

**TOWARDS SECURE SMART CONTRACTS: A DEEP LEARNING
APPROACH FOR DETECTING SECURITY THREATS**

by

TAMER ABDELAZIZ ABDELMEGID MOHAMED

(M.Sc., Helwan University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2023

Supervisors:

Siau-Cheng Khoo, Associate Professor , Main Supervisor
Aquinas Adam Hobor, Associate Professor , Co-Supervisor


Examiners:

Seth Lewis Gilbert, Associate Professor
Henz, Martin J, Associate Professor
Chang Ee Chien, Associate Professor

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

A handwritten signature in blue ink, appearing to read 'Tamer', with a large, stylized initial 'T'.

Tamer Abdelaziz Abdelmegid Mohamed

December 31, 2023

Alhamdulillah, by Allah's grace, I've earned my PhD! All praise and thanks to Him, the source of all knowledge.

Acknowledgments

Reaching the pinnacle of my Ph.D. in Computer Science at the National University of Singapore's School of Computing has been a monumental feat, paved with challenges and triumphs. None of it would have been possible without the invaluable support and encouragement of a remarkable group of individuals.

First and foremost, my deepest gratitude goes to my advisors, Dr. Siau-Cheng Khoo and Dr. Aquinas Adam Hobor. Their expertise, unwavering support, and insightful guidance have been instrumental in shaping my research and academic growth. From the initial research proposal to the final defense, their mentorship was a constant source of inspiration and motivation.

My thesis examiners deserve immense appreciation for their insightful critiques and encouragement. Their diverse perspectives pushed me to broaden my research, resulting in a richer and more robust dissertation.

My labmates, with their collaboration, constructive criticism, and stimulating discussions, have been my academic lifeline. Their diverse perspectives and research interests not only enriched my research experience but also broadened my horizons. I am deeply grateful for their camaraderie, friendship, and unwavering support, making my journey as a researcher both impactful and enjoyable.

My research wouldn't have been possible without the generous support of the Singapore International Graduate Award (SINGA), Dr. Joxan Jaffar and the NUS CRYSTAL Centre. This funding provided crucial resources and financial stability, allowing me to fully immerse myself in my research pursuits.

Beyond academia, my family has been my rock throughout this journey. My lovely wife Ola, my son Anas, my parents Abdelaziz and Hanya, and my siblings Mahmoud and Abeer, have showered me with unwavering love, support, and belief in my abilities. Their patience, understanding, and guidance were critical in sustaining me through the highs and lows. I am eternally grateful for their contributions to my personal and professional growth.

Finally, I express my heartfelt appreciation to the entire computer science department at the School of Computing. The department's vibrant culture of innovation, excellence, and collaboration has been a constant source of inspiration

and support. I am deeply grateful for the opportunities and resources provided, which have shaped my research and academic success.

In conclusion, my profound gratitude goes to Dr. Khoo, Dr. Hobor, my thesis examiners, labmates, funding sources, family, friends, and the entire School of Computing community. Your invaluable support, guidance, and encouragement have been the cornerstone of my Ph.D. journey, and I am eternally grateful for your unwavering belief in me.

Contents

Acknowledgments	ii
Abstract	viii
List of Figures	x
List of Tables	xiii
1 Introduction	1
1.1 Research Questions	4
1.2 Contributions	7
1.3 Thesis Statement	10
1.4 Tools Specification	10
1.5 Thesis Organization	11
2 Background	13
2.1 Ethereum	13
2.2 Smart Contracts	14
2.3 Ethereum Virtual Machine	15
2.4 Smart Contract Vulnerabilities	19
2.5 Deep Learning: Methods and Applications	31
2.5.1 Learning Methods	33
2.5.2 The Mechanics of a Basic Neural Network	38
2.5.3 Graph Neural Networks (GNNs)	39
2.6 Evaluation Metrics	41
3 Related work	44
3.1 Static and Dynamic Analysis Methods for SC	45

3.1.1	Symbolic Execution (SymEx) Studies/Tools:	49
3.1.2	Fuzzing (Fuz) Studies/Tools:	54
3.1.3	Static Analysis (StAn) Studies/Tools:	57
3.2	Learning-based Techniques for SC	60
3.2.1	Machine Learning (ML) Studies/Tools:	64
3.2.2	Sequential Deep Learning (Seq. DL) Studies/Tools:	72
3.2.3	Graph Deep Learning (Graph DL) Studies/Tools:	84
3.3	Learning-based Techniques for PL	89
3.3.1	Sequential Deep Learning (Seq. DL) Studies/Tools:	89
3.3.2	Graph Deep Learning (Graph DL) Studies/Tools:	94
3.4	Data Sources and Benchmarks	96
3.4.1	Manually Crafted Datasets:	97
3.4.2	Real World Datasets:	99
3.5	Summary of Related Work	103
3.6	Comparing Our Approach to State-of-the-Art	110
3.7	Ethical Disclosure	122
3.8	Threats to Validity	122
4	Supervised Deep Learning: DLVA	125
4.1	Introduction	126
4.2	Designing DLVA	131
4.2.1	Preprocessing	136
4.2.2	Unsupervised Node Feature Extraction: N2V	139
4.2.3	Supervised Training: SC2V and CC	140
4.2.4	Selection of hyperparameters	142
4.2.5	Sibling Detector (SD)	143
4.2.6	Tweaking for smaller contracts	144
4.2.7	Final details	144
4.3	Experiments and Evaluation	145
4.3.1	Designing benchmark datasets	145
4.3.2	DLVA’s neural nets vs. alternatives	149
4.3.3	Evaluating DLVA’s models against Slither	154
4.3.4	DLVA vs. state-of-the-art tools	158

4.3.5	Discussion	164
4.4	Key Comparative Studies	169
4.5	Summary	174
4.6	Availability	175
5	Semi-Supervised Learning: SCooLS	176
5.1	Introduction	177
5.2	Deep learning styles	180
5.3	Design of data sets	182
5.4	Designing SCooLS	183
5.4.1	Preprocessing	183
5.4.2	Graph Neural Networks (GNNs)	185
5.4.3	Semi-Supervised Self-Training	187
5.4.4	Final trained models	188
5.4.5	Discussion	188
5.5	Auto-Exploit Generator Design	189
5.6	Experiments and Evaluation	193
5.6.1	Evaluative Metrics	193
5.6.2	Experimental setup	194
5.6.3	SCooLS vs. state-of-the-art tools	195
5.6.4	Auto-exploit generator results	197
5.7	Key Comparative Studies	198
5.8	Summary	200
5.9	Availability	201
6	Conclusion and Future Work	202
6.1	Conclusion	202
6.2	Future Research Directions	203
	Bibliography	205
A	Word vs. Sentence Embeddings	223
A.1	Word-level Embeddings	224
A.1.1	Recurrent Neural Networks (RNNs)	224

A.1.2	Long Short Term Memory Networks (LSTMs)	225
A.1.3	Applying the word-level embeddings during research	227
A.2	Sentence-level Embeddings	229
	Publications during PhD Study	230

Abstract

The Ethereum blockchain has experienced a substantial surge in popularity in recent years, which has manifested in its increased adoption and widespread usage, leading to the development of decentralized applications (dApps) built on smart contracts. However, smart contracts are susceptible to various security vulnerabilities that can lead to devastating consequences. In this thesis, we propose a deep learning-based approach to detect and exploit vulnerabilities in Ethereum smart contracts.

The first part of the thesis focuses on automated detection of vulnerabilities in smart contracts, without requiring prior source code access. We use supervised deep learning to identify vulnerabilities directly from publicly available blockchain bytecode. This has resulted in the creation of a *Deep Learning Vulnerability Analyzer* (**DLVA**) [3, 4, 5], which is a fast and efficient solution for smart contract vulnerability detection. DLVA has a generic design and can be trained to recognize future vulnerabilities easily without using any painstakingly-crafted expert rules or predefined patterns. DLVA checks contracts for 29 distinct vulnerability types in 0.2 seconds, a speedup of 5-1,000x+ compared to traditional tools. Impressively, it achieves this while maintaining an optimal balance between high true positive rates and minimal false positive rates.

In the second part of this thesis, we demonstrate that the lack of large, labeled data sets for training deep learning models poses a significant challenge for the effective detection of vulnerabilities. To address this challenge, we use semi-supervised learning to produce more accurate models than unsupervised learning, while not requiring the large oracle-labeled training set that supervised learning requires. We propose a second solution called *Smart Contract Learning (Semi-supervised)* (**SCoolS**) [2], which represents a pioneering application of semi-supervised learning techniques in the realm of smart contracts vulnerability analysis. It uniquely enables the precise detection and exploitation of specific vulnerable functions. Significantly, it's the first tool to not only identify these vulnerable functions but also to generate authentic attack demonstrations for end-users and developers. This approach diverges from the traditional method of simply labeling the entire contract as vulnerable, providing developers with a tangible method to test the exploitability of their contracts. SCoolS exhibits superior performance when compared to existing

tools, showcasing exceptional accuracy, a notable F1 score, and an impressively low false positive rate. Additionally, SCoolS demonstrates remarkable speed in analyzing contract's functions. Leveraging its capability to pinpoint specific vulnerable functions, we successfully developed an exploit generator. This generator effectively extracted Ether from a significant portion of the identified vulnerable functions considered true positives.

Our deep learning approach is capable of detecting a higher number of vulnerabilities with a lower false positive rate, while being computationally efficient, making it a promising solution for enhancing the security of smart contracts.

List of Figures

1.1	An illustration depicting the reentrancy attack on DAO.	3
1.2	DLVA Specification.	10
1.3	SCoolS Specification.	11
2.1	Ethereum Virtual Machine (EVM), adapted from ethereum.org	16
2.2	Sample representation of Solidity source code	17
2.3	Sample representation of EVM bytecode	17
2.4	Sample representation of EVM opcodes	17
2.5	Smart Contract Control Flow Graph (adapted from [37]).	18
2.6	Reentrancy Exploit (DAO) Example	30
2.7	The relationship between artificial intelligence, machine learning, and deep learning.	32
2.8	Machine Learning vs Deep Learning.	34
2.9	Artificial Neural Network.	38
2.10	Multilayer Perceptron Neural Network.	39
2.11	Graph-structured data.	40
3.1	Static and Dynamic Analysis for SC in primary studies.	49
3.2	Learning-based Techniques for SC in primary studies; “Available*” with the asterisk denoting potential availability of some studies upon request approval; “Source code*” with the asterisk denoting the analysis level for some studies requires source code and/or potential alternatives like transactions or account data.	66
3.3	Learning-based Techniques for PL in primary studies.	90

3.4	Distribution of Tool Availability in SC vulnerability detection studies over time; “Available*” with the asterisk denoting potential availability of some studies upon request approval.	105
3.5	Distribution of Analysis Levels in SC vulnerability detection studies over time; “Source code*” with the asterisk denoting the analysis level for some studies requires source code and/or potential alternatives like transactions or account data.	106
3.6	DLVA vs. STATE-OF-THE-ART Tools; Completion Rate (i.e., the percentage of contracts for which a tool produces an answer rather than, e.g., raising an exception, timing out, running out of memory, the higher the better) tested on the <i>Elysium</i> _{benchmark} [7], <i>Reentrancy</i> _{benchmark} [11], and <i>SolidiFI</i> _{benchmark} [13]; star ★ indicates the mean; plus + indicates outliers	112
3.7	DLVA vs. STATE-OF-THE-ART Tools; True Positive Rate (i.e., detection rate; the higher the better) tested on the <i>Elysium</i> _{benchmark} [7], <i>Reentrancy</i> _{benchmark} [11], and <i>SolidiFI</i> _{benchmark} [13]; star ★ indicates the mean; plus + indicates outliers	113
3.8	DLVA vs. STATE-OF-THE-ART Tools; False Positive Rate (i.e., false alarm rate; the lower the better) tested on the <i>Elysium</i> _{benchmark} [7], <i>Reentrancy</i> _{benchmark} [11], and <i>SolidiFI</i> _{benchmark} [13]; star ★ indicates the mean; plus + indicates outliers	114
3.9	DLVA vs. STATE-OF-THE-ART Tools; Accuracy (the higher the better) tested on the <i>Elysium</i> _{benchmark} [7], <i>Reentrancy</i> _{benchmark} [11], and <i>SolidiFI</i> _{benchmark} [13]; star ★ indicates the mean; plus + indicates outliers	115
3.10	DLVA vs. STATE-OF-THE-ART Tools; Average analysis time per contract (the graph is in log scale, the lower the better) tested on the <i>Elysium</i> _{benchmark} [7], <i>Reentrancy</i> _{benchmark} [11], and <i>SolidiFI</i> _{benchmark} [13]; star ★ indicates the mean; plus + indicates outliers	116
3.11	SCoolS vs. STATE-OF-THE-ART Tools; Accuracy (the higher the better); F1 (i.e., the harmonic mean of precision and recall; the higher the better) False Positive Rate (FPR) (i.e., false alarm rate; the lower the better); Average analysis time per function (the lower the better); tested on the <i>ReentrancyBook</i> [12]	119

3.12	Comparison of reported tool performance by its authors versus independent benchmarking results tested on the <i>Elysium</i> _{benchmark} [7], <i>Reentrancy</i> _{benchmark} [11], and <i>SoliddiFI</i> _{benchmark} [13]	121
4.1	DLVA vs. alternatives on the <i>Elysium</i> _{benchmark} [7], <i>Reentrancy</i> _{benchmark} [11], and <i>SoliddiFI</i> _{benchmark} [13]; star ★ indicates the mean; plus + indicates outliers	129
4.2	The Deep Learning Vulnerability Analyzer (DLVA).	135
4.3	EthereumSC Data Set Histogram.	137
4.4	Vulnerability Frequencies of EthereumSC Dataset.	138
4.5	Sentence Embeddings using Universal Sentence Encoder based on Deep Averaging Network(DAN).	140
4.6	USE-generated vector embeddings	141
4.7	Evaluating SC2V vs. state-of-the-art GNNs	150
4.8	t-SNE-Embeddings for the “unchecked-lowlevel” Vulnerability.	151
4.9	DLVA-CC vs. ten “off-the-shelf” ML classifiers and a majority voting strategy (★ is the average; + are outliers)	153
4.10	Deep Learning Vulnerability Analysis Tool Score Summary for 29 Vulnerabilities of <i>EthereumSC</i> _{large} Dataset (The star symbol ★ represents the average, while the plus + represents outliers)	155
5.1	Fuzzy C-Means clustering algorithm for reentrancy.	181
5.2	Data Preprocessing and Graph Neural Networks (GNNs) Design.	184
5.3	Architecture of a Transformer with six encoder layers.	186
5.4	The Smart Contracts Semi-Supervised Learning (SCoolS).	187
5.5	Training Accuracy and Loss, and Validation Accuracy and Loss	195
A.1	Recurrent Neural Networks (RNNs).	225
A.2	Long Short Term Memory Networks (LSTMs).	226
A.3	Node Instructions Composition using Bidirectional Long Short-Term Memory.	228

List of Tables

3.1	Static and Dynamic Analysis for SC in primary studies.	50
3.2	Learning-based Techniques for SC in primary studies; “Available*” with the asterisk denoting potential availability of some studies upon request approval; “Source code*” with the asterisk denoting the analysis level for some studies requires source code and/or potential alternatives like transactions or account data.	65
3.3	Learning-based Techniques for PL in primary studies.	91
3.4	Data Sources and Benchmarks.	98
4.1	29 vulnerabilities in <i>EthereumSC_{large}</i> (200+ times); ★ indicates 21 vulnerabilities in <i>EthereumSC_{small}</i> (30+ times)	132
4.2	Datasets used for benchmarking DLVA	146
4.3	Best of the ten commonly used machine learning supervised binary classifiers results	152
4.4	DLVA Trained on <i>EthereumSC_{large}</i> (use DLVA’s core classifier only for the entire test set)	156
4.5	DLVA Trained on <i>EthereumSC_{large}</i> (Sibling Detector Results)	157
4.6	DLVA Trained on <i>EthereumSC_{large}</i> (Core Classifier Results)	158
4.7	DLVA Trained on <i>EthereumSC_{large}</i> (Results when SD and CC are working together)	159
4.8	DLVA Trained on <i>EthereumSC_{small}</i> (use DLVA’s core classifier for the entire test set)	160

4.9	Comparison of DLVA vs. state-of-the-art tools; Input: (S: Source code, B:Bytecode, S/B ⁻ : Source preferred, bytecode possible); Method: (SA/DA:Static/Dynamic Analysis, ML/DL:Machine/Deep Learning); Vul: # of vulnerability detectors; Year: year of release of the used version; Cits: number of citations from Google Scholar on 01/12/2023.	161
4.10	Small contracts, bytecode analyzers; Exp: Exceptions; Vulnerability: {RE:Reentrancy, PB:Parity Bug}; GP: Ground Positives; GN: Ground Negatives; FN: False Negatives; FP: False Positives; Σ F: Sum of Failures	165
4.11	Small contracts, source code analyzers; Exp: Exceptions; Vulnerability: {RE:Reentrancy, PB:Parity Bug}; GP: Ground Positives; GN: Ground Negatives; FN: False Negatives; FP: False Positives; Σ F: Sum of Failures	166
4.12	Large contracts; Exp: Exceptions; Vulnerability: (RE:Reentrancy, TS:Timestamp-Dependency, OU:Over/Underflow, TX:tx.origin); GP: Ground Positives; GN: Ground Negatives; FN: False Negatives; FP: False Positives; Σ F: Sum of Failures	167
5.1	The <i>BigBook</i> data set.	194
5.2	SCooLS vs. state-of-the-art tools.	196
5.3	Results on <i>ReentrancyTestBook</i> .	197
5.4	The Auto-Exploit Generator.	198

Chapter 1

Introduction

Blockchain is a decentralized and distributed digital ledger that records transactions in a secure and transparent manner. It is essentially a database that stores a continuously growing list of records called blocks, which are linked together and secured using cryptographic techniques. Each block contains a cryptographic hash of the previous block, a timestamp, and a set of transactions that have been verified and validated by network participants called nodes. The blockchain ledger is maintained and updated by a decentralized network of nodes, which work together to reach a consensus on the state of the ledger.

Cryptocurrency, on the other hand, is a digital or virtual currency that uses cryptography for security and operates independently of a central bank. Cryptocurrencies are based on blockchain technology and use a decentralized network of nodes to validate transactions and maintain the integrity of the ledger. Cryptocurrencies enable peer-to-peer transactions without the need for intermediaries, such as banks, and provide users with a high degree of privacy and anonymity. Bitcoin and Ethereum are two prominent cryptocurrencies that are based on blockchain technology but differ in their purpose, features, and use cases.

Bitcoin was the first cryptocurrency, created in 2009 by an unknown individual or group using the pseudonym Satoshi Nakamoto [102]. It was designed as a decentralized and trustless digital currency that operates without the need for intermediaries, such as banks. Bitcoin uses a proof-of-work consensus algorithm, where miners compete to solve complex mathematical puzzles to validate transactions and add blocks to the blockchain. Bitcoin transactions are irreversible and transparent, with all transactions recorded on the public blockchain. Bitcoin's main use case is as a digital currency for peer-to-peer transactions, allowing users to send and receive

payments globally without the need for traditional financial institutions. Bitcoin has gained popularity as a store of value and a hedge against inflation, with some investors considering it as a digital gold.

Ethereum, on the other hand, is a blockchain platform that goes beyond being just a cryptocurrency. It was proposed by Vitalik Buterin in 2013 and launched in 2015 [145]. Ethereum's primary purpose is to serve as a decentralized platform for building decentralized applications (dApps) and smart contracts (SC). Ethereum has its own cryptocurrency called Ether (ETH), which is used to pay for transaction fees and incentivize network participants. Ethereum introduced the concept of smart contracts, which are self-executing digital contracts that run on the Ethereum blockchain. Smart contracts allow developers to create decentralized applications, ranging from DeFi protocols and digital identity solutions to decentralized games and decentralized exchanges. One of the key features of Ethereum is its programmability, which enables developers to create their own tokens, implement custom logic, and build decentralized applications with their own rules and functionality. Ethereum also allows for more complex and flexible transactions, including multi-signature wallets and decentralized autonomous organizations (DAOs).

In 2016, the DAO was a decentralized investment fund built on the Ethereum blockchain, which raised over \$150 million worth of Ether in a crowdsale. However, the DAO's smart contract had a vulnerability that allowed an attacker to siphon off over \$50 million worth of Ether from the fund [95, 121]. As shown in Figure 1.1, the attack was a reentrancy attack, which involved a flaw in the smart contract's code that allowed the attacker to repeatedly withdraw funds from the DAO without the contract properly accounting for the withdrawals. The attacker's malicious contract was able to repeatedly call the "withdrawBalance" function in the DAO contract before it could complete its initial execution, allowing the attacker to siphon off Ether from the DAO contract. The attacker then moved the stolen Ether into a child DAO, effectively bypassing the DAO's intended security measures. After the attack was discovered, the Ethereum community proposed two options for addressing the issue: a soft fork to blacklist the attacker's address and prevent them from accessing the stolen Ether or a hard fork to roll back the entire Ethereum blockchain to the point before the attack occurred. Eventually, the hard fork was implemented, resulting in the creation of Ethereum and Ethereum Classic.

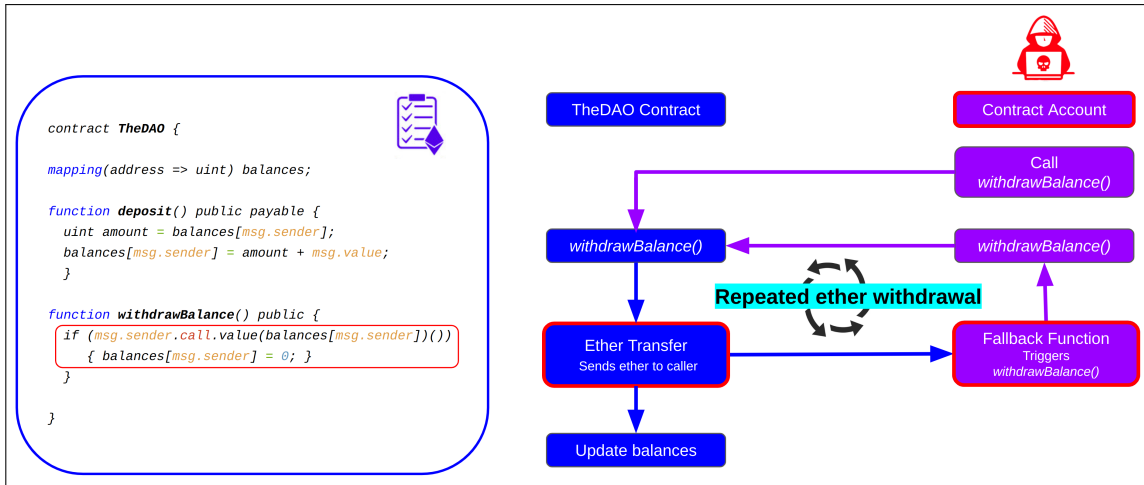


Figure 1.1: An illustration depicting the reentrancy attack on DAO.

In July 2017, a vulnerability was discovered in Parity’s multi-sig wallet, which allowed an attacker to steal funds from a number of deployed contracts. The vulnerability was traced back to an issue in the wallet’s smart contract code that enabled an attacker to trigger a function that changed the ownership of the wallet, without proper authorization. Specifically, the vulnerability was related to the way the wallet’s ownership structure was implemented. The wallet used a “library” contract to handle its logic, and the library contract was designed to be upgradeable by the wallet’s owner. However, the library contract was not properly protected against unauthorized changes, and the attacker was able to take advantage of this flaw to call a function that transferred ownership of the wallet to the attacker’s address. As a result, the attacker gained full control over the wallet and was able to drain the funds from the affected multi-sig wallets. The incident resulted in the loss of approximately 153,037 ETH, which was worth \$30 million of dollars at the time.

In November 2017, another vulnerability was discovered in Parity’s multi-sig wallet, which led to the freezing of approximately 513,774 ETH, worth \$150 million of dollars [129]. This vulnerability was caused by a similar issue related to the ownership structure and upgradeability of the wallet’s smart contract code. In this case, a user accidentally triggered a function in the wallet’s code that had been previously identified as vulnerable, resulting in a “freeze” of the affected multi-sig wallets. The function was intended to initialize the wallet’s internal state, but it

was not designed to handle re-initialization, and as a result, it locked the funds in the affected wallets, rendering them inaccessible. Despite efforts by the Parity team to address the issue and recover the frozen funds, it was ultimately not possible to unfreeze the wallets, and the funds remained locked.

The occurrence of smart contract attacks such as the DAO attack and the Parity wallet attack highlighted the criticality of proper smart contract development and program analysis in detecting vulnerabilities prior to deployment on the blockchain. These incidents exemplified the difficulties associated with addressing smart contract vulnerabilities and mitigating unauthorized access and potential financial loss. Therefore, it is essential to have robust security measures in place to ensure the integrity of smart contracts, as well as to provide timely responses to mitigate the impact of any potential attacks.

One of the foremost challenges in the detection of vulnerabilities within smart contracts is the limited availability of their source code. A mere 0.3% of the roughly 49 million smart contracts presently deployed on the Ethereum blockchain offer public access to their source code. As of March 2022, out of the 49,183,523 smart contracts recorded on the Ethereum blockchain [56], merely 152,996, as verified by the Smart Contract Sanctuary project ¹, have exposed their source code to the public. This indicates that a vast majority, approximately 99.7% of smart contracts, lack publicly accessible source code [48]. Moreover, the interconnected nature of smart contracts, where they can invoke functions in other smart contracts, implies that security vulnerabilities present in bytecode-only or “closed-source” contracts have the potential to impact even those with readily available source code.

1.1 Research Questions

The aforementioned security breaches involving smart contracts raise questions about their overall safety. While Ethereum’s powerful features enable cutting-edge applications, its use of Turing-complete languages for smart contract creation introduces potential vulnerabilities and casts doubt on the guaranteed security of blockchain technology. Despite the widespread belief that blockchain applications are highly secure and impregnable, the DAO and Parity wallet hacks illustrate how

¹<https://github.com/tintinweb/smart-contract-sanctuary-ethereum>

vulnerabilities in smart contracts can render them susceptible to attacks. While extensive efforts have focused on static analysis tools to find security flaws in smart contracts, relatively little research has explored leveraging deep learning techniques to build practical tools for assessing whether malicious actors could exploit real-world vulnerabilities in smart contracts. *Why learning-based approaches can significantly contribute to smart contract security and vulnerability detection:*

- 1) **Adaptability to Complex Contracts:** Modern smart contracts are highly expressive, meaning they can perform intricate tasks and calculations. Traditional security methods often struggle to keep up with this complexity. Learning-based approaches, however, can adapt to the evolving nature of smart contracts by learning from data and identifying patterns, allowing them to detect vulnerabilities in even the most complex contracts.
- 2) **Proven Track Record in Security:** Beyond smart contracts, learning-based approaches have already demonstrated success in securing other areas of computing [16]. For example, they've been effective in detecting malware, preventing software vulnerabilities, and protecting networks from intrusion. This proven track record in diverse security domains suggests that they can be similarly effective in securing smart contracts.

Challenges: While state-of-the-art approaches have made significant progress in identifying vulnerabilities and potential exploits in smart contracts, they still face several key challenges:

- 1) **Source Code Scarcity:** Most existing approaches rely heavily on analyzing the source code of smart contracts. However, only a tiny fraction (less than 1%) of deployed contracts have their source code publicly available. In contrast, all contracts have their bytecode readily accessible. This poses a key research question:

RQ1: How can we leverage deep learning to identify potential vulnerabilities in smart contract bytecode, even without access to the source code?

- 2) **Maintainability:** Static analysis approaches rely on predefined rules to detect vulnerabilities, are often limited in their effectiveness. The accuracy of these tools depends heavily on the completeness and quality of their rule sets. Outdated or incomplete rules can lead to missed vulnerabilities, creating a constant need for manual updates to keep up with evolving threats and attack vectors. This raises another research question:

RQ2: Can we develop deep learning techniques that effectively learn the characteristics of vulnerable bytecodes, without relying on expert rules or predefined patterns?

- 3) **Scalability:** Many static analysis approaches use symbolic execution that explores multiple paths through a program, resulting in “path explosion problem”, especially in complex programs with loops, recursive functions, or large inputs. This can lead to scalability issues, significantly increasing computational resources and analysis time. Therefore, a crucial research question emerges:

RQ3: Can we develop deep learning techniques to overcome the scalability challenge and efficiently analyze complex smart contracts?

- 4) **Lack of labels:** A major drawback of supervised learning-based approach lies in its reliance on vast amounts of labeled training data. Acquiring such data can be a costly and time-consuming endeavor, especially for niche areas like smart contract vulnerabilities. Furthermore, labeling data can be subjective and prone to errors, potentially leading to biased models with compromised accuracy. This poses a critical research question:

RQ4: How can we effectively train deep learning models for smart contract vulnerability detection when labeled data is scarce, expensive, or difficult to obtain?

- 5) **Localization and Auto-Exploit Generation:** Many learning-based tools for smart contract vulnerability detection fall short in providing specific and actionable insights: a) They often classify contracts as simply “vulnerable” or “non-vulnerable” without pinpointing the exact vulnerability type or its location within the bytecode. b) They fail to demonstrate how an attacker

could exploit the vulnerabilities, limiting developers’ ability to understand and address potential threats. This highlights a crucial research question:

RQ5: How can we develop mechanisms to confirm detected vulnerabilities using exploit generation techniques, effectively demonstrating to developers the potential actions of attackers against their smart contracts?

These questions are designed to guide our investigation of detecting security threats in smart contracts using deep learning-based approaches.

1.2 Contributions

Through our study, we have contributed to the field of blockchain security by exploring the effectiveness of using deep learning to enhance the security of smart contracts. The main contributions of this thesis are summarized as follows:

- 1) To address **RQ1**, **RQ2**, and **RQ3** — We design and develop Deep Learning Vulnerability Analyzer (DLVA) [3, 4, 5] — a vulnerability detection tool for Ethereum smart contracts based on powerful deep learning techniques adapted for smart contract bytecode.
 - We train DLVA to judge bytecode *even though the supervising oracle, Slither, can only judge source code*. DLVA’s training algorithm is general: we “extend” a source code analysis to bytecode without any manual feature engineering, predefined patterns, or expert rules.
 - We develop a Smart Contract to Vector (SC2V) engine that maps smart contract bytecode into a high-dimensional floating-point vector space. SC2V uses a mix of neural nets trained in both unsupervised and supervised manners. We use Slither for supervision, labeling each contract as vulnerable or non-vulnerable for each of the 29 vulnerabilities we handle. *We provide no expert rules or other “hints” during training*. We evaluate the SC2V engine against four state-of-the-art graph neural networks and show it is 2.2% better than the average competitor and 1.2% better than the best.

- Our Sibling Detector (SD) classifies contracts according to the labels of other contracts Euclidian-nearby in the vector space. Our SD is highly accurate, showing the quality of SC2V: on the 55.7% of contracts in our test set that it can judge, it has an accuracy (to Slither) of 97.4% and an associated FPR of only 0.1%.
 - We design the Core Classifier (CC) of DLVA using additional neural networks, trained in a supervised manner using the same labeled dataset as SC2V. On the “harder” 44.3% of our test set, the CC has an accuracy (to Slither) of 80.0% with an associated FPR of 21.4%. The CC is solving a harder problem than the SD, at it must classify contracts that are quite different from those seen during training. We evaluate the CC against ten off-the-shelf machine learning methods and show that it beats the average competitor by 11.3% and the best by 8.4%.
 - DLVA is the combination of SC2V, SD, and CC. This whole is greater than its parts: DLVA judges every contract in the test set, with an average accuracy (to Slither) of 87.7% and FPR of 12.0%.
 - Small contracts are simpler than larger ones. We tweak our design to better handle such contracts and retrain. On small contracts, DLVA has an average accuracy (to Slither) of 97.6% with a FPR of 2.3%.
 - Accordingly, DLVA’s overall accuracy (average of large and small) is 92.7% with a FPR of 7.2%.
 - We propose and evaluate six datasets to benchmark DLVA and its components. We benchmark DLVA against eight static analyzers and three learning-based analyzers.
 - DLVA is much faster than conventional tools for smart contract vulnerability detection based on formal methods: DLVA checks contracts for 29 vulnerabilities in 0.2 seconds, a speedup of 5-1,000x+ compared to traditional tools that do not scale nearly as well as program complexity and length grows.
- 2) To address **RQ4**, and **RQ5** — We design and implement Smart Contract Learning (Semi-supervised) SCoolS [2].

- SCooLS takes smart contract vulnerability analysis to a new level with its groundbreaking application of semi-supervised learning. It's the first tool to not only pinpoint specific vulnerable functions in bytecode but also generate real-world attack demonstrations for developers and users. This revolutionizes vulnerability detection by moving beyond simply labeling the entire contract as vulnerable. Instead, SCooLS provides developers with concrete ways to test and understand the exploitability of their contracts, empowering them to take proactive security measures.
 - Semi-supervised learning method that aims to address the challenge of limited availability of high-confidence labeled code data in practical smart contract vulnerability classification tasks.
 - In total we train 120 distinct models derived from applying a variety of hyperparameters to five state-of-the-art graph neural networks, using a voting system to smooth out the variance in individual models during training.
 - We measure the performance of SCooLS and compare it with three state-of-the-art tools. SCooLS dominates the competition, obtaining a higher accuracy level of 98.4%, a higher F1 score of 90.5%, and the lowest false positive rate of just 0.8%. Moreover, the analysis is fast, requiring only 0.05 seconds per function.
 - SCooLS implements an auto-exploit generator to prove that the detected vulnerabilities can be exploited by attackers to steal contract funds. The exploit generator was able to attack 76.9% of the true positive instances for which an ABI was available.
- 3) DLVA, SCooLS, data sets, and benchmarks are publicly available to enable contract developers to assess the security of their smart contracts prior to deployment on the blockchain.

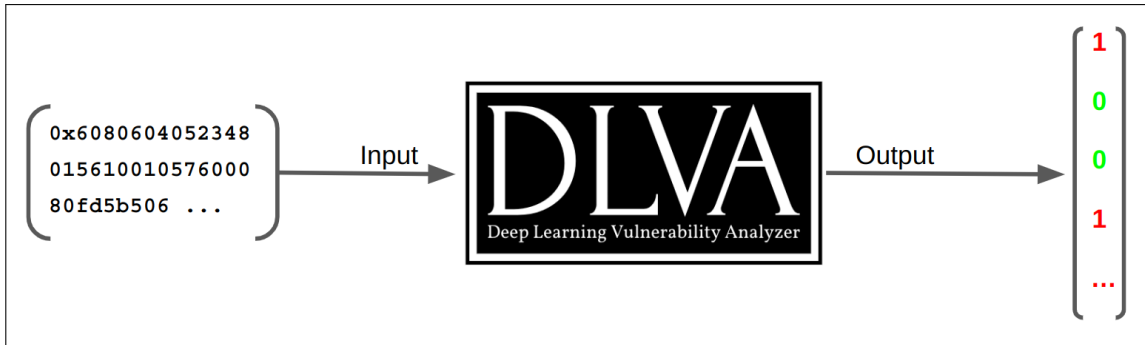


Figure 1.2: DLVA Specification.

1.3 Thesis Statement

Deep learning enables rapid and efficient detection of vulnerabilities in smart contracts, along with accurate exploit demonstrations. This leads to significantly improved accuracy and reduced false positives, paving the way for practical and reliable security models.

1.4 Tools Specification

DLVA Specification Figure 1.2 illustrates the input and output of DLVA:

- **Input:** Let C be a smart contract (in bytecode).
- **Output:** A 29-dimensional binary vector corresponding to the 29 distinct vulnerabilities. Each element in the vector indicates:
 - **1:** Contract C is vulnerable to the corresponding vulnerability.
 - **0:** Contract C is secure from the corresponding vulnerability.

SCoolS Specification Figure 1.3 illustrates the input and output of SCoolS:

- **Input:** Let C be a smart contract (in bytecode).
- **Output:** N -dimensional binary vector, where N is the number of functions in the contract:

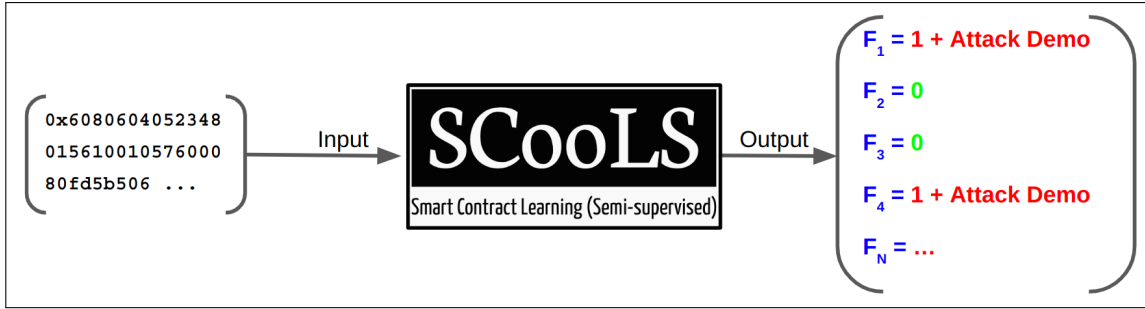


Figure 1.3: SCooLS Specification.

- **1**: Function F_i in C is exploitable for reentrancy, with attack demonstration.
- **0**: Function F_i in C is secure from reentrancy.

1.5 Thesis Organization

This thesis dives into the world of Ethereum smart contracts and their security flaws by leveraging the power of deep learning. Here is a roadmap to guide you through the journey:

Chapter 2: Dive into the realm of Ethereum smart contracts – what they are, how they work, and the vulnerabilities lurking within. We will also equip you with an understanding of deep learning techniques, so you are ready to see how they combat these security threats.

Chapter 3: Comprehensive survey of existing solutions – what others have done, how effective they were, and where they stumbled. This lays the groundwork for our own innovative approach.

Chapter 4: Introducing DLVA, our deep learning vulnerability analyzer! We will break down its design, show you how it works, and present the results of its real-world testing. Buckle up for some exciting experiments!

Chapter 5: What happens when data for training our defender is scarce or costly? SCooLS to the rescue! We will build a semi-supervised technique for exploits detection and present an auto-exploit generator to confirm the detected vulnerabilities are actually real threats.

CHAPTER 1. INTRODUCTION

Chapter 6: We will summarize our findings, offer some concluding thoughts, and even peek into the future, showcasing potential directions for further research in this ever-evolving field.

Chapter 2

Background

The purpose of this chapter is to provide a fundamental understanding of the concepts necessary for comprehending the research presented in this thesis. To achieve this goal, an overview of Ethereum and smart contracts is presented, including a detailed explanation of their technical aspects, as well as a discussion of the common vulnerabilities associated with smart contracts. Furthermore, the chapter provides a contextual overview of deep learning techniques, thus enabling readers to better understand the research that will be presented in subsequent chapters.

2.1 Ethereum

Ethereum is a blockchain-based decentralized platform for developing and executing smart contracts and decentralized applications (DApps), introduced in 2013 [145], the network was launched on July 30, 2015. Like Bitcoin, Ethereum is a distributed ledger technology that utilizes cryptography to maintain a secure and transparent record of transactions. At its core, the Ethereum blockchain is a decentralized database that consists of a network of interconnected computers, each of which contains a copy of the same data. The data is stored in blocks, and each block is linked to the previous block in a chain-like structure, hence the name “blockchain.”

One of the key differences between Ethereum and Bitcoin is that Ethereum was designed to be a platform for building decentralized applications, while Bitcoin was primarily designed to serve as a digital currency. Ethereum’s smart contract functionality allows developers to create self-executing contracts that can be used to automate a wide variety of processes and transactions.

In addition to smart contracts, Ethereum also has its own cryptocurrency, called Ether (ETH), which serving not only as a means of exchange but also as an economic incentive for users to contribute computational resources to the network. The smallest unit of ether is known as wei, which is equivalent to 10^{-18} ether.

On September 15, 2022, Ethereum successfully completed a significant upgrade process known as “the Merge,” transitioning its consensus mechanism from proof-of-work (PoW) to proof-of-stake (PoS). This transition led to a remarkable reduction in Ethereum’s energy consumption by approximately 99.95%. Ethereum supports two types of accounts: Externally Owned Accounts (EOAs), which are controlled by private keys and have no associated code, and Contract Accounts, which are controlled by smart contracts and have code associated with them [25].

Overall, the Ethereum blockchain provides a secure and decentralized platform for developers to build and deploy decentralized applications, while also providing a cryptocurrency that can be used to power these applications and facilitate transactions on the network.

