	0.928605	0.920718	0.887975	0.913099	0.968689
Recall	0.931851	0.923372	0.889454	0.913699	0.968155
Weighted F1-score	0.928301	0.921411	0.884900	0.912275	0.968094

8.3 Result comparison

The micro F1-score for both our models is an average micro F1-score calculated across all five vulnerability classes on the validation split of the dataset. Table 7.3.1 displays the micro F1-scores that different models trained by Rossini, Zichichi, & Ferret (2022) achieved on the validation split of the dataset. The last two entries of Table 4 show the micro F1-score that our models achieved on the same split. They clearly perform better than the other models.

Table 4. Comparison of micro F1-scores between different models.

Model	**Micro F1-score**
ResNet1D	0.8381
ResNet	0.7928
Inception	0.8015
LSTM Baseline	0.7953
Our RNN model	0.93
Our CNN model	0.89

CHAPTER 9

CONCLUSIONS

9.1 Conclusions

Based on the results achieved during testing, we conclude that EVM bytecode is effective in showing smart contract vulnerabilities. We base our conclusion on the fact that we managed to achieve an average micro F1-score of 0.93 and 0.89 across all five vulnerabilities for our RNN model and CNN model, respectively.

9.2 Recommendations for future research

For future initiatives, we recommend stepping up data gathering efforts and convolutional block complexity in the model architecture. The quantity and diversity of the dataset can be increased to facilitate the model's generalization and to provide more detailed insights and applications. The auditing of smart contracts could be performed by experts in the field to ensure the correct labeling of the smart contracts. Finally, we encourage developing a concatenation between two different types of inputs in a single model. One input layer would expect the bytecode of the smart contract and the other input layer would expect the source code of the same smart contract.

REFERENCES

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2020). A transformer-based approach for source code summarization.

Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code.

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., . . . Zanella-Béguelin, S. (2016). Formal Verification of Smart Contracts: Short Paper. *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, (pp. 91–96).

Bigi, G., Bracciali, A., Meacci, G., & Tuosto, E. (2015). Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods. In C. Bodei, G. Ferrari, & C. Priami, *Programming Languages with Applications to Biology and Security.*

Brent, L., Grech, N., Lagouvardos, S., Scholz, B., & Smaragdakis, Y. (2020). Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities.

Buterin, V. (2013). Ethereum white paper: a next generation smart contract & decentralized application platform.

Buterin, V., Conner, E., Dudley, R., Slipper, M., Norden, I., & Bakhta, A. (2019). EIP1559: Fee market change for eth 1.0 chain.

Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2020). Deep learning based vulnerability detection: Are we there yet?

Cheng, X., Wang, H., Hua, J., Xu, G., & Sui, Y. (2021). "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology*.

Chittoda, J. (2019). Mastering Blockchain Programming with Solidity.

Consesys. (2018). *Mythril.* Retrieved from Github: https://github.com/ConsenSys/mythril

Deng, W., Wei, H., Huang, T., Cao, C., Peng, Y., & Hu, X. (2023). Smart Contract Vulnerability Detection Based on Deep Learning and Multimodal Decision Fusion.

Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., & Wang, K. (2020). Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs.

Ding, Y., Suneja, S., Zheng, Y., Laredo, J., Morari, A., Kaiser, G., & Ray, B. (2021). VELVET: A novel ensemble learning approach to automatically locate VulnErable sTatements.

Eshghie, M., Artho, C., & Gurov, D. (2021). Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning.

Evangelos, D., Katsadouros and Charalampos, Z., & Patrikakis. (2022). A Survey on Vulnerability Prediction using GNNs.

Fan, Y., Shang, S., & Ding, X. (2021). Smart Contract Vulnerability Detection Based on Dual Attention Graph Convolutional Network.

Farnaghi, M., & Mansourian, A. (2020). Blockchain, an enabling technology for transparent and accountable decentralized public participatory GIS.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., . . . Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages.

Gao, Z., Jiang, L., Xia, X., Lo, D., & Grundy, J. (2020). Checking Smart Contracts With Structural Code Embedding.

Gogineni, A. K., Swayamjyoti, S., Sahoo, D., & Sahu, K. K. (2022). Multi-Class classification of vulnerabilities in Smart Contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing.

Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., & Zohar, Y. (2018). Online detection of effectively callback free objects with applications to smart contracts. *PACMPL 2, POPL.* New York, United States: Association for Computing Machinery.

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., & Bieber, D. (2020). Global relational models of source code.

Hin, D., Kan, A., Chen, H., & Babar, M. A. (2022). LineVD: Statement-level Vulnerability Detection using Graph Neural Networks.

Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification.

Huang, J., Zhou, K., Xiong, A., & Li, D. (2022). Smart Contract Vulnerability Detection Model Based on Multi-Task Learning.

Huang, T. H.-D. (2018). Hunting the Ethereum smart contract: Colorinspired inspection of potential attacks.

Kalra, S., Goel, S., Dhawan, M., & Sharma, S. (2018). *ZEUS: Analyzing Safety of Smart Contracts.*

Li, Y., Wang, S., & Nguyen, T. N. (2021). Vulnerability detection with finegrained interpretations. *European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 292–303). Association for Computing Machinery.

Loi Luu, D.-H. C. (2016). Making Smart Contracts Smarter. *ACM SIGSAC Conference on Computer and Communications Security*.

Lutz, O., Chen, H., Fereidooni, H., Sendner, C., Dmitrienko, A., Koushanfar, F., & Sadeghi, A. R. (2023). ESCORT: Ethereum Smart COntRacTs Vulnerability Detection using Deep Neural Network and Transfer Learning.

Meiying, W., Xie, Z., Wen, X., Li, J., & Zhou, K. (2023). Ethereum Smart Contract Vulnerability Detection Model Based on Triplet Loss and BiLSTM.

Merity, S., Keskar, N. S., & Socher, R. (2017). Regularizing and optimizing LSTM language models.

Mi, F., Wang, Z., Zhao, C., Guo, J., Ahmed, F., & Khan, L. (2021). VSCL: Automating Vulnerability Detection in Smart Contracts with Deep Learning.

Nguyen, V.-A., Nguyen, D. Q., Nguyen, V., Le, T., Tran, Q. H., & Phung, D. (2022). ReGVD: Revisiting Graph Neural Networks.

Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., & Hobor, A. (2018). Finding The Greedy, Prodigal, and Suicidal Contracts at Scale.

Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., & Jin, Z. (2015). Building program vector representations for deep learning.

Rossini, M. (2022). *Slither Audited Smart Contracts Dataset*. Retrieved from HuggingFace: https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts

Rossini, M., Zichichi, M., & Ferret, S. (2022). Smart Contracts Vulnerability Classification Through Deep Learning.

Şahin, S. E., Özyedierler, E. M., & Tosun, A. (2022). Predicting vulnerability inducing function versions using node embeddings and graph neural networks.

Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-attention with relative position representations.

Sun, Y., & Gu, L. (2021). Attention-based Machine Learning Model for Smart Contract Vulnerability Detection.

Suneja, S., Zheng, Y., Zhuang, Y., Laredo, J., & Morari, A. (2020). Learning to map source code to software vulnerability using code-as-a-graph.

Tann, W. J.-W., Han, X. J., Gupta, S. S., & Ong, Y.-S. (2018). Towards safer smart contracts: A sequence learning approach to detecting security threats.

Tarlow, D., Moitra, S., Rice, A., Chen, Z., Manzagol, P.-A., Sutton, C., & Aftandilian, E. (2019). Learning to fix build errors with graph2diff neural networks.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., & Veche, M. (2018). *Securify: Practical Security Analysis of Smart Contracts.*

Vytovtov, P., & Chuvilin, K. (2019). Unsupervised classifying of software source code using graph neural networks.

Wang, X., Sun, J., Hu, C., Yu, P., Zhang, B., & Hou, D. (2022). EtherFuzz: Mutation Fuzzing Smart Contracts for TOD Vulnerability Detection.

Wang, Y., Wang, K., Gao, F., & Wang, L. (2020). Learning semantic program embeddings with graph interval neural network.

Wood, G. (2018). Ethereum: a secure decentralized generalized transaction ledger, Byzantium version.

Wu, H., Zhang, Z., Wang, S., Lei, Y., Lin, B., Qin, Y., . . . Mao, X. (2021). Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques.

Xie, X., Kersting, K., & Neider, D. (2022). Neuro-Symbolic Verification of Deep Neural Networks.

Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. *IEEE Symposium on Security and Privacy* (pp. 590–604). IEEE.

Yiu, N. C. (2021). Toward Blockchain-Enabled Supply Chain Anti-Counterfeiting and Traceability.

Zhang, L., Wang, J., Wang, W., Jin, Z., Zhao, C., Cai, Z., & Chen, H. (2022). A Novel Smart Contract Vulnerability Detection Method Based on Information Graph and Ensemble Learning.

Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., & He, Q. (2020). Smart Contract Vulnerability Detection Using Graph Neural Networks.