2.2 Smart Contracts

A smart contract on the Ethereum blockchain is simply a program has a set of functions paired with some associated data, located at a specific address in the Ethereum blockchain. Smart contracts 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. User accounts can then interact with a smart contract by submitting transactions that execute a function defined on the smart contract. They are frequently used to automate the execution of an agreement without the involvement of a third party (i.e., bank or government). Contracts used in financial applications, for example, are typically used to manage, collect, or distribute an asset. Furthermore, its immutability makes it an ideal choice for storing important data (e.g., ownership, provenance) for notary purposes. Smart contracts can define rules, like a regular contract, and automatically enforce them via the code. Smart contracts cannot be deleted by default, and interactions with them are irreversible. Most Ethereum smart contracts are written in a high-level language such as Solidity.

Solidity is an object-oriented, high-level language especially developed for contract writing, and it is currently the most prominent programming language for developing smart contracts in Ethereum. The Solidity compiler compiles the source code into bytecode. Smart contracts on the Ethereum blockchain are stored as bytecode, a machine-readable instruction set understood by the Ethereum Virtual Machine (EVM). This means the original code written by the developer (source code) is not directly visible. Users typically interact with smart contracts through their public bytecode, which can be difficult to understand without the source code.

In Solidity smart contracts, the *fallback function* serves a crucial role in exception handling and unexpected interactions, it is a special function that automatically executes under two specific conditions:

- Non-existent function call: When a transaction attempts to call a function that does not exist within the contract, instead of reverting the transaction with an error, the fallback function takes over. This allows for graceful handling of invalid function calls, potentially logging the event, sending a message, or redirecting the transaction to a suitable function.
- Direct Ether transfer: If a transaction sends Ether directly to the contract without specifying a specific function, the fallback function acts as the entry point. This enables the contract to receive and process the payment, potentially updating internal state or performing other actions based on the received Ether.

2.3 Ethereum Virtual Machine

The Ethereum Virtual Machine or EVM is the runtime environment for smart contracts in Ethereum that runs low-level bytecode and supports a Turing-complete set of instructions. As shown in Figure 2.1, the EVM follows the Harvard architecture by separating code and data into four parts: (a) an immutable EVM code, which contains the smart contract's bytecode, (b) a mutable but persistent storage to persist smart contract data across executions, (c) a mutable but volatile memory that acts as a temporary data storage during execution, and finally (d) a stack that allows smart contracts to pass arguments to instructions at runtime. Since the smart

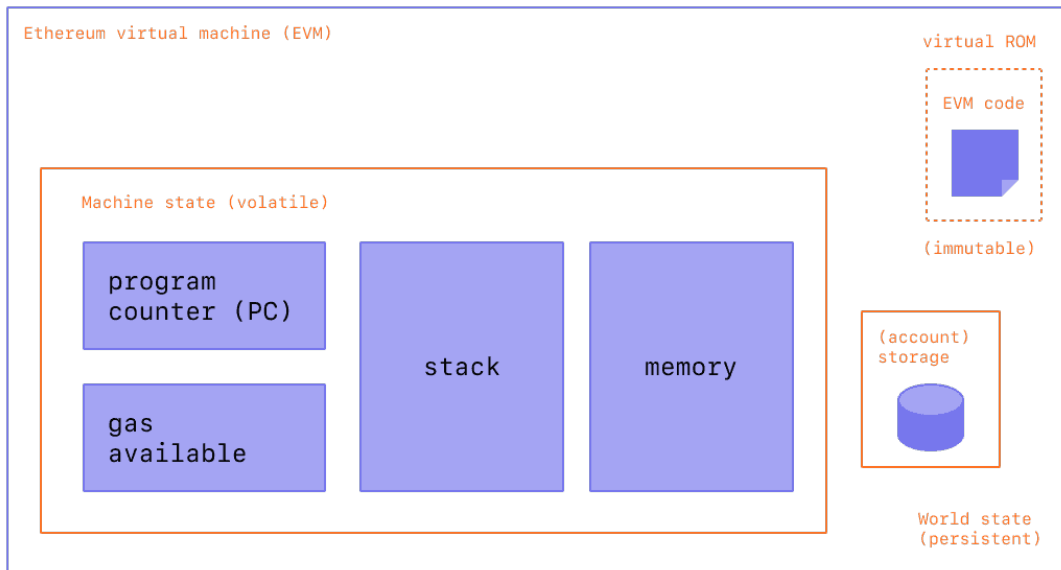


Figure 2.1: Ethereum Virtual Machine (EVM), adapted from ethereum.org

contract bytecode is stored on the blockchain, making it accessible to both regular users and attackers.

EVM *bytecode* is represented by a very long hexadecimal number such as `0x6080604052348...` EVM bytecode consists of two main parts: deployment bytecode and runtime bytecode. Deployment bytecode is the binary code that will be sent to the network for the creation of the smart contract, initializing state variables, and reading constructor arguments appended at the end of the Ethereum bytecode. The runtime bytecode is the binary code that will be stored on the blockchain, contains the runtime logic (i.e., runtime bytecode), and executed whenever the contract is invoked by transaction.

There is a simple injective relationship between valid hexadecimal *bytecode* sequences and a list of valid human-readable *opcodes* such as “PUSH1” (encoded as `0x60`), “MSTORE” (`0x52`), and so forth. Our approach takes these hexadecimal bytecode sequences as input and disassembles them into opcode sequences. Ethereum’s “Yellow Paper” defines the EVM as a variant of a stack machine with 150 distinct opcodes [145]. In Figures 2.2, 2.3, and 2.4, we give examples of these various representations of programs for reference.


```

1  pragma solidity ^0.4.22;
2  contract Bank {
3      mapping(address => uint) private balances;
4      function withdraw() public {
5          uint amount = balances[msg.sender];
6          msg.sender.call{value: amount}("");
7          balances[msg.sender] = 0;
8      }
9  }

```

Figure 2.2: Sample representation of Solidity source code

```

1  608060405234801561001057600080fd5b50610160806100206000396000f3fe60806040523480156
2  1001057600080fd5b506004361061002b5760003560e01c80633ccfd60b14610030575b600080fd5b
3  61003861003a565b005b60008060003373ffffffffffffffffffffffffffffffffffffffff1673fff
4  ffffffffffffffffffffffffffffffffffffffffff1681526020019081526020016000205490503373fff
5  ffffffffffffffffffffffffffffffffff168160405180600001905060006040518083038185875
6  af1925050503d80600081146100db576040519150601f19603f3d011682016040523d82523d600060
7  2084013e6100e0565b606091505b50505060008060003373fffffffffffffffffffffffffffff
8  fffffff1673fffffffffffffffffffffffffffffffff168152602001908152602001600020
9  819055505056fea2646970667358221220983da010e57932d9b85216b1c0b75842ec7e5b30f3cf702
10 fc9c43d8c2d6d3cc64736f6c63430007000033

```

Figure 2.3: Sample representation of EVM bytecode

```

1  PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI PUSH1 0x0
2  DUP1 REVERT JUMPDEST POP PUSH2 0x160 DUP1 PUSH2 0x20 PUSH1 0x0 CODECOPY PUSH1
3  0x0 RETURN INVALID PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2
4  0x10 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0x4 CALLDATASIZE LT PUSH2
5  0x2B JUMPI PUSH1 0x0 CALLDATALOAD PUSH1 0xE0 SHR DUP1 PUSH4 0x3CCFD60B EQ PUSH2
6  ...
7  POP JUMP INVALID LOG2 PUSH5 0x6970667358 0x22 SLT KECCAK256 SWAP9 RETURNDATASIZE
8  LOG0 LT 0xE5 PUSH26 0x32D9B85216B1C0B75842EC7E5B30F3CF702FCF9C43D8C2D6D3CC
9  PUSH5 0x736F6C6343 STOP SMOD STOP STOP CALLER

```

Figure 2.4: Sample representation of EVM opcodes

The aforementioned representations of smart contract as source code, bytecode, and opcode are text and we must convert smart contract to numerical feature vectors because sophisticated machine learning models typically work with numerical feature vectors rather than text. The proposed learning methods in this thesis automatically learns vulnerability features of smart contract from the control flow graph extracted from the runtime bytecode.

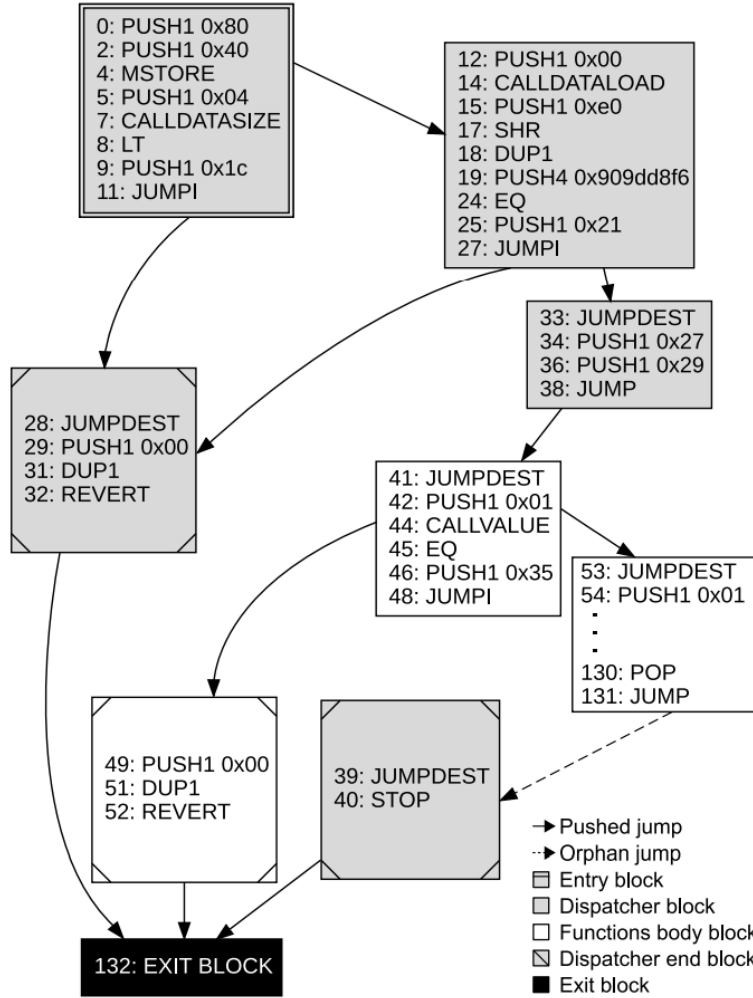


Figure 2.5: Smart Contract Control Flow Graph (adapted from [37]).

In program analysis, a *control-flow graph* (CFG) is a graphical representation of the flow of control within a program. It is a directed graph that represents the order in which the instructions of a program are executed. The nodes in the graph represent *basic blocks*, which are sequences of instructions that have no branching or looping. The edges between the nodes represent the control flow between the basic blocks, which can be conditional or unconditional, as shown in Figure 2.5.

The control-flow graph can be used to analyze and understand the behavior of a program. For example, it can be used to identify potential bugs or vulnerabilities in the code, to optimize the performance of the program, or to verify that the program meets certain security or safety requirements.

To construct a control-flow graph, the program is first divided into basic blocks. Each basic block starts with the first instruction in the program or the target of a branch or jump instruction, and ends with the last instruction before the next branch or jump instruction. The basic blocks are then connected by edges that represent the control flow between them.

The control-flow graph can be analyzed using various techniques, such as data flow analysis, symbolic execution, or graph neural networks. These techniques use the graph to reason about the behavior of the program and to identify possible errors or vulnerabilities. They can also be used to generate test cases or to optimize the performance of the program. CFG is more useful for analysis than linear representations of the code because it captures important semantic structures within the contract.

2.4 Smart Contract Vulnerabilities

Smart contract vulnerabilities refer to security weaknesses or flaws that exist in the runtime bytecode of a smart contract. Smart contracts are self-executing programs that run on a blockchain network and are designed to automate the execution of a contract agreement between two or more parties. After deployment on the chain, the smart contract becomes immutable, and any vulnerabilities cannot be directly patched. Due to their decentralized nature and the fact that smart contracts are executed automatically, any vulnerabilities or bugs in their code can have severe consequences, including financial losses or the compromise of sensitive information.

In the past years, many attacks on deployed smart contracts have been reported [15]. This includes many well-known vulnerabilities including “reentrancy-eth” (the DAO hack, USD 50 million in losses [95, 121]) and “suicidal” (the Parity bug, USD 280 million in losses [129]). Some well-known vulnerabilities are tagged with Smart contract Weakness Classification (SWC) numbers for ease of reference [111].

Finding vulnerabilities in smart contracts is crucial to ensuring their secure execution. However, *there is currently no standardized definition or description of these vulnerabilities*. To address this, we rely on the vulnerability descriptions provided by Slither (version 0.8.0, build date: May 7, 2021) [45]. Slither is a widely

used and accurate tool that can identify a diverse range of vulnerabilities (74 types!) from source code.

However, only 29 of these vulnerability types have enough examples (at least 200 positive samples) for training our deep learning models. We focus on these 29 types due to the limited data for the remaining ones. This section provides detailed explanations of each of the 29 vulnerabilities our models target:

- 1) ★ The vulnerability of **shadowing-state** (SWC-119) in smart contracts is a potential security vulnerability that arises due to the ambiguous naming of state variables when inheritance is used. In this scenario, a contract may inherit from another contract that has a state variable with the same name as the one defined in the inheriting contract. This can lead to the creation of two distinct versions of the variable, one accessible from the inheriting contract and the other from the inherited contract. In more complex contract systems, this ambiguity can easily go unnoticed and result in unintended consequences, potentially leading to security breaches. Shadowing state variables can also occur within a single contract, where multiple definitions exist at the contract and function level, further exacerbating the risk of unintended behavior.

To mitigate this vulnerability, Solidity developers must be mindful of naming conventions and ensure that variables are named uniquely across all contracts and functions.

- 2) ★ The **suicidal** vulnerability (SWC-106) in smart contracts occurs when the **SELFDESTRUCT** instruction is triggered by an arbitrary account, causing the contract to be destroyed. This vulnerability can arise when a contract includes a security fallback option that allows it to be destroyed by its owner or trusted addresses in the event of an emergency, such as a security breach or malfunction. However, if this functionality is not properly secured and can be triggered by any arbitrary account, it renders the contract susceptible and potentially suicidal. Once the **SELFDESTRUCT** instruction is executed, all remaining Ether in the contract is sent to a designated address and the contract is permanently destroyed, leading to loss of data and potentially disrupting the blockchain network.

To mitigate this vulnerability, developers must ensure that the **SELFDESTRUCT** instruction can only be triggered by authorized accounts, typically the contract owner or a set of trusted addresses. Because failure to address this vulnerability can have serious consequences, potentially leading to significant financial losses and reputational damage.

- 3) ★ The **uninitialized-state** vulnerability (SWC-109) in smart contracts arises when local storage variables are not initialized properly, resulting in unexpected storage locations within the contract. This can create intentional or unintentional vulnerabilities that can be exploited by attackers to compromise the security and integrity of the contract. Uninitialized storage variables can cause unexpected behavior when they are accessed or modified by other parts of the contract. In some cases, uninitialized variables may point to storage locations that contain sensitive data or important functionality, allowing attackers to manipulate or corrupt the contract’s state.

To mitigate this vulnerability, developers must ensure that all local storage variables are properly initialized before they are used in the contract.

- 4) ★ The **arbitrary-send** vulnerability in smart contracts refers to the potential for malicious actors to exploit unprotected calls to functions that send Ether to arbitrary addresses. This vulnerability arises due to missing or insufficient access controls, allowing malicious parties to withdraw some or all Ether from the contract account. The arbitrary-send vulnerability can be particularly damaging as it can result in significant financial losses. Malicious actors can exploit this vulnerability by using unprotected function calls to send Ether to their own accounts, effectively stealing funds from the contract. In some cases, attackers may also be able to compromise the contract’s security and execute further attacks on the network.

To mitigate this vulnerability, developers must implement robust access controls and ensure that functions that send Ether are protected against unauthorized access. This can be achieved by implementing proper authentication mechanisms, such as requiring specific authorization or permissions to access the function.

- 5) ★ The **controlled-array-length** vulnerability pertains to functions within a smart contract that enable direct assignment of an array's length using a variable that is controlled by a user, which could potentially be manipulated by an attacker. This vulnerability arises when there are no proper checks or validations on the user-controlled variable, allowing malicious actors to tamper with the length of the array in an unauthorized manner. Such unauthorized manipulation of array length can result in unexpected behavior, data corruption, or other security risks within the smart contract. It is crucial for smart contract developers to implement robust input validation and access control mechanisms to safeguard against this vulnerability and prevent potential attacks.
- 6) ★ The **controlled-delegatecall** (SWC-112) vulnerability relates to a specific type of message call called `delegatecall`, which is similar to a regular message call except that the code executed at the target address runs within the context of the calling contract, with the values of `msg.sender` and `msg.value` remaining unchanged. This functionality allows a smart contract to dynamically load code from a different address during execution. However, calling into untrusted contracts using `delegatecall` is highly risky because the code at the target address is granted complete access to the caller's balance and can modify any storage values within the caller's contract. This presents significant security risks, as the untrusted contract could potentially manipulate or steal the caller's funds, or compromise the integrity of the calling contract's storage data. It is critical for smart contract developers to exercise caution when utilizing `delegatecall` and thoroughly audit any external code that will be executed within the context of their contract.
- 7) ★ The **reentrancy-eth** (SWC-107) vulnerability is associated with the risk of calling external contracts, as they can potentially manipulate the control flow. In a reentrancy attack, also known as a DAO or recursive call attack, a malicious contract invokes a function within the calling contract before the first invocation of the function has completed, leading to unintended interactions among different invocations of the function. This type of vulnerability can result in undesirable behavior, including the potential theft of Ether, as the malicious contract may repeatedly call back into the calling contract, exploiting

the reentrancy vulnerability to drain Ether from the target contract. Proper validation of external contracts, appropriate use of locks or mutexes, and careful management of control flow can help mitigate the risks associated with reentrancy attacks and prevent unauthorized access to Ether or other critical resources within a smart contract system.

- 8) ★ The **unchecked-transfer** vulnerability refers to a scenario where the return value of an external `transfer/transferFrom` call is not properly validated within a smart contract system. Specifically, the return value of a message call is not checked, and execution continues even if the called contract throws an exception. This type of vulnerability can result in unexpected behavior in the subsequent program logic, particularly if the call fails due to accidental error or is manipulated by an attacker. In such cases, the program may proceed as if the call succeeded, leading to unintended consequences and potential security breaches. To mitigate the risks associated with unchecked transfers, developers should carefully validate the return values of message calls, implement robust error handling mechanisms.
- 9) ★ The **erc20-interface** vulnerability pertains to incorrect return values for ERC20 functions within a smart contract system. Specifically, when a contract compiled with Solidity version greater than 0.4.22 interacts with these functions, the execution may fail as the expected return values are missing. To address this vulnerability, it is crucial to ensure that the appropriate return values and types are correctly defined for the ERC20 functions. This includes thorough validation of the return values and types during the development, testing, and deployment phases of smart contract development. By adhering to proper return value conventions, developers can prevent potential execution failures and ensure the reliable functionality of ERC20 token contracts within the Ethereum ecosystem.
- 10) ★ The **incorrect-equality** vulnerability, as described in (SWC-132), relates to the improper use of strict equality in smart contract systems. When contracts assume a specific Ether balance based on strict equality comparisons, erroneous behavior may result. To mitigate this vulnerability, it is recommended that

strict equality should not be used to determine if an account has sufficient Ether or tokens. Instead, contracts should use safe comparison operations that take into account possible rounding errors and other unpredictable factors that may affect the accuracy of balance calculations.

- 11) ★ The **locked-ether** is a vulnerability that affects contracts with a payable function, but without a corresponding withdrawal mechanism. Such contracts can accept Ether but are unable to send it out, either because they lack instructions that send Ether out or because these instructions are not reachable. As a result, Ether sent to these contracts can become permanently locked, leading to financial loss for the contract owner and users. The recommended solution is to either remove the payable attribute or add a withdraw function to enable the contract to send Ether out when necessary.
- 12) The **mapping-deletion** vulnerability refers to a situation where a contract contains a structure with a mapping, and the contract tries to delete the structure. However, deleting the structure does not delete the mapping, and the data in the mapping can still be accessed, potentially allowing an attacker to breach the contract. To mitigate this vulnerability, it is recommended to use a lock mechanism instead of deleting the structure. This involves adding a boolean variable to the structure that can be set to true to disable the structure, preventing any further use of the mapping data. It is also important to ensure that all references to the deleted structure are removed from the contract code to prevent any unintentional use.
- 13) The **shadowing-abstract** is the shadowing state variables in abstract contracts can lead to confusion and unintended behavior, as the derived contract may not behave as intended. Therefore, it is recommended to remove any state variable shadowing in abstract contracts.
- 14) The **tautology** is tautological expressions or contradictions in smart contract code can lead to unexpected behavior and security vulnerabilities. A tautology is a logical statement that is always true, while a contradiction is a logical statement that is always false. In the context of smart contracts, such expressions can lead to erroneous results, causing the contract to behave in unintended

ways. To avoid such issues, it is recommended to carefully review the code and identify any instances of tautological expressions or contradictions. Once identified, the appropriate corrective action can be taken to fix the incorrect comparison by changing the value type or the comparison itself.

- 15) The **write-after-write** vulnerability in smart contracts refers to variables that are written but never read, resulting in unnecessary or redundant writes. This can occur when a variable is first assigned to one value and then immediately reassigned to another value, effectively negating the effect of the first write operation. To address this issue, it is recommended to review the code and identify any instances where variables are written multiple times without being read or where subsequent writes override previous writes without any meaningful purpose. Such redundant writes can be fixed or removed to optimize the contract's efficiency and reduce unnecessary gas costs. This may involve reevaluating the logic of the contract and ensuring that variable writes are performed only when necessary and with a clear purpose, avoiding unnecessary or redundant operations.
- 16) ★ In the **constant-function-asm** vulnerability, **constant/pure/view** functions are intended to be read-only functions that do not modify state. However, if a function is declared as **constant/pure/view** using assembly code, it may not be correctly labeled, which can cause issues when called in Solidity 0.5.0 or later. In particular, a call to a function declared as **constant/pure/view** uses the **STATICCALL** opcode, which reverts in case of state modification. This means that a call to an incorrectly labeled function may trap a contract compiled with Solidity 0.5.0 or later. To avoid such issues, it is recommended to ensure that the attributes of contracts compiled prior to Solidity 0.5.0 are correct. In particular, functions should be properly labeled as **constant/pure/view** if they are intended to be read-only functions that do not modify state. This can help prevent unexpected behavior and ensure the correct functioning of the contract.
- 17) The **constant-function-state** is vulnerability with functions declared as **constant/pure/view** is that they are not supposed to change the state, but

in some cases they do. This was not strictly enforced prior to Solidity 0.5.0. Starting from Solidity 0.5.0, a call to a **constant/pure/view** function uses the **STATICCALL** opcode, which reverts in case of state modification. As a result, a call to an incorrectly labeled function may trap a contract compiled with Solidity 0.5.0 or later.

- 18) ★ The **divide-before-multiply** vulnerability where solidity integer division can truncate the result. Therefore, performing multiplication before division can avoid loss of precision that might occur due to integer division.
- 19) ★ The **reentrancy-no-eth** vulnerability refers to a scenario where a contract function can be called recursively before the initial invocation of the function completes, potentially leading to unexpected or malicious behavior. This vulnerability is not specific to Ether transactions and can also occur with other types of function calls.

To address this vulnerability, the check-effects-interactions pattern can be applied. This pattern involves first performing all state changes and calculations before interacting with external contracts or making additional function calls. By following this pattern, the risk of reentrancy attacks can be mitigated. Therefore, it is recommended to carefully analyze the code and apply the check-effects-interactions pattern to prevent reentrancy attacks that don't involve Ether.

- 20) The **tx-origin** vulnerability refers to the use of **tx.origin** for authorization, which can be exploited by a malicious contract if a legitimate user interacts with the malicious contract. The **tx.origin** variable contains the address that originated the transaction, which can be different from the address that actually sent the transaction. This means that a contract relying on **tx.origin** for authorization can be tricked by a malicious contract to perform unauthorized actions on behalf of the user.

To mitigate this vulnerability, it is recommended to avoid using **tx.origin** for authorization. Instead, contracts should use **msg.sender** to determine the address that sent the current message. This ensures that only the intended

user can perform the authorized actions, regardless of whether the transaction was initiated by a contract or an external account.

- 21) ★ The **unchecked-lowlevel** vulnerability can pose a security risk, as the return value of such calls is not checked, which may result in the locking of Ether in the contract. This issue can be particularly problematic when low-level calls are used to prevent blocking operations. To mitigate this risk, it is recommended that developers ensure that the return value of low-level calls is either checked or logged. By doing so, failed calls can be identified and appropriate measures can be taken to prevent the locking of Ether in the contract.
- 22) ★ The **unchecked-send** (SWC-105) vulnerability: The return value of the `send()` function in Solidity is not checked, which can lead to a situation where the Ether sent is locked in the contract. This can happen when the recipient of the sent Ether is a contract that has a fallback function which either reverts or runs out of gas. It is important to note that the `send()` function should not be used for critical operations. If it is used to prevent blocking operations, then it is recommended to log the failed send transactions to detect potential issues. Therefore, it is crucial to ensure that the return value of the `send()` function is checked or logged to prevent Ether from getting locked in the contract.
- 23) ★ The **uninitialized-local** (SWC-109) vulnerability: Uninitialized local storage variables in Solidity can lead to unexpected behavior, as they may point to unexpected storage locations in the contract. This can result in unintended consequences and vulnerabilities in the contract logic. To mitigate this risk, it is recommended to initialize all variables properly before using them. If a variable is intended to be initialized to zero, it is important to explicitly set it to zero during declaration or assignment, as it improves code readability and reduces the risk of uninitialized variables pointing to unexpected storage locations.
- 24) ★ The **unused-return** (SWC-104) vulnerability: In Solidity, when the return value of an external call is not stored in a local or state variable, it can result in unintended consequences or vulnerabilities in the contract logic. The return

values of function calls can contain important data that may be necessary for further processing or decision-making within the contract. To prevent this issue, it is recommended to ensure that all return values of function calls are properly captured and used in the contract logic. This can be achieved by storing the return values in local or state variables, which can then be utilized as needed in the contract's logic or further interactions with other contracts.

- 25) ★ The **incorrect-modifier** vulnerability: In Solidity, if a modifier does not execute `_` or `revert`, the execution of the function will return the default value, which can be misleading for the caller. Therefore, it is important to ensure that all paths in a modifier execute either `_` or `revert`. This will help to prevent unexpected behavior and ensure that the function behaves as intended.
- 26) ★ The **shadowing-builtin** vulnerability: Shadowing built-in symbols using local variables, state variables, functions, modifiers, or events can lead to unexpected behavior in Solidity smart contracts. When a symbol is shadowed, the Solidity compiler may interpret the code in a way that is different from what the programmer intended. To avoid such issues, it is recommended to rename any local variables, state variables, functions, modifiers, and events that could potentially shadow a built-in symbol. By doing so, the code will be clearer and easier to read, while also reducing the risk of unexpected behavior.
- 27) ★ The **shadowing-local** vulnerability: Shadowing using local variables that shadow another component can introduce confusion and ambiguity in Solidity smart contracts. When a local variable has the same name as another component, such as a function parameter or a state variable, it can lead to unintended behavior and make the code harder to understand and maintain. To avoid such issues, it is recommended to rename any local variables that may shadow another component. By choosing descriptive and unique names for local variables, the code will be more readable and less prone to confusion.
- 28) The **variable-scope** vulnerability: When declaring variables in Solidity, it is important to ensure that they are declared before they are used. This is because Solidity allows the use of variables before they are declared, which can lead to unintended consequences if the variable is declared in another

scope or at a later time. To prevent this, it is recommended to move all variable declarations prior to any usage of the variable, and to ensure that reaching a variable declaration does not depend on some conditional if it is used unconditionally. This ensures that the variable is properly initialized and avoids any potential issues related to the variable’s scope.

- 29) The **void-cst** vulnerability pertains to the situation where a constructor is called, but the constructor has not been implemented. This can lead to unexpected behavior and errors in the contract execution. The recommended solution is to remove the constructor call. This can be achieved by removing the instantiation of the contract object or by ensuring that the constructor is implemented before the contract is deployed.

Example:

To illustrate a reentrancy attack stealing Ether, consider the “Bank.sol” contract (Figure 2.6). This vulnerable contract contained security flaws similar to those exploited in the infamous DAO attack, where a “reentrancy-eth” vulnerability allowed attackers to steal funds. Let’s break down the attack steps:

- The Attacker calls *deposit* function of “Bank.sol” to register his address in the contributors list, then invokes *withdraw* function.
- The “Bank.sol” transfers an amount of money and calls the fallback function of the Attacker.
- The fallback function of the attacker recursively calls the withdraw function again, to gain more payment.
- Within an iteration limit, extra ether will be transferred many times to the “thief.sol” contract.
- The attacker calls *steal* function to transfer the stolen money from the “thief.sol” contract to his account.

Available solutions for “reentrancy-eth”:

- Use `send()` or `transfer()` to send funds instead of `call.value()`.

```

1 // Bank.sol
2 pragma solidity ^0.4.22;
3 contract Bank {
4     mapping(address => uint) balances;
5     constructor() public { }
6     function deposit() public payable {
7         uint amount = balances[msg.sender];
8         balances[msg.sender] = amount + msg.value;
9     }
10    function withdraw() public {
11        if (msg.sender.call.value(balances[msg.sender])) {
12            balances[msg.sender] = 0;
13        }
14    }
15 }
16

```

```

1 // Thief.sol
2 pragma solidity ^0.4.22;
3 import "./Bank.sol";
4 contract Thief {
5     Bank public target;
6     address public owner;
7     constructor(address _target) public {
8         target = Bank(_target);
9         owner = msg.sender;
10    }
11    function collect() public payable {
12        target.deposit.value(msg.value)();
13        target.withdraw();
14    }
15    function() public payable {
16        if (address(target).balance >= msg.value) {
17            target.withdraw();
18        }
19    }
20    function steal() payable public{
21        owner.transfer(address(this).balance);
22    }
23 }
24

```

```

1 // truffle console
2 // const bank = await Bank.deployed() // const thief = await Thief.deployed()
3 // const accounts = await web3.eth.getAccounts()
4 // bank.deposit({from:accounts[1], value: 100});
5 // bank.deposit({from:accounts[2], value: 100});
6 // await web3.eth.getBalance(bank.address)
7 // thief.collect({value:10}); // thief.steal();
8

```

Figure 2.6: Reentrancy Exploit (DAO) Example

- Change the internal state first and then call external function, and use a mutex when the external calls are unavoidable.

Due to the complexity of smart contracts, it is possible for developers to inadvertently introduce security vulnerabilities into the code. Such vulnerabilities can be exploited by attackers to steal or manipulate funds or to disrupt the normal functioning of the DApp. Therefore, it is essential to use tools that can detect security vulnerabilities in smart contracts and provide developers with the necessary information to fix them.

2.5 Deep Learning: Methods and Applications

Deep learning is a specific subfield of machine learning, which in turn is a subfield of artificial intelligence. Figure 2.7 provides a visual representation of these relationships. Artificial intelligence is the broadest category, encompassing all systems that exhibit intelligent behavior. Machine learning is a subset of artificial intelligence that focuses on developing algorithms that can learn from data without being explicitly programmed. Finally, deep learning is a subfield of machine learning that is specifically concerned with neural networks and the development of algorithms that can model complex relationships in data.

One example of the application of deep learning is in the field of medical imaging. Medical professionals use medical imaging techniques such as X-rays, MRIs, and CT scans to diagnose diseases and injuries. Analyzing and interpreting these images can be challenging, as there is often a large amount of visual data to process. Deep learning algorithms can be trained to recognize patterns and identify anomalies in medical images, which can assist medical professionals in making diagnoses and developing treatment plans. This is an example of how deep learning can be used as a method to implement a machine learning task, which is necessary for the interpretation of medical images because it is not feasible to explicitly program the machine to recognize all possible patterns and anomalies.

The primary difference between machine learning and deep learning is in their algorithms. Machine learning relies on feature engineering, which involves manually selecting and extracting relevant features from data, whereas deep learning uses

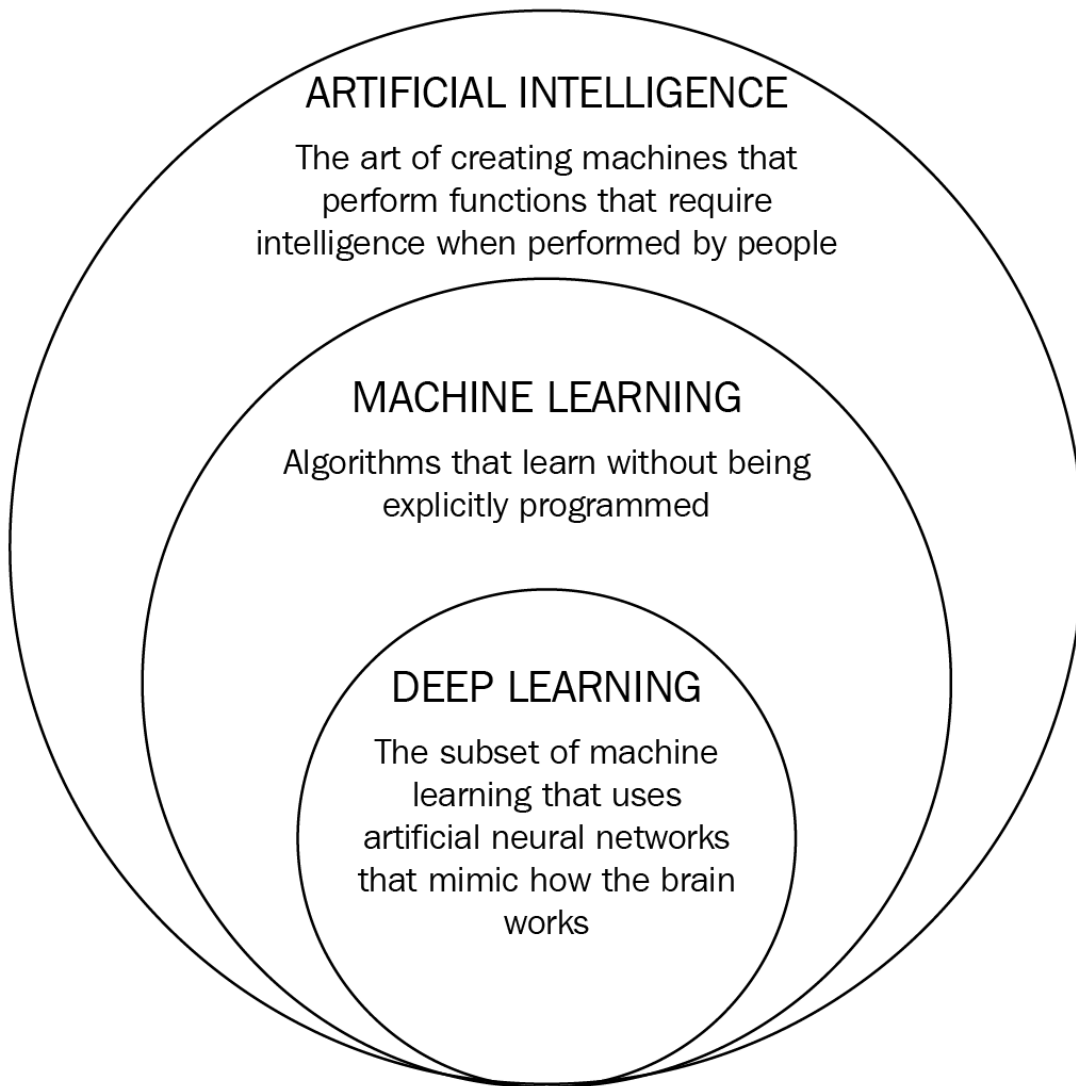


Figure 2.7: The relationship between artificial intelligence, machine learning, and deep learning.

neural networks and multiple layers of data processing to automatically extract relevant features, as shown in Figure 2.8. Deep learning algorithms can recognize more complex patterns than machine learning algorithms, which allows them to achieve higher accuracy in decision-making tasks. Deep learning models can have dozens or even hundreds of layers, allowing them to learn increasingly complex representations of the data. Another difference is the amount of data required to train deep learning models. Deep learning models typically require large amounts of labeled data to achieve high levels of accuracy, whereas some other machine learning algorithms can work well with smaller datasets. Finally, deep learning models are often used for tasks that involve unstructured data, such as images, audio, or text, whereas other machine learning algorithms are often used for more structured data, such as numerical or categorical data.

In learning-based model training, the loss function is a mathematical metric quantifying the discrepancy between a model's predictions and the actual values. It acts as a guide during training, providing feedback on the model's performance. By minimizing the loss function through optimization algorithms, the model learns to adjust its internal parameters and improve its predictions. Choosing the appropriate loss function is crucial for effective training, as it influences the model's bias-variance trade-off and sensitivity to outliers.

2.5.1 Learning Methods

There are four main types of learning-based methods:

- 1) **Supervised learning** is a type of machine learning where a model learns to make predictions or classifications based on labeled examples or input-output pairs. In other words, the model is trained on a dataset where the desired output or label is already known, and it tries to learn a function that maps input to output. Here are some examples of supervised learning.

Image classification: In this task, the goal is to classify images into different categories or classes. For instance, a learning-based model can be trained to recognize whether an image contains a dog or a cat. The model is trained on a dataset of labeled images where each image is labeled as either a dog or a cat. Once the model is trained, it can be used to classify new, unseen images.

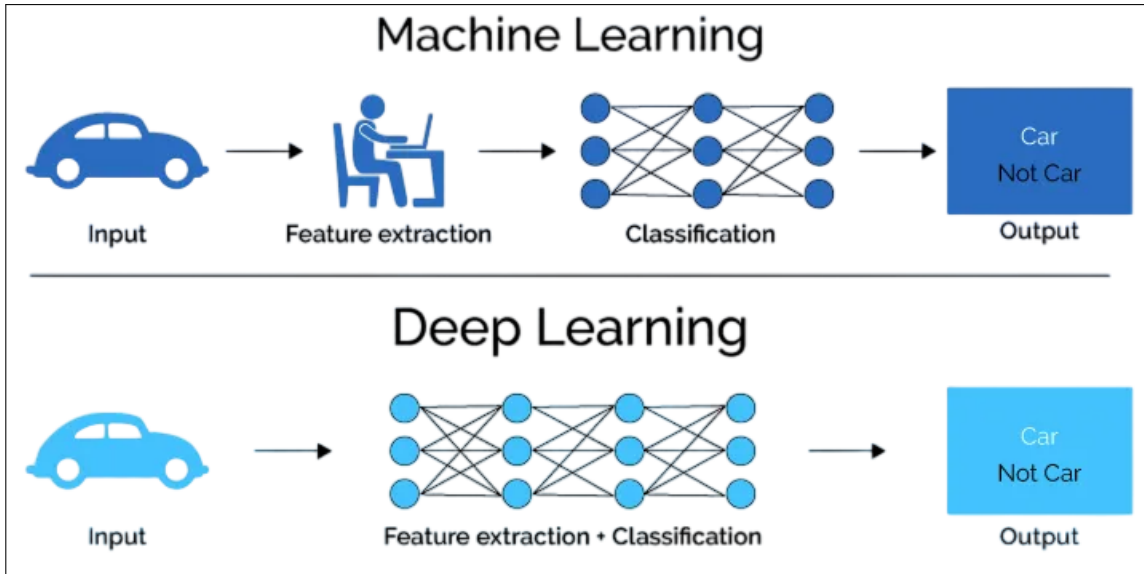


Figure 2.8: Machine Learning vs Deep Learning.

Sentiment analysis: In this task, the goal is to classify text as positive, negative, or neutral. For example, a learning-based model can be trained to analyze customer reviews of a product and determine whether the review is positive or negative. The model is trained on a dataset of labeled reviews where each review is labeled as either positive or negative. Once the model is trained, it can be used to classify new, unseen reviews.

Speech recognition: In this task, the goal is to transcribe spoken words into text. For example, a learning-based model can be trained to transcribe a person's spoken words into written text. The model is trained on a dataset of labeled audio recordings where each recording is transcribed into text. Once the model is trained, it can be used to transcribe new, unseen audio recordings.

Fraud detection: In this task, the goal is to detect fraudulent transactions. For example, a learning-based model can be trained to identify credit card transactions that are likely to be fraudulent. The model is trained on a dataset of labeled transactions where each transaction is labeled as either fraudulent or not. Once the model is trained, it can be used to identify new, potentially fraudulent transactions.

These are just a few examples of supervised learning applications. The key

idea is that the model is trained on labeled data, and it tries to learn a function that maps input to output. Once the model is trained, it can be used to make predictions or classifications on new, unseen data.

- 2) **Unsupervised learning** is a type of machine learning where a model is trained on unlabeled data to discover patterns and relationships without any pre-existing knowledge of the target variable or output. In other words, the model learns to identify hidden structures and patterns in the data without any guidance or supervision. Here are some examples of unsupervised learning.

Clustering: In this task, the goal is to group similar items together based on their features or characteristics. For example, a learning-based model can be trained to group customers based on their purchasing behavior or group images based on their visual features. The model is trained on unlabeled data and learns to identify similarities and differences between items to create clusters.

Anomaly detection: In this task, the goal is to identify unusual or rare occurrences in the data. For example, a learning-based model can be trained to identify fraudulent transactions or detect anomalies in medical images. The model is trained on unlabeled data and learns to identify patterns that are different from the norm.

Dimensionality reduction: In this task, the goal is to reduce the number of features or variables in the data while preserving as much of the original information as possible. For example, a learning-based model can be trained to compress high-dimensional data such as images or text into a lower-dimensional space. The model is trained on unlabeled data and learns to identify the most important features or dimensions.

Generative modeling: In this task, the goal is to generate new data that is similar to the original data. For example, a learning-based model can be trained to generate realistic images, text, or sound. The model is trained on unlabeled data and learns to capture the underlying distribution of the data to generate new samples.

The key idea is that the model is trained on unlabeled data and learns to identify patterns and relationships without any pre-existing knowledge of the

target variable or output. Once the model is trained, it can be used to generate new data or identify unusual occurrences in the data.

- 3) **Semi-supervised learning** is a type of machine learning where a model is trained on a combination of labeled and unlabeled data. The model learns to make predictions or classifications based on the labeled examples while also discovering patterns and relationships in the unlabeled data. This type of learning is especially useful when labeled data is limited or expensive to obtain. Here are some examples of semi-supervised learning.

Text classification: In this task, the goal is to classify text into different categories or classes. For example, a learning-based model can be trained to classify news articles into different topics such as sports, politics, or business. The model is trained on a small set of labeled articles and a large set of unlabeled articles. The model learns to identify patterns and relationships in the unlabeled articles to improve its classification accuracy on the labeled data.

Image segmentation: In this task, the goal is to partition an image into different regions or objects. For example, a learning-based model can be trained to segment medical images into different organs or structures. The model is trained on a small set of labeled images and a large set of unlabeled images. The model learns to identify similar patterns and relationships in the unlabeled images to improve its segmentation accuracy on the labeled data.

Speech recognition: In this task, the goal is to transcribe spoken words into text. For example, a learning-based model can be trained to transcribe a person's spoken words into written text. The model is trained on a small set of labeled audio recordings and a large set of unlabeled recordings. The model learns to identify common speech patterns and relationships in the unlabeled recordings to improve its transcription accuracy on the labeled data.

Fraud detection: In this task, the goal is to detect fraudulent transactions. For example, a learning-based model can be trained to identify credit card transactions that are likely to be fraudulent. The model is trained on a small set of labeled transactions and a large set of unlabeled transactions. The

model learns to identify similar patterns and relationships in the unlabeled transactions to improve its fraud detection accuracy on the labeled data.

The key idea is to leverage both labeled and unlabeled data to improve the model's accuracy and generalization performance. Semi-supervised learning is especially useful when labeled data is limited or expensive to obtain.

- 4) **Reinforcement learning** is a type of machine learning where a model learns to make decisions by interacting with an environment and receiving feedback or rewards for its actions. The model learns to maximize its rewards by exploring different actions and observing the outcomes. This type of learning is especially useful for tasks where there is no clear right or wrong answer, such as game playing, robotics, and autonomous driving. Here are some examples of reinforcement learning.

Game playing: In this task, the goal is to learn to play a game such as chess or video games. The learning-based model learns to make moves based on the current state of the game and the expected rewards or penalties for each move. The model explores different moves and learns from its mistakes to improve its game playing performance over time.

Robotics: In this task, the goal is to learn to control a robot to perform a task such as grasping an object, navigating through an environment, or manipulating an object. The learning-based model learns to control the robot based on sensory input such as camera images or laser scans and receives rewards or penalties based on its actions. The model explores different actions and learns to perform the task more efficiently over time.

Autonomous driving: In this task, the goal is to learn to drive a vehicle autonomously by making decisions such as accelerating, braking, and steering. The learning-based model learns to make decisions based on sensory input such as camera images, lidar scans, and GPS data and receives rewards or penalties based on its actions such as reaching the destination safely or colliding with an obstacle. The model explores different actions and learns to drive more safely and efficiently over time.

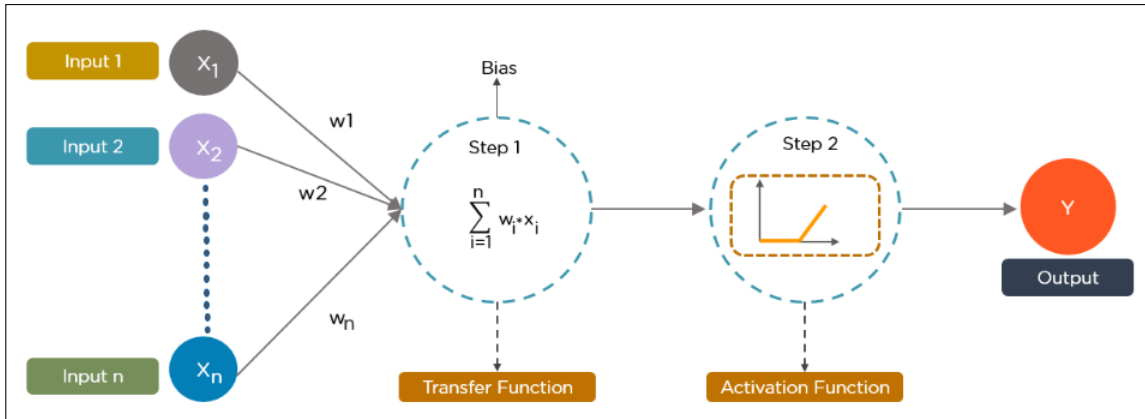


Figure 2.9: Artificial Neural Network.

Dialogue systems: In this task, the goal is to learn to generate natural language responses to user queries. The learning-based model learns to generate responses based on the user’s input and receives rewards or penalties based on the relevance and coherence of its responses. The model explores different responses and learns to generate more natural and informative responses over time.

The key idea is that the model learns to make decisions by interacting with an environment and receiving feedback or rewards for its actions. Reinforcement learning is especially useful for tasks where there is no clear right or wrong answer and the model must learn through trial and error.

2.5.2 The Mechanics of a Basic Neural Network

The basic structure of a deep learning algorithm is a neural network, which is comprised of interconnected nodes or artificial neurons that communicate with each other like the human brain. The network is divided into layers, including the input layer, one or more hidden layers, and the output layer, as shown in Figure 2.9. During the training process, the neural network takes in input data and processes it through a series of transformations, with each layer of the network using its own set of learned parameters to perform its specific computation. These computations involve adjusting the weights and biases of the connections between the nodes to minimize the difference between the actual output and the expected output. Deep

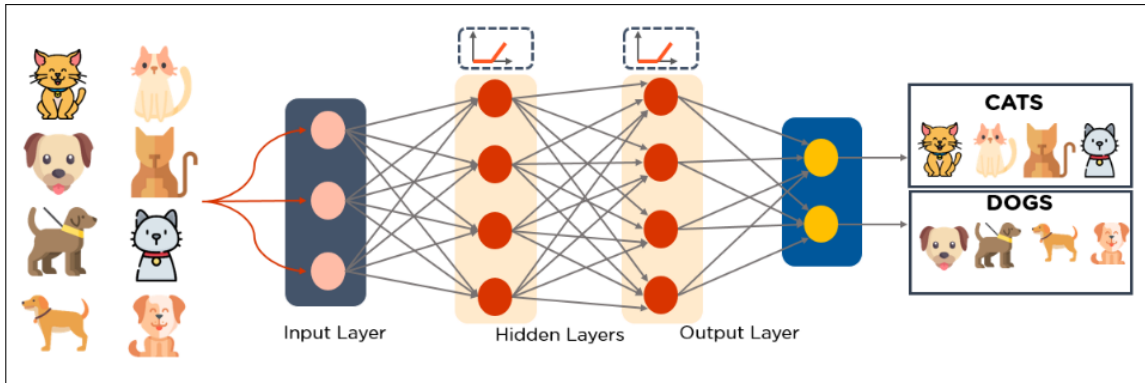


Figure 2.10: Multilayer Perceptron Neural Network.

learning algorithms use a process called backpropagation to adjust the weights and biases of the network. During this process, the error or loss between the predicted output and the actual output is calculated and fed back through the network to update the weights and biases of the connections. This process is repeated until the error is minimized, and the network is able to make accurate predictions on new data. Once the network is trained, it can be used to make predictions on new data by feeding the input data through the network and obtaining the output from the final layer, as shown in Figure 2.10.

2.5.3 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) are a type of neural network that operate on graph-structured data. A graph is a mathematical representation of a set of objects, where the objects are represented by nodes (or vertices) and their relationships are represented by edges, as shown in Figure 2.11. GNNs are designed to learn representations of nodes and edges in a graph, and to use these representations to perform various tasks, such as node classification, link prediction, and graph classification.

The basic building block of a GNN is a graph convolutional layer, which applies a linear transformation to each node's features and aggregates information from its neighbors. This process is repeated for multiple layers to learn increasingly abstract representations of the graph's structure. GNNs can also incorporate additional features, such as edge and node attributes, to further enhance their representational

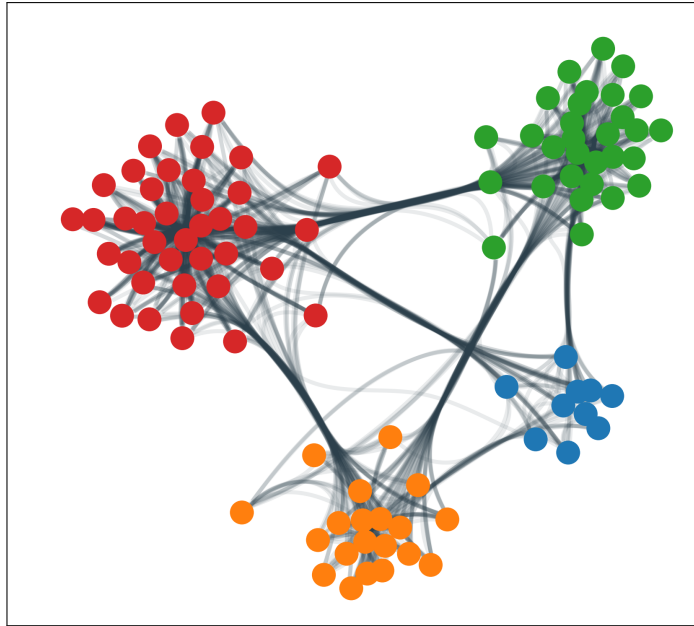


Figure 2.11: Graph-structured data.

power. One of the key challenges in designing GNNs is handling the variable size and connectivity of graphs. Different approaches have been proposed to address this, including message-passing schemes, graph attention mechanisms, and graph pooling operations. GNNs have been applied to a wide range of tasks, including node classification, link prediction, and graph classification. They have shown great promise in domains such as social networks, chemistry, and recommendation systems, where data is naturally represented as graphs.

Graph Neural Networks (GNNs) can be applied to a variety of tasks in graph analysis. Below are some of the types of GNN tasks with examples:

- **Graph Classification:** This task involves classifying an entire graph into one of several categories based on its structure or properties. For example, social network analysis can classify a graph of users and their interactions into different communities based on common interests or behaviors.
- **Node Classification:** In this task, the goal is to predict missing node labels in a graph based on the labels of neighboring nodes. For example, in a citation network, we can predict the topic of a scientific paper based on the topics of its citing papers.

- **Link Prediction:** This task involves predicting the likelihood of a link between two nodes in a graph based on the graph’s structure. For example, in a social network, we can predict the likelihood of two users becoming friends based on their common interests or mutual friends.
- **Community Detection:** This task involves dividing nodes into clusters or communities based on their structural similarity. For example, in a co-authorship network, we can identify groups of researchers who collaborate frequently based on their publication history.
- **Graph Embedding:** This task involves mapping graphs into low-dimensional vectors while preserving the graph’s structure and properties. Graph embeddings can be used for downstream tasks such as node classification, link prediction, and community detection.
- **Graph Generation:** This task involves learning from the distribution of sample graphs to generate new graphs that are similar in structure and properties. Graph generation can be used for applications such as drug discovery, where new molecular structures need to be generated and evaluated.

In summary, Graph Neural Networks are a type of neural network that can operate on graph-structured data, and are designed to learn representations of nodes and edges in a graph. They are built using graph convolutional layers and can incorporate additional features to enhance their representational power. GNNs have been shown to be effective for a variety of applications that involve graph-structured data, including graph classification, node classification, link prediction, community detection, graph embedding, and graph generation. These tasks have applications in various fields such as social network analysis, text classification, and drug discovery.

2.6 Evaluation Metrics

The learning-based classification has four possible outcomes: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). We adopted the evaluation metrics of accuracy, true positive rate (TPR), true negative rate (TNR), false positive rate (FPR), false negative rate (FNR), F1-score, and area under the

curve (AUC) scores on the test dataset. We could not use accuracy in (2.1) as the only metric for evaluation because our datasets are imbalanced (the total number of vulnerable smart contracts are scarce compared to the non-vulnerable), as the model could easily achieve high accuracy by labeling all samples as the majority class (non-vulnerable class) and neglect the minority (vulnerable class). Accuracy works best if false positives and false negatives have similar cost. In our problem false negative has dire consequences more than false positive.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Tested}} \quad (2.1)$$

Precision in (2.2) is a metric used in machine learning to evaluate the accuracy of a model in correctly identifying positive instances. It is calculated as the ratio of true positives (TP) to the sum of true positives and false positives (FP):

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \quad (2.2)$$

In other words, precision measures the proportion of true positives out of all instances predicted as positive. A high precision score indicates that the model is making fewer false positive predictions and is, therefore, more reliable in identifying positive instances. This metric is particularly important in situations where false positive predictions are costly, such as in fraud detection.

Recall in (2.3) is another metric used in machine learning to evaluate the performance of classification models. Recall is also known as sensitivity, hit rate, or true positive rate (TPR). Recall is calculated as the ratio of true positives (TP) to the sum of true positives and false negatives (FN):

$$\text{True Positive Rate (Recall)} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \quad (2.3)$$

Recall measures the ability of the model to correctly identify all positive instances, regardless of whether it also predicts some false positives. A high recall score indicates that the model can identify a high proportion of positive instances out of all actual positive instances. Recall is particularly important in situations where false negative predictions are costly, such as in disease diagnoses, where a false negative result can result in a delayed or incorrect treatment.

False positive rate in (2.4) (also known as probability of false alarm, fall-out) is a statistical metric that measures the proportion of negative instances that are incorrectly classified as positive.

$$\text{False Positive Rate} = \frac{\text{False Positives}}{\text{True Negatives} + \text{False Positives}} \quad (2.4)$$

The value of TPR measures the ability not to miss any vulnerable contract (how many of the actual positives our model is able to capture through labeling it as positive and it is true positive), while the value of FPR measures the ability of the model to reduce false alarms. These metrics help to measure the detection rate of vulnerable contracts, and calculate the false alarms of mislabeling an innocent contract as vulnerable. The receiver operating characteristic (ROC) curve is frequently used for evaluating the performance of binary classification algorithms. ROC is produced by calculating and plotting the true positive rate against the false positive rate for a single classifier at a variety of thresholds. AUC stands for area under the ROC curve and represents the degree or measure of separability. AUC tells how much the model is capable of distinguishing between non-vulnerable and vulnerable classes¹.

The F1 score in (2.5) is a statistical metric that represents the harmonic mean of precision and recall. It is frequently used as an evaluation metric in both binary and multi-class classification tasks, as it combines the precision and recall metrics into a single value that provides insight into model performance.

$$\text{F1 score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \quad (2.5)$$

As such, the F1 score serves as a valuable metric for evaluating the effectiveness of classification models, allowing for the identification of optimal models that provide both high precision and recall. Additionally, the F1 score provides a convenient method for comparing the performance of different models across varying datasets and problem domains.

¹AUC visualization: <https://arize.com/blog/what-is-auc/>

Chapter 3

Related work

This chapter delves into the ever-evolving realm of Ethereum smart contract (SC) analysis tools, conducting a comprehensive survey of their diverse types and methodical approaches. To accomplish this, we meticulously explored 83 research papers, tools, and datasets spanning the literature and online resources published from 2016 to 2023, this comprehensive analysis illuminates the current state of the art in smart contract security. Beyond simply charting the existing terrain, this exploration aims to empower developers and researchers with insightful knowledge on available tools, datasets, and benchmarks, ultimately paving the way for the development of even more robust and impactful solutions. Our analysis focuses on four key areas:

- 1) **Static and Dynamic Analysis (SA/DA) Methods for SC:** We begin by exploring well-established static and dynamic analysis (SA/DA) methods for scrutinizing smart contract code. This includes examining 21 primary studies and tools that leverage SA/DA techniques to identify vulnerabilities
- 2) **Learning-based Techniques for SC:** We then shift focus to the cutting-edge field of applying machine learning and deep learning (ML/DL) techniques to smart contract security. Here, we analyze 35 primary studies and tools that utilize various ML/DL algorithms to automatically detect vulnerabilities in smart contracts.
- 3) **Learning-based Techniques for PL:** To further enrich our understanding, we draw insights from the application of ML/DL approaches to vulnerability detection in traditional programming languages (PL) like C/C++, Java, and

Python. We review 12 primary studies and tools in this space, seeking valuable parallels and complementary perspectives for smart contract security.

- 4) **Data Sources and Benchmarks for SC:** Finally, we turn our attention to the data resources and benchmarks available for testing and comparing the performance of smart contract security tools. We assess 15 primary datasets, evaluating their suitability for rigorous and comprehensive evaluation of different tools and techniques.

By systematically exploring these four key areas, this chapter not only illuminates the current state of smart contract security but also charts a valuable roadmap for future advancements. Understanding the tools and datasets available, the depths of analysis employed, and the diverse techniques utilized in each area empowers researchers and developers to build upon existing knowledge and tackle the evolving challenges of securing these critical blockchain components. This comprehensive analysis lays the foundation for a future where smart contracts are even more resilient and reliable, enabling the continued growth and innovation within the blockchain ecosystem.

3.1 Static and Dynamic Analysis Methods for SC

This section explores various static and dynamic analysis (SA/DA) techniques for smart contract vulnerability detection, categorized into symbolic execution (SymEx), fuzzing (Fuz), and static analysis (StAn).

- Symbolic execution (SymEx) is a powerful technique for analyzing the security of smart contracts. Instead of running a smart contract with concrete data, symbolic execution uses symbolic parameters to represent the possible inputs. This allows researchers to explore a wide range of execution paths and uncover potential vulnerabilities that might be missed by traditional testing methods. Popular symbolic execution tools like Oyente, Mythril, Maian, Manticore, Osiris, and teEther leverage this core principle to analyze smart contracts. These tools typically take the contract and a set of security specifications (e.g., predefined rules by experts) as input. They then employ an SMT (Satisfiability

Modulo Theories) solver to systematically explore all possible execution paths under different symbolic values. Finally, they generate a human-readable report highlighting any potential security issues identified, such as the range of values that could trigger certain vulnerabilities.

However, symbolic execution comes with certain drawbacks:

- Path Explosion: Symbolic execution explores multiple paths through a program, resulting in a combinatorial explosion of paths, especially in complex programs with loops, recursive functions, or large inputs. This can lead to scalability issues, significantly increasing computational resources and analysis time.
- Path-Insensitivity: Symbolic execution may be path-insensitive, i.e., it might not cover all possible program paths due to conditional branches or loops that rely on symbolic values. This could result in missing potential bugs or vulnerabilities that occur in specific paths.
- Over-approximation and False Positives: In some cases, symbolic execution might over-approximate the set of possible program behaviors, leading to false positives or reporting potential issues that are not actual vulnerabilities.

Despite these drawbacks, researchers continue to work on improving symbolic execution techniques and addressing these limitations to make it more effective and scalable for diverse vulnerability analysis scenarios.

- Fuzzing technique (Fuz) offers a powerful approach to smart contract testing by iteratively generating test cases designed to expose vulnerabilities. This addresses two key challenges in software testing: limited input range and path explosion. Popular tools like ContractFuzzer, EthRacer and Confuzzius employ various methods to generate subsets of test inputs that can reveal vulnerable paths within the smart contract’s execution.

The process begins with the smart contract as input. A fuzzing iterator acts as a test case generator, creating new inputs for the fuzzing engine. This engine often utilizes a blockchain virtual machine to simulate transactions with the

generated inputs. Each test transaction is executed, and the resulting state change is fed back to the iterator, influencing the generation of the next test case. This iterative process continues until the desired number of fuzzing cycles is completed. Finally, the collected data is analyzed to identify potentially vulnerable contracts and, if possible, the specific transactions that trigger the vulnerability.

By systematically exploring a wide range of possible inputs and execution paths, fuzzing efficiently detects vulnerabilities that might be missed by traditional testing methods. This makes it a valuable tool for ensuring the security and robustness of smart contracts.

However, fuzzing comes with certain drawbacks:

- Limited Coverage: While fuzzing explores a broad range of inputs, it doesn't guarantee complete code coverage. Certain portions of the codebase or intricate program paths might remain untested, possibly concealing hidden vulnerabilities. Fuzzing tends to excel at finding “surface-level” bugs but struggles with deeper issues, potentially leading to low code coverage and many false negatives.
- Challenges in Input Generation: The effectiveness of fuzzing heavily relies on the quality of generated inputs. If the test cases don't adequately cover potential vulnerabilities, bugs may go undetected. Crafting effective test cases requires understanding the specific vulnerabilities being targeted and tailoring the fuzzing process accordingly.

Despite these drawbacks, fuzzing remains a valuable and widely used technique in identifying vulnerabilities. Combining fuzzing with other analysis methods or employing advanced fuzzing techniques (e.g., coverage-guided or hybrid fuzzing) can help mitigate some of these limitations and improve overall effectiveness in vulnerability discovery.

- Static analysis technique (StAn) provides a valuable approach for detecting vulnerabilities in smart contracts through automated analysis of their source code (e.g., SmartCheck and Slither tools) or bytecode (e.g., Vandal tool). This process begins with the extraction of relevant information, or facts, from the

contract’s source code or bytecode. In some cases, additional metadata may be added to the contract to enhance semantic clarity.

Model construction follows, where the contract is transformed into an intermediate representation (IR). This involves the creation of a structured model that accurately reflects the contract’s behavior, often incorporating manual fact extraction using techniques such as regular expressions or parsing. A control flow graph (CFG) is then constructed to represent the contract’s execution paths.

The analyzer forms the core of the static analysis solution, employing multiple data sources alongside the intermediate representation. These inputs may include blockchain transactions, execution traces produced by instrumented nodes, or vulnerability signatures in the form of formal rules or security patterns. The culmination of the static analysis process is the generation of a human-readable report, outlining the identified vulnerabilities and providing insights into the contract’s security posture.

However, static analysis comes with certain drawbacks:

- Dependency on Expert Rules: The effectiveness of static analysis is highly dependent on the quality of the comprehensiveness of its rule sets. Outdated or incomplete rules can lead to missed vulnerabilities, necessitating continuous updates to keep pace with evolving threats and attack vectors.
- False Positives: Static analysis tools may report potential vulnerabilities that aren’t genuine issues, leading to unnecessary time and effort spent on investigation.

Understanding these limitations is crucial to using static analysis effectively and considering its complementary techniques to achieve more comprehensive program analysis.

Having outlined various static and dynamic analysis techniques, along with their strengths and weaknesses, we now turn to a closer examination of specific tools within each technique. Figure 3.1 offers a visual overview of 21 primary studies/tools,

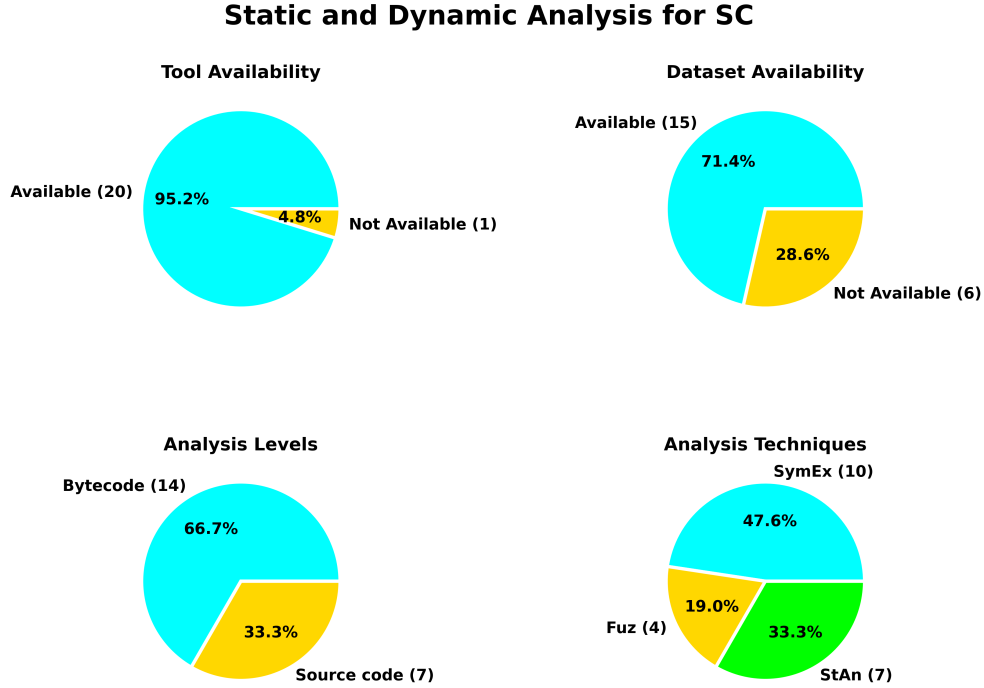


Figure 3.1: Static and Dynamic Analysis for SC in primary studies.

categorized by four key aspects: tool availability (readily available, not available), dataset availability (publicly available, not available), analysis levels (source code, bytecode), and analysis techniques (symbolic execution (SymEx), fuzzing (Fuz), static analysis (StAn)). Table 3.1 further details the corresponding paper references for each category.

In the subsequent subsections, we will delve into each study in sequence, grouped by their respective analysis technique category. This systematic approach allows for a focused and comprehensive exploration of the individual tools and their contributions to the field.

3.1.1 Symbolic Execution (SymEx) Studies/Tools:

Luu et al. [93] (2016)¹ introduced Oyente, a pioneering static analysis tool specifically designed to detect vulnerabilities in smart contracts. Oyente operates

¹Luu et al. [93] (Oyente) is available: <https://github.com/oyente/oyente>

Metric	Category	#Studies	References
Tool Availability	Available	20	[93, 132, 82, 23, 100, 136, 105, 134, 76, 135, 45, 99, 81, 123, 116, 32, 122, 34, 133, 22]
	Not Available	1	[78]
Dataset Availability	Available	15	[93, 78, 134, 76, 135, 45, 99, 81, 123, 116, 32, 122, 34, 133, 22]
	Not Available	6	[132, 82, 23, 100, 136, 105]
Analysis Levels	Bytecode	14	[93, 82, 23, 100, 136, 105, 134, 76, 135, 99, 81, 116, 32, 34]
	Source code	7	[132, 78, 45, 123, 122, 133, 22]
Analysis Techniques	SymEx	10	[93, 82, 100, 105, 134, 135, 99, 32, 122, 22]
	Fuz	4	[76, 81, 34, 133]
	StAn	7	[132, 23, 136, 78, 45, 123, 116]

Table 3.1: Static and Dynamic Analysis for SC in primary studies.

on the bytecode and state of smart contracts, employing symbolic execution to analyze control flow paths and identify potential vulnerabilities based on path constraints (i.e., conditions that must hold true for a specific execution path). The paper demonstrates Oyente’s effectiveness in detecting four common vulnerabilities: transaction-ordering dependence, timestamp dependence, mishandled exceptions, and reentrancy. Additionally, the authors enhanced the open-source code to address integer overflow vulnerabilities. As the first of its kind, Oyente made a significant contribution to the field of smart contract security, laying the foundation for subsequent research and development of more advanced static analysis tools.

Krupp et al. [82] (2018)² introduced teEther, a tool designed to simplify exploit creation for vulnerable smart contracts based solely on their bytecode representation. Its exploit generation process comprises five specialized modules: constructing the contract’s control flow graph (CFG), identifying exploitable execution paths within

²Krupp et al. [82] (teEther) is available: <https://github.com/nescio007/teether>

the CFG, generating constraints associated with those paths, and finally, generating the actual exploit code. For constraint resolution, teEther leverages the powerful Z3 constraint solver. In a comprehensive analysis of 38,757 contracts, teEther identified 815 as vulnerable and subsequently generated 1,564 functional exploits targeting these vulnerable contracts. This impressive outcome highlights the urgency of prioritizing smart contract security measures and addressing potential vulnerabilities to prevent real-world exploits and financial losses. Although there is no common type of vulnerability between teEther and our tools for direct comparison, this highlights the possibility of complementary insights through diverse analysis approaches.

Mueller et al. [100] (2018)³ introduced Mythril, a powerful tool for analyzing the security of smart contracts. Mythril uses symbolic execution to identify potential vulnerabilities. It also incorporates taint analysis and control flow inspection to enhance its detection capabilities. This allows Mythril to detect 14 vulnerabilities at the bytecode level, including common issues like reentrancy, integer overflow and underflow, and timestamp dependencies. Mythril is available as a free open-source version and a paid version called MyTh, which offers additional features such as improved scalability, integration with development tools, and more comprehensive reporting.

Nikolic et al. [105] (2018)⁴ presented MAIAN, a tool for dynamic analysis of smart contracts that focuses on identifying vulnerabilities through trace-based analysis. MAIAN operates by comprehensively analyzing the execution traces of a contract across multiple invocations, allowing it to pinpoint three distinct categories of vulnerabilities: greedy contracts, suicidal contracts (formerly self-destructive), and prodigal contracts. To minimize false positives, MAIAN employs a unique approach: it deploys the analyzed contracts on a private blockchain and then confirms the identified vulnerabilities by initiating targeted transactions that exploit the specific vulnerability type. This innovative validation process strengthens the accuracy of MAIAN’s findings and sets it apart from other vulnerability detection tools. Notably, *MAIAN’s focus on real-world exploitability inspired the development of SCooLS, which incorporates a real exploit generator to further validate identified vulnerabilities on a private blockchain. While both tools validate vulnerabilities through on-chain*

³Mueller et al. [100] (Mythril) is available: <https://github.com/Consensys/mythril>

⁴Nikolic et al. [105] (MAIAN) is available: <https://github.com/ivicanikolicsg/MAIAN>

execution, they do not directly share vulnerabilities for comparison due to their differences in analysis methods and targeted vulnerabilities.

Torres et al. [134] (2018)⁵ introduced Osiris, an advanced static analysis tool for smart contracts built upon the foundation of Oyente. Specifically designed to detect integer-related vulnerabilities, Osiris tackles critical security concerns such as integer overflow, underflow, and type conversion leading to value truncation. Its core strength lies in the synergistic combination of symbolic execution and taint analysis, meticulously tailored to pinpoint these prevalent flaws within Ethereum smart contracts. Compared to existing tools, Osiris boasts a significantly broader spectrum of bug detection, while simultaneously maintaining a higher level of precision in its identification process. This enhanced accuracy and comprehensiveness solidify Osiris’s position as a valuable asset for safeguarding smart contracts against integer-related vulnerabilities.

Torres et al. [135] (2019)⁶ introduced HONEYBADGER, a tool utilizing symbolic execution and specific heuristics, automatically detects smart contract honeypots. In the first systematic analysis of its kind, HONEYBADGER examined over 2 million smart contracts, uncovering at least 690 deployed honeypots. Analysis of transactions associated with a subset of these honeypots revealed that 240 users have fallen victim to them, resulting in a cumulative profit exceeding 90,000 USD for the creators. This research highlights the prevalence of honeypots and the potential financial losses they pose to users. We did not benchmark HONEYBADGER due to the absence of shared vulnerabilities with our tools for comparison.

Mossberg et al. [99] (2019)⁷ introduced Manticore, a powerful dynamic symbolic execution framework specifically designed for analyzing smart contracts and binaries. Unlike other tools that rely on specific execution models, Manticore employs a platform-agnostic execution engine that can adapt to different environments. It analyzes smart contracts by simulating their execution through various possible states, such as function calls and control flow changes. Notably, Manticore sets itself apart by its ability to analyze multiple contracts simultaneously, offering insights into complex interactions and dependencies between them. Inspired by this capability,

⁵Torres et al. [134] (Osiris) is available: <https://github.com/christoftorres/Osiris>

⁶Torres et al. [135] (HONEYBADGER) is available: <https://github.com/christoftorres/HoneyBadger> and <https://honeybadger.uni.lu/>

⁷Mossberg et al. [99] (Manticore) is available: <https://github.com/trailofbits/manticore>

we have designed our tools with two modes: a single mode for individual contract analysis and a batch mode for analyzing multiple contracts concurrently.

Chen et al. [32] (2021)⁸ introduced DefectChecker, a symbolic execution to pinpoint eight smart contract defects directly within the bytecode, bypassing the need for source code analysis. This direct bytecode analysis offers several advantages, including potentially uncovering vulnerabilities missed by traditional source-code-based methods. The authors craft eight distinct rules, each tailored to exploit the signature of a specific contract defect. These rules leverage the CFG, stack event, and identified features to pinpoint the presence of vulnerabilities within the bytecode. This rule-based approach offers both precision and efficiency. However, one noteworthy aspect of DefectChecker involves its exhaustive path exploration. It considers all potential execution paths, even those seemingly unreachable due to conditional expressions consistently evaluating to false. While this ensures comprehensive vulnerability detection, it may also lead to false positives. In cases where such unlikely paths trigger the defect-specific rules, the tool might flag non-existent vulnerabilities. Notably, *DefectChecker currently lacks readily available instructions for use*. The authors responded to our inquiries, and clarified that the tool is specifically designed for use with the Solidity compiler version v0.4.25 and EVM version 1.8.14. Running the tool with higher versions may lead to issues, so users should be aware of these compatibility limitations.

So et al. [122] (2021)⁹ introduced SMARTTEST, a novel approach to symbolic execution for smart contracts that focuses on identifying vulnerable transaction sequences. It employs a unique hybrid strategy that combines symbolic execution with a vulnerability-aware language model. This model prioritizes program paths during symbolic execution based on their likelihood of leading to vulnerabilities, improving efficiency and focusing on exploit-relevant areas. SMARTTEST autonomously learns this vulnerability knowledge by analyzing a corpus of training transaction sequences extracted from known vulnerable contracts through unguided symbolic execution. This training process allows it to establish a probability distribution over potential vulnerable sequences. In our attempts to utilize VERISMART for benchmarking

⁸Chen et al. [32] (DefectChecker) is available: <https://github.com/Jiachi-Chen/DefectChecker>

⁹So et al. [122] (SMARTTEST) is available: <http://prl.korea.ac.kr/smartest>

purposes, we encountered installation and configuration challenges. Unfortunately, our efforts to contact the authors for guidance did not yield a response before this writing. Consequently, we were unable to incorporate VERISMART into our evaluation.

Bose et al. [22] (2022)¹⁰ introduced SAILFISH, a scalable system designed to automatically detect state-inconsistency bugs in smart contracts source code. SAILFISH stands apart from other tools by employing a hybrid strategy that combines efficient exploration with precise symbolic evaluation. In the first phase, EXPLORE, SAILFISH performs a lightweight analysis to identify potentially vulnerable code paths. This reduces the computational burden compared to full symbolic analysis, making SAILFISH scalable for analyzing large numbers of contracts. In the second phase, REFINE, SAILFISH applies its innovative value-summary analysis (VSA) to capture the global state of the contract across its storage variables. This VSA efficiently generates comprehensive constraints on possible contract states, which are then fed to a symbolic evaluator. This combined approach significantly reduces false positives compared to traditional methods, leading to highly accurate bug detection in SAILFISH. In our benchmarking, SAILFISH exhibited promising performance in terms of scalability and accuracy. However, we encountered some exceptions due to SAILFISH’s current implementation relying on an older version of Slither that is not fully compatible with recent Solidity updates. This compatibility issue necessitates further development and updates to the tool to maintain its effectiveness as Solidity languages evolve.

3.1.2 Fuzzing (Fuz) Studies/Tools:

Jiang et al. [76] (2018)¹¹ introduced ContractFuzzer, a pioneering tool that leverages fuzz testing to uncover vulnerabilities in smart contracts. Unlike traditional black-box fuzzing approaches, ContractFuzzer takes advantage of the contract’s ABI specification interface. This interface defines the functions and data structures exposed to the outside world. ContractFuzzer analyzes this specification to generate fuzzed inputs that adhere to the specific syntax and semantics of the contract

¹⁰Bose et al. [22] (SAILFISH) is available: <https://github.com/ucsb-seclab/sailfish>

¹¹Jiang et al. [76] (Contractfuzzer) is available: <https://github.com/gongbell/ContractFuzzer>

functions. Furthermore, ContractFuzzer employs targeted defect oracles to identify different vulnerability types with high precision. These oracles define specific patterns within the contract’s execution that signify vulnerabilities like reentrancy, timestamp dependency, or transaction order dependency. For instance, to detect reentrancy, ContractFuzzer crafts a dedicated attack contract. When invoked on the target contract, this attack contract attempts to trigger the reentrancy flaw and confirm its presence. This targeted approach significantly reduces false positives. In our research, we evaluated the performance of DLVA and SCoolS, our smart contract analysis tools, alongside recent fuzzers like ConFuzzius. This comparative analysis allowed us to assess the strengths and weaknesses of different approaches to smart contract security analysis, ultimately highlighting the complementary benefits of fuzzing techniques like ContractFuzzer.

Kolluri et al. [81] (2019)¹² introduced EthRacer, a powerful tool for automatically detecting Event Order (EO) bugs. Unlike other tools requiring user-provided hints, EthRacer analyzes contracts directly on Ethereum bytecode, making it user-friendly and scalable. EthRacer’s core mission is to determine whether modifying the sequence of events within a contract alters its final outputs. If different outputs emerge under rearranged event orders, the contract is flagged as exhibiting an EO bug (EO-unsafe). Conversely, contracts unaffected by event reordering are classified as EO-safe. This analysis leverages a unique combination of symbolic execution and randomized fuzzing of event sequences. To optimize its search space, EthRacer utilizes the Happens-Before relation, a concept defining causal dependencies between events. This prevents the tool from wasting time on irrelevant or impossible event reorderings. Our benchmarks do not shard vulnerabilities directly comparable to EthRacer’s capabilities.

Choi et al. [34] (2021)¹³ introduced SMARTIAN, a smart contract fuzzing tool that leverages a unique combination of SA/DA to optimize its seed generation and management. This hybrid approach enhances the effectiveness of the fuzzing process and ultimately boosts the detection of vulnerabilities. In the initial stage, SMARTIAN employs static analysis to examine smart contract bytecode. This analysis predicts promising transaction sequences that are likely to uncover vulnerabilities

¹²Kolluri et al. [81] (EthRacer) is available: <https://github.com/aashishkolluri/EthRacer>

¹³Choi et al. [34] (SMARTIAN) is available: <https://github.com/SoftSec-KAIST/Smartian>

during fuzzing. It further identifies specific constraints that each transaction should adhere to, ensuring focused exploration of the contract’s state space. This valuable information lays the foundation for the fuzzing phase, informing the creation of an initial seed corpus of high-quality inputs. Throughout the fuzzing campaign, SMARTIAN continuously performs a lightweight dynamic data-flow analysis. This analysis monitors the execution of generated inputs, observing how data flows through the contract. By analyzing these data-flow patterns, SMARTIAN gathers crucial feedback that guides the fuzzing process in real-time. This feedback helps prioritize promising paths and avoid unproductive exploration, leading to a more efficient and effective vulnerability discovery process.

Torres et al. [133] (2021)¹⁴ introduced ConFuzzius, a hybrid fuzzer that combines the strengths of symbolic execution and fuzzing to uncover vulnerabilities in smart contracts. Unlike traditional fuzzers, ConFuzzius leverages symbolic taint analysis to track the flow of information through the contract and generate path constraints based on tainted inputs. These constraints represent the conditions that must be met for the fuzzer to reach specific parts of the code. When the fuzzer encounters a challenging contract condition that blocks its progress, ConFuzzius activates a constraint solver. This solver attempts to find concrete values that satisfy the constraints, allowing the fuzzer to bypass the obstacle and explore new paths in the code execution. The solutions generated by the solver are then added to a mutation pool, which serves as a reservoir of potential inputs for the fuzzer to explore further. Furthermore, ConFuzzius utilizes dynamic data dependency analysis to identify relationships between variables and transactions within the contract. This information is then used to generate sequences of transactions that are more likely to trigger specific contract states where bugs might be hidden. This targeted approach significantly increases the efficiency and effectiveness of the fuzzing process compared to purely random fuzzing techniques.

¹⁴Torres et al. [133] (ConFuzzius) is available: <https://github.com/christoftorres/ConFuzzius>

3.1.3 Static Analysis (StAn) Studies/Tools:

Tikhomirov et al. [132] (2018)¹⁵ presented SmartCheck, a static analysis tool designed to detect potential vulnerabilities in Solidity smart contracts. SmartCheck operates by employing a syntactic pattern recognition approach. It first translates Solidity code into an XML-based syntax tree, creating a structured representation of the code’s elements and relationships. Vulnerabilities are then defined using XQuery path expressions, a powerful query language specifically designed for navigating XML data structures. These expressions enable SmartCheck to efficiently search for specific vulnerability patterns within the XML tree, pinpointing potential security flaws within the code. This technique offers a lightweight and targeted approach to vulnerability identification, focusing on the structural elements of the code that often signal security issues. Notably, SmartCheck is able to detect 21 vulnerabilities in Solidity contracts, but its project has been deprecated since 2020, and the analysis may work incorrectly for Solidity versions starting with 0.6.0 as mentioned in its repository.

Brent et al. [23] (2018)¹⁶ introduced Vandal, a comprehensive security analysis tool for smart contracts bytecode. Vandal’s core strength lies in its sophisticated analysis pipeline. This pipeline translates low-level bytecode into a higher-level format that facilitates efficient vulnerability detection through symbolic execution and logic-based analysis. The pipeline begins by retrieving the contract’s bytecode from the blockchain and transforms it into a series of opcode instructions. Next, the decompiler analyzes these instructions and captures the contract’s logic as relationships between data and control flow, revealing potential vulnerabilities like reentrancy and unchecked-send issues based on predefined logic patterns. Vandal’s ability to analyze bytecode of smart contracts makes it a promising tool for developers, auditors, and researchers. However, during our exploration, *we encountered difficulties installing Vandal due to the lack of readily available instructions*. Despite contacting the authors, we haven’t received a response at the time of writing. This lack of user support currently presents a barrier to wider adoption of the tool, but its functionality holds value for those able to overcome this hurdle.

¹⁵Tikhomirov et al. [132] (SmartCheck) is available: <https://github.com/smartdec/smartcheck>

¹⁶Brent et al. [23] (Vandal) is available: <https://github.com/usyd-blockchain/vandal>

Tsankov et al. [136] (2018)¹⁷ introduced Securify, a scalable, fully automated analyzer for Ethereum smart contracts that assesses their safety or unsafety based on user-defined security properties. Securify operates at the bytecode level, extracting precise semantic details from the code through a symbolic analysis of its dependency graph. These semantics, expressed in Datalog syntax, capture the contract’s behavior and internal relationships. Securify then cross-references these extracted semantics against predefined security property rules categorized into compliance and violation modes. This enables it to verify the contract’s adherence to the desired security properties, effectively detecting potential vulnerabilities and unsafe behaviors. Notably, *Securify requires the user to specify the Solidity version before building the tool, hindering its applicability in benchmarking scenarios involving large datasets with contracts written in various Solidity versions. Due to this limitation, Securify was not included in the current benchmarking process, as the datasets under consideration contain thousands of contracts with diverse Solidity versions, making it impractical.*

Kalra et al. [78] (2018) introduced ZEUS, an automated formal verification tool for smart contracts. ZEUS translates Solidity source code into the LLVM intermediate language, a format that facilitates efficient analysis. Employing the eXtensible Access Control Markup Language (XACML), ZEUS constructs customized verification rules atop this representation. These rules are specifically designed to detect various security vulnerabilities, enabling ZEUS to assess the security posture of the target smart contract throughout the formal verification process. While the tool itself is currently not available for public benchmarking, the authors released a benchmark of seven vulnerabilities, which we discuss further in the benchmarking section.

Feist et al. [45] (2019)¹⁸ developed Slither, a static analysis framework dedicated to comprehensively detecting vulnerabilities in smart contract source code. Slither operates at the source code level, performing both lexical and syntactic analyses. Slither leverages abstract syntax trees (ASTs) to construct crucial code representations like inheritance graphs, control flow graphs, and contract expressions. Furthermore, Slither introduces an intermediate language called SlitherIR, which serves as the platform for all static program analysis operations. One defining feature

¹⁷Tsankov et al. [136] (Securify) is available: <https://github.com/eth-sri/securify2>

¹⁸Feist et al. [45] (Slither) is available: <https://github.com/crytic/slither>

of Slither is its active development and frequent updates. While the paper by Feist et al. reported Slither’s ability to detect 28 vulnerabilities, the current version boasts an impressive repertoire of 74 vulnerability types. This continually expanding spectrum makes Slither a powerful tool for identifying a wide range of security flaws in smart contracts. Another advantage of Slither lies in its efficiency. Its analysis speed surpasses other static analyzers, with contracts typically taking only 2-3 seconds to process. This rapid analysis makes Slither an ideal choice as a “supervising oracle” for labeling large datasets. Slither is employed to label the datasets mentioned in references [98, 150, 127], providing a combination of thoroughness and effectiveness in identifying vulnerabilities. We use Slither to label two datasets [9, 8] for DLVA’s training.

So et al. [123] (2020)¹⁹ introduced VERISMART, a dedicated arithmetic safety verifier for Ethereum smart contracts source code. Notably, it leverages an algorithmic approach to automatically identify transaction invariants. These invariants represent properties that hold true under any possible transaction order, enabling VERISMART to analyze contracts exhaustively without individually exploring every program path. This contrasts with traditional methods that often require manual effort to define invariants, making VERISMART more automated and potentially scalable. Unfortunately, our attempts to install and benchmark VERISMART were unsuccessful. Despite reaching out to the authors for guidance, we have not yet received a response. Therefore, we are unable to provide a comprehensive evaluation of its performance compared to other safety analyzers.

Schneidewind et al. [116] (2020)²⁰ introduced eThor, a sound static analysis tool for smart contracts that guarantees the absence of the single-entrancy vulnerabilities. eThor works by translating the low-level bytecode of smart contracts into logical relationships expressed as Horn clauses. These clauses capture the contract’s behavior and allow eThor to formulate reachability queries about security and functional properties. Reachability queries ask whether certain states can be reached during contract execution, and eThor uses the Z3 solver to answer them. eThor is designed to be sound in detecting a specific vulnerability (e.g. single-entrancy), this guarantee comes with the trade-off of focusing on a single vulnerability type and at the

¹⁹So et al. [123] (VERISMART) is available: <http://prl.korea.ac.kr/verismart>

²⁰Schneidewind et al. [116] (eThor) is available: <https://secpriv.wien/ethor/>

cost of potentially raising many false alarms due to its formal constraints. *Our experimentation on eThor discovered a significant number of false positives when compared against a ground truth based on SWC-107. Moreover, eThor considers any contract containing a DELEGATECALL or CALLCODE opcode to be out of scope; in practice, this eliminates many important examples.*

3.2 Learning-based Techniques for SC

This section delves into different machine learning (ML) and deep learning (DL) methods used in smart contract vulnerability detection. Three dominant paradigms in learning-based techniques—machine learning (ML), sequential deep learning (Seq. DL), and graph deep learning (Graph DL)—stand out in their application to the detection of vulnerabilities in smart contracts, each bringing distinct characteristics to the fore.

- Machine learning (ML) constitutes a vast domain involving algorithms and statistical models enabling systems to learn from data and make decisions sans explicit programming. Within smart contract vulnerability detection, ML relies heavily on manual feature engineering, wherein relevant attributes are manually extracted from contract code or related data. Subsequently, these features are employed by diverse algorithms like decision trees (DT), support vector machines (SVM), or random forests (RF) to identify vulnerabilities through learned data patterns.
- Deep learning (DL), a subset of ML, centers on learning data representations through hierarchical layers in neural networks—artificial neural networks. In the sphere of smart contract vulnerability detection, DL branches into two categories: sequential deep learning (Seq. DL) and graph deep learning (Graph DL).
 - Sequential deep learning (Seq. DL) effectively processes sequential tokens prevalent in smart contract code. Models such as Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), or Gated Recurrent Units (GRUs) are commonly employed in DL for analyzing text token sequences in natural language processing. These models can capture

dependencies and patterns within contract code, aiding in identifying potential vulnerabilities.

- Graph deep learning (Graph DL) delves into analyzing complex interconnectedness. Techniques like Graph Neural Networks (GNNs) enable the extraction of insights from intricate network structures. This makes Graph DL well-suited for analyzing the control flow graph of smart contracts, where entities (functions, variables) and relationships (calls, dependencies) form a complex graph/network. By comprehending these intricate connections, Graph DL can uncover vulnerabilities that might evade other detection approaches.

The majority of ML/DL learning-based solutions for smart contract vulnerability detection rely on supervised learning, requiring a substantial dataset of labeled smart contract examples. These methods typically follow a multi-step workflow:

- 1) **Data Collection:** Building a robust vulnerability detection model starts with gathering relevant data. This includes both vulnerable and non-vulnerable smart contracts, serving as positive and negative examples for the model to learn from. Manual labeling of vulnerabilities is often crucial, though static analyzer outputs can also contribute to the labeling process.
- 2) **Data Representation:** Once collected, the data needs to be transformed into a format suitable for ML/DL models. This involves choosing an appropriate representation, such as graphs, trees, or token sequences, that captures the essential structure and semantics of the smart contracts.
- 3) **Embedding:** To bridge the gap between symbolic code and numerical representation, the chosen representation is further converted into vectors or embeddings. These embeddings capture the complex relationships and features within the smart contracts, allowing the models to process and analyze them effectively.
- 4) **Model Selection and Architecture Design:** Choosing the right ML/DL model is critical for success. The optimal choice depends on the specific vulnerability detection task. From simple ML algorithms like Support Vector Machines

(SVM) and Random Forests (RF) to sequential DL architectures like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), or even more advanced DL options like Graph Neural Networks (GNNs), a wide range of models can be considered. The model architecture is then designed to extract relevant features and patterns from the embedded data.

- 5) **Training:** The core of the process, training involves feeding the labeled data into the chosen model. The model iteratively learns from the data, adjusting its internal parameters to minimize prediction errors. Optimization techniques like gradient descent guide this learning process.
- 6) **Evaluation and Validation:** After training, the model’s performance is rigorously evaluated on a separate dataset unseen during training. Metrics like accuracy, precision, recall, false alarm rate, and F1 score assess the model’s ability to correctly identify vulnerabilities. Additionally, validating the model against real-world vulnerabilities further strengthens its practical applicability. *Later, after meticulously training and internally evaluating DLVA, we will take further step of assessing its performance on four independent benchmarks unseen by the model during training.*

By following these steps, ML/DL models can be effectively leveraged to detect vulnerabilities in smart contracts, enhancing the security of blockchain applications and the broader ecosystem. However, they are not without limitations:

- **Supervised Learning Dependence:** A major drawback of supervised learning lies in its reliance on vast amounts of labeled training data. Acquiring such data can be a costly and time-consuming endeavor, especially for niche areas like smart contract vulnerabilities. Furthermore, labeling data can be subjective and prone to errors, potentially leading to biased models with compromised accuracy. This necessitates exploring alternative strategies like semi-supervised learning, which leverages both labeled and unlabeled data, to overcome data scarcity and labeling challenges (*Our SCoolS introduces an interesting approach using semi-supervised learning to solve this limitation*).

- **Computational Cost:** Training complex ML/DL models can be computationally expensive, requiring significant resources and infrastructure. This can pose challenges for smaller scale projects or resource-constrained environments.
- **Explainability and Interpretability:** The complex nature of ML/DL algorithms can make their decision-making process opaque. This lack of interpretability hinders developers' ability to understand why a specific vulnerability was flagged and how to effectively address it. Efforts are underway to develop more transparent models that provide insights into their reasoning, but this remains an ongoing challenge.

Leveraging machine learning and deep learning (ML/DL) models presents a powerful approach to vulnerability detection in smart contracts. These models offer several compelling advantages, which significantly enhance their suitability for this task.

- **Automation:** The ML/DL models ability to automate vulnerability detection holds immense value. ML/DL models can autonomously scan and analyze vast codebases, identifying potential vulnerabilities without requiring human involvement. This automation drastically accelerates the detection process, allowing developers to address security concerns much faster.
- **Performance:** ML/DL models demonstrate impressive performance in terms of analysis speed. Their capacity to process and analyze massive datasets in parallel enables them to generate vulnerability predictions rapidly, dramatically reducing the time needed to secure smart contracts. This efficiency is crucial in the fast-paced world of blockchain, where prompt vulnerability identification can prevent costly exploits and safeguard user trust.

By combining these strengths, ML/DL models empower developers to continuously secure their smart contracts with a high degree of efficiency and accuracy. This paves the way for a more robust and resilient blockchain ecosystem, where innovation and adoption can flourish with greater confidence.

Having explored the landscape of various machine and deep learning techniques for vulnerability detection, along with their benefits and limitations, we delve deeper

into specific tools within each category. Figure 3.2 provides a comprehensive visual overview of 35 primary studies and tools, classified based on four key aspects: tool and dataset availability (indicated by “Available*” with the asterisk denoting potential availability of some studies upon request), analysis levels (“Source code*” encompassing both source code and/or potential alternatives like transactions or account data), and analysis techniques (distinguished by machine learning (ML), sequential deep learning (Seq. DL), and graph deep learning (Graph DL)). Table 3.2 further details the corresponding paper references for each category.

The upcoming subsections will delve into each study sequentially, grouped by their corresponding analysis technique category. This methodical approach ensures a focused and comprehensive exploration of individual tools and their contributions within the field of smart contracts vulnerability detection.

3.2.1 Machine Learning (ML) Studies/Tools:

Gao et al. [50, 49, 51] (2019, 2020)²¹ proposed SMARTEMBED, an automated tool that harnesses code embedding techniques and similarity analysis to identify vulnerabilities and code clones in Solidity smart contracts. This approach, applicable for clone detection, bug detection, and contract validation, targets nine specific vulnerabilities, including overflow, honeypot, and reentrancy. SMARTEMBED functions by first converting source code tokens into numerical vectors using FastText. These vectors are then aggregated into single contract embeddings and stored in a database. Notably, a separate bug database stores embeddings derived from 63 manually labeled buggy statements across 52 contracts. This database, however, might suffer from limited diversity, potentially leading to missed vulnerabilities not present in the training data. During prediction, new contracts are embedded and compared against the bug database. If similarities are detected, SMARTEMBED flags potential issues based on known vulnerabilities.

This publicly available tool allows for community benchmarking and independent evaluation, a significant advantage highlighted by the authors. Our own benchmarking on three benchmarks [13, 7, 11] for reentrancy and integer overflow vulnerabilities revealed that SMARTEMBED exhibits low false alarm rates (0.4%) compared to

²¹Gao et al. [50, 49, 51] (SmartEmbed) is available: <https://github.com/beyondacm/SmartEmbed>

CHAPTER 3. RELATED WORK

Metric	Category	#Studies	References
Tool Availability	Available *	5	[128, 50, 49, 51, 89, 103, 104, 127]
	Not Available	30	[98, 66, 140, 92, 109, 150, 42, 75, 67, 14, 154, 139, 96, 91, 162, 146, 158, 126, 159, 148, 147, 142, 119, 90, 85, 69, 62, 72, 61, 125]
Dataset Availability	Available *	10	[128, 50, 49, 51, 89, 109, 42, 14, 154, 146, 103, 104, 127]
	Not Available	25	[98, 66, 140, 92, 150, 75, 67, 139, 96, 91, 162, 158, 126, 159, 148, 147, 142, 119, 90, 85, 69, 62, 72, 61, 125]
Analysis Levels	Bytecode	11	[128, 140, 92, 14, 96, 158, 126, 147, 69, 62, 72]
	Source code *	24	[50, 49, 51, 98, 89, 66, 109, 150, 42, 75, 67, 154, 139, 91, 162, 146, 159, 148, 142, 119, 90, 85, 61, 103, 104, 127, 125]
Analysis Techniques	ML	11	[50, 49, 51, 98, 89, 66, 140, 150, 42, 67, 158, 147, 119]
	Seq. DL	18	[128, 92, 109, 75, 14, 154, 139, 96, 146, 126, 148, 85, 69, 62, 72, 61, 127, 125]
	Graph DL	6	[91, 162, 159, 142, 90, 103, 104]

Table 3.2: Learning-based Techniques for SC in primary studies; “Available*” with the asterisk denoting potential availability of some studies upon request approval; “Source code*” with the asterisk denoting the analysis level for some studies requires source code and/or potential alternatives like transactions or account data.

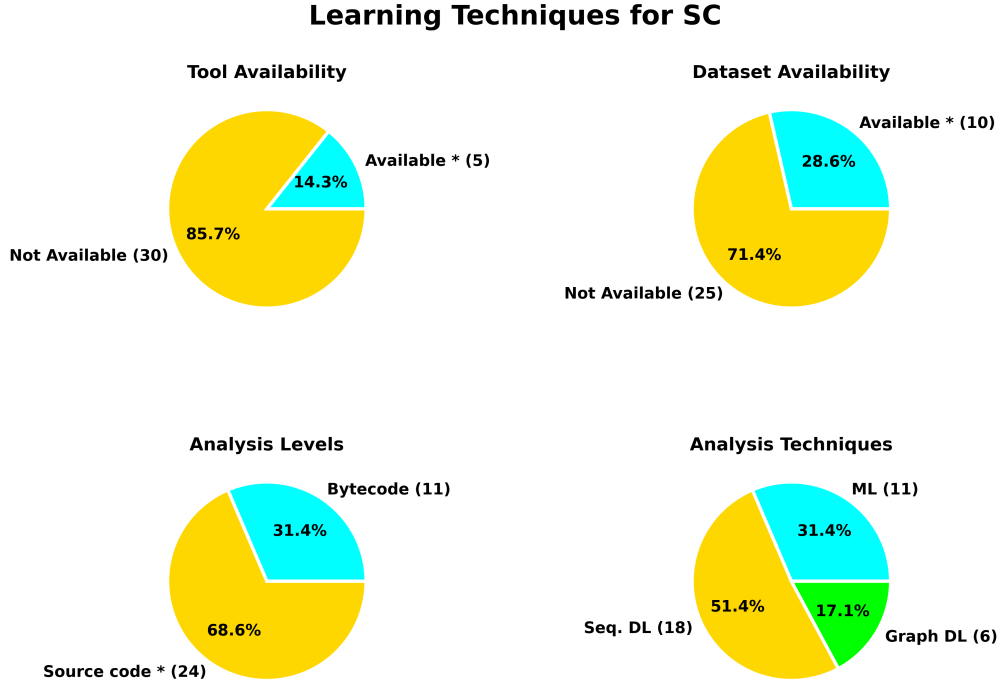


Figure 3.2: Learning-based Techniques for SC in primary studies; “Available*” with the asterisk denoting potential availability of some studies upon request approval; “Source code*” with the asterisk denoting the analysis level for some studies requires source code and/or potential alternatives like transactions or account data.

state-of-the-art tools [93, 134, 105, 136]. However, *it completely missed all reentrancy vulnerabilities in the benchmarks due to the limited size of its bug database.* Expanding and diversifying this database with more examples could significantly enhance SMARTEMBED’s effectiveness in detecting a wider range of vulnerabilities. We will delve deeper into this comparison in the later DLVA chapter of this thesis. In summary, SMARTEMBED represents a valuable tool for smart contract analysis, but further development focusing on diversifying its bug database could unlock its full potential for enhanced vulnerability detection.

Momeni et al. [98] (2019) developed a machine learning model for detecting 16 security vulnerabilities within smart contract source code, including reentrancy and suicide attacks. Their approach relies on extracting 17 predefined features

from the Abstract Syntax Tree (AST) of the code, focusing on Solidity code structures that appeared more than 1000 times within their dataset. Four distinct machine learning classifiers—Support Vector Machines (SVM), Neural Networks (NN), Random Forests (RF), and Decision Trees (DT)—were then employed to identify vulnerabilities.

The authors reported promising results, achieving an average accuracy of 95% in predicting major vulnerabilities and outperforming two static analyzers, Slither and Mythril, in terms of speed. However, the study has limitations that hinder its broader assessment and adoption. First, the artifacts of this work, such as the model, dataset, and source code, are not publicly available. This lack of accessibility prevents independent investigation and benchmarking by the research community, making it challenging to fully evaluate the model’s performance on independent test sets and compare it with other approaches. Second, the dataset used in the study is limited to Solidity version 0.4.18. This raises questions about the model’s generalizability to contracts written in newer Solidity versions, which may introduce new vulnerabilities or code patterns. Despite these limitations, Momeni et al.’s work demonstrates the potential of machine learning techniques for smart contract vulnerability detection. Further research and the release of publicly available artifacts would enable a more comprehensive evaluation of this approach and facilitate its potential adoption in practice. *Our work, DLVA, is inspired by the improvements achieved by Momeni et al., but operates directly on bytecode without requiring source code and utilizes unsupervised feature engineering, thus not relying on predefined features.*

Liao et al. [89] (2019)²² introduced SoliAudit, a smart contract vulnerability assessment tool that leverages both machine learning and fuzz testing. This powerful approach enables detection of 13 vulnerabilities, including reentrancy, access control issues, and denial-of-service attacks. SoliAudit’s strength lies in its combined approach. It uses word2vec and n-gram models with term frequency-inverse document frequency (tf-idf) to effectively represent each opcode in a smart contract as a vector. These vectors are then aggregated to form a comprehensive feature matrix. Subsequently, SoliAudit employs various machine learning classifiers trained on this

²²Liao et al. [89] (SoliAudit) is available: <https://github.com/jianwei76/SoliAudit/tree/master/va>

data to identify vulnerabilities. Furthermore, SoliAudit automates the creation of a fuzzer contract specifically designed for abnormal analysis, conducting in-depth fuzz testing to uncover hidden vulnerabilities. This combination of machine learning and fuzzing provides a powerful and comprehensive detection capability.

Crucially, SoliAudit is publicly available, fostering independent evaluation and benchmarking by the research community. *We evaluated SoliAudit on three benchmarks [13, 7, 11], achieving an average accuracy of 81.9%. A detailed comparison with other tools is presented in our DLVA paper [3], and we will delve deeper into this comparison in the DLVA chapter of this thesis.* SoliAudit’s public availability and impressive performance make it a valuable tool for developers and security researchers alike, advancing the field of smart contract vulnerability detection.

Hao et al. [66] (2020) introduced SCScan, a smart contract vulnerability scanning system that leverages Support Vector Machine (SVM) classifier for detection. SCScan relies primarily on pattern matching against predefined expert rules to identify five specific features: presence of a fallback function; number of lines in the fallback function; number of `<` and `>` characters; number of `call.value`, `send`, and `transfer` functions; and ability for the last user to retrieve the bid amount. Based on these features, SCScan constructs a five-dimensional feature vector and utilizes an SVM classifier to detect six vulnerabilities, including reentrancy and denial-of-service attacks.

While the authors report promising results, the study suffers from several limitations. Firstly, the research artifacts, including the trained model, dataset, and source code, are not publicly available. This lack of accessibility prevents independent investigation and benchmarking by the broader research community, making it difficult to verify the reported performance and compare it with other tools. Secondly, the study lacks a comprehensive comparison with state-of-the-art tools. Without benchmarking against established methods on independent datasets, it’s challenging to assess the relative effectiveness and generalizability of SCScan’s approach. Therefore, while SCScan presents an interesting approach to smart contract vulnerability detection using SVMs, further research and improved transparency are necessary for a more comprehensive evaluation and potential adoption in practical settings.

Wang et al. [140] (2020) proposed ContractWard, a vulnerability detection method for smart contracts, operates by first transforming source code into opcodes.

Opcodes are then simplified by removing operands and grouping functionally similar opcodes. Bigram features are extracted from the simplified opcode fragments, resulting in a 1619-dimensional feature space used for vulnerability identification. ContractWard employs One vs. Rest (OvR) algorithms for multi-label classification within this feature space. Its design targets the detection of six specific vulnerabilities: Integer Overflow, Integer Underflow, Transaction-Ordering Dependence, Callstack Depth Attack, Timestamp Dependency, and Reentrancy.

However, ContractWard faces limitations stemming from the small window size of its bigram language model. This constraint renders it unsuitable for lengthy contracts and hinders its ability to capture long-term dependencies within the code. To effectively address this issue, a larger window size is necessary, but this poses challenges due to the high dimensionality of the data, which can impede model training. Additionally, the authors reported ContractWard’s performance on their test set without conducting comparisons to state-of-the-art tools, raising concerns about its relative performance. Unfortunately, further investigation and benchmarking of ContractWard are obstructed by the lack of publicly available tools, training and testing datasets, and source code.

Eshghie et al. [42] (2021) introduced Dynamit, a novel smart contract vulnerability detection approach that stands out for its reliance solely on publicly available transaction and balance data from the blockchain. Unlike other methods, Dynamit doesn’t require access to the contract source code. It operates by extracting key features from transaction data and training machine learning models (e.g. Naive Bayes (NB), Logistic Regression (LR), K-nearest neighbors (K-NN), SVM, and RF) to classify them as benign or harmful. Notably, Dynamit specializes in detecting reentrancy vulnerabilities, even generating an attack execution trace. The authors meticulously crafted four specific features and trained five different machine learning classifiers, achieving promising results.

However, Dynamit lacks benchmarking against state-of-the-art tools, hindering a comparison of its relative performance. While the source code is available, the end-user tool for wider community testing on independent datasets is missing. This restricted accessibility prevents comprehensive evaluation and hinders practical adoption. *We plan to explore transaction utilization as an anomaly detection technique for vulnerability detection in future work.*

Xu et al. [150] (2021) introduced a novel machine learning-based approach for smart contract vulnerability detection that utilizes shared child nodes within Abstract Syntax Trees (ASTs). This method focuses on predicting eight vulnerability types, including reentrancy, arithmetic errors, access control issues, and denial-of-service vulnerabilities. The core idea lies in analyzing the shared child nodes between the AST of a target smart contract and a set of predefined ASTs representing known malicious contracts. These shared child nodes act as indicators of potential vulnerabilities, and their frequency is quantified to create a feature vector. For instance, if four known reentrancy vulnerabilities share three, five, two, and seven child nodes with the analyzed contract, its feature vector would be (3, 5, 2, 7).

The authors reported promising performance compared to existing analysis tools like Oyente and SmartCheck. However, a crucial limitation hinders further investigation and evaluation: the lack of publicly available artifacts. Neither the tool itself nor the dataset and source code are publicly accessible, preventing independent benchmarking and broader community scrutiny. This lack of transparency restricts comprehensive evaluation of the approach’s strengths, weaknesses, and real-world applicability. Therefore, while Xu et al.’s shared child node analysis presents a promising direction for vulnerability detection, its lack of accessibility hinders its potential for wider adoption and thorough validation.

Zhang et al. [158] (2021) proposed a methodology for detecting Ponzi schemes in smart contracts by extracting distinctive features from contract bytecode and opcodes. They employed the Term Frequency-Inverse Document Frequency (TF-IDF) technique to create a vector space representation of opcode occurrences, enabling the measurement of bytecode similarity between contracts under scrutiny and a predefined set of known Ponzi schemes. This approach aimed to detect Ponzi schemes by identifying resemblances in bytecode structures. Their experiments demonstrated outstanding performance with precision and recall reaching 97%. However, the study lacked comparisons with state-of-the-art tools, and the trained model was not made available for independent benchmarking and further investigation, limiting its generalizability and potential for broader adoption.

Hara et al. [67] (2021) presented a novel machine learning model for identifying honeypot vulnerabilities in smart contracts. The proposed approach leverages TF-IDF and word2vec for feature extraction and representation learning, followed by

XGBoost classification for honeypot detection. Term-frequency inverse document frequency (TF-IDF) identifies prominent keywords within the contract’s bytecode, highlighting potential honeypot patterns. Word2vec then creates distributed representations of these keywords, capturing their semantic relationships and context, enriching the model’s understanding.

Despite achieving promising results, the paper fails to provide crucial comparisons against existing state-of-the-art tools. This omission makes it difficult to assess the model’s true effectiveness and advancement over established solutions. Furthermore, the lack of publicly available artifacts, such as trained model or datasets, prevents independent validation and benchmarking by the research community. This hinders the model’s potential for wider adoption and collaborative improvement. In summary, while Hara et al.’s work offers an interesting exploration of honeypot detection in smart contracts, its limitations in comparative evaluation and open accessibility significantly restrict its broader scientific contribution and practical applicability.

Wu et al. [147] (2022) proposed a smart contract vulnerability detection approach that utilizes opcodes as static features and employs various machine learning algorithms for classification. The method extracts opcode features using both unigram and bigram techniques, generating 141-dimensional and 20087-dimensional feature representations, respectively. These features are then transformed into contract embeddings using the mean term frequency-inverse document frequency (Mtfidf) technique. The model subsequently employs several multi-label classification algorithms, including KNN, DT, RF, CNN, LSTM, CNN-BiLSTM, and ResNets, to identify vulnerabilities.

This approach is specifically designed to detect six types of vulnerabilities: Overflow, Underflow, Call, TOD, Timestamp, and Reentrance. The authors reported promising results, achieving a Macro-F1 score of 82% on their own test set. However, the study’s conclusions are constrained by the lack of benchmarking against state-of-the-art tools, which hinders the assessment of its relative performance. Furthermore, the absence of publicly available artifacts, including the tool itself, datasets, and source code, prevents independent investigation and benchmarking by the broader community. This limited accessibility restricts a comprehensive evaluation of the approach’s strengths and weaknesses, making it challenging to fully understand its practical potential.

Shakya et al. [119] (2022) introduced SmartMixModel, a smart contract vulnerability detection model that leverages an expanded feature space encompassing both source code and bytecode information. This unique approach goes beyond typical methods by combining high-level syntactic features extracted from the Solidity source code (using Code2Vec) with low-level features gleaned from the compiled bytecode (using an n-gram algorithm). This dual-level feature space enables SmartMixModel to capture a more comprehensive picture of the contract, leading to improved vulnerability detection accuracy. Notably, the model targets ten different types of vulnerabilities, expanding its coverage compared to many existing tools.

However, despite its promising capabilities, the study lacks crucial elements for a comprehensive evaluation. The authors solely reported performance based on ground truth data generated by the SmartCheck static analyzer, neglecting comparisons with other state-of-the-art tools. Furthermore, the absence of publicly available artifacts, including SmartMixModel itself, the datasets, and the source code, hinders independent benchmarking and prevents a thorough assessment of its strengths and weaknesses relative to existing approaches. This limited accessibility makes it difficult to fully evaluate SmartMixModel’s potential and effectiveness in real-world applications.

3.2.2 Sequential Deep Learning (Seq. DL) Studies/Tools:

Tann et al. [128] (2018)²³ introduced SaferSC, the first work to leverage Long Short-Term Memory (LSTM) neural networks for analyzing smart contract opcodes and detecting vulnerabilities. Pioneering this approach, SaferSC targeted three specific categories: suicidal, prodigal, and greedy contracts. However, the paper ultimately combined these vulnerabilities into a single “vulnerable” vs. “non-vulnerable” classification. Despite not achieving perfect granularity, SaferSC still showcased significant accuracy improvements compared to the symbolic analysis tool Maian. However, LSTMs have limitations. While excelling at sequence learning, they struggle with control-flow vulnerabilities like reentrancy, which exploit recursive function calls to manipulate contract behavior and steal funds. Unfortunately, LSTMs’ focus on sequential patterns makes them ill-equipped to capture these

²³Tann et al. [128] (SaferSC) is available: <https://github.com/wesleyjtann/Safe-SmartContracts>

intricate control-flow issues, hindering their effectiveness in detecting and addressing such vulnerabilities.

Crucially, SaferSC stands out for its public availability, enabling benchmarking and independent evaluation by the research community. *Inspired by this transparency, we followed suit and made all our tools publicly accessible. Interestingly, we benchmarked two pre-trained SaferSC models: “LSTM” and “Improved_LSTM” using our Elysium benchmark [7]. While the “LSTM” model failed spectacularly, classifying every contract as suicidal, the “Improved_LSTM” model demonstrated significant progress with an accuracy of 91.9%.* This discrepancy warrants further investigation into potential overlap between the SaferSC training set and the Elysium benchmark. Notably, SaferSC’s reported 99.57% accuracy significantly outperformed Maian’s 89%, but our results highlight the importance of robust benchmarks in evaluating true performance. We will delve deeper into this comparison in the later DLVA chapter of this thesis. Nevertheless, the SaferSC team has done the right thing releasing their tool as it allows for a more scientific understanding of their technique’s accuracy and performance.

Qian et al. [109] (2020) proposed a novel approach for detecting only one type of vulnerabilities (e.g., reentrancy) in smart contracts, known as BLSTM-ATT. This model leverages various deep learning architectures, including GRU, LSTM, BLSTM, and LSTM-Attention, alongside N-grams and word2vec word embeddings. The key innovation of BLSTM-ATT lies in its code snippet condensation technique. It identifies semantically related sections of the source code, focusing specifically on snippets involving “call.value”, a crucial element of reentrancy vulnerabilities. This condensed representation allows the model to focus on relevant code sections, potentially improving its detection accuracy.

The authors reported promising results, demonstrating superior performance compared to four established reentrancy detection tools: Oyente, Mythril, Securify, and SmartCheck. However, our attempt to evaluate BLSTM-ATT further encountered limitations: While the source code is available, crucial files for building and training the models as a practical tool are missing. This lack of readily available tools hinders independent benchmarking and wider adoption. Attempts to contact the authors for assistance with the missing artifacts have been unsuccessful, further hindering our evaluation efforts.

Despite these limitations, Qian et al.’s work demonstrates the potential of deep learning for smart contract vulnerability detection, particularly with the innovative code snippet condensation technique. However, the lack of accessible artifacts and limited communication from the authors significantly hinder a comprehensive assessment and practical deployment of this promising approach.

Lou et al. [92] (2020) took a unique approach to identify Ponzi schemes within smart contracts by building a dedicated Convolutional Neural Network (CNN) model. Their method stands out for its preprocessing step, transforming the contract’s bytecode into visual representations for the CNN to analyze. This involves: converting the hexadecimal bytecode to its decimal equivalent, standardizing the decimal values for consistent scaling, and generating images from the standardized values, leveraging the CNN’s ability to learn features from visual data.

The model performs well on a dataset of 3774 samples, achieving a promising F-score of 95.9% according to the reported results by the authors. However, the study has limitations that hinder a definitive evaluation: The CNN’s performance is not benchmarked against other established Ponzi scheme detection methods, making it difficult to assess its relative effectiveness. The model, dataset, and source code are not publicly available, preventing independent investigation and further research by the broader community. The relatively small dataset raises questions about the model’s generalizability and ability to detect diverse Ponzi schemes in different contract structures. While the reported results are encouraging, further research with larger datasets, comparisons to existing tools, and public accessibility of artifacts are necessary to fully assess the potential of this novel CNN for real-world Ponzi scheme detection in smart contracts.

Wu et al. [146] (2021) introduced Peculiar, a smart contract vulnerability detection model utilizing a pre-trained language models and focusing on analysis of crucial data flow graphs (CDFGs). CDFGs are subgraphs of data flow graphs (DFGs) containing critical information potentially leading to vulnerabilities. Peculiar leverages this focused analysis, capturing essential data flow information directly from the CFG instead of explicitly outlining the entire program as a graph. This approach balances feature preservation for model generalization across diverse contracts while maintaining efficiency. The model operates by first extracting the DFG and CDFG from the source code’s abstract syntax tree (AST) and then performing vulnera-

bility detection based on its pre-trained architecture. Notably, Peculiar currently targets only specific vulnerabilities (e.g., reentrancy) but demonstrates the authors’ dedication to meticulous dataset labeling efforts.

Peculiar is currently unavailable as a ready-to-use tool, and our attempt to replicate Peculiar’s results encountered obstacles . *While the source code is available, we use it to reproduce the reported results, but training the model beyond the first epoch proved unsuccessful. Despite contacting the authors for assistance, we haven’t received a response at the time of writing.* Further investigation is necessary to diagnose and overcome this training hurdle.

Sun et al. [126] (2021) proposed a novel method for smart contract vulnerability detection, targeting three specific types: reentrancy, arithmetic errors, and time manipulation. This approach leverages a Convolutional Neural Network (CNN) fused with a self-attention mechanism for improved accuracy and efficiency. The process starts by transforming each opcode instruction into a 78-dimensional vector using one-hot encoding. Subsequently, the self-attention mechanism extracts a feature vector representing the entire contract, capturing key relationships between instructions. This allows the CNN to learn complex patterns indicative of vulnerabilities within the contract’s code.

Sun et al. report promising results, showcasing lower missing rates and faster detection times compared to two popular static analysis tools, Oyente and Mythril. However, a crucial limitation of this work is the lack of publicly available artifacts. To fully assess the approach’s strengths and weaknesses, the broader research community requires access to the model, datasets, and source code used in the study. Without such accessibility, independent investigation and benchmarking remain impossible, hindering the potential for further development and adoption of this promising vulnerability detection technique.

Jeon et al. [75] (2021) presented SmartConDetect, a vulnerability detection tool designed to identify security flaws in Solidity smart contracts. It employs a pre-trained BERT model to extract code segments and pinpoint vulnerable patterns. SmartConDetect targets a broad range of 23 vulnerability classes, utilizing SmartCheck’s outputs as ground truth for training. The process operates as follows: Each function within a contract is encoded using BERT, resulting in a 768-dimensional vector representation that captures its semantic meaning. These

vectors are then fed into a fully connected layer, which condenses the embedding output for further processing. The softmax function is employed as the activation mechanism, and the binary cross-entropy function is used to classify the presence or absence of vulnerabilities within the analyzed code.

The authors reported an impressive F1-Score of 90.9% for SmartConDetect analysis of smart contract source code, indicating a high degree of both precision and recall in vulnerability detection. The tool’s ability to address a wide spectrum of 23 vulnerability classes further highlights its potential versatility. However, a significant limitation of this work is the lack of publicly available artifacts, such as the model, training, and testing datasets. This lack of accessibility prevents independent evaluation and benchmarking by the broader research community, making it challenging to verify the reported performance and compare it with other approaches. Without such transparency and reproducibility, the adoption of SmartConDetect in practical settings remains hindered.

Wang et al. [139] (2021) proposed AFS, a model that carefully analyzes both structure and semantics within smart contracts to detect vulnerabilities. It accomplishes this through a unique combination of techniques, including Abstract Syntax Tree (AST) serialization, program slicing, word2vec representation, and Long Short-Term Memory (LSTM) networks. AFS begins by translating a contract’s functions into ASTs, which represent the code’s structural organization. A depth-first search then traverses these ASTs, ensuring that any vulnerability-related structural information is preserved during serialization. To extract the semantic meaning embedded within the code, AFS applies program slicing techniques, isolating and focusing on code segments that are potentially relevant to vulnerabilities. The word2vec method, commonly used in natural language processing, is then employed to capture a comprehensive feature representation of the code, effectively encoding both structural and semantic information. Finally, AFS utilizes LSTM and BLSTM-ATT (Bidirectional LSTM with Attention Mechanism), two neural network architectures, to analyze the extracted features and semantic information, classifying the code as either vulnerable or non-vulnerable.

However, AFS faces limitations in terms of resource availability and dataset accessibility. Crucial resources like source code, datasets, and trained models are not publicly available, hindering independent evaluation and benchmarking by the

wider research community. Additionally, while the authors claim significant efforts in manually labeling their dataset, it remains inaccessible for further investigation and validation of their results. The authors reported AFS’s performance on their test set without conducting comparisons to state-of-the-art tools, raising concerns about its relative performance.

Ashizawa et al. [14] (2021) introduced Eth2Vec, a machine learning-based static analysis tool for smart contract vulnerability detection. Its unique approach involves learning from smart contract source code through their EVM bytecode, assembly code, and abstract syntax trees. Eth2Vec identifies vulnerabilities by comparing the code similarity between a target contract’s EVM bytecode and the ones it has already learned. The tool operates through two key modules: the EVM Extractor, which prepares inputs for the PV-DM model by processing Solidity source code, and the PV-DM model itself. This unsupervised neural network excels at paragraph-level text processing, transforming code data into vector representations that capture both individual words and entire paragraphs. These vectors then become the fuel for Eth2Vec’s detection of both code clones and vulnerabilities.

However, despite its potential, Eth2Vec faces some significant limitations. Notably, it disregards the valuable information encoded within a program’s graph structure, treating the code purely as textual data. Additionally, *its ability to generalize beyond its original training dataset is limited, as reported by its authors, raising concerns about its practical effectiveness in real-world scenarios.* Furthermore, while the paper offers a link to its implementation code, the provided repository lacks a complete and user-friendly tool for end-user deployment. Attempts to independently build and train the tool have been hampered by missing files within the codebase. Finally, efforts to contact the authors for clarification and assistance have not yielded any response.

In summary, while Eth2Vec’s code similarity and vector representation approach holds promise for smart contract vulnerability detection, addressing its limitations in graph structure utilization, generalization capabilities, tool accessibility, and author responsiveness is crucial for its practical impact and wider adoption within the field of smart contract security. *In a comparison, our tools and datasets are readily available and perform effectively on multiple benchmarks disjoint from training sets.*

Yu et al. [154] (2021) introduced DeeSCVHunter, a model designed to enhance

deep learning-based smart contract vulnerability detection by leveraging data and control dependencies within the source code. However, DeeSCVHunter focuses solely on two specific vulnerabilities: reentrancy and time dependence. To achieve this, DeeSCVHunter dissects the smart contract into smaller code segments and generates “vulnerability candidate slices” (VCS) based on specific match patterns. For example, “call.value” and “block.timestamp” patterns respectively target reentrancy and time dependence vulnerabilities. DeeSCVHunter then expands these VCS by tracing data flows and control flows from the matched statements, gathering relevant code segments that might contribute to the potential vulnerability.

This reliance on specific pattern matching poses a significant limitation. Expanding DeeSCVHunter’s scope to detect other vulnerabilities would require identifying unique patterns for each, potentially demanding considerable effort and expertise. Further hindering broad adoption, DeeSCVHunter’s availability remains shrouded in uncertainty. While promising public access, the currently available repository only offers data pre-processing code, lacking the crucial training code and final tool. Despite author confirmation that the repository will be updated and fully open-sourced, this has not yet materialized as of now. Additionally, attempts to contact the authors have yielded no response.

In summary, while DeeSCVHunter’s approach based on vulnerability-specific pattern matching and deep learning holds promise for certain targeted vulnerabilities, its limitations in coverage and accessibility significantly hinder its broader impact. Addressing these limitations, through pattern development for additional vulnerabilities and complete open-sourcing of the tool, is crucial for DeeSCVHunter to realize its full potential in detecting smart contract vulnerabilities.

Mi et al. [96] (2021) introduced VSCL, a model for detecting vulnerabilities in smart contracts bytecode. Its distinctive feature lies in prioritizing code structure by meticulously reordering bytecode sequences. This reordering, guided by control flow graphs (CFGs) and depth-first search (DFS) algorithms, aims to elevate the most critical code segments for vulnerability analysis. VSCL’s journey begins by translating the smart contract’s bytecode, a low-level representation, into a sequential format. Then, it constructs CFGs, and by utilizing DFS, VSCL meticulously traverses these CFGs, rearranging the bytecode sequence. This reordered sequence of tokens is then transformed into a numerical feature vector using the n-gram/TFIDF technique,

capturing its key characteristics. Finally, a deep neural network analyzes this vector, classifying the contract as either vulnerable or non-vulnerable. *It’s noteworthy that in contrast to VSCL’s focus on bytecode sequence reordering, our tools harness the power of graph neural networks to directly extract valuable features from the CFG structure, offering a distinct approach to vulnerability detection.*

While VSCL’s approach shows promise, limitations hinder its practical impact. Notably, VSCL only detects vulnerabilities without specifying their types. This lack of specificity can impede developers’ ability to prioritize and address critical issues, as some vulnerabilities pose significantly higher risks than others. Additionally, the authors reported VSCL’s performance on their test set without conducting comparisons to state-of-the-art tools, raising concerns about its relative performance. Furthermore, crucial resources for independent evaluation and research, including source code, training and testing datasets, trained models, and an executable version of the tool, are not publicly available. This lack of transparency raises concerns about the reproducibility and validity of VSCL’s results. Additionally, attempts to contact the authors for clarification, access to tools, and further information regarding training methods and dataset labeling have not yielded any response.

In summary, VSCL’s focus on code structure through bytecode reordering presents a potentially valuable approach to smart contract security. However, addressing the limitations in vulnerability specificity, resource availability, and author communication is crucial for VSCL to realize its full potential and contribute meaningfully to safeguarding the smart contract landscape.

Guo et al. [61] (2022) proposed SCVSN, a Siamese network model for detecting only one type of vulnerabilities (e.g., reentrancy) in smart contracts. This model operates by comparing the similarity of two contracts: a sample contract under scrutiny and a set of predefined “bug-vectors” representing known vulnerabilities. The process involves extracting a feature vector from the sample contract and calculating its Euclidean distance to each bug-vector. If the distance to any bug-vector falls below a predetermined threshold, the sample contract is flagged as vulnerable to the corresponding reentrancy vulnerability.

While the authors reported promising performance results and provided a link to their artifacts, our investigation revealed several concerns. The link to the supposed GitHub repository (<https://github.com/xiaoaochen/SCVSN>) is non-existent,

suggesting a lack of transparency and accessibility of the proposed method. Additionally, the reported comparisons with other tools seem to be based on secondary sources rather than direct experimentation, raising questions about the validity and objectivity of the performance claims. Ultimately, the absence of readily available artifacts and independent verification hinders the broader community’s ability to assess and validate the effectiveness of SCVSN. This lack of transparency and independent benchmarking significantly limits the trustworthiness and potential for wider adoption of the proposed approach.

Xu et al. [148] (2022) proposed HAM-BiLSTM, a method for detecting reentrancy vulnerabilities in smart contracts that integrates both source code and account information. The model employs a sequential neural network with an attention mechanism, designed to specifically identify reentrancy vulnerabilities. HAM-BiLSTM operates by first converting the contract source code and account information into word vectors using the word2vec model. These vectors are then fed into a bidirectional LSTM (BiLSTM) network for classification. The attention mechanism within the network enables it to focus on the most relevant parts of the input data for vulnerability detection.

While the authors reported promising performance results compared to three state-of-the-art tools (Oyente, Osiris, and Mythril), the study has limitations that hinder its generalizability. The ground truth of the test set used for evaluation is not publicly known, making it difficult to assess the accuracy of the results independently. Additionally, the lack of publicly available artifacts, including the tool itself, datasets, and source code, prevents further investigation and benchmarking by the broader research community. This lack of transparency and accessibility restricts the comprehensive evaluation of HAM-BiLSTM’s strengths and weaknesses, ultimately limiting the assessment of its practical potential.

Li et al. [85] (2022) proposed Link-DC, a vulnerability detection model for smart contracts that leverages both structural and pattern-based features. The model operates through two key components: a contract graph built from the smart contract’s control and data flow, and expert-defined pattern features. The contract graph serves as a structural representation of the contract, where key variables and function calls act as nodes connected by edges reflecting control and data flow. One-hot encoding is applied to both nodes and edges to create a graph embedding.

Meanwhile, expert knowledge is incorporated through sub-pattern one-hot encoding, capturing crucial conditions often associated with vulnerabilities. These encoded patterns are then combined with the graph embedding to form a comprehensive feature vector for vulnerability detection using a Multilayer Perceptron (MLP) classifier.

Link-DC’s detection capabilities are focused on three specific vulnerabilities: reentrancy, timestamp dependence, and infinite loops. While the authors reported high performance compared to CGE through cited results [91], they conducted no direct comparisons with state-of-the-art tools. Furthermore, the lack of publicly available artifacts, including the tool itself, datasets, and source code, hinders independent investigation and benchmarking, limiting a thorough evaluation of Link-DC’s strengths and weaknesses.

Hu et al. [69] (2022) designed SCSGuard, a deep learning framework specifically aimed at detecting scam smart contracts, including honeypots, Ponzi schemes, and phishing scams. This approach leverages the power of n-gram features and an attention-based neural network to analyze smart contract bytecodes and effectively identify potential phishing scripts. SCSGuard works by first slicing the bytecode into a series of two-character bytes, each representing specific features of the contract. These bytes are then transformed into vector representations using an n-gram method, where each distinct combination of n consecutive bytes becomes a unique feature. Finally, these bytecode n-grams are converted into vectors and fed into an attention-based neural network for classification.

While the reported accuracy and performance of SCSGuard appear promising, several limitations hinder further evaluation and comparative analysis. Firstly, the study lacks any comparison or benchmarking with established state-of-the-art tools, making it difficult to assess its relative effectiveness. Additionally, the authors haven’t made SCSGuard’s artifacts publicly available, including the tool itself, datasets, and source code. This limited accessibility impedes independent investigation and benchmarking, preventing a more comprehensive understanding of its strengths and weaknesses.

Gupta et al. [62] (2022) introduced a novel approach for smart contract categorization that incorporates a reward-penalty system based on security assessments. Their methodology involves converting contract opcodes into one-hot encoded vectors,

which are subsequently transformed into feature vectors of length 256, corresponding to the possible opcodes. Each entry in the vector represents the count of the respective opcode within the contract. The generated feature vectors are then fed into deep learning models, such as artificial neural networks (ANN), long short-term memory (LSTM), and gated recurrent unit (GRU) models, for classification purposes in the prediction layer.

However, despite the reported high accuracy, the study’s findings are limited by several factors. The authors did not conduct comparisons or benchmarking with state-of-the-art tools, hindering the assessment of its relative performance. Additionally, the approach treats contract bytecode solely as a text sequence, disregarding its execution flow, which may overlook potential vulnerabilities. Furthermore, the method only provides a binary classification of secure or malicious, without identifying specific vulnerability types. Further investigation and benchmarking are impeded by the unavailability of tools, datasets, or source code.

Hwang et al. [72] (2022) proposed CodeNet, a novel approach to smart contract vulnerability detection that takes a unique visual route. Instead of relying on traditional text analysis, CodeNet transforms smart contracts into images, capturing their structure and logic while preserving local context. This visual representation allows CodeNet to leverage the power of convolutional neural networks (CNNs) for vulnerability identification. The transformation process involves three steps: first, the smart contract source code is compiled into bytecode, a lower-level instruction set. Then, the bytecode is adjusted to a fixed size to ensure consistency in image formation. Finally, each bytecode instruction is meticulously mapped to a specific RGB pixel value, creating a contract-specific image. This image, fed into the CNN model, becomes the basis for vulnerability detection.

However, our investigation revealed some limitations. Currently, CodeNet targets only four specific vulnerabilities: reentrancy, unchecked low-level calls, tx.origin, and timestamp dependency. This limited scope restricts its overall effectiveness. Additionally, the reported results haven’t been validated using real-world vulnerable contracts, leaving questions about its practical performance. Furthermore, the tool, source code, and datasets remain unavailable, hindering independent evaluation and further research.

In summary, while CodeNet introduces a promising visual-based approach to

smart contract vulnerability detection, addressing its limitations in scope, real-world validation, and resource availability is crucial for its practical adoption and advancement. Expanding its vulnerability detection capabilities, validating its performance on real-world contracts, and making its resources readily available would significantly enhance CodeNet’s potential in securing the smart contract landscape.

Sun et al. [125] (2023) proposed a novel framework, ASSBert, that combines active and semi-supervised learning techniques to enhance smart contract vulnerability detection. It targets six specific vulnerabilities, including reentrancy and timestamp dependence. ASSBert combines the strengths of active and semi-supervised learning to improve its effectiveness. Active learning identifies uncertain segments from unlabeled Solidity files, manually annotates them, and adds them to the training set. This process refines the model’s understanding of vulnerable code patterns. Semi-supervised learning further leverages unlabeled data by selecting high-confidence code segments and assigning pseudo-labels based on the model’s predictions. These pseudo-labeled segments are then incorporated into the training set, further enhancing the model’s performance.

However, active learning’s dependence on manual annotation can be a bottleneck. *Addressing this limitation, our concurrent work, SCooLS (2023), employs an automated committee voting strategy. This eliminates the need for manual annotation by leveraging multiple models to vote on uncertain code, achieving comparable accuracy without human intervention. SCooLS further innovates by going beyond mere vulnerability detection. It generates realistic attack scenarios to exploit the identified bugs, providing richer insights for developers and security professionals.*

While Sun et al. report promising improvements compared to baseline language models, a comprehensive evaluation remains difficult. The lack of public availability of artifacts including the trained model and datasets prevents benchmarking against state-of-the-art tools and independent investigation. Overall, Sun et al.’s work shows promise in its novel approach to smart contract vulnerability detection. However, wider accessibility and rigorous comparison with existing tools are crucial for a definitive assessment of ASSBert effectiveness and practical applicability.

Tang et al. [127] (2023)²⁴ presented Lightning Cat, a novel model that demonstrates significant advancements in vulnerability detection within smart contracts. Notably, it leverages the pre-trained CodeBERT model for data preprocessing, enhancing its ability to comprehend source code semantics. This model effectively addresses a common challenge faced by large language models (LLMs), namely the limitation in processing lengthy texts. Lightning Cat accepts Solidity functions in text format, conducts thorough analysis, and accurately identifies potential vulnerabilities.

However, at the time of writing, accessing the tool itself requires contacting the authors, who haven't yet responded. *Building upon these advancements in LLMs, our future research will explore LLM techniques for vulnerability detection directly within bytecode, further expanding the scope and effectiveness of smart contract security.*

3.2.3 Graph Deep Learning (Graph DL) Studies/Tools:

Zhuang et al. [162] (2021) investigated the potential of graph neural networks (GNNs) for smart contract vulnerability detection. Their approach involves constructing a “contract graph” that encodes both syntactic and semantic information from the source code. Nodes in this graph represent key elements like function calls and variables, while edges capture their temporal execution relationships. To prioritize crucial nodes, the authors propose an elimination phase for graph normalization. Subsequently, they introduce two dedicated models for vulnerability detection: a degree-free graph convolutional neural network (DR-GCN) and a temporal message propagation network (TMP). Both models operate on the normalized graph to identify vulnerabilities.

Our examination revealed limitations in the scope and generalizability of this approach. The model currently targets only three specific vulnerabilities (reentrancy, timestamp dependence, and infinite loop). Furthermore, *the method for identifying critical nodes lacks generalizability and necessitates manual customization for different vulnerabilities.* For instance, detecting reentrancy relies on recognizing invocations of “transfer” and “call.value” functions, while timestamp dependence

²⁴Tang et al. [127] (Lightning Cat) is available: upon reasonable request it could be provided.

focuses on the “block.timestamp” function, and infinite loop detection treats all custom functions as crucial nodes. Despite this limited scope, the model achieves an average F1 score of 77% across these three vulnerabilities. *We hypothesize that this performance limitation stems from the graph normalization process potentially eliminating malicious nodes crucial for vulnerability exploitation.* In contrast, *our models excel in their ability to prioritize the most significant nodes within the CFG, extracting the final contract or function embedding without relying on any predefined elimination patterns. This distinction liberates our models from the constraints that hinder the extension of Zhuang et al.’s approach to a wider array of vulnerabilities.*

Unfortunately, the paper lacks readily available resources for further exploration. Despite finding multiple GitHub repositories containing source code, we couldn’t locate pre-trained models or deployable tools. Our attempts to contact the authors for clarification and access to tools haven’t yielded any response at the time of writing. Further investigation is required to address these limitations and assess the full potential of this GNN-based approach for smart contract vulnerability detection.

Liu et al. [91] (2021) present a novel smart contract vulnerability detection system from source code that leverages both graph neural networks and expert knowledge. They propose a temporal message propagation network (TMP) to extract features from a normalized contract graph, similar to Zhuang et al. [162] (2021) due to shared authorship. This extracted feature is then combined with pre-defined expert patterns to form the final detection system. Notably, Liu et al. build upon their previous work (Zhuang et al. [162], 2021) where they explored different GNN architectures, including TMP. Here, they focus on enhancing TMP’s performance by integrating it with expert patterns. This combined approach specifically targets the same three vulnerabilities: reentrancy, timestamp dependence, and infinite loops.

The paper details the implementation of pattern extraction, highlighting both simple and complex methods. Simple patterns like function calls and loop statements are identified through keyword matching, while others like balance deductions and timestamp assignments involve syntax analysis. For the complex “timestamp contamination” pattern, taint analysis is employed to track data flow and identify potentially affected variables. The final vulnerability detection integrates graph features from critical variables and function calls with expert-defined security patterns. This combined approach offers potentially improved accuracy compared to

solely relying on GNNs. *However, it’s crucial to acknowledge that the dependence on expert-defined security patterns could necessitate considerable effort for expansion to encompass the vast array of vulnerabilities addressed by our tools.*

However, despite the promising approach, our evaluation encountered limitations because readily available tools for benchmarking and deployment are missing. While the source code is available, it lacks crucial files necessary for building and training the models or deploying them as a tool. Unfortunately, our attempts to contact the authors for assistance haven’t been successful. In summary, Liu et al.’s work presents a promising direction for integrating expert knowledge with GNNs for smart contract vulnerability detection. However, the lack of readily available tools and unresponsive authors hinder further assessment and potential real-world application.

Zhang et al. [159] (2022) presented two machine learning techniques, ASGVulDetector and BASGVulDetector, for pinpointing four specific vulnerabilities (reentrancy, timestamp dependency, block info dependency, and tx.origin) in smart contracts. Their approach analyzes contracts from both source code and bytecode perspectives. The key innovation lies in the “abstract semantic graph” (ASG), designed to capture the syntax and semantics of smart contract code. For source code analysis, ASGVulDetector builds upon the traditional Abstract Syntax Tree (AST) by incorporating control and data flow information. When source code is unavailable, BASGVulDetector constructs the ASG by decompiling and enriching basic block sequences with control flow data. This ASG serves as the input for graph neural networks (GNNs) and graph matching networks (GMNs), which assess contract similarities. By comparing contracts to labeled vulnerable ones, these networks can identify potential vulnerabilities. The process involves multi-layer perceptrons (MLPs) to encode node and edge features, followed by a gated graph neural network (GGNN) for graph embedding.

However, a critical limitation lies in the dependency on a limited set of labeled vulnerable contracts. Zhang et al.’s model relies on a predetermined similarity threshold of 0.85 for vulnerability detection. This means any new contract with vulnerabilities, especially those dissimilar to the established dataset, will likely go undetected. In contrast, our DLVA model bypasses this limitation, expertly analyzing and judging contracts even if their similarity to known vulnerabilities falls below the predefined threshold.

While Zhang et al. reported promising performance, all artifacts related to their work, including the trained models, remain unavailable, hindering independent benchmarking and further investigation by the research community. This lack of transparency limits the broader adoption and generalizability of their proposed methods.

Nguyen et al. [103, 104] (2022)²⁵ introduced MANDO-GURU, a deep learning tool utilizing control-flow and call graphs from Solidity source code to detect seven vulnerabilities, including reentrancy and time manipulation. Its novel aspect lies in heterogeneous graph attention neural networks that capture complex relationships within these graphs, enabling vulnerability detection. MANDO-GURU builds heterogeneous contract graphs merging the graphs to capture Solidity’s unique features and leverage specialized metapaths to construct its neural networks. These networks learn multi-tiered embeddings of contracts, enabling vulnerability recognition in new code.

While MANDO-GURU exhibits promise, concerns arise regarding its accessibility and generalizability. *The training data for each vulnerability is limited (less than 200 samples), potentially hindering the model’s performance on larger datasets.* Furthermore, the tool’s current availability is restricted – accessible only through a web-based system and potentially via an API, though the latter remains untested and the authors haven’t provided further assistance. Consequently, we were unable to benchmark MANDO-GURU against our datasets. Instead, *we test our own tool, DLVA, on MANDO-GURU’s dataset for the shared vulnerabilities (reentrancy and time manipulation). DLVA achieved superior Macro-F1 scores (91.5% and 95%, respectively) compared to MANDO-GURU’s reported scores (80.78% and 86.76%).*

While this suggests promise for DLVA, limitations remain. MANDO-GURU’s paper doesn’t clarify which dataset subsets were used for training and testing, hindering a more accurate comparison. Therefore, running DLVA on the entire MANDO-GURU dataset provides only partial insight. In summary, MANDO-GURU exhibits promising features for smart contract vulnerability detection. Our initial comparison using DLVA shows promise, further investigation with improved transparency from MANDO-GURU’s authors is crucial for a comprehensive evaluation.

²⁵Nguyen et al. [103, 104] (MANDO-GURU) is available: <http://mandoguru.com/>, this web-based system allows to test the contracts one by one.

Wang et al. [142] (2022) developed GVD-net, a vulnerability detection method specifically designed for Solidity smart contracts. It operates by first constructing a Control Flow Graph (CFG) that captures the relationships between variables and function calls within the source code. This CFG serves as a blueprint to create a non-Euclidean graph representing the contract’s structural features. Subsequently, GVD-net employs the Node2Vec graph embedding algorithm to generate a 256-dimensional vector, encoding the contract’s characteristics. This vector becomes the basis for conducting similarity analysis and identifying potential vulnerabilities.

GVD-net focuses on detecting three specific vulnerability types: arithmetic issues, access control flaws, and asset freezing vulnerabilities. The authors reported promising results, demonstrating GVD-net’s effectiveness in comparison with well-known tools like Oyente and Manticore. However, the lack of publicly available artifacts, including the tool itself, datasets, and source code, prevents independent benchmarking and evaluation on an independent test set. This limited accessibility hinders a thorough assessment of GVD-net’s strengths and weaknesses relative to other approaches, making it difficult to fully gauge its practical potential.

Liu et al. [90] (2022) proposed S_HGTNs, a Heterogeneous Graph Transformer Network framework designed for smart contract anomaly detection on the Ethereum platform, specifically targeting financial fraud. This approach leverages a rich set of contract information by incorporating both transaction data and source code features. The source code undergoes preprocessing and Doc2Vec embedding to translate it into vector representations. S_HGTNs operates by constructing a Heterogeneous Information Network (HIN) that captures diverse relationships between nodes in the network. It utilizes the concept of meta-paths, essentially multi-hop connections among nodes, to effectively navigate the HIN’s complex structure. This allows the model to extract comprehensive information from the network and represent it as a meta-path network. Finally, S_HGTNs leverages the learned node embeddings to perform the anomaly detection task.

While the classification results demonstrate the effectiveness and stability of S_HGTNs, the study’s conclusions are limited by two key factors. Firstly, the authors primarily compared S_HGTNs to various neural network variants, lacking direct comparisons with existing state-of-the-art anomaly detection tools. Additionally, the absence of publicly available artifacts, including the tool itself, datasets, and

source code, prevents independent investigation and comprehensive benchmarking. *Furthermore, the lack of common vulnerabilities shared with our tools precludes comparative analysis.* This limited accessibility hinders a thorough evaluation of S_HGTNs’s strengths and weaknesses relative to other approaches, making it difficult to fully assess its potential in real-world applications.

3.3 Learning-based Techniques for PL

This section explores the evolution of learning-based methods for vulnerability detection in mainstream programming languages (PL). Recent years have seen the emergence of learning-based models as powerful tools for detecting vulnerabilities in the source and binary code of general programming languages like C/C++, Python, and Java. These models excel at identifying subtle patterns and correlations within large datasets, allowing them to automatically extract meaningful features from raw code and pinpoint hidden patterns indicative of vulnerabilities. This capability is invaluable in vulnerability detection, as vulnerabilities often manifest through intricate code characteristics and dependencies.

Unlike the smart contracts domain, which lacks readily available and stable datasets for benchmarking and research, the field of general programming languages (PL) boasts a wealth of resources. This abundance of data facilitates the development and deployment of increasingly sophisticated models.

Figure 3.3 visually summarizes 12 key studies and tools, categorized by four crucial aspects: tool availability, dataset availability, analysis level (source code or binary), and analysis technique (sequential deep learning or graph deep learning). Table 3.3 provides corresponding paper references for each analysis technique.

The following subsections will delve deeper into each study, grouped by their analysis technique. This methodical approach ensures a focused exploration of individual tools and their contributions to the field of general programming languages (PL) vulnerability detection.

3.3.1 Sequential Deep Learning (Seq. DL) Studies/Tools:

Phan et al. [108] (2017) proposed a novel two-step method for automatically discovering software defects. The first step involves constructing Control Flow Graphs

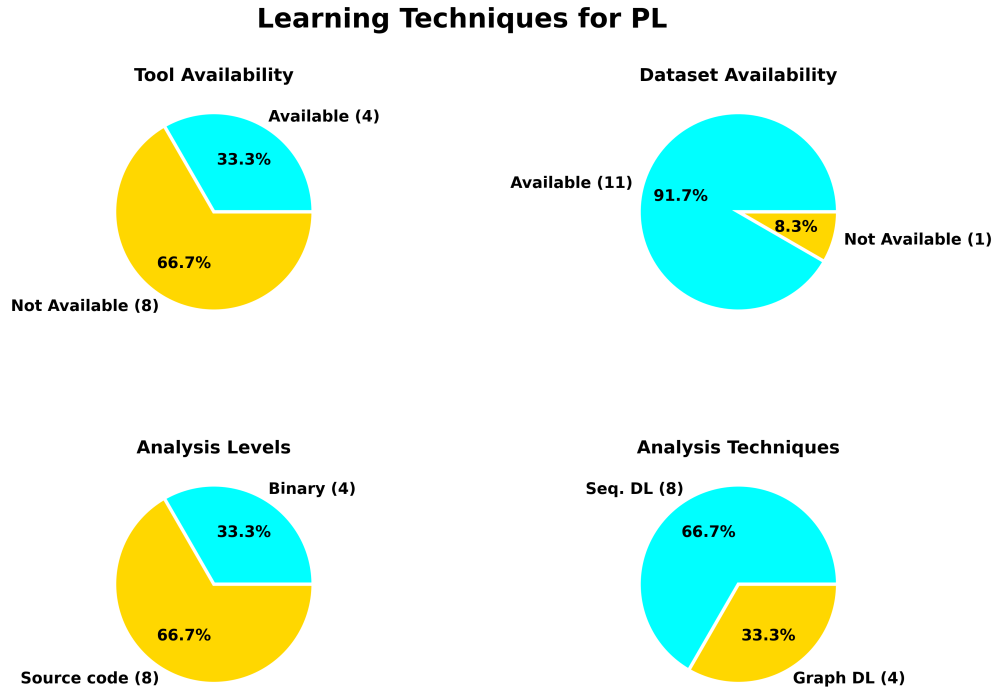


Figure 3.3: Learning-based Techniques for PL in primary studies.

(CFGs) from the assembly instructions generated during source code compilation. These graphs visually represent the program’s execution flow, providing valuable insights into its behavior. In the second step, a deep neural network called a Directed Graph-based Convolutional Neural Network (DGCNN) is applied to the extracted CFG datasets. This powerful network automatically learns hidden features within the CFGs that are indicative of defects. By leveraging the rich information within CFGs and the robust learning capabilities of DGCNNs, this approach enables the automatic construction of predictive models for defect detection. This paves the way for significant advancements in software quality assurance by automating the crucial step of defect identification.

Russell et al. [113] (2018) utilized the extensive repository of open-source C and C++ code to create a comprehensive machine learning-based system for detecting vulnerabilities at the function level on a large scale. They employ feature-extraction methods akin to those utilized in sentence sentiment classification, employing Con-

Metric	Category	#Studies	References
Tool Availability	Available	4	[94, 87, 30, 143]
	Not Available	8	[108, 113, 88, 161, 18, 131, 151, 155]
Dataset Availability	Available	11	[108, 113, 88, 94, 161, 18, 151, 155, 87, 30, 143]
	Not Available	1	[131]
Analysis Levels	Binary	4	[108, 94, 131, 151]
	Source code	8	[113, 88, 161, 18, 155, 87, 30, 143]
Analysis Techniques	Seq. DL	8	[108, 113, 88, 18, 131, 151, 87, 143]
	Graph DL	4	[94, 161, 155, 30]

Table 3.3: Learning-based Techniques for PL in primary studies.

volutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) for classifying vulnerabilities at the function-level in source code. This approach disregards the program code’s execution flow, treating it solely as a sequence of tokens without considering its execution flow.

Li et al. [88] (2018) introduced VulDeePecker, which has an innovative dataset tailored for assessing the effectiveness of deep learning-based vulnerability detection systems. Central to VulDeePecker’s methodology is the proposal of “code gadgets” or slices, which represent segments of programs. These code gadgets encompass semantically related lines of code, allowing for vectorization and subsequent utilization as input for deep learning models. Within the VulDeePecker framework, the learning phase involves processing a significant volume of training programs, comprising both vulnerable and non-vulnerable instances. Initially, it extracts library/ABI function calls from these programs and then identifies program slices linked to the arguments of these calls. Subsequently, it encodes the symbolic representations of these code gadgets into vectors using word2vec. These vectorized representations serve as input for training a Bidirectional Long Short-Term Memory (BLSTM) neural network. The outcome of the learning phase is the extraction of vulnerability patterns, which are encoded within the BLSTM neural network. This approach aims to effectively discern vulnerabilities within program code, establishing a mechanism for deep learning-based vulnerability detection systems. However, it’s important to note that

a model trained solely on a synthetic dataset composed of simple patterns may encounter limitations in detecting only those basic patterns that infrequently occur in real-life scenarios. Additionally, it's essential to acknowledge another limitation: the token-based model operates under the assumption of linear token dependency. Consequently, it primarily considers lexical dependencies among tokens, neglecting semantic dependencies that frequently hold significant roles in vulnerability prediction.

Bilgin et al. [18] (2020) investigated the use of machine learning to predict software vulnerabilities from source code. It first divides source code into smaller units like functions. Next, for each function, it generates and extracts an abstract syntax tree (AST), employing a lexer for tokenization. The extracted AST is then transformed into a complete binary tree with a deterministic shape, where the number of nodes per level is predefined. Finally, each token in the complete binary AST is encoded as a pre-defined numerical tuple and subsequently concatenated from the root to leaves to obtain a one-dimensional numerical array representation of the corresponding function-level source code. This representation serves as the input for the machine learning model for vulnerability prediction. However, there are concerns regarding the manual feature engineering aspect of the process. The method involves the manual construction of a set of predetermined numerical features for each token, resulting in a labor-intensive and resource-costly procedure. This manual feature engineering approach may be time-consuming and inefficient, potentially hindering scalability and adaptability in handling diverse codebases and environments.

Tian et al. [131] (2021) introduced BinDeep, a deep learning methodology tailored to identify similarities between functions within binary code. The process begins by extracting the instruction sequences from individual functions, which are then transformed into numerical features using a word2vec embedding model. Through this conversion, the instructions are represented as numerical values. Subsequently, a Recurrent Neural Network (RNN) is employed to generate embeddings for each function. These embeddings encapsulate the fundamental functionality of the respective functions. BinDeep further employs Siamese neural networks, integrating Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) architectures, to evaluate the similarity between the embedded representations of two functions. This approach showcases a notable efficiency in identifying resemblances

between functions, even in scenarios where the functions are obfuscated or compiled using different tools.

Yan et al. [151] (2021) introduced HAN-BSVD, a hierarchical attention network designed specifically for detecting vulnerabilities in binary software. HAN-BSVD employs a multi-step process. Initially, it enhances contextual information by eliminating irrelevant jump addresses and normalizing instructions. Following this enhancement, it constructs an instruction embedding network utilizing Bi-GRU and a word-attention module to effectively preserve the enriched contextual information. Finally, a feature extraction network, incorporating Text-CNN with a spatial-attention module, facilitates the capture of local features and accentuates critical regions, ultimately enhancing the overall performance of vulnerability detection.

Li et al. [87] (2021)²⁶ proposed SySeVR which introduces two fundamental concepts: Syntax-based Vulnerability Candidates (SyVCs) and Semantics-based Vulnerability Candidates (SeVCs). SyVCs capture syntax-related vulnerability characteristics, while SeVCs extend SyVCs to encompass semantic aspects induced by data and control dependencies. In its approach to vulnerability detection, SySeVR employs various neural networks. Initially, it leverages syntax traits to identify SyVCs, serving as an initial step for vulnerability detection. However, SyVCs lack sufficient semantic information necessary for training deep learning models. SeVCs bridge this gap by extending SyVCs with semantically related statements, induced by control and data dependencies. To encode SeVC symbols into fixed-length vectors, SySeVR utilizes word2vec. This framework facilitates the use of multiple neural networks for detecting diverse vulnerabilities. Notably, Bidirectional Gated Recurrent Units (BGRU) within Bidirectional Recurrent Neural Networks (RNNs) demonstrate heightened effectiveness compared to unidirectional RNNs and Convolutional Neural Networks (CNNs). Nonetheless, it's crucial to acknowledge the potential limitations of models trained solely on synthetic datasets, as they may primarily detect simple patterns that are rare in real-world scenarios.

Wartschinski et al. [143] (2022)²⁷ introduced Vudenc, a deep learning-based

²⁶Li et al. [87] (SySeVR) is available: <https://github.com/SySeVR/SySeVR>

²⁷Wartschinski et al. [143] (VUDENC) is available: <https://github.com/LauraWartschinski/VulnerabilityDetection>

tool designed for automated feature learning from an extensive real-world Python code repository to detect vulnerabilities. Leveraging a word2vec model, Vudenc identifies code tokens with semantic similarity, generating vector representations. It subsequently employs a network of Long Short-Term Memory cells (LSTMs) for fine-grained classification of sequences of vulnerable code tokens. This process not only pinpoints potential vulnerability areas within the source code but also offers confidence levels for its predictions.

3.3.2 Graph Deep Learning (Graph DL) Studies/Tools:

Massarelli et al. [94] (2019)²⁸ introduced a novel method for analyzing binary code by applying graph embedding techniques to control flow graphs (CFGs). This innovative approach leverages the power of machine learning to extract meaningful features from assembly instructions and identify relationships within the program’s execution flow. At the heart of the method lies word2vec, a popular word embedding model. Similar to how word2vec learns representations for words in natural language, Massarelli et al. adapt it to capture the semantic meaning of individual assembly instructions. This allows them to encode the essence of each instruction within a low-dimensional vector space. To capture the global structure of the CFG, Massarelli et al. employ Structure2Vec. This powerful technique combines the node features with the graph’s structural information, generating a comprehensive embedding vector that encapsulates the entire program’s control flow. The resulting CFG embeddings are then used for two distinct tasks: binary similarity by comparing the cosine similarity between CFG embeddings, and compiler provenance identification by training a feed-forward neural network on labeled CFG embeddings to predict the compiler used to generate the binary code. Massarelli et al.’s work demonstrates the effectiveness of graph embedding techniques for analyzing binary code.

Zhou et al. [161] (2019) proposed Devign, a novel graph neural network model for vulnerability detection in source code. It leverages a rich set of semantic representations extracted from the source code and constructs a comprehensive graph. A novel Conv module efficiently extracts features from the learned node representations for graph-level classification. The model is trained on manually

²⁸Massarelli et al. [94] is available: <https://github.com/lucamassarelli/Unsupervised-Features-Learning-For-Binary-Similarity>

labeled datasets built on 4 large-scale, diverse open-source C projects, reflecting real-world complexity and variety, unlike previous work that relied on synthetic code.

Zeng et al. [155] (2021) proposed GCN2defect, an extension of GCN designed to enhance software defect prediction by learning to encode software dependency networks automatically. It begins by constructing a Class Dependency Network for a program. Through node2vec, it learns embedded representations, capturing the structural features of the network based on a set of manually designed metrics. Subsequently, GCN2defect merges these learned structural features with conventional source code manually designed features to initialize node attributes within the Class Dependency Network. This combined information is then fed into GCN, allowing for the generation of deeper representations of the class. It’s important to note that this approach treats the entire class as a single node within the project graph and does not consider the Control Flow Graph (CFG) of the class code.

Chakraborty et al. [30] (2021)²⁹ introduced REVEAL, a novel approach by employing a graph-based embedding derived from real-world source code using a Gated Graph Neural Network (GGNN). Initially, it extracts both syntax and semantics from the code via a Code Property Graph (CPG). Each code fragment within the graph is then encoded into a feature vector using a word2vec embedding model. The final step in preprocessing is to aggregate all the nodes embedding to create a single vector representing the whole CPG. Following this encoding process, REVEAL proceeds to train a representation learner on the extracted features. This learner aims to derive an optimal representation that effectively distinguishes vulnerable source code from non-vulnerable instances. It’s noteworthy that REVEAL utilizes GGNN-based feature embedding, complemented by techniques like SMOTE (Synthetic Minority Over-sampling Technique) and representation learning. These methods address challenges associated with data duplication, imbalanced data distribution, and lack of separability in the dataset. As a result, REVEAL demonstrates significant improvements, achieving up to a 33.57% increase in precision and a 128.38% increase in recall compared to existing state-of-the-art methods.

²⁹Chakraborty et al. [30] (REVEAL) is available: <https://git.io/Jf6IA>

3.4 Data Sources and Benchmarks

This section focuses on data sources and benchmarks availability. While manual efforts to classify Ethereum smart contracts exist and deserve recognition, achieving a comprehensive and universally accepted ground truth remains elusive. To address this gap, we delve into publicly available and documented Ethereum smart contract benchmark sets containing manually verified ground truth data. We focus on datasets generated through meticulous manual contract checking or deliberate bug injection to ensure the highest quality ground truth possible. These datasets fall into two distinct categories based on their origin, summarized in Table 3.4 (The final column in the table indicates whether our tools have been tested on these datasets or not):

- **Manually Crafted Datasets (4 datasets):** A significant proportion of these contracts are real-world contracts that may or may not have been modified by the authors. Notably, some contracts within these datasets may be author-created but not publicly accessible on the chain. Analysis of these contracts necessitates the explicit provision of their source code and/or bytecode.
- **Real World Datasets (11 datasets):** These datasets offer a crucial testing ground for smart contract vulnerability detection tools. They contain contracts that have actually been deployed on the public Ethereum chain, providing a realistic representation of the complexities and challenges faced in real-world scenarios. Importantly, all contracts within these datasets, except for self-destructed ones, have unique addresses on the chain. This allows researchers to conveniently retrieve their source code (if available) and/or bytecode for analysis directly using these addresses. While this process can be time-consuming for exceptionally large datasets, it offers a significant advantage over manually crafted datasets, providing invaluable insights into the vulnerabilities present in actively deployed contracts.

The following subsections will delve into each dataset within their respective groups, providing a detailed examination of their characteristics and suitability for various evaluation purposes. This comprehensive analysis aims to equip researchers and developers with a deeper understanding of available ground truth resources, facilitating the development of more robust and effective tools for Ethereum smart

contract security analysis. Our findings show that *there’s a lack of reliable benchmarks for assessing smart contract vulnerability detection*. Developing robust benchmarks is a critical need for the blockchain security community.

3.4.1 Manually Crafted Datasets:

Durieux et al. [41] (2020)³⁰ propose SmartBugs Curated, which is a valuable resource for researchers in Solidity smart contract security. It comprises 69 carefully chosen contracts annotated with 115 vulnerabilities falling into ten categories including reentrancy, integer overflow, and access control issues. The lines containing each vulnerability are precisely tagged by the authors, facilitating the evaluation of new analysis tools. Compared to other datasets, SmartBugs focuses on diverse and realistic vulnerabilities, making it highly relevant for practical security research. However, it’s important to note that the dataset primarily includes smaller contracts and may not fully represent the entire range of real-world smart contracts.

SWC Dataset (2020)³¹ is a taxonomy of 37 vulnerabilities and weaknesses found in smart contracts, aimed at standardizing the way these issues are identified and categorized. It helps developers, security researchers, and other stakeholders understand the nature and potential impact of different smart contract problems. It categorizes weaknesses like reentrancy and access control flaws, providing a common language for communication and facilitating security analysis tools.

Ghaleb et al. [54] (2020)³² present SolidiFI benchmark as a dataset of buggy smart contracts designed to evaluate the effectiveness of static analysis tools for Solidity code. It contains over 9,300 bugs injected into 350 contracts across 7 categories, including reentrancy, integer overflows, and timestamp dependencies. These “buggy” contracts serve as test cases for analysis tools, allowing researchers and developers to assess their ability to detect and flag vulnerabilities. Additionally, the benchmark provides injection logs revealing where and how each bug was inserted, enabling further analysis and understanding of smart contract weaknesses. Overall, SolidiFI Benchmark can play a crucial role in advancing the development of reliable and

³⁰Durieux et al. [41] (SmartBugs Curated Dataset) is available: https://github.com/smartbugs/smartbugs-curated/blob/main/ICSE2020_curated_69.txt

³¹SWC Dataset is available: <https://swcregistry.io/>

³²Ghaleb et al. [54] (SolidiFI Dataset) is available: <https://github.com/DependableSystemsLab/SolidiFI-benchmark>

Category	Dataset	#Samples	#Vul	Benchmarked by our tools
Manually Crafted Datasets	SmartBugs Curated [41]	69	10	Easy: It is included as part of our benchmark [7].
	SWC	37	37	Easy: It isn't included due to limited number of samples per vulnerability.
	SolidiFI [54]	350	7	Easy: We created our challenging benchmark [13] based on SolidiFI bug injection method.
	Gigahorse	275	9	Easy: Most samples are part of our benchmark [11].
Real World Datasets	Contract-fuzzer [76]	416	7	Possible: It isn't included due to lacking samples of non-vulnerable contracts.
	Zeus [78]	1,524	7	Hard, it isn't included due to its labelling inconsistency.
	SmartBugs Wild [41]	47,398	0	Difficult: Not included because it is an unlabeled dataset, requiring huge effort to label.
	Ever-evolving Game [160]	1,265	6	Possible: It isn't included due to lacking source and bytecode.
	TSE-Contract-Defect [31]	587	9	Possible: It isn't included due to lacking both source code and bytecode, and potential inconsistencies in its labels.
	eThor [116]	605	1	Possible: it is included as part of our benchmark [10].
	Horus [47]	1,655	4	Easy: it is included as part of our benchmark [7].
	Scrawld [152]	6,780	8	Easy: it isn't included due to the lack of manual verification by the authors necessitates caution regarding the accuracy of the labels.
	Forta	753	1	Easy: it isn't included due to the absence of a shared vulnerability detectable by our tools.
	MANDO-GURU [103]	493	7	Easy: it is included in our comparison with [103, 104].
SC_UEE [71]	4,364	10	Possible: it isn't included because access to the dataset is granted upon reasonable request.	

Table 3.4: Data Sources and Benchmarks.

effective smart contract analysis tools, ultimately contributing to the security of blockchain applications.

Gigahorse Dataset (2021)³³ offers a diverse collection of 275 smart contracts (source and bytecode) labeled with nine common vulnerabilities (e.g., reentrancy, integer overflow). Some contracts are derived from SmartBugs Curated, engineered to showcase typical vulnerability patterns. This balanced mix of 109 vulnerable and 166 safe contracts makes it suitable for evaluating both vulnerability detection and false positive rates of analysis tools.

3.4.2 Real World Datasets:

Jiang et al. [76] (2018)³⁴ provide Contractfuzzer dataset as a valuable resource for researchers and developers working on smart contract security. Contractfuzzer dataset features 416 manually verified vulnerable smart contracts of seven vulnerabilities, analyzed using the ContractFuzzer tool. This collection includes both source code and bytecode for further analysis, offering valuable insights into real-world vulnerabilities. However, it's important to note that the dataset focuses solely on confirmed vulnerable examples, lacking cases of non-vulnerable contracts. This makes it suitable for testing and evaluating the accuracy of smart contract analysis tools, but not for measuring false positives rate.

Kalra et al. [78] (2018)³⁵ present Zeus dataset, which offers a collection of 1,524 contracts categorized based on the Zeus tool's vulnerability assessment for seven types of vulnerabilities. However, no source code or bytecode is provided, and the authors of the dataset haven't responded to requests for further details on their labeling methodology. Notably, the eThor paper [115] identified inconsistencies in the Zeus dataset's labeling practices, particularly within section D.5 of the eThor extended version. Through careful filtering, the eThor authors reduced the dataset from 1,524 to 720 contracts deemed more reliable for their analysis. While the Zeus dataset provides a starting point for exploring potential vulnerabilities, its limitations and inconsistencies warrant caution in interpreting its results.

³³Gigahorse Dataset is available: <https://github.com/nevillegrech/gigahorse-benchmarks>

³⁴Jiang et al. [76] (Contractfuzzer Dataset) is available: <https://github.com/gongbell/ContractFuzzer/tree/master/examples>

³⁵Kalra et al. [78] (Zeus Dataset) is available: <https://goo.gl/kFNHy3>

Durieux et al. [41] (2020)³⁶ propose SmartBugs Wild, comprising 47,398 Solidity smart contract source codes extracted from the Ethereum blockchain, presents a unique challenge for researchers and developers. While the exact vulnerabilities within these contracts remain unknown, the dataset’s vast size and real-world origin make it invaluable for identifying potential vulnerabilities and gauging their prevalence. However, it’s important to acknowledge the limitations of working with unknown vulnerabilities, such as the possibility of false positives or negatives. Despite these challenges, the SmartBugs Wild Dataset has already been successfully used to identify several real-world vulnerabilities and compare various analysis tools, demonstrating its potential for advancing smart contract security research.

Zhou et al. [160] (2020)³⁷ present Ever-evolving Game dataset as a collection of 1,265 smart contracts categorized based on six prevalent vulnerabilities, including reentrancy and integer overflow. To ensure accurate vulnerability identification, the authors meticulously validate the labels through manual analysis of the contracts’ transaction histories. Notably, Ever-evolving Game dataset only offers contract addresses and vulnerability labels, lacking both source code and bytecode representations.

Chen et al. [31] (2020)³⁸ present TSE-ContractDefect dataset as a collection of 587 manually labeled smart contracts exhibiting 20 diverse contract defects, including 9 classified as security vulnerabilities. While the dataset’s size is relatively small, we identified potential inconsistencies in its labels. For example, some contracts tagged with security vulnerabilities like reentrancy (such as the address 0x53a54c442583d2e844733f179d142c32f3c004b7) seemed protected by the onlyOwner modifier, potentially mitigating real-world exploitability despite the theoretical vulnerability. Further investigation into such cases is encouraged to refine the dataset’s accuracy and enhance its usefulness for research and evaluation purposes. Notably, TSE-ContractDefect dataset only offers contract addresses and vulnerability labels, lacking source code and bytecode representations.

³⁶Durieux et al. [41] (SmartBugs Wild Dataset) is available: <https://github.com/smartbugs/smartbugs-wild>

³⁷Zhou et al. [160] (Ever-evolving Game Dataset) is available: <https://drive.google.com/file/d/1xLssDXyWyKFCwS5HUrQaSex0uwJRSvDi/view>

³⁸Chen et al. [31] (TSE-ContractDefect Dataset) is available: <https://github.com/Jiachi-Chen/TSE-ContractDefects/blob/master/ContractDefects.csv>

Schneidewind et al. [116] (2020)³⁹ provide eThor dataset, which is a collection of 605 smart contracts with source code and bytecode, focusing solely on the reentrancy vulnerability. However, the authors’ definition of reentrancy diverges significantly from the common description (e.g., SWE-107), leading to a “wide divergence in ground truth.” Additionally, the dataset focuses on a specific type of reentrancy (“single-entrancy”), further complicating comparison with other datasets. These limitations present challenges for interpreting the eThor dataset and utilizing it for standard benchmarking practices. While its unique focus on a specific vulnerability and the availability of source code and bytecode offer potential benefits for certain research tasks, researchers should carefully consider these limitations when utilizing the eThor dataset for their studies.

Ferreira et al. [47] (2021)⁴⁰ introduce Horus dataset, which stands out by focusing on real-world attack detection rather than just identifying vulnerable contracts. It presents a collection of 1,655 unique smart contracts, each vulnerable to one of four common vulnerabilities like reentrancy and parity bug. The dataset’s true strength lies in its rich collection of 129,863 annotated transactions, meticulously labeled as benign (122,830) or attacker-initiated (7,041). This unique combination provides valuable insights into real-world attack patterns and the dynamics of on-chain exploits. Importantly, Horus offers bytecode representations for all contracts, facilitating straightforward benchmarking and enabling researchers to test and evaluate attack detection tools.

Yashavant et al. [152] (2022)⁴¹ present Scrawld dataset, which offers a valuable resource for researchers studying smart contract security, featuring 6,780 real-world contracts from the Ethereum network. This large dataset covers eight diverse vulnerability categories, labeled using five automated analysis tools (Slither, Smartcheck, Mythril, Oyente, Osiris) through majority voting. While the lack of manual verification by the authors necessitates caution regarding the accuracy of the labels, Scrawld’s size and diversity make it valuable for studying the prevalence of vulnerabilities in real-world contracts, and analyzing the performance of different analysis methods.

³⁹Schneidewind et al. [116] (eThor Dataset) is available: <https://secpriv.wien/ethor/>

⁴⁰Ferreira et al. [47] (Horus Dataset) is available: <https://github.com/christoftorres/Elysium/tree/main/evaluation/datasets/Horus>

⁴¹Yashavant et al. [152] (Scrawld Dataset) is available: <https://github.com/sujeetc/Scrawld>

Forta Dataset (2022)⁴² provides two valuable datasets for researching smart contract security. The first, hosted on GitHub, consists of 753 contracts identified through their association with theft-related exploits, fraud-linked addresses, and other malicious activities. This repository offers contract addresses, creator addresses, and labels, but lacks source code and bytecode representations. Forta’s second dataset, available on Hugging Face, comprises a larger collection of 70,000 contracts with creation bytecode and malicious label. This rich dataset presents an excellent opportunity for training anomaly detection models, making it a promising resource for future research.

Nguyen et al. [103] (2022)⁴³ introduce MANDO-GURU dataset, a comprehensive compilation comprising 493 Solidity vulnerable contracts sourced from various prior studies. This dataset encompasses seven distinct vulnerability types, such as reentrancy and time manipulation, each containing a range of 44 to 95 positive samples. Of significance, the MANDO-GURU dataset provides both source code and bytecode representations for all contracts, housed within a separate repository (<https://github.com/MANDO-Project/ge-sc/tree/master/experiments/ge-sc-data>). This accessibility not only simplifies benchmarking procedures but also empowers researchers by facilitating the evaluation and testing of attack detection tools.

Hu et al. [71] (2023)⁴⁴ present SC_UEE dataset, which is a recently released dataset of 4,364 real-world Solidity smart contracts manually labeled with ten types of vulnerabilities, including reentrancy, integer overflow, and timestamp dependency issues. What sets SC_UEE apart is its unique focus on exploitability. Unlike other datasets, the authors don’t just label contracts as vulnerable or not; they meticulously analyze the source code of vulnerable contracts to judge whether they are actually exploitable in practice. This additional layer of information makes SC_UEE a valuable resource for researchers and developers working on smart contract security, as it helps assess the true risk posed by vulnerabilities and prioritize mitigation efforts.

⁴²Forta Dataset is available: <https://github.com/forta-network/labelled-datasets>, and <https://huggingface.co/datasets/forta/malicious-smart-contract-dataset>

⁴³Nguyen et al. [103] (MANDO-GURU) is available: https://github.com/MANDO-Project/ge-sc-machine/tree/master/sco/graph_labels

⁴⁴Hu et al. [71] (SC_UEE Dataset) is available: https://github.com/1052445594/SC_UEE

3.5 Summary of Related Work

While Ethereum smart contract security is paramount, ensuring their safety is a complex challenge. Most modern smart contract languages are Turing-complete, allowing them to implement complex algorithms using languages like Solidity and Vyper. The very expressiveness that makes them powerful is a double-edged sword, as it also complicates their analysis. Moreover, many smart contract vulnerabilities are more complex than many of the traditional bugs targeted by static analysis techniques (*e.g.*, null pointer dereferences). The intricacy inherent in smart contract vulnerabilities poses significant challenges for analysis, and means that traditional analysis techniques may not be sufficient.

Existing efficient techniques often rely on predefined patterns, which frequently prove inadequate for handling real-world code scenarios. Conversely, methods that yield more precise results often demand substantial computational resources, leading to extended and/or infeasible processing times. Learning-based approaches, particularly modern models like deep neural networks, offer a promising avenue. Unlike the typical hand-coded patterns utilized in most static analyzers, these advanced models possess the capability to comprehend exceedingly intricate feature spaces. This advancement paves the way for more comprehensive and robust detection of vulnerabilities in smart contracts.

There has been a notable surge in the adoption of learning-based techniques for smart contract vulnerability detection in recent years. *While their practical implementation and readily available tools have not yet caught up to this growth*, the inherent flexibility and adaptability of these methods suggest a bright future in bolstering smart contract defenses. Their potential to navigate the vast and evolving landscape of vulnerabilities holds significant promise for surpassing the limitations of traditional static and formal analysis methods. Therefore, it is highly likely that learning-based techniques will play an increasingly important role in safeguarding smart contracts against ever-sophisticated security threats.

Our Analysis Summary for:

- 1) **Studies in Static and Dynamic Analysis (SA/DA) Methods for SC:** While the aforementioned limitations persist (§3.1), SA/DA methods

demonstrate a notable advantage: 95.2% of studies offer readily available tools, enabling researchers to effectively compare and benchmark proposed solutions. 71.4% of studies also release a dataset, although in most cases this dataset lacks ground truth labels; the authors estimate accuracy by manually checking a small sample of the output (without disclosing which contracts or the associated labels).

Two-thirds of studies possess the capability to directly analyze bytecode, circumventing the need for often-unavailable source code, with an average detection capacity of 7.7 vulnerabilities. This enhanced accessibility and source code independence represent significant strengths of SA/DA methods within the realm of smart contract vulnerability detection.

- 2) **Studies in Learning-based Techniques for SC:** Our survey indicates that the application of ML/DL techniques for smart contracts vulnerability detection has increased substantially in the past few years, with a large proportion of papers are published in 2021 and 2022 (§3.2). However, current solutions suffer from critical limitations that impede both their practical implementation and our analysis.

First, very few (14.3%) of studies have available tools or artifacts for the community to use, hindering adoption and real-world impact. By keeping their tools and code closed-source, researchers create a significant barrier to entry for other researchers and practitioners. This stifles collaboration, prevents independent verification of results, and ultimately limits the broader application of valuable discoveries. Figure 3.4 clearly demonstrates the stark difference between open-source availability. In sharp contrast to the near-universal open-source availability of SA/DA tools (only one of which—Zeus [78]—is closed-source), learning-based tools lag far behind. *This stark contrast is the primary reason why, in subsequent chapters, we will focus on comparing our open-source tools primarily to many existing SA/DA tools, and a few learning-based tools.*

Alarming, only 28.6% of studies released their datasets and benchmarks, further hindering research and development efforts. Without shared datasets and benchmarks, objective comparison and validation of different approaches

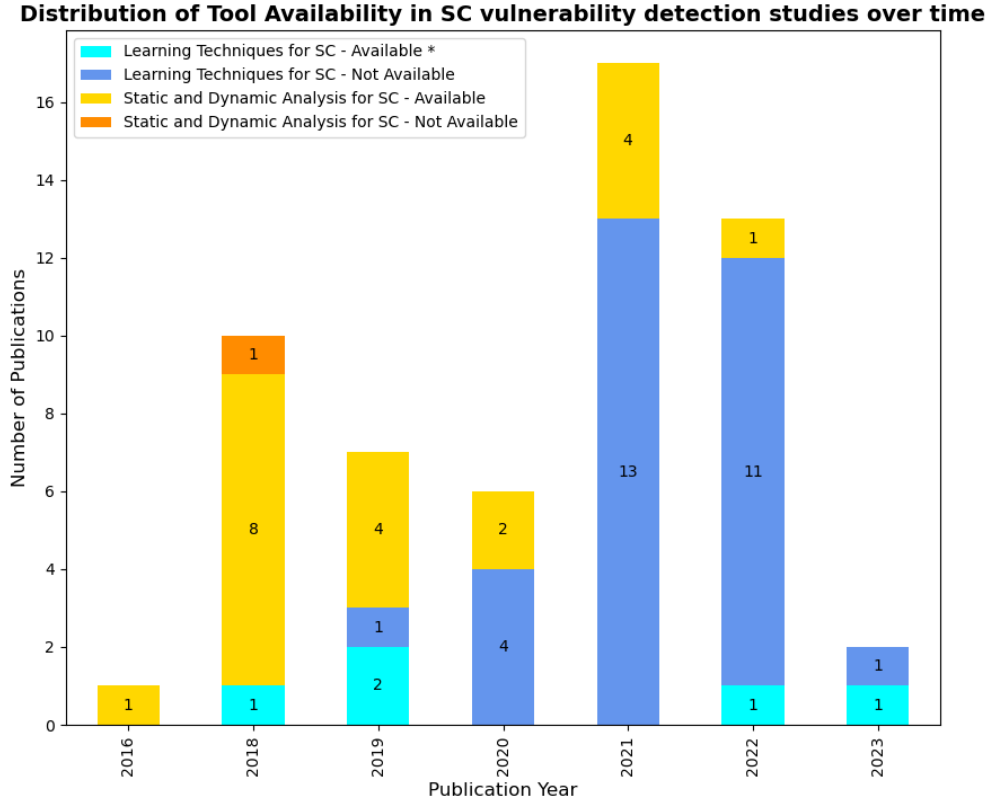


Figure 3.4: Distribution of Tool Availability in SC vulnerability detection studies over time; “Available*” with the asterisk denoting potential availability of some studies upon request approval.

becomes difficult, impeding progress in the field. Moreover, since ML/DL tools are built from the application code to training data, withholding the training data is almost as bad as withholding the code.

Moreover, only 31.4% of studies can work directly with bytecode. Since only 1% of deployed contracts have their source code publicly available (according to studies from [105] and [48]), this considerably restricts the applicability of these tools. Figure 3.5 shows that static/dynamic analysis tools have tended to focus on bytecode, whereas learning-based tools have tended to focus on source code. *This bytecode-centric approach means that static/dynamic analysis tools can typically analyze a much broader range of contracts.*

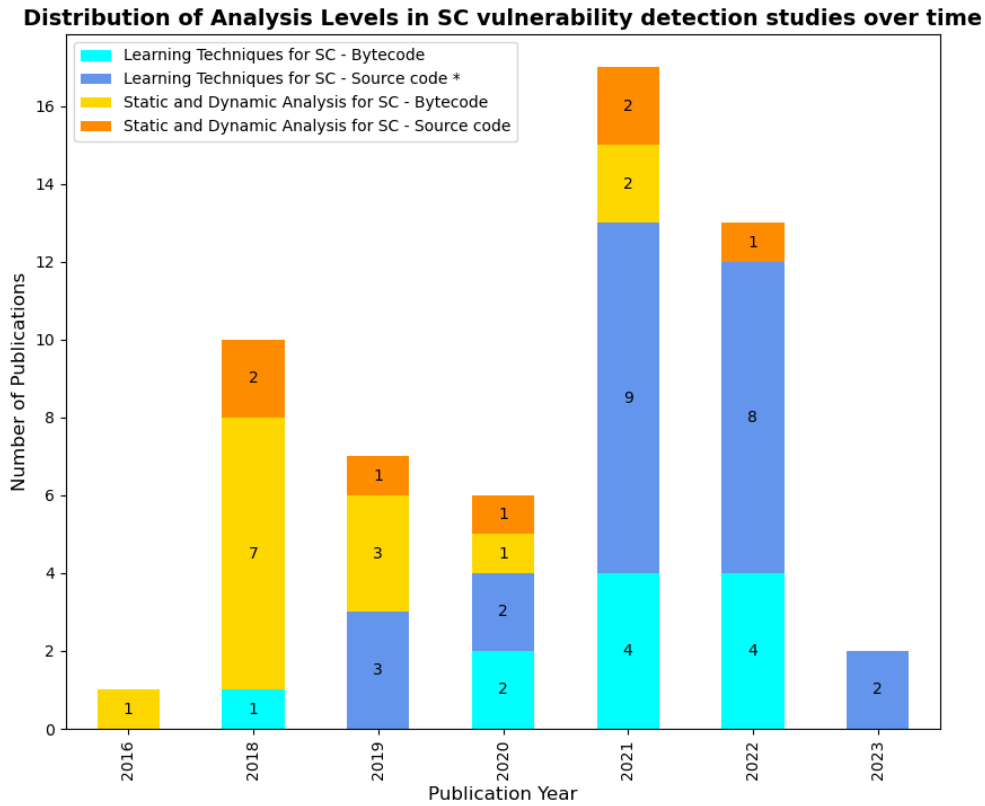


Figure 3.5: Distribution of Analysis Levels in SC vulnerability detection studies over time; “Source code*” with the asterisk denoting the analysis level for some studies requires source code and/or potential alternatives like transactions or account data.

Our survey noted that 82.8% of studies used classic ML or sequential DL models: treating programs as linear data, much like a natural language. Only 17.1% treated programs non-linearly by considering the execution flow of smart contract code, suggesting a potential avenue for improvement in code analysis. Our own DLVA and SCoolS tools use graph DL models to better extract program semantics for classification.

Finally, ML/DL tools often exhibit a limited scope, with an average detection capacity of only 4.6 vulnerabilities. Indeed, some do not distinguish the kind of vulnerability at all. Merely flagging a contract as “vulnerable” without specifying what kind of vulnerability is clearly less valuable than specifying

which vulnerabilities are present (as DLVA does). More valuable still is specifying which functions are concerning (as SCooLS does), or even which program points. These are deep and ongoing challenges, and the trade-offs are not obvious since the ML/DL techniques investigated to date seem to be less compositional than SA techniques. Thus, identifying vulnerable functions (as SCooLS does) can easily miss vulnerabilities that occur across function boundaries; as we will see, DLVA has higher accuracy despite its less-specific error localisation.

In summary, these limitations highlight the benefits of a paradigm shift. First, a move towards open-source tools and publicly available datasets is crucial. Second, methods that can function without relying on source code are preferable. Third, exploring graph DL models that capture code execution flow could enhance vulnerability detection capabilities. Finally, future research should prioritize the development of tools capable of fine-grained vulnerability type detection, enabling more precise and nuanced security solutions for the rapidly growing smart contract ecosystem.

- 3) **Studies in Learning-based Techniques for PL:** The field of general programming language (PL) vulnerability detection boasts a wealth of readily available datasets and benchmarks, providing fertile ground for researchers to develop innovative solutions (§3.3). In contrast to the poor availability of ML/DL tools and datasets for smart contract analysis, the more established field of general programming language vulnerability detectors has an abundance of available resources, which helps researchers build new and improved tools in this domain. Beyond the lack of public tools and datasets, smart contract vulnerability detection presents a greater challenge than PL vulnerability detection due to several key challenges:

Firstly, unlike traditional PL, smart contracts operate on a decentralized and immutable blockchain, where deployed code cannot be patched or modified. This means that any vulnerability, once exposed, becomes a permanent exploit point. This permanence further increases the value of high accuracy in vulnerability detection.

Secondly, compared to the extensive datasets available for PL vulnerabilities, the field of smart contract security lacks a robust collection of labeled vulnerabilities. This scarcity of data hinders the development of accurate and comprehensive vulnerability detection models, as machine learning algorithms rely on large datasets to train and generalize effectively.

Thirdly, smart contracts often utilize specialized languages like Solidity or Vyper, which differ significantly from mainstream languages like Java or Python. These languages introduce new concepts and features, such as blockchain-specific transactions and storage mechanisms, that can be challenging for existing PL vulnerability detection techniques to handle effectively. This necessitates the development of specialized analysis tools and techniques tailored to the unique characteristics of smart contract languages.

Finally, the rapidly evolving nature of the blockchain ecosystem presents a moving target for vulnerability detection. New attack vectors and exploit patterns emerge constantly, demanding continual adaptation and refinement of detection techniques. This dynamism requires solutions that can not only identify known vulnerabilities but also anticipate and adapt to unforeseen threats.

In summary, while the field of PL vulnerability detection benefits from established resources and techniques, the unique characteristics and challenges of smart contracts necessitate a tailored approach. Addressing the limitations in training data availability, adapting to smart contract languages, and continuously evolving to counter new attack vectors are crucial for ensuring the security of smart contracts and the broader blockchain ecosystem.

- 4) **Studies in Data Sources and Benchmarks for SC:** The scarcity of high-quality benchmarks poses a significant obstacle in the advancement of smart contract vulnerability detection. Two primary challenges hinder the development and evaluation of robust detection methods: breadth and ground truth. Breadth refers to both the quantity and diversity of contracts within a benchmark. It's crucial to have a dataset that encompasses a substantial number of contracts to ensure the generalizability of detection models. Additionally, these contracts should mirror real-world examples rather than simplistic, hand-

crafted test cases to ensure the practical relevance of findings. Ground truth, on the other hand, emphasizes the accuracy and reliability of vulnerability labels within a dataset. Unreliable or inaccurate labels can lead both to flawed model development, and—even worse—an inaccurate understanding of tool performance.

Despite the existence of 15 identified datasets and benchmarks (§3.4), researchers must carefully consider the following challenges:

- (A) The availability of samples for specific vulnerability types might be restricted, hindering the application of machine learning techniques to a broader range of vulnerabilities. Data augmentation techniques can potentially alleviate this issue by artificially expanding the dataset.
- (B) Class imbalance, *i.e.* where the number of samples in one class (e.g., vulnerable contracts) is significantly lower than the other class (e.g., non-vulnerable contracts). This imbalance can negatively impact the model’s ability to accurately detect the minority class. Therefore, it is essential to ensure sufficient representation of vulnerable contracts in the training set to achieve a high true positive rate.

Addressing these challenges in data availability and quality is critical for the development of effective smart contract vulnerability detection methods. Efforts in dataset curation, augmentation, and addressing class imbalance are crucial for fostering a secure and reliable smart contract ecosystem.

The subsequent chapters of DLVA and SCoolS take a crucial step by delving into the construction of our training datasets and test datasets/benchmarks. In the spirit of open research and shared progress, *we have made all datasets publicly available [9, 8, 13, 7, 11, 10, 6, 12]. This includes both the training data that fuels our research and the rigorous test datasets/benchmarks used to evaluate our proposed methods.* By readily sharing these valuable resources, we aim to empower the community with the tools and data needed to accelerate the development and comparison of emerging smart contract vulnerability detection techniques. This open and collaborative approach will undoubtedly

lead to more robust and effective solutions, ultimately advancing the security of the entire blockchain ecosystem.

Our DLVA paper [3], presented at USENIX Security 2023, received the highest recognition from the USENIX Artifact Evaluation Committee: all three badges (Available, Functional, and Reproduced) [5]. This achievement means that not only are all the research materials, including tools and data, freely available to the public, but independent experts have also confirmed the validity and reliability of the paper’s main findings. To the best of our knowledge, *DLVA is the first learning-based smart contract vulnerability detection research to earn this distinction.*

3.6 Comparing Our Approach to State-of-the-Art

This thesis contributes to the field of smart contract vulnerability detection by introducing novel approaches and tool implementations. To rigorously evaluate the efficacy of our techniques, we conducted extensive benchmarking on benchmark datasets unseen during training.

This rigorous testing against state-of-the-art tools provides valuable insights into the relative strengths and weaknesses of our methods. The results demonstrate impressive performance, highlighting the potential of our work to enhance smart contract security and mitigate emerging vulnerabilities. We conclude by our benchmarking results and the significance of independent evaluation as follows:

- (A) **Benchmarking results:** Our extensive analysis of 35 studies in §3.2 revealed a concerning trend: nearly half (48.6%) lack any comparisons to existing state-of-the-art real tools when evaluating their smart contract learning-based techniques. This dearth of benchmarking, crucial for accurate performance assessment, is further amplified by the limited scope of comparisons observed in studies that do include them. On average, studies only compare their proposed methods to 1.9 other tools, with a maximum of 8 competitors evaluated in a single study. Our work stands out in stark contrast to this trend. *In the case of DLVA, we rigorously benchmarked against a record-breaking eleven competitors, while SCoolS was compared to three, exceeding the average by a significant margin.* This commitment to comprehensive and transparent

benchmarking sets our work apart and ensures a more reliable evaluation of our proposed techniques.

- DLVA [3, 4, 5]: As shown in Figures 3.6, 3.7, 3.8, 3.9, 3.10, DLVA is benchmarked against eleven competitors (8 SA/DA analyzers + 3 ML/DL analyzers). DLVA is on the far right. We use bar-and-whiskers where star \star represents the mean and plus $+$ represents outliers. Along the bottom we put the competitors, and in parenthesis the number of tests we include in the benchmark for that competitor (not every tool can handle every vulnerability).

We present five measures of performance. Overall, DLVA performed extremely well. Figure 3.6 shows the Completion Rate (i.e., the percentage of contracts for which a tool produces an answer rather than, *e.g.*, raising an exception, timing out, running out of memory, the higher the better). Most suffered from the occasional timeout or etc. Many of the source code analyzers were not able to analyze some contracts since the Solidity version was too old or new⁴⁵. eThor refused to analyze many contracts with `DELEGATECALL` or `CALLCODE` opcodes, because eThor’s formal definition is not accept these opcodes. Only DLVA, SaferSC, SMARTEMBED and SmartCheck answered every query.

Arguably the most important metrics are Accuracy, the True Positive Rate (TPR), and the False Positive Rate (FPR). We exclude any contract that failed to complete from these metrics (*i.e.*, we do not double count failures). Figure 3.7 shows the True Positive Rate (i.e., detection rate; the higher the better), eThor had a 100.0% TPR; SaferSC followed with 99.8% TPR, Slither with 99.4% TPR, and DLVA came in fourth with 98.7% TPR. Figure 3.8 shows the False Positive Rate (i.e., false alarm rate; the lower the better), SAILFISH boasts an impressive 0.1% FPR, followed by SMARTEMBED at 0.4% FPR, DLVA at 0.6%, and SmartCheck at 2.4% FPR.

⁴⁵We made a good-faith effort to lightly clean source code to help them, but in many cases it was not enough. We did exclude any contract for which source code was unavailable; Completion Rates would have been far worse for source-only competitors otherwise.

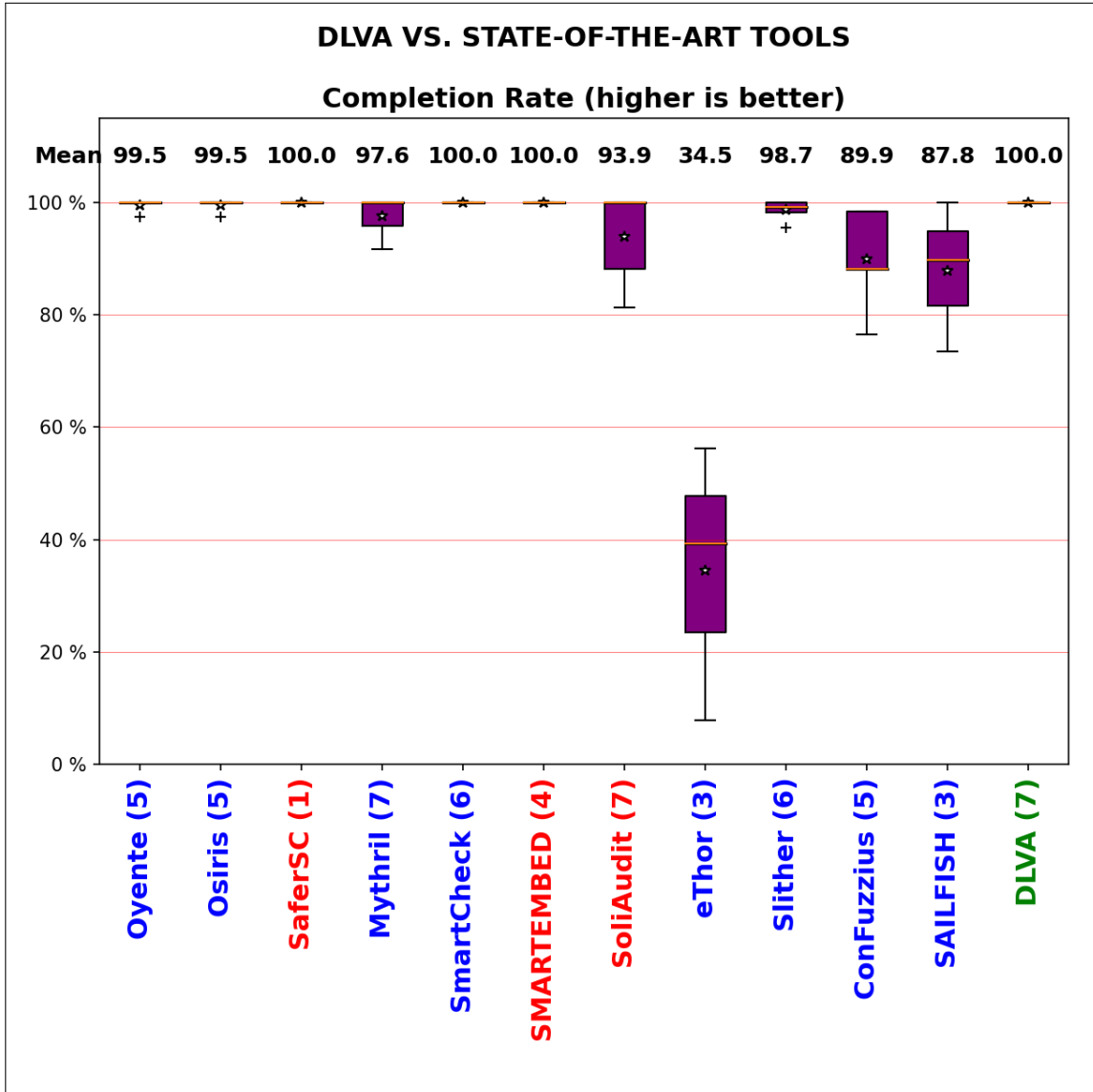


Figure 3.6: DLVA vs. STATE-OF-THE-ART Tools; Completion Rate (i.e., the percentage of contracts for which a tool produces an answer rather than, *e.g.*, raising an exception, timing out, running out of memory, the higher the better) tested on the *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11], and *SolidiFI*_{benchmark} [13]; star \star indicates the mean; plus $+$ indicates outliers

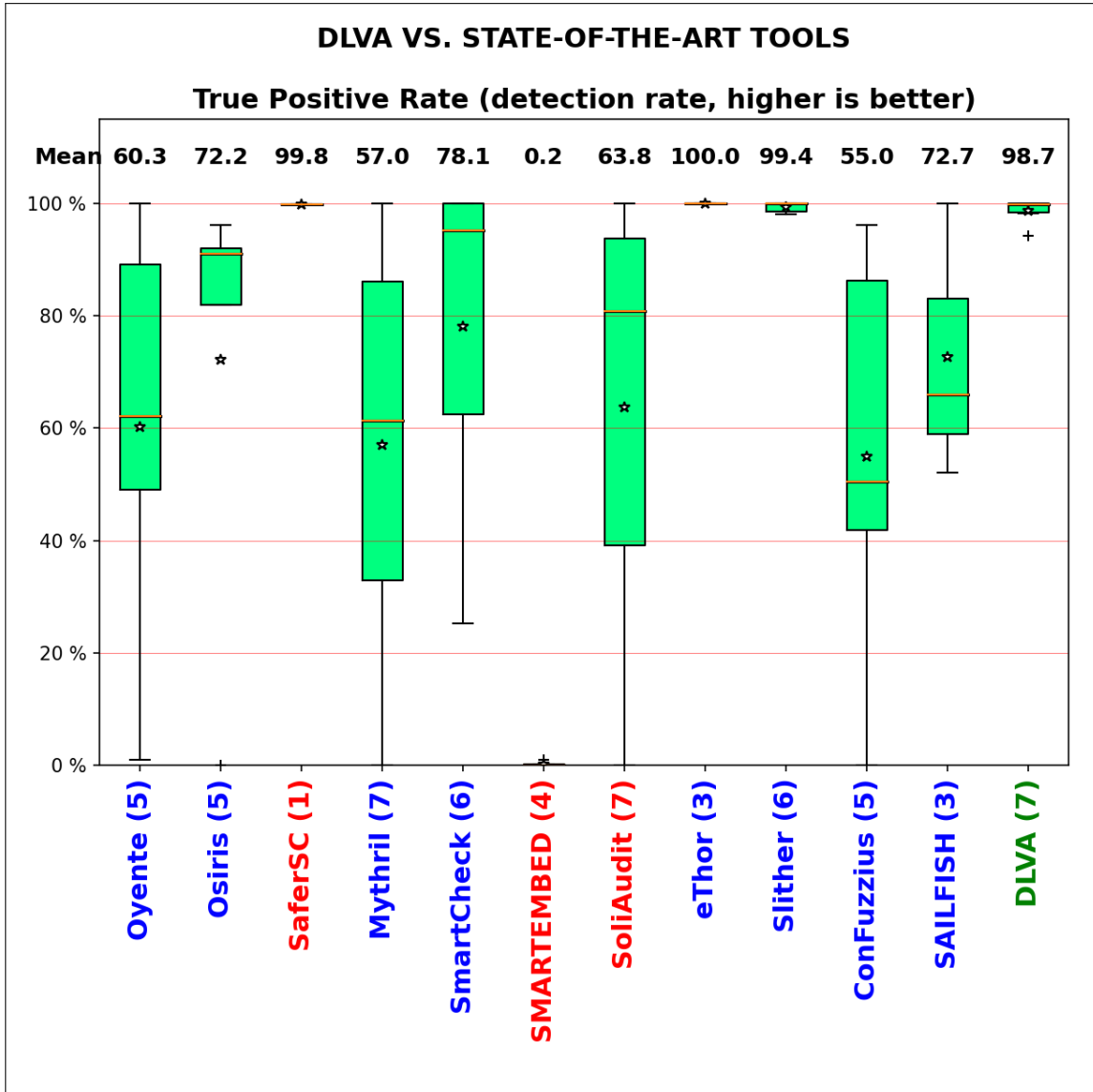


Figure 3.7: DLVA vs. STATE-OF-THE-ART Tools; True Positive Rate (i.e., detection rate; the higher the better) tested on the *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11], and *SolidiFI*_{benchmark} [13]; star \star indicates the mean; plus $+$ indicates outliers

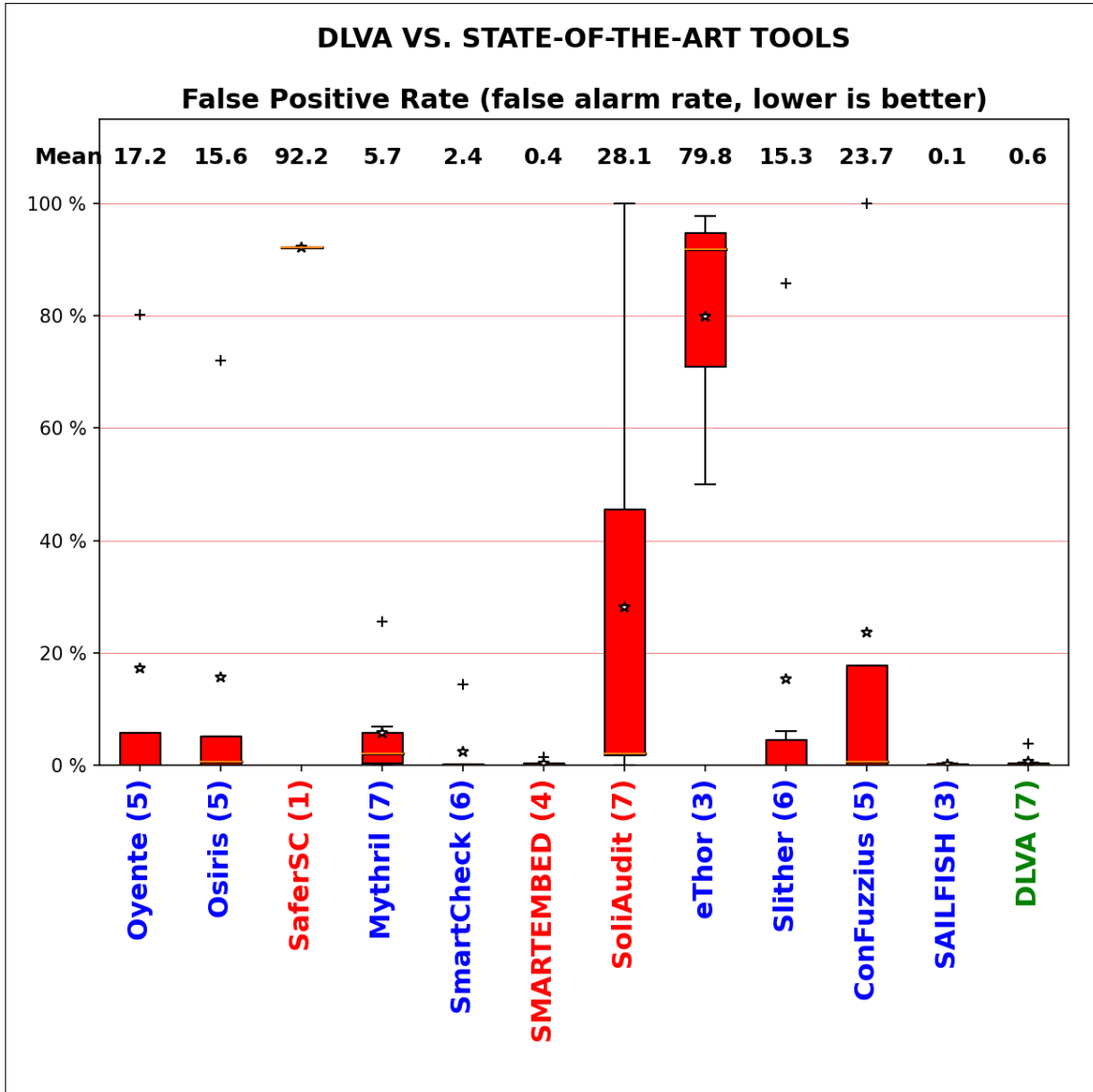


Figure 3.8: DLVA vs. STATE-OF-THE-ART Tools; False Positive Rate (i.e., false alarm rate; the lower the better) tested on the *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11], and *SolidiFI*_{benchmark} [13]; star * indicates the mean; plus + indicates outliers

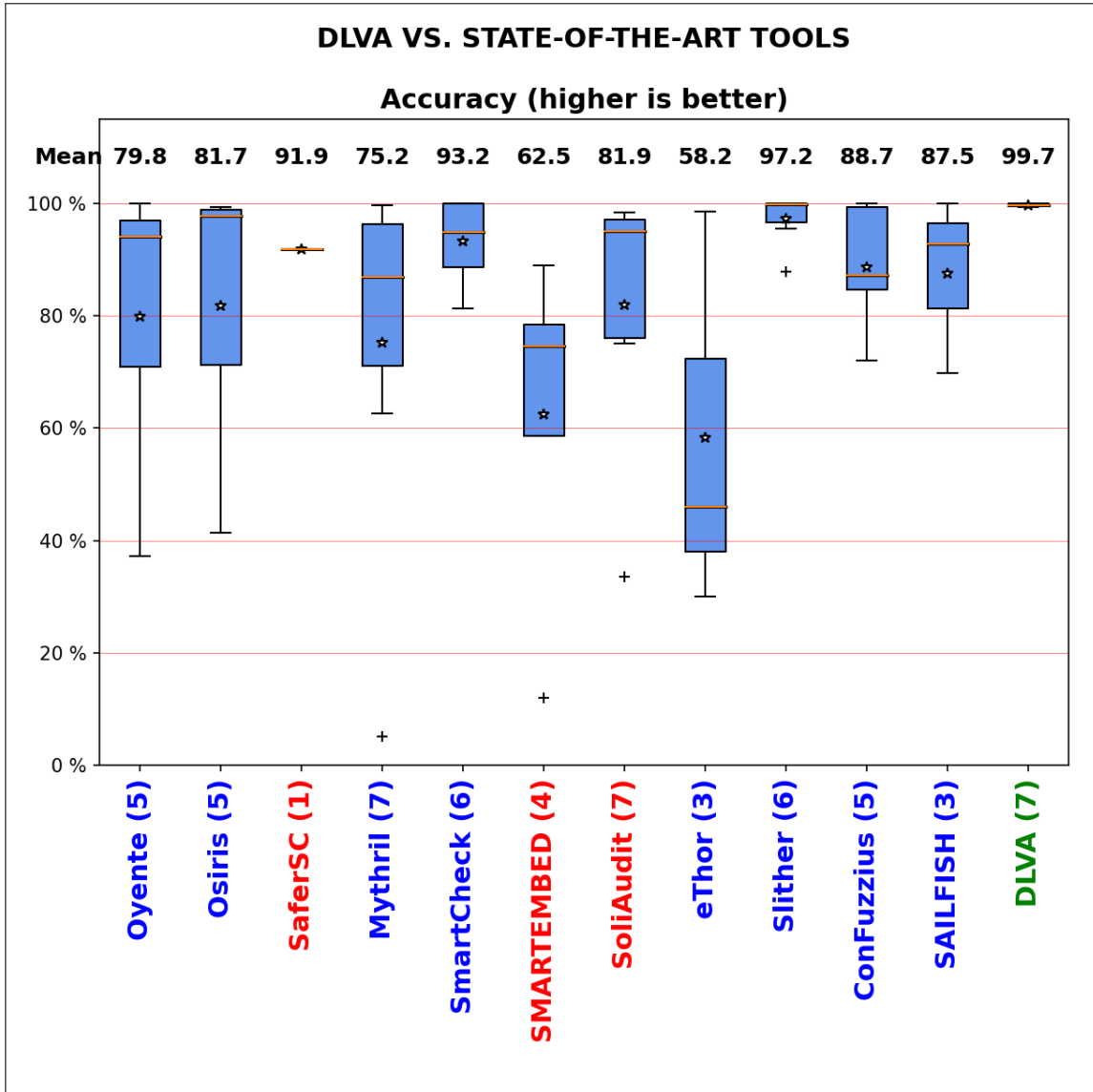


Figure 3.9: DLVA vs. STATE-OF-THE-ART Tools; Accuracy (the higher the better) tested on the *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11], and *SolidiFI*_{benchmark} [13]; star \star indicates the mean; plus $+$ indicates outliers

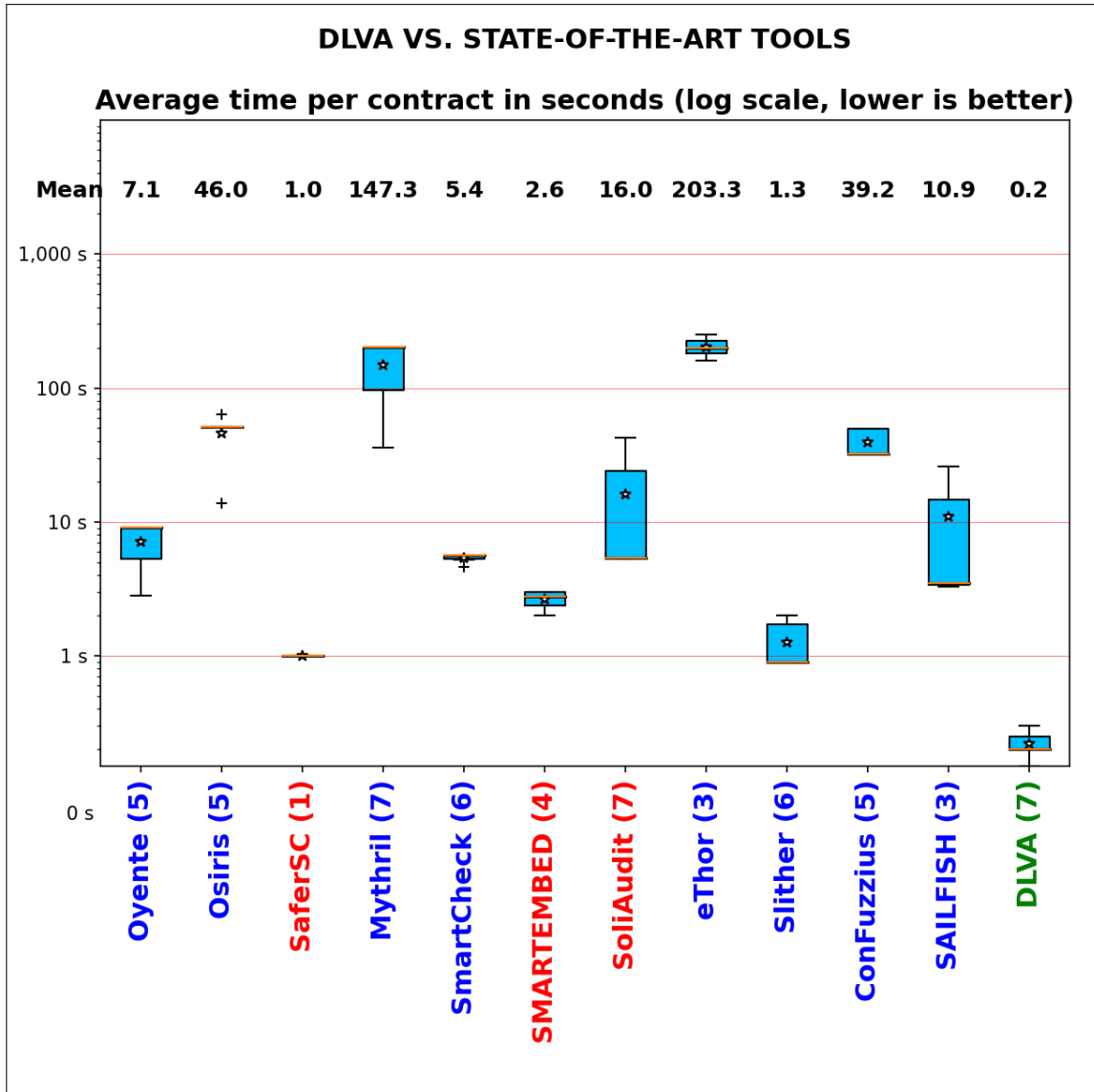


Figure 3.10: DLVA vs. STATE-OF-THE-ART Tools; Average analysis time per contract (the graph is in log scale, the lower the better) tested on the *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11], and *SolidiFI*_{benchmark} [13]; star \star indicates the mean; plus $+$ indicates outliers

In Figures 3.9, DLVA led the pack in accuracy at 99.7%, Slither came in second at 97.2%, SmartCheck came in third at 93.2%, and SaferSC came in fourth at 91.9%. (Moreover, recall that DLVA judges bytecode whereas Slither and Smartcheck require source code!)

In Figures 3.10, DLVA led the pack in average analysis time per contract (the graph is in log scale, lower better) at only 0.2 seconds, SaferSC

came in second at 1.0 seconds, Slither came in third at 1.3 seconds, and SMARTEMBED came in fourth at 2.6 seconds.

In our benchmarking, we found that DLVA, Slither, and SmartCheck had the best overall performance: their high accuracy (99.7%, 97.2%, and 93.2%, respectively) reflects a good balance between a high TPR (98.7%, 99.4%, and 78.1%) and a low FPR (0.6%, 15.3%, and 2.4%). Generally speaking, other tools with good TPR suffered with poor FPR, and vice versa.

A few of these competitors deserve special attention to contextualize their results. eThor’s focus is entirely on soundness, and indeed we were never able to produce a false negative with it; the authors are to be commended. However, the cost to the other metrics is severe: their 34.5% completion rate and 79.8% false positive rate are pitiful; moreover, their analysis time is three orders of magnitude slower than DLVA.

SAILFISH leads the pack with 0.1% FPR, but their TPR is a mediocre 72.7%, explaining their unexceptional 87.5% overall accuracy.

SaferSC’s results seem to have benefited greatly from the nature of our test suite. Of all the vulnerabilities we test, it is only sensitive to “suicidal,” which is why we have only 1 relevant benchmark (Elysium_{benchmark} [7]); all other tools have at least 3 relevant benchmarks. We suspect that SaferSC is strongly biased to report a vulnerability. This naturally leads to a fantastic true positive rate (99.8% tested), but also results in a very poor false positive rate as well (92.2%). However, since the number of suicidal contracts in the Elysium test set is much higher than the number of non-suicidal contracts, SaferSC’s benchmarked accuracy of 91.9% looks better than we think it would, if benchmarked against a test suite that had a more realistic balance of suicidal and non-suicidal contracts. Nevertheless, since SaferSC was one of the few available ML/DL tools, we included it in our benchmark.

SMARTEMBED suffers from the opposite problem: due to a small predefined bug dataset, it is strongly biased to consider contracts as safe. This naturally leads to a fantastic false positive rate (0.4%), at the cost

of a very poor true positive rate (0.2%). Accuracy (62.5%) is not as bad as might be anticipated since most contracts in the 4 test datasets are in fact safe for the “reentrancy” and “over/underflow” vulnerabilities. Like SaferSC, we included SMARTEMBED primarily since it was one of the few available ML/DL tools.

SoliAudit, the third easily available ML/DL tool, performed better than SaferSC or SMARTEMBED on balance, despite its unexceptional overall performance. Its middling TPR of 63.8% was in line with a number of other tools; its FPR of 28.1% was markedly higher than almost every other tool. Still, unlike SaferSC or SMARTEMBED, these results indicate that SoliAudit is not very strongly biased positively or negatively. Accordingly, we think that SoliAudit’s benchmarked accuracy of 81.9% is a fair measure of its performance.

The tools that seem to be most widely used in the community at the moment are Mythril (including its commercial version MythX) and Slither. ConFuzzius is seeing increasing use in the fuzzing community.

We will discuss these experimental results in more detail in the following DLVA Chapter §4.

- SCooLS [2]: As shown in Figure 3.11, SCooLS has an overall accuracy 98.4% and F1 score 90.4% with an associated false positive rate of only 0.8%. It enjoys the highest accuracy among the tools, the highest F1 score, and the lowest false positive rate. Moreover, the average time to analyze a function was only 0.05 seconds, tied for first place. We discuss Figure 3.11 in more detail in the following SCooLS Chapter §5.

(B) **The importance of independent evaluation:** The importance of independent evaluation for new tools and techniques is underscored by the work of Chakraborty et al. [30] (2021). In their study, they assessed the performance of state-of-the-art learning-based techniques for general programming languages in a real-world vulnerability prediction scenario using *independent benchmarks*. Shockingly, their findings revealed that the tools’ performance plummeted by over 50% when evaluated on these external datasets. This stark decrease

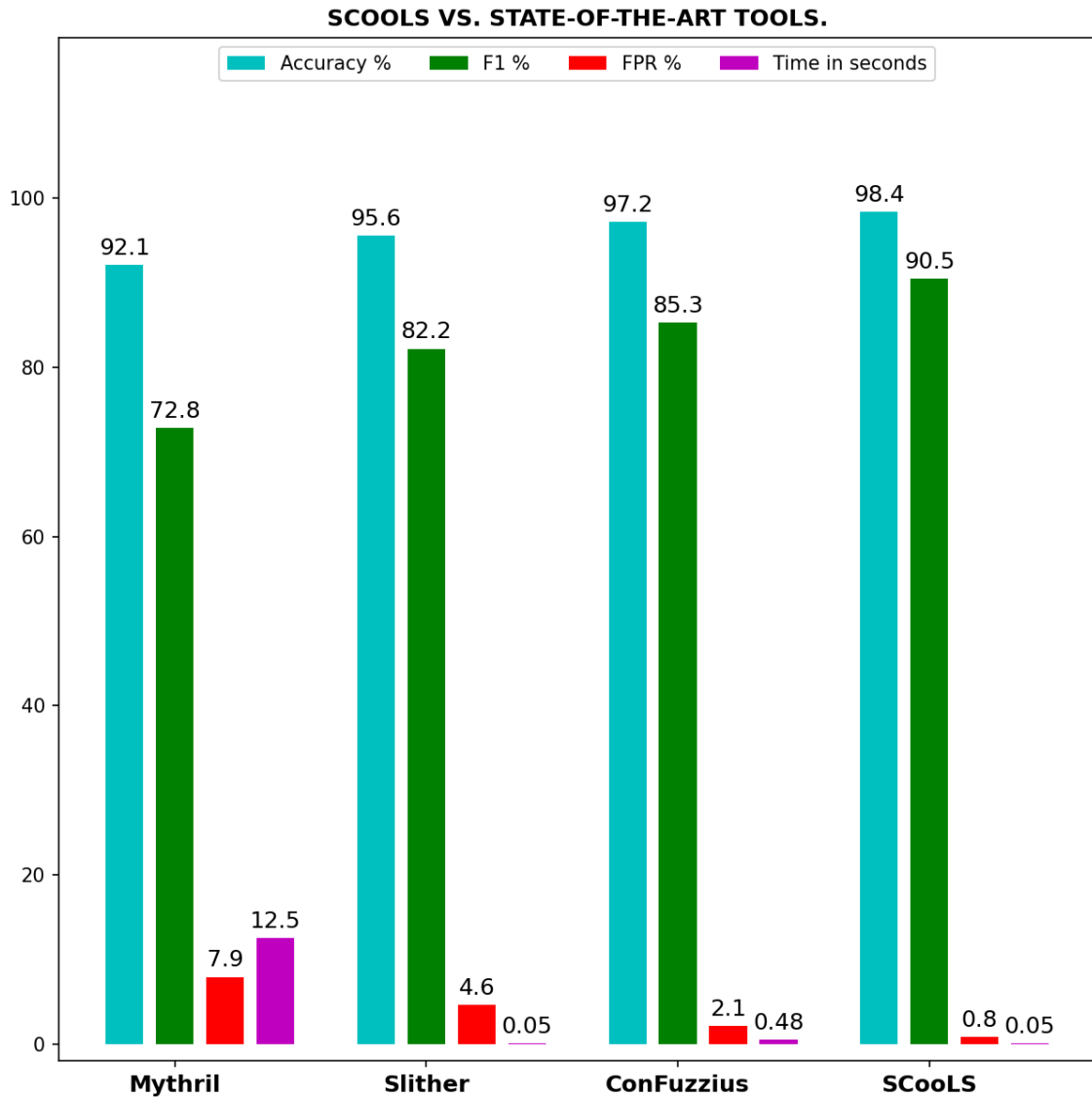


Figure 3.11: SCooLS vs. STATE-OF-THE-ART Tools; Accuracy (the higher the better); F1 (i.e., the harmonic mean of precision and recall; the higher the better) False Positive Rate (FPR) (i.e., false alarm rate; the lower the better); Average analysis time per function (the lower the better); tested on the *ReentrancyBook* [12]

points to critical limitations in existing approaches, including data duplication, imbalanced data handling, lack of real-world training data, inadequate semantic information learning, and poor class separability.

Chakraborty et al.’s findings offer a compelling argument for rigorous independent benchmarking as a cornerstone for assessing the true efficacy of novel tools and techniques. It exposes hidden weaknesses that might otherwise remain concealed within the confines of internal evaluations, ultimately promoting transparency and fostering the development of robust and reliable solutions. By embracing independent validation, we can ensure that promising innovations stand the test of real-world challenges and ultimately deliver tangible benefits in practical applications.

This is precisely why, after meticulously training and internally evaluating DLVA, we took the crucial step of assessing its performance on four independent benchmarks unseen by the model during training. These rigorous benchmarks, including *Reentrancy_{benchmark}* [11], *SolidiFI_{benchmark}* [13], *Elysium_{benchmark}* [7], and *MANDO-GURU* [103]. These benchmarks, meticulously crafted through manual labeling, bug injection, and peer-reviewed research, represent real-world challenges unseen by DLVA during training. It is with great satisfaction that *we report DLVA’s high performance across all benchmarks, a testament to its efficacy and generalizability.* This outcome not only validates the robustness of our approach but also underscores the invaluable role of independent benchmarking in fostering trust and propelling the development of reliable solutions in the evolving landscape of smart contract security.

With regards to SCoolS, we are committed to conducting similarly rigorous independent benchmarking in the future, contingent upon the availability of a suitable function-level benchmark dataset from the community. This commitment aligns with our unwavering belief in the transformative power of independent validation to advance the field and ensure the development of truly dependable smart contract security tools.

To assess the real-world effectiveness of state-of-the-art learning-based vulnerability prediction techniques, we benchmarked the three readily available tools against three benchmark datasets: *Elysium_{benchmark}* [7], *Reentrancy_{benchmark}* [11],

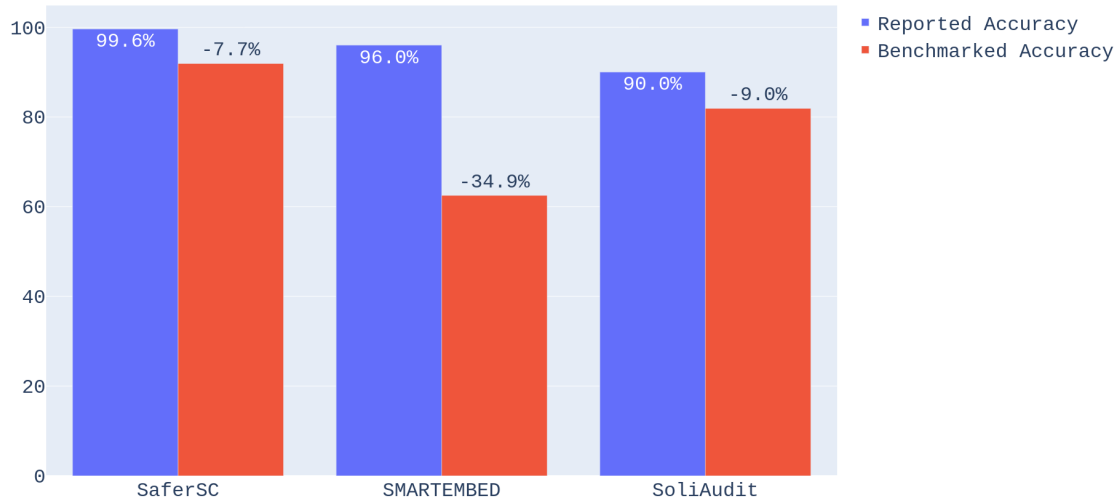


Figure 3.12: Comparison of reported tool performance by its authors versus independent benchmarking results tested on the *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11], and *SoliddiFI*_{benchmark} [13]

and *SoliddiFI*_{benchmark} [13]. As shown in Figure 3.12, our own benchmarking was more challenging than the benchmarking done by the authors: benchmarked performance scores were lower than reported performance scores across all tools, with declines ranging from a modest 7.7% to a substantial 34.9%. Nevertheless, we applaud the authors for making their tools publicly available for independent analysis, especially as the vast majority of ML/DL tools are not available.

These striking results underscore the critical importance of independent benchmarking. Only through rigorous testing on diverse and unseen datasets can we truly gauge the effectiveness of vulnerability prediction techniques in real-world settings. Unfortunately, the lack of open access to tools and datasets remains a significant hurdle in this field. The prevalence of closed-source tools impedes independent verification and limits our ability to fully assess the generalizability and practical applicability of these techniques. Our confidence in the reported results of closed-source tools is low. We urge researchers in this field to embrace open-source practices and prioritize transparency to foster trust

and accelerate the development of robust smart contract security solutions.

3.7 Ethical Disclosure

Blockchain vulnerability disclosure presents a unique dilemma: attackers actively hunt for weaknesses, while anonymity makes it difficult to alert participants to potential threats [20]. To address this challenge, we prioritized responsible disclosure, aligned with USENIX’s Research Ethics Committee feedback. We focused on mitigating real-world impacts before releasing our vulnerability analysis tools, DLVA and SCoolS.

While temporarily closed for safety enhancements, both tools are now re-opened, empowering the community to contribute to a more secure blockchain ecosystem. This responsible approach ensures vulnerabilities are addressed before malicious actors can exploit them, building trust and strengthening the overall blockchain landscape.

3.8 Threats to Validity

- **Internal Validity:**

Our bytecode analysis tools are fast and efficient, but they come with two potential blind spots:

- **Blind Spot 1: Maliciously-created Contracts:** We only analyze publicly available bytecode, which mostly comes from good-faith developers. This misses a smaller but crucial group: contracts with intentional flaws designed for malicious purposes. We can’t effectively detect these yet because our training data lacks enough positive examples. This means our findings might miss a portion of the actual security threats on Ethereum. We’re working on gathering more data and improving our tools to identify these bad actors in the future.
- **Blind Spot 2: Source Code Vulnerabilities:** Some vulnerabilities stem from source code flaws that don’t directly translate to the bytecode level. We compared our detection accuracy with source code analysis tools to assess

this impact. While our bytecode approach generally performed better, for specific vulnerabilities like “uninitialized-local,” which are easier to identify in source code, our detection rate fell below average. This highlights the complementary strengths of both approaches, emphasizing the need for a comprehensive analysis strategy that combines both source code and bytecode insights for optimal vulnerability detection.

- **External Validity:**

Addressing external validity challenges is essential for building robust and trustworthy learning-based smart contract vulnerability detection tools. Ensuring the real-world effectiveness of learning-based smart contract vulnerability detection tools is crucial for building trust and widespread adoption. This involves addressing two key challenges:

- **Explainability and Interpretability:** Firstly, understanding why these models flag certain contracts as vulnerable can be difficult, which can impede trust and adoption. To partially address this, we have built SCoolS’s auto-exploit generator, which proves the identified vulnerabilities are indeed exploitable by external attackers. This adds a tangible layer of validation to the model’s predictions. Secondly, we need to ensure that the vulnerabilities detected by the model are relevant and exploitable in real-world scenarios. While the auto-exploit generator tackles this to some extent, future work will involve developing techniques to explain the model’s reasoning, providing insights into the importance of different features, and empowering developers to validate the findings themselves. These steps will further solidify the trustworthiness and real-world applicability of our vulnerability detection tools.
- **Evolving Vulnerabilities and Attack Patterns:** As new vulnerabilities emerge, models require continuous retraining to maintain effectiveness. DLVA and SCoolS have been designed with adaptability in mind, allowing for continuous model updates and retraining. It leverages domain knowledge for adaptive threat modeling, but this process does require

some positive training samples to initiate supervised or semi-supervised model retraining.

- Robustness Against Adversarial Attacks: Malicious actors could attempt to manipulate smart contracts or data fed to the model to evade detection. This could involve crafting subtle changes to the contract code, poisoning the training data with false positives or negatives, or exploiting specific vulnerabilities in the model architecture. Future work will explore more advanced techniques like adversarial training and resilient model architectures to further harden our tools' defenses. This involves deliberately exposing the model to adversarial examples during training, allowing it to learn to identify and resist such attacks. Additionally, research into more robust model architectures, such as those with built-in redundancy or fault tolerance, could further improve defense capabilities.

Chapter 4

Supervised Deep Learning: DLVA

In this chapter, we focus on detecting of potential security vulnerabilities in smart contracts using a supervised deep learning methodology. Specifically, our focus is on leveraging advanced computational techniques to accurately detect and classify these vulnerabilities in a systematic and efficient manner.

Smart contracts represent a highly promising and attractive concept within the realm of Ethereum blockchain technology, as they offer several key advantages, including immutability, transparency, and decentralization. These features make Ethereum smart contracts an appealing option for a wide range of applications. However, the adoption of smart contracts also poses unique challenges, particularly in regards to security. Given the potential financial and reputational risks associated with smart contract vulnerabilities, vulnerability analyzers are needed to proactively identify and mitigate potential security threats before they result in significant losses.

We introduce *Deep Learning Vulnerability Analyzer (DLVA)* [3, 4, 5], a vulnerability detection tool for Ethereum smart contracts based on powerful deep learning techniques adapted for smart contract bytecode. We train DLVA to judge bytecode *even though the supervising oracle, Slither, can only judge source code*. DLVA’s training algorithm is general: we “extend” a source code analysis to bytecode without any manual feature engineering, predefined patterns, or expert rules. DLVA’s training algorithm is also robust: it overcame a 1.25% error rate mislabeled contracts, and—the student surpassing the teacher—found vulnerable contracts that Slither mislabeled. In addition to extending a source code analyzer to bytecode, DLVA is much faster than conventional tools for smart contract vulnerability detection based on formal methods: DLVA checks contracts for 29 vulnerabilities in 0.2 seconds, a

speedup of 5-1,000x+ compared to traditional tools that do not scale nearly as well as program complexity and length grows.

DLVA has three key components. First, *Smart Contract to Vector* (SC2V) uses neural networks to map arbitrary smart contract bytecode to an high-dimensional floating-point vector. We benchmark SC2V against 4 state-of-the-art graph neural networks and show that it improves model differentiation by an average of 2.2%. Second, *Sibling Detector* (SD) classifies contracts when a target contract’s vector is Euclidian-close to a labeled contract’s vector in a training set; although only able to judge 55.7% of the contracts in our test set, it has an average Slither-predictive accuracy of 97.4% with a false positive rate of only 0.1%. Third, *Core Classifier* (CC) uses neural networks to infer vulnerable contracts regardless of vector distance. We benchmark DLVA’s CC with 10 “off-the-shelf” machine learning techniques and show that the CC improves average accuracy by 11.3%. Overall, DLVA predicts Slither’s labels with an overall accuracy of 92.7% and associated false positive rate of 7.2%.

Lastly, we benchmark DLVA against eleven well-known smart contract analysis tools. Despite using much less analysis time, DLVA completed every query, leading the pack with an average accuracy of 99.7%, pleasingly balancing high true positive rates with low false positive rates.

4.1 Introduction

The transparency of the smart contract bytecode, which is publicly accessible, allows potential attackers to thoroughly analyze the code for vulnerabilities [93]. This poses a significant risk, especially considering that some smart contracts manage digital assets with a substantial combined value in the hundreds of millions of US dollars. As a result, the motivation for attackers to exploit vulnerabilities in these contracts is exceedingly high, with potentially severe consequences for the affected parties.

We developed the *Deep Learning Vulnerability Analyzer* (DLVA) to help developers and users of Ethereum smart contracts detect security vulnerabilities. DLVA uses deep learning (neural networks) to analyze smart contracts. DLVA has no built-in expert rules or heuristics, learning which contracts are vulnerable during an

initial training phase.

We focus on Ethereum blockchain since it has the largest developer and user bases: about 6,000 active monthly developers (5x growth in monthly active Ethereum developers, from 1,084 during 2018 to 5,819 during 2022. 800+ new Ethereum developers per month since February 2021) [27]. Ethereum distributed applications (dApps) target domains including financial services, entertainment, and decentralized organizations. Ethereum smart contracts are usually written in high-level programming languages such as Solidity, before being compiled to Ethereum Virtual Machine (EVM) bytecode and deployed on the blockchain [145]. Ethereum smart contracts pair (1) a set of functions with (2) dedicated state stored in the blockchain. In turn, functions are sequences of EVM instructions that define how the smart contract behaves. Once deployed on the blockchain, the code of a smart contract is immutable and publicly available. Unfortunately, being computer programs, smart contracts are prone to bugs.

Bugs occur in smart contracts for many reasons, *e.g.* the semantics for Ethereum Virtual Machine (EVM) instructions is more subtle than is typically understood [118]. Poor software engineering techniques, *e.g.* widespread copying/pasting/modifying [50, 78] lead to rapid propagation of buggy code.

Since smart contract bytecode—and for about a third of the contracts, source code—is public, attackers can analyze a smart contract’s code for vulnerabilities [93, 163]. With some contracts controlling digital assets valued in the hundreds of millions of US dollars, the motivation to attack is significant. Smart contract bugs have caused major financial losses, with various bugs costing tens or even hundreds of millions of US dollars [121, 129]. Unlike with conventional financial systems, users typically have no recourse to recover losses.

Approximately two-thirds of *unique* smart contracts do not have source code available, but most previous vulnerability analyzers require (or at least meaningfully benefit from) source code availability. *DLVA works directly on bytecode*. Moreover, most previous tools require significant time to analyze contracts, especially as the contracts get longer. *DLVA checks a typical contract in 0.2 seconds*, 5-1,000+ times faster than competitors, enabling vulnerability detection at scale.

We trained DLVA using contracts labeled by the Slither [45] static analyzer. Slither is state of the art but requires source code, and so can only label 32.6% of

the unique contracts in our data set. *Although Slither can only label source code, we train DLVA to judge bytecode, thus “extending” a source code analyzer to bytecode.* Slither taught DLVA 29 vulnerabilities for long contracts (750+ opcodes) and 21 for shorter contracts.

Previous bug-finding approaches for smart contracts fall into three camps: static analyzers, fuzzing, and machine learning. Most previous work has been in the static analyzers camp, *e.g.* Oyente [93], Mythril [100], Osiris [134], SmartCheck [132], and Slither [45]. Most require source code, although a few can handle bytecode. Fuzzing is a dynamic analysis technique that attempts to falsify user-defined predicates or assertions. Contractfuzzer [76] and Echidna [59] are important examples.

DLVA is the first publicly available smart contract vulnerability analyzer using deep learning (graph neural nets), but a few pioneers have tried other machine learning techniques; unfortunately, only SaferSC [128], SMARTEMBED [50, 49, 51], and SoliAudit [89] have an available tool, enabling benchmarking.

Figure 4.1 benchmarks DLVA against eleven competitors. DLVA is on the far right. We use bar-and-whiskers where star \star represents the mean and plus $+$ represents outliers. Our average Completion Rate (*i.e.*, the percentage of contracts for which a tool produces an answer, the higher the better) is 100.0%. Our average accuracy is 99.7% (the higher the better), with a True Positive Rate (*i.e.*, detection rate; the higher the better) of 98.7% and a False Positive Rate (*i.e.*, false alarm rate; the lower the better) of 0.6%. Our average analysis time per contract (the graph is in log scale, lower better) is 0.2 seconds. We discuss Figure 4.1 in more detail in §4.3.4.

Smart learning pays off: DLVA beats Slither on every statistic except for TPR (where it lags by 0.7%). Recall also that Slither requires source whereas DLVA needs only bytecode.

Our main contributions are as follows:

- 1) §4.2.2, 4.2.3.1, 4.3.2 We develop a Smart Contract to Vector (SC2V) engine that maps smart contract bytecode into a high-dimensional floating-point vector space. SC2V uses a mix of neural nets trained in both unsupervised and supervised manners. We use Slither for supervision, labeling each contract as vulnerable or non-vulnerable for each of the 29 vulnerabilities we handle. *We provide no expert rules or other “hints” during training.* We evaluate the

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

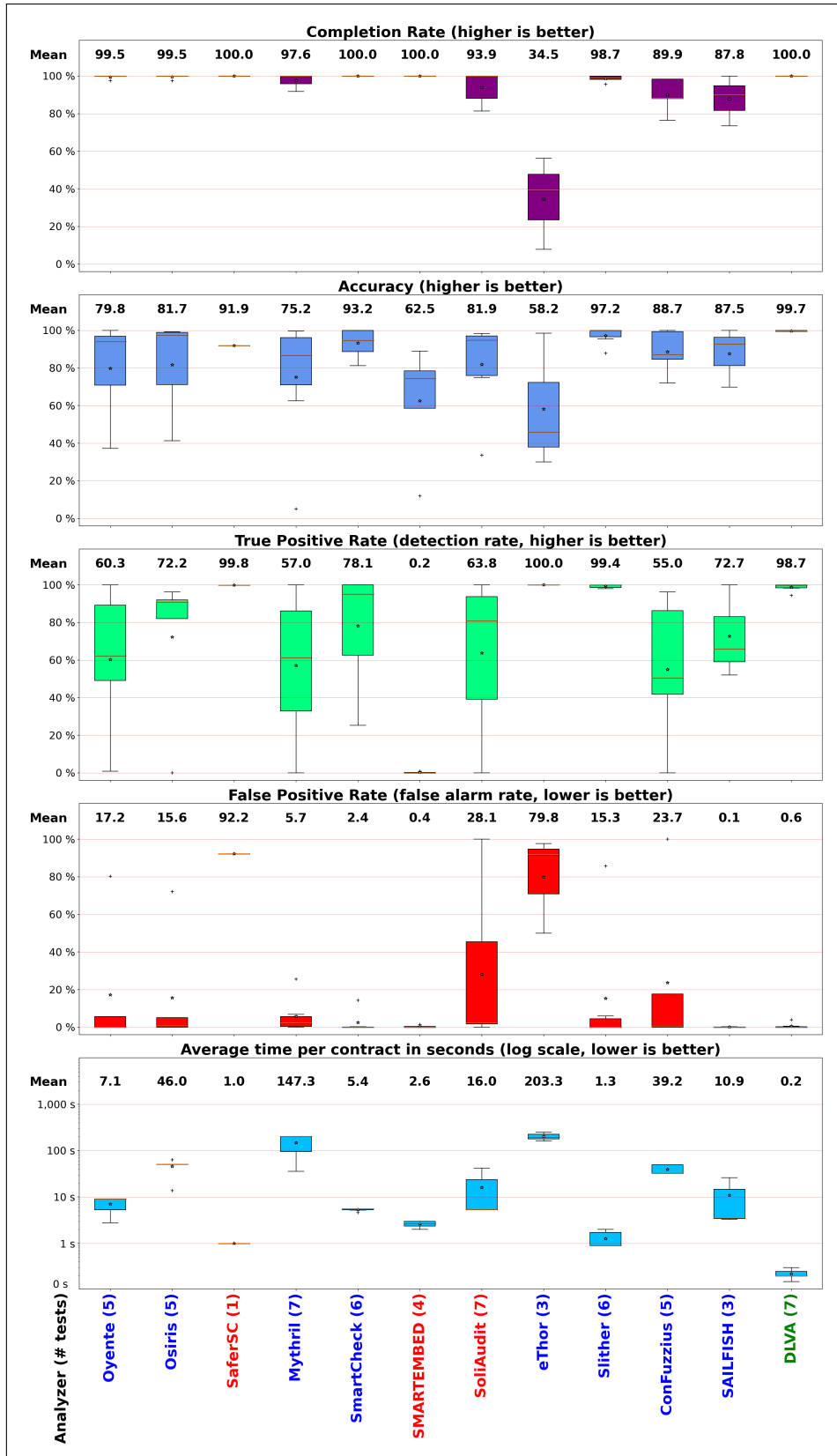


Figure 4.1: DLVA vs. alternatives on the *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11], and *SolidiFI*_{benchmark} [13]; star \star indicates the mean; plus $+$ indicates outliers

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

SC2V engine against four state-of-the-art graph neural networks and show it is 2.2% better than the average competitor and 1.2% better than the best.

- 2) §4.2.5, 4.3.3.1 Our Sibling Detector (SD) classifies contracts according to the labels of other contracts Euclidian-nearby in the vector space. Our SD is highly accurate, showing the quality of SC2V: on the 55.7% of contracts in our test set that it can judge, it has an accuracy (to Slither) of 97.4% and an associated FPR of only 0.1%.
- 3) §4.2.3.2, 4.3.3.1, 4.3.2 We design the Core Classifier (CC) of DLVA using additional neural networks, trained in a supervised manner using the same labeled dataset as SC2V. On the “harder” 44.3% of our test set, the CC has an accuracy (to Slither) of 80.0% with an associated FPR of 21.4%. We evaluate the CC against ten off-the-shelf machine learning methods and show that it beats the average competitor by 11.3% and the best by 8.4%.
- 4) §4.2, 4.3.3.1 DLVA is the combination of SC2V, SD, and CC. This whole is greater than its parts: DLVA judges every contract in the test set, with an average accuracy (to Slither) of 87.7% and FPR of 12.0%.
- 5) §4.2.6, 4.3.3.2 Small contracts are simpler than larger ones. We tweak our design to better handle such contracts and retrain. On small contracts, DLVA has an average accuracy (to Slither) of 97.6% with a FPR of 2.3%.
- 6) §4.2, 4.3.3.3 Accordingly, DLVA’s overall accuracy (average of large and small) is 92.7% with a FPR of 7.2%.
- 7) §4.3.1, 4.3.4 We propose and evaluate six datasets to benchmark DLVA and its components. As presented in Figure 4.1, we benchmark DLVA against eight static analyzers and three learning-based analyzers

In addition to the main contributions itemized above, the discussion of related work in §4.4. Some supplemental material is included in the appendix: Appendix A contains a discussion of alternative deep learning models we explored, which may be of interest to those building related tools.

DLVA availability and ethical considerations Any vulnerability analyzer can be used with ill intent. Blockchains are tricky for responsible disclosure [20]. Not only are attackers incentivized to find and attack vulnerabilities, but due to the pseudonymous nature of the blockchain, it is impossible to quietly inform participants of discovered vulnerabilities.

On the other hand, since DLVA requires labelled data sets to train, none of our detected vulnerabilities are “zero-day.” Moreover, honest actors benefit from DLVA too: everyone wants to know if the contracts they use are vulnerable.

On balance, the community benefits from access to DLVA, and so like other smart contract vulnerability analyzers [93, 134, 100, 132, 89, 116, 45, 133, 22], we will release DLVA.

DLVA is available for download from <https://secartifacts.github.io/use-nixsec2023/appendix-files/sec23winterae-final67.pdf> [5]. The instructions contain explanation for how to analyze a single contract or a batch of contracts. It takes 1–2 minutes to load the models into memory (≈ 2 seconds per model); afterwards, each contract is judged extremely quickly (≈ 0.2 seconds per contract).

DLVA is easy to operate, delivers quick results, and has demonstrated high accuracy in identifying vulnerabilities. It has user-friendly interface and efficient performance make it highly suitable for integration into the development process by engineers. The adoption of DLVA has the potential to significantly enhance the security and reliability of smart contracts, mitigating the risks associated with vulnerabilities and contributing to the advancement of blockchain technology. Further research can be done to explore DLVA’s potential for other cybersecurity applications beyond smart contract vulnerability classification.

4.2 Designing DLVA

Given a smart contract c (expressed in bytecode) as input to DLVA. For each vulnerability v of the 29 vulnerabilities listed in Table 4.1, our Deep Learning Vulnerability Analyzer’s job is to predict label c_v , where $c_v = 1$ means that c is vulnerable to v and $c_v = 0$ means c is secure from v .

Developing a tool that uses deep learning involves several steps. First, the overall architecture must be designed. Second, the resulting model must be trained on a

suitable *training* data set. Third, substantial testing with a disjoint *validation* set is used to tune *hyperparameters*. These steps are the focus of §4.2. Evaluating the model on (disjoint) *testing* sets is in §4.3.

Table 4.1: 29 vulnerabilities in *EthereumSC_{large}* (200+ times); ★ indicates 21 vulnerabilities in *EthereumSC_{small}* (30+ times)

	Smart contract vulnerabilities	Large	Small
High Severity	★ shadowing-state (SWC-119): state variables with multiple definitions at contract and function level.	3,602	52
	★ suicidal (SWC-106): the <code>selfdestruct</code> instruction that is triggered by an arbitrary account.	374	49
	★ uninitialized-state (SWC-109): local storage variables are not initialized properly, and can point to unexpected storage locations in the contract.	3,260	56
	★ arbitrary-send : unprotected call to a function sending Ether to an arbitrary address.	6,499	338
	★ controlled-array-length : functions that allow direct assignment of an array’s length.	5,282	61
	★ controlled-delegatecall (SWC-112): <code>delegatecall</code> or <code>callcode</code> instructions to external address.	1,485	37
	★ reentrancy-eth (SWC-107): usage of the fallback function to re-execute function again, before the state variable is changed (a.k.a. recursive call attack); reentrancies without Ether not reported.	3,962	39
	★ unchecked-transfer : the return value of an external transfer/transferFrom call is not checked.	14,151	262
	★ erc20-interface : incorrect return values for ERC20 functions.	9,017	161
★ incorrect-equality (SWC-132): improper use of strict equality comparisons.	8,604	95	
Continued on next page			

Table 4.1 – continued from previous page

	Smart contract vulnerabilities	Large	Small
Medium Severity	★ locked-ether : contract with a payable function, but without withdrawal ability.	12,164	398
	mapping-deletion : deleting a structure containing a mapping will not delete the mapping, and the remaining data may be used to breach the contract.	235	0
	shadowing-abstract : state variables shadowed from abstract contracts.	2,894	9
	tautology : expressions that are tautologies or contradictions.	2,441	17
	write-after-write : variables that are written but never read and written again.	467	6
	★ constant-function-asm : functions declared as constant/pure/view using assembly code.	4,019	49
	constant-function-state : calling to a constant/pure/view function that uses the <code>staticcall</code> opcode, which reverts in case of state modification, and breaking the contract execution.	210	3
	★ divide-before-multiply : imprecise arithmetic operations order; because division might truncate.	14,529	176
	★ reentrancy-no-eth (SWC-107): report reentrancies that don't involve Ether.	14,982	130
	tx-origin : tx.origin-based protection for authorization can be abused by a malicious contract if a legitimate user interacts with the malicious contract.	347	19
	★ unchecked-lowlevel : return value of a low-level call is not checked.	1,419	68
	★ unchecked-send : return value of a send is not checked, so if the send fails, the Ether will be locked.	593	68
Continued on next page			

Table 4.1 – continued from previous page

	Smart contract vulnerabilities	Large	Small
	★ uninitialized-local (SWC-109): uninitialized local variables; if Ether is sent to them, it will be lost.	6,843	114
	★ unused-return (SWC-104): return value of an external call is not stored in a local or state variable.	11,222	2,427
Low Severity	★ incorrect-modifier : modifiers that can return the default value, that can be misleading for the caller.	1,273	171
	★ shadowing-builtin : shadowing built-in symbols using variables, functions, modifiers, or events.	1,536	35
	★ shadowing-local : shadowing using local variables.	26,259	174
	variable-scope : variable usage before declaration (<i>i.e.</i> , declared later or declared in another scope).	1,484	27
	void-cst : calling a constructor that is not implemented.	341	3

Overview DLVA’s design is in Figure 4.2. DLVA begins with the selection of a large training set, which is labeled for supervisory purposes as vulnerable or non-vulnerable for each attack vector. Next, a control-flow graph is extracted (§4.2.1).

The first neural net maps CFG Nodes to Vectors (N2V) using the Universal Sentence Encoder (USE), trained in unsupervised mode (§4.2.2). The second and third neural nets form the heart of DLVA. The Smart Contract to Vector (SC2V) engine maps smart contract into vectors; the Core Classifier (CC) classifies contracts as vulnerable or non-vulnerable by looking for 29 vulnerabilities. The design and training of these neural nets, including choices for hyperparameters, is in §4.2.3 and §4.2.4. Lastly, the Sibling Detector (SD) applies a simple heuristic to improve accuracy for “simple cases” (§4.2.5).

Once training has finished, analyzing a fresh contract proceeds as follows. First, bytecode is transformed to a CFG, and N2V summarizes its nodes into vectors. Next, SC2V uses those node summaries to summarize the entire CFG as a vector.

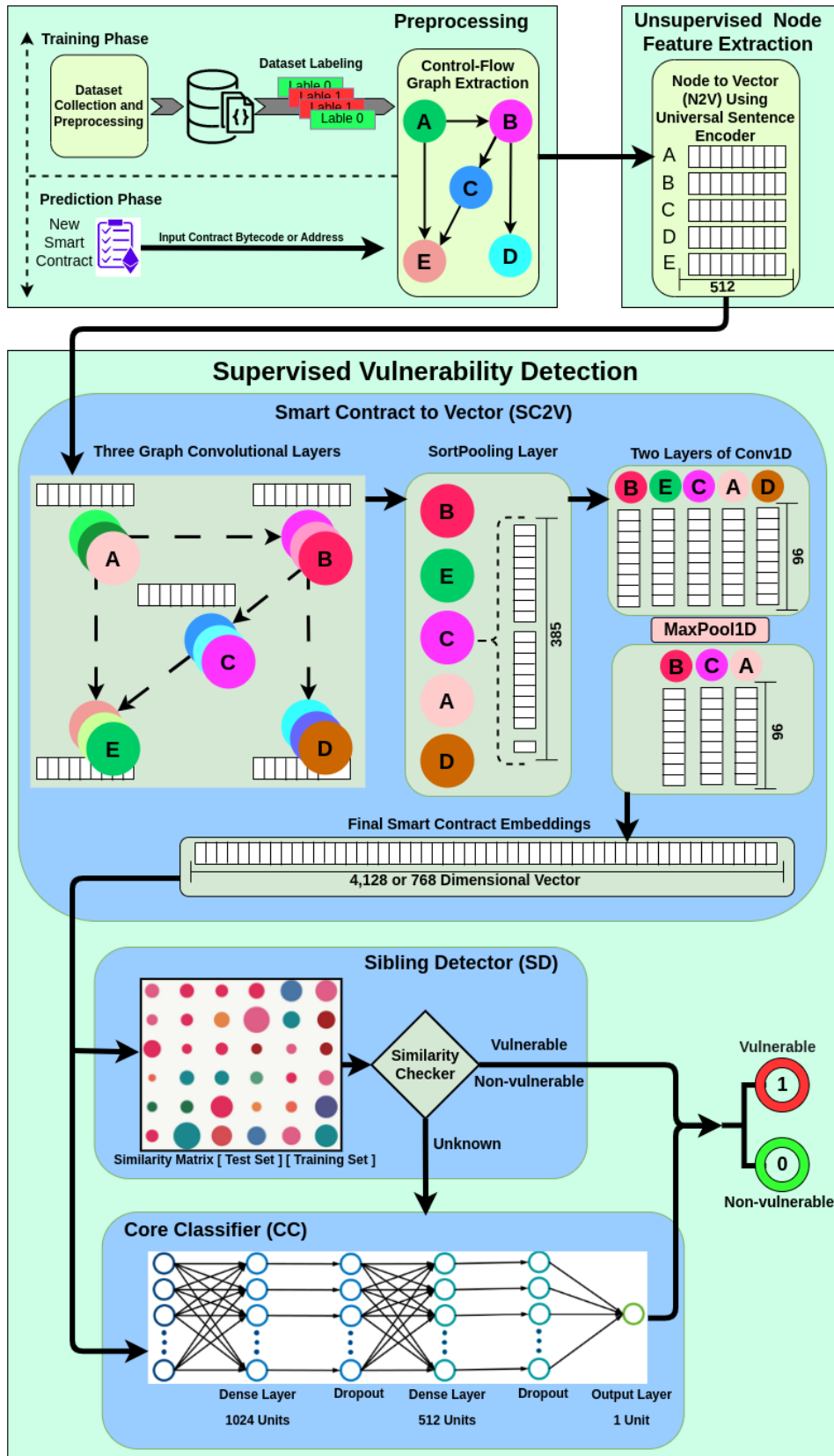


Figure 4.2: The Deep Learning Vulnerability Analyzer (DLVA).

This vector is given to the SD to see if it is close to a contract in the training set. If so, DLVA is done. If not, DLVA passes the vector to the CC, which renders its judgment. Finally, DLVA outputs a 29-dimensional binary vector corresponding to the 29 distinct vulnerabilities. Each element in the binary vector indicates: 1 means contract c is vulnerable, while 0 means contract c is secure from the corresponding vulnerability.

4.2.1 Preprocessing

Data Collection We downloaded our Ethereum smart contract data set from Google BigQuery [57]. The dataset contains 51,913,308 contracts, but many are redundant: 99.3%, in fact. Removing redundant contracts leaves 368,438 distinct contracts, which we dub the *EthereumSC* data set, summarized in the histogram in Figure 4.3.

Data labeling Two of our neural nets require labeled datasets to train. We chose Slither [45] (v0.8.0, committed on May 07, 2021, build 4b55839) to label because it covers a wide variety of vulnerabilities (74!), is more accurate than competitors for most vulnerabilities, and is relatively quick. Slither requires access to Solidity source code (rather than bytecode). The Slither analyzer employs an intermediate representation called SlithIR, which utilizes the static single assignment (SSA) technique to conduct a series of predefined analyses aimed at detecting vulnerabilities in Solidity code. These analyses assess the code’s impact levels, categorizing them as high, medium, low, or informational. Despite its effectiveness, Slither was only able to label 32.6% of the dataset, primarily due to the absence of source code for the majority of contracts on Etherscan, the leading blockchain explorer for Ethereum [43] (of the 368,438 contracts in our data set, only 120,365 (32.6%) had source available). It took 13.6 days (using 1 core/16gb) for Slither to label them.

Quality training requires a reasonable number of positive examples, so we chose the 29 vulnerabilities that occurred at least 200 times for DLVA. Although some of the 29 vulnerabilities are more serious than others, it was troubling to discover that Slither considered only 37,574 contracts (31.3%) pristine from all 29 vulnerabilities.

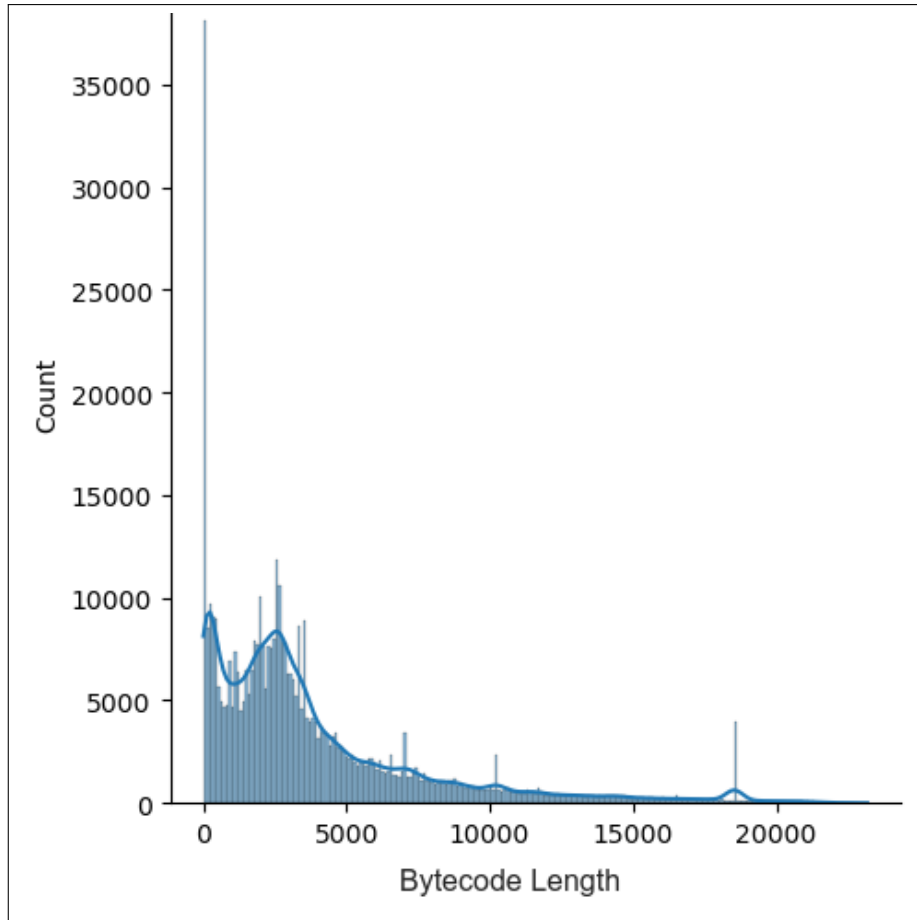


Figure 4.3: EthereumSC Data Set Histogram.

The remaining 82,609 vulnerable contracts had vulnerabilities distributed as shown in Table 4.1 (some contracts exhibit multiple vulnerabilities).

DLVA must cope with messy realities, among them that Slither is impressive, but not foolproof. A manual inspection of 50 positive “reentrancy” vulnerabilities provided some evidence for a false positive rate of approximately 10% [45].

Figure 4.4 visually presents the vulnerability frequencies within the EthereumSC dataset, showing the distribution and occurrence of each vulnerability type with a bar graph. Each bar corresponds to a specific vulnerability type, while the height of the bars signifies the frequency or occurrence of that particular vulnerability type within the dataset.

Control-Flow Graph extraction A Control-Flow Graph (CFG) makes a program’s structure more apparent than a list of syntactic tokens does. We use

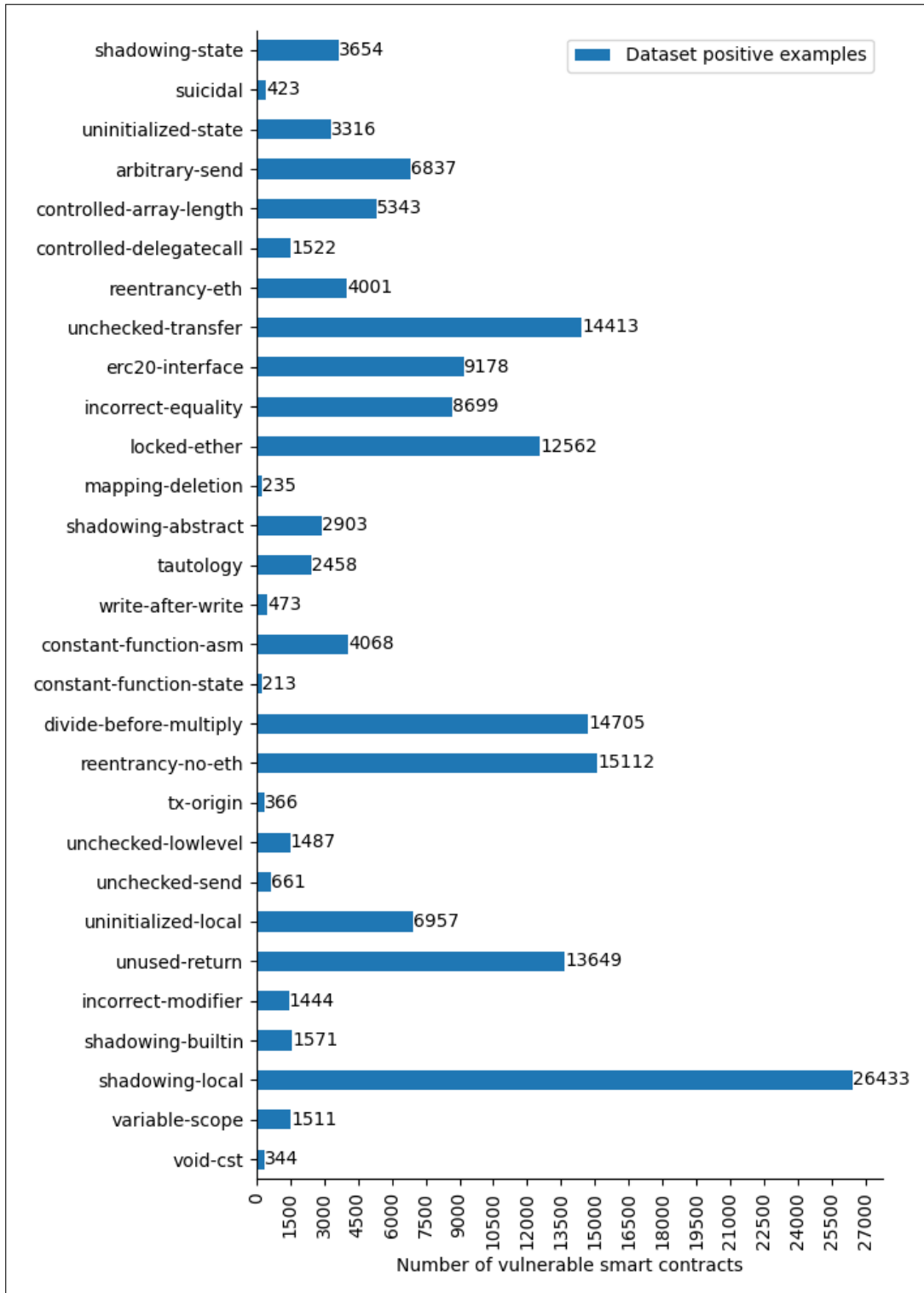


Figure 4.4: Vulnerability Frequencies of EthereumSC Dataset.

EtherSolve [37] to disassemble a smart contract [145] and generate the associated opcode CFG. EtherSolve failed to create a CFG for 182 contracts (0.1% of the labelled data set), leaving us with 120,183 contracts suitable for training and testing. The average contract has 228 basic blocks, with 551 edges between them.

Dividing our dataset The deep learning techniques in DLVA work better if trained on contracts of similar size to the contract being analyzed, so we split the 120,183 distinct contracts with labels in *EthereumSC* into two datasets depending on length. The 7,017 contracts with fewer than 750 opcodes become the *EthereumSC_{small}* data set, whereas the 113,166 contracts with between 750 and 10,000 opcodes become the *EthereumSC_{large}* data sets. Both data sets are public [9, 8].

As is typical, we divide each data set into three disjoint subsets. The first 60% (in the order given by the Google BigQuery after filtering) we call the “training set,” the next 20% is the “validation set,” and the last 20% is the “test set.”

4.2.2 Unsupervised Node Feature Extraction: N2V

Sophisticated machine learning models typically work with numerical feature vectors rather than text. Our Node to Vector (N2V) component translates the opcode text within each CFG node (basic block) into such a feature vector to enable more sophisticated processing. We treat each basic block as a textual sentence of instructions (e.g. “PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE...”). We then use the *Universal Sentence Encoder (USE)* [29] to transform these sentences into 512-dimensional vectors. We train USE by feeding it the ≈ 21.9 million basic blocks in our training & validation sets. We do not provide any expert rules or guidance.

There are two variants of USE. The Transformer Architecture (TA) [138] yields more accurate models than Deep Averaging Networks (DAN) [74], at the cost of increased model complexity. We found the cost of TA too high: the 20-core/96-gb time-unlimited configuration ran out of memory, and the 12 hours available on the 24-core/180gb configuration were insufficient to finish training. DAN can be trained in linear time and was accurate enough for our purposes. To train DAN to summarize basic blocks took only 10.5 hours with a 12-core/16gb configuration. Figure 4.5 illustrates DAN architecture where the embeddings for word and bi-grams present

in a sentence are averaged together then passed through 4-layer feed-forward neural network to produce 512-dimensional sentence embedding as output, the embeddings for word and bi-grams are learned during training.

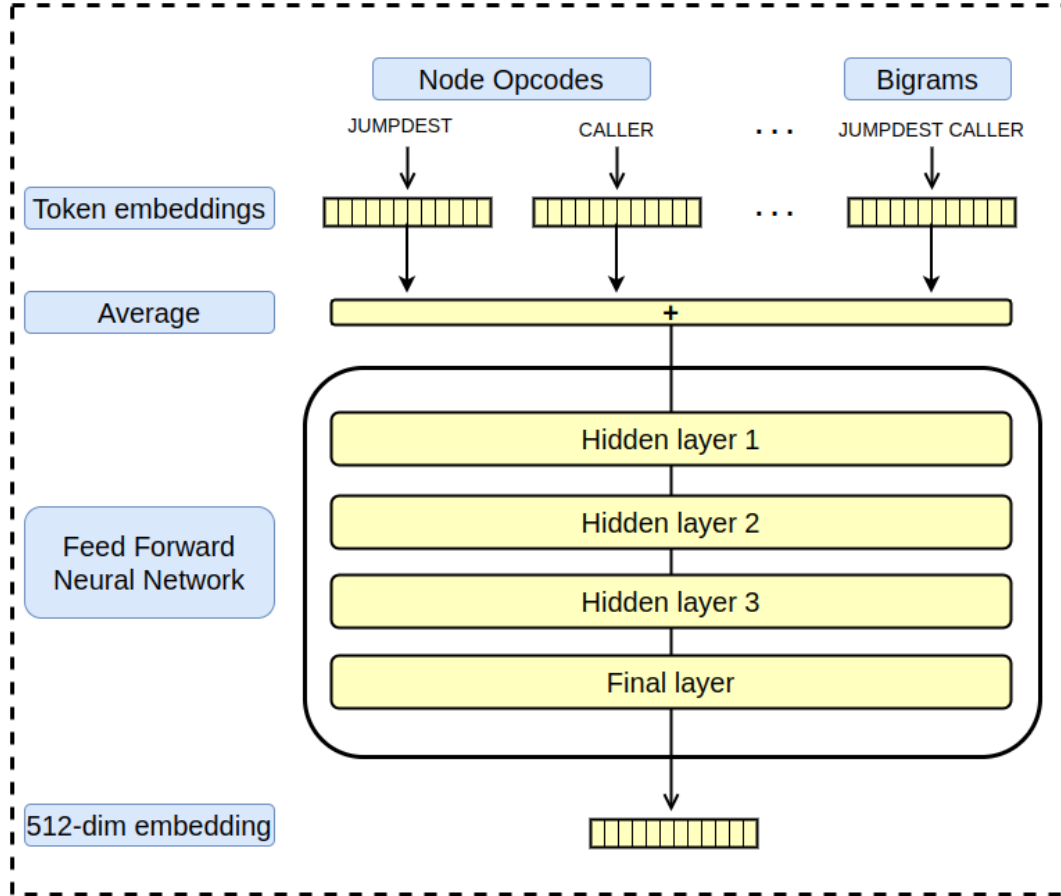


Figure 4.5: Sentence Embeddings using Universal Sentence Encoder based on Deep Averaging Network(DAN).

In Figure 4.6 we put USE/DAN’s summary of the smart contract from Figure 2.2. Basic block nodes are labeled by the address of the first opcode in the block, and the $f_0 \dots f_{511}$ give the corresponding 512-dimensional vector for the node.

4.2.3 Supervised Training: SC2V and CC

Our Smart Contract to Vector (SC2V) engine and Core Classifier (CC) form the heart of DLVA. As may be apparent from Figure 4.2, they have a relatively complex structure. Both are trained in supervised mode using the Slither-generated labels.

Node	f0	f1	...	f510	f511
0	-0.029561	0.022146	...	-0.036117	-0.116207
12	0.009187	0.018610	...	0.041887	-0.101014
...
368	-0.015426	0.032837	...	-0.062078	0.015451
371	-0.030309	0.054876	...	0.009683	-0.023788

Figure 4.6: USE-generated vector embeddings

Although they are distinct components, SC2V and CC are trained together. Key idea: *by coupling their training we increase the accuracy of our predictive models.* Rather than having one universal SC2V model and 29 vulnerability-specific CC models, we actually have 29 SC2V/CC model pairs.

4.2.3.1 Smart Contract to Vector (SC2V)

Key idea: *treating programs as a graph rather than just a sequence of textual symbols increases the accuracy of our models.* SC2V maps smart contract CFGs to high-dimensional vectors. It takes as input the graph structure of the CFG (which we handle with Python’s NetworkX library [63]), together with the USE-generated 512-dimensional vector embeddings for the associated basic blocks. We add self-loops to every node to increase the feedback in the neural net.

We use a modified *Graph Convolutional Network (GCN)* [80] combined with the SortPooling layer from the *Deep Graph Convolutional Neural Network (DGCNN)* [157] to analyze the complex structure of CFG graphs. We used three layers of GCN with 256, 128, and 1 neuron(s). The graph convolution aggregates a node’s information with the information from neighboring nodes. The three layers propagate information to neighboring nodes (up to three “hops” away, and including the node itself due to self-loops), extracting local substructure and inferring a consistent node ordering.

We incorporated the SortPooling layer to sort the nodes using the third graph convolution (whose output is a single channel). After sorting the node summaries in ascending order by this channel, SortPooling selects the highest-valued 100 nodes, whose summaries are from the GCN layers, *i.e.* a $256 + 128 + 1 = 385$ -dimensional vector. We feed these sets of 385-dimensional vectors into a pair of traditional *Conv1D*

convolutional layers, which further transform the 385-dimensional summaries into 96-dimensional vectors using rectified linear activation functions (“ReLU”). Between the Conv1D layers we use a *MaxPool1D* layer, which discards the half of the vectors with least magnitude. After the second Conv1D layer, we use a *Flatten* layer to produce the final vector representing the smart contract: contracts become 4,128-dimensional vectors. As an example, here is part of the vector for the smart contract from Figures 2.2 and 4.6:

[0.032677 0.027598 0.014528 0.021004 ... 0.107124 0.114875 0.121002]

4.2.3.2 Core Classifier (CC)

As shown in Figure 4.2, the last neural net is a Feed Forward Network (FFN). The goal of the CC is to use the contract embeddings generated by SC2V to predict the label for arbitrary contracts. The structure of the FFN is three “Dense” layers with 1,024, 512, and 1 neuron(s) respectively. These layers use standard activation functions to activate said neurons: the first two layers use ReLU activation functions, whereas the final layer uses a sigmoid activation function. Between the layers we put standard “Dropout” filters with a 0.5 cutoff.

4.2.4 Selection of hyperparameters

Machine learning hyperparameters play a crucial role in model performance. Hyperparameters are set prior to training and affect the behavior of the learning algorithm. In our case we considered the following hyperparameters:

- 1) the number of graph convolutional layers (from {2, 3, 4}) and associated neuron sizes (from {128, 256});
- 2) aggregation methods (from {mean, sum, sort-top-k}), followed by {1, 2, 3} layers of Conv1D to reduce the size of the final embedding vector;
- 3) the number of Dense FFN layers (from {1, 2, 3}) with associated neuron sizes (from {256, 512, 1024}); and
- 4) activation functions {Hyperbolic Tangent, ReLU}.

In total we have $3 \times 2 \times 3 \times 3 \times 3 \times 3 \times 2 = 972$ possible hyperparameter settings. To reach the design given in Figure 4.2, we selected constant-function-asm in *EthereumSC_{large}*, trained all 972 possible models for that vulnerability, and selected the hyperparameters that performed best according to the validation set (disjoint from the training and test sets). We chose constant-function-asm because the number of positive examples were in the middle of the pack; the vulnerability is mostly only detected by Slither, thus minimizing the danger of biasing testing due to overfitting; and because we believed Slither’s detector for this vulnerability was generally of high quality, with minimal false positives/negatives.

We used the same hyperparameters to train the other 28 models. In addition to giving us confidence that the models have not been overfitted, this implies that our architecture is relatively generic for smart contract vulnerability detection. With support of NUS High Performance Computing (NUS HPC)¹ for research computation, we able to run 1,112 experiments with different hyperparameters to find best parameters for DLVA design. DLVA does not rely on any manually designed expert rules or other human-generated hints. Accordingly, given suitable labeling oracles, training DLVA to recognize additional vulnerabilities is straightforward (we do this in §4.3.2 and §4.3.4).

4.2.5 Sibling Detector (SD)

Given the smart contract embeddings generated by SC2V, we create a similarity matrix using Euclidean distance: $\sqrt{\sum_{i=1}^N (Q_i - P_i)^2}$, where Q is a (previously unseen) contract embedding vector from the test set and P is a contract embedding vector from the training set (with known label). The Sibling Detector labels Q with the same label as the closest contract in the training set, as long as one exists within distance 0.1. Otherwise, the SD reports “unknown.” SD starts with a distance of 0.0 and gradually increases it by 0.00001 until a contact is found or the maximum allowable distance of 0.1 is reached, whichever comes first. Sometimes Q has multiple neighbors whose distances to Q are within 0.00001. When this happens, the SD counts votes instead; if a strict majority are vulnerable, then SD reports Q as “vulnerable.”

¹NUS HPC — <https://hpcportal.nus.edu.sg/>

4.2.6 Tweaking for smaller contracts

With the overall design settled, we make a few tweaks to better handle shorter contracts (under 750 opcodes). Since there are many fewer distinct small contracts than large ones, we were only able to train 21 of the 29 vulnerability models on the *EthereumSC_{small}* data set, despite lowering the minimum threshold to only 30 positive occurrences; we mark those 21 vulnerabilities in Table 4.1 with a \star .

We tweak SC2V’s SortPooling layer² to select the highest-valued 30 nodes (down from 100), which induces the Flatten layer to produce a 768-dimensional vector (down from 4,128). We use the same hyperparameters as for large contracts. The training set has fewer contracts so we turn off the SD.

4.2.7 Final details

Engineering choices We use Python’s Keras framework to train our models. We train for 100 epochs (stopping early when *callbacks=20*). In each epoch, Keras feeds the networks the training set and adjusts their weights using the loss function *binary_crossentropy*. Keras uses the *Adam optimizer* with a categorical cross-entropy loss function to train more efficiently. We set the *learning_rate* to $5e - 4$ and the *batch_size* to 512. We used BatchNormalization and Dropout layers to enhance the model’s generalization and prevent overfitting.

Training setup and time A training machine has 96 GB of memory and a 20-core “Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz” CPU. We used “CentOS Linux 7 (Core),” tensorflow 2.12.0 [1], tensorflow_hub 0.13.0, and Miniconda.

Since we had $972 + 28 + 21 = 1,021$ models to train, we used 10 training machines in parallel (200 cores, 960 GB). Total wall-clock training time was approximately four days.

²For large contracts, SortPool selects the top 100 nodes. If we did this for small contracts we would have to insert many dummy nodes, leading to significant artificial similarity. Accordingly, for small contracts SortPool selects 30 nodes.

4.3 Experiments and Evaluation

We evaluate the quality of DLVA from several complementary perspectives. In §4.3.1 we find and build benchmarks to help us understand DLVA’s performance from a number of complementary perspectives. In §4.3.2 we analyze the performance of our neural nets vs. various ML alternatives, comparing our SC2V engine with four GNNs and our Core Classifier with ten machine learning alternatives. In §4.3.3 we measure how well DLVA can predict our oracle Slither. In §4.3.4 we compare DLVA with eleven competitors. Lastly, in §4.3.5 we draw some conclusions from our experiments.

Testing setup Our test machine is a desktop with a 12-core 3.2 GHz Intel(R) Core(TM) i7-8700 and 16 GB of memory.

Evaluative metrics The learning-based classification results are divided into *true positives* (TP), *true negatives* (TN), *false positives* (FP), and *false negatives* (FN). Derivative metrics include *accuracy*; *true positive rate* (TPR), also known as recall, sensitivity, probability of detection, & hit rate; *false positive rate* (FPR), also known as probability of false alarms, & fall-out.

Although “accuracy” is important, it is not sufficient. Our data set is imbalanced: vulnerable smart contracts are scarce. Accordingly, a bogus model that simply labels all contracts as non-vulnerable will be surprisingly “accurate.” Accordingly we focus on TPR, which measures how often we catch vulnerable contracts; and FPR, which measures how often we issue false alarms. We formally define these metrics in §2.6.

4.3.1 Designing benchmark datasets

It is challenging to pin down “ground truth” for tools that operate over large data sets. Most human-curated benchmarks contain fewer than 100 examples, and many of those are unrealistic, *e.g.* stripped to minimum size for pedagogical purposes. This is not how vulnerabilities occur in the real world.

Machine-curated benchmarks, such as *EthereumSC_{large}* and *EthereumSC_{small}* that we defined in §4.2.1, can contain large numbers of realistic contracts. However, it is hard to be totally confident about their labels. Tools capable of processing contracts

Table 4.2: Datasets used for benchmarking DLVA

Dataset	Contracts	Vul	Sz	Ground Truth
<i>EthereumSC_{large}</i> [8]	22,634	29	L	Slither
<i>EthereumSC_{small}</i> [9]	1,381	21	S	Slither
<i>Elysium_{benchmark}</i> [7]	900 (57)	2	S	Peer-reviewed
<i>Reentrancy_{benchmark}</i> [11]	473 (472)	1	S	GP: Manual-labelled, GN: 2 static analyzers
<i>SolidiFI_{benchmark}</i> [13]	444	4	L	GP: Bug injection, GN: 5 static analyzers
<i>Zeus/eThor_{benchmark}</i> [10]	583	1	S/L	Peer-reviewed

at scale suffer from weaknesses that include: unsoundness, incompleteness, bugs, timeouts, and/or considering important classes of contracts to be out-of-scope.

The simple truth is that there are no existing benchmarks for Ethereum smart contract analysis tools that label large numbers of realistic contracts in a truly reliable way. We considered six benchmarks, summarized in Table 4.2, to help us evaluate DLVA from a variety of perspectives. “Contracts” indicates the number of contracts in test sets. Two benchmarks include contracts for which source code is unavailable; in this case the (parenthetical) gives the number that do have source code available. To help source code-based analyzers, in some cases we “lightly cleaned” the source code (*e.g.*, removing unicode from comments, moderately upgrading Solidity versions). “Vul” indicates the number of vulnerabilities; “Sz” whether the contracts are (mostly) Large (750–10,000 opcodes) or Small (less than 750); and the source of ground truth. All six datasets are disjoint from DLVA’s training sets and are publicly available [8, 9, 13, 7, 11, 10]. We discussed *EthereumSC_{large}* and *EthereumSC_{small}* in §4.2.1; they are used in §4.3.2 and §4.3.3 to tell *how closely DLVA corresponds to Slither*. Three others are used to *evaluate DLVA’s behaviour directly* in §4.3.4.

Elysium_{benchmark} [48] This human-curated data set is labelled by Torres *et al.*. *Elysium_{benchmark}* combines the SmartBugs [46] and Horus [47] data sets for “Re-entrancy” (reentrancy-eth, 75 positive examples, 825 negative) and “Parity bug” (suicidal, 823 positive examples, 77 negative). *Elysium_{benchmark}* contains many contracts that have been exploited in the real world. However, only 57 have available source (most suicided contracts are no longer available). We cleaned 2 contract

sources. Most contracts are under 750 opcodes, a few 750-900.

Reentrancy_{benchmark} We sourced 53 contracts that exhibit the “Re-entrancy” (e.g. reentrancy-eth) vulnerability from the academic literature [76, 78, 112], reported attacks on GitHub, and various Ethereum blogs. We took well-reported vulnerabilities as positive ground truth. Almost all (52) had source code on Etherscan, and when so we manually confirmed the vulnerability. We cleaned 19 contract sources. We considered the 420 contracts that *both* Slither and Mythril labelled as safe from the 1,381 contracts in our *EthereumSC_{small}* test set to be negative ground truth. All contracts are under 750 opcodes.

SolidiFI_{benchmark} To benchmark larger contracts, we used SolidiFI [54], a systematic method for bug injection that has been used in previous work to evaluate smart contract analysis tools [70, 156, 150, 103] to build the *SolidiFI_{benchmark}*.

Negative ground truth is established by the intersection of five static analyzers (Oyente, Mythril, Osiris, Smartcheck, and Slither). Positive ground truth is by injecting bugs from four different categories: Reentrancy (specifically, reentrancy-eth), Timestamp-Dependency, Overflow-Underflow, and tx.origin.

SolidiFI is used to inject security bugs into Solidity smart contracts. In [54] SolidiFI is employed to evaluate all ten analysis tools for Ethereum contracts, focusing on false negatives and false positives. SolidiFI, the tool, formulates distinct code snippets representative of exploitable vulnerabilities corresponding to each bug type. The benchmark is constructed as follows:

- (a) We select five static analyzers: Oyente 0.2.7, Mythril 0.21.20, Osiris 0.0.1, Smartcheck 2.0, and Slither 0.8.0. The analyzers flag various vulnerabilities, but four are largely in common[54]: Re-entrancy, Timestamp-Dependency, Overflow-Underflow, and tx.origin.
- (b) Out of our test set of 22,634 contracts in *EthereumSC_{large}*, we isolate the 553 contracts that are all considered safe for all four vulnerabilities, by all static analyzers that can detect them. We consider these 553 contracts to be the “negative ground truth.”

- (c) For each of these four vulnerabilities, SolidiFI has 40-45 code snippets exhibiting said vulnerability. By default, SolidiFI injects many vulnerabilities into target contracts. This makes the vulnerability detector’s task too easy. To make the task more challenging, we modified SolidiFI to inject only a single randomly-chosen vulnerability into a given contract. Starting from a negative contract, we thus reach a “positive ground truth.”
- (d) For each vulnerability, we inject into each safe contract in a single randomly-chosen place of source code, yielding $553 \times 4 = 2,212$ vulnerable contracts with unique bytecode for each contract after compilation. By the nature of the SolidiFI injection, none of these contracts have been seen before by any of the tools. Moreover, for a given vulnerability X, there will be 553 X-vulnerable contracts and 1,659 contracts that do not have X.
- (e) We divide the contracts into three buckets: 60% for training, 20% for validation, and 20% for testing.
- (f) This yields 111 contracts in the test set for each vulnerability (444 in total), with another 444 contracts in the validation set, and 1,324 in the training set.

We generated 2,212 contracts, of which 80% were reserved for training/validation, with 20%—*i.e.*, 444 in total—available for testing, with each vulnerability occurring exactly 111 times. All contracts have source available, all of which we cleaned. All are over 750 opcodes. The contracts are complex but the injected vulnerabilities are simple; accordingly, performance may be better than for real-world vulnerable contracts.

Zeus/eThor_{benchmark} [78, 116] A reentrancy benchmark used to evaluate Zeus [78] and eThor [116]. Zeus’s ground truth labels differ substantially from eThor’s, making results hard to interpret and reinforcing the slippery nature of ground truth. Contract sizes are a mix of small and large; the subset we used have source. We discuss this dataset in the Appendix of [4].

4.3.2 DLVA’s neural nets vs. alternatives

Node to Vector (N2V) Appendix §A discusses other models/techniques we considered for N2V before settling on the Universal Sentence Encoder [29], including *fastText* [21], *word2vec* [97], Recurrent Neural Networks (RNNs) such as Long Short-Term Memory Networks (LSTMs) [68, 52], and Bidirectional Long Short-Term Memory (BiLSTMs) [117].

Smart Contract to Vector (SC2V) To evaluate SC2V we used *SoliddiFI_{benchmark}*, since we consider its labels to be more reliable than *EthereumSC_{large}*. We used *SoliddiFI_{benchmark}*’s training set for five state-of-the-art networks: a *Graph Convolutional Network* (GCN) [80], a *Gated Graph Sequence Neural Network* (GGC) [86], a *Graph Isomorphism Network* (GIN) [149], a *Deep Graph Convolutional Neural Network* (DGCNN) [157], and of course our own SC2V. For consistency, we trained all five competitors with DLVA’s CC.

The results of our experiment are in Figure 4.7. We use the AUC “area under the receiver operating characteristic curve” metric, which measures the ability of the model to differentiate vulnerable from non-vulnerable cases, with higher scores better; AUC was explained in §2.6. SC2V has the highest score on Reentrancy and Overflow-Underflow, and ties with GCN for tx.origin; on Timestamp-Dependency, SC2V is a hair weaker than GCN. Averaged over all four vulnerabilities, SC2V leads with 99.0%, followed by GIN at 97.8%, GCN and GGC both at 97.5%, and finally DGCNN at 94.5%. Thus, SC2V beats the best competitor by 1.2% and the average competitor by 2.2%. SC2V wins outright for Reentrancy and Overflow/Underflow; ties for tx.origin; and comes in second for Timestamp-Dependency. SC2V performs better than competing models due to its more complex design: the convolutional layers of GCN and the SortPooling layer of DGCNN, followed by a pair of traditional convolutional layers.

The point of SC2V is to Euclidian-cluster smart contracts that share a specific vulnerability distinctly from contracts that are safe from that vulnerability. The underlying vectors for large contracts have 4,128 dimensions. We use SMOTEENN (combining over- and under-sampling using SMOTE and Edited Nearest Neighbours) to balance positive and negative examples. Afterward, we employ t-Distributed

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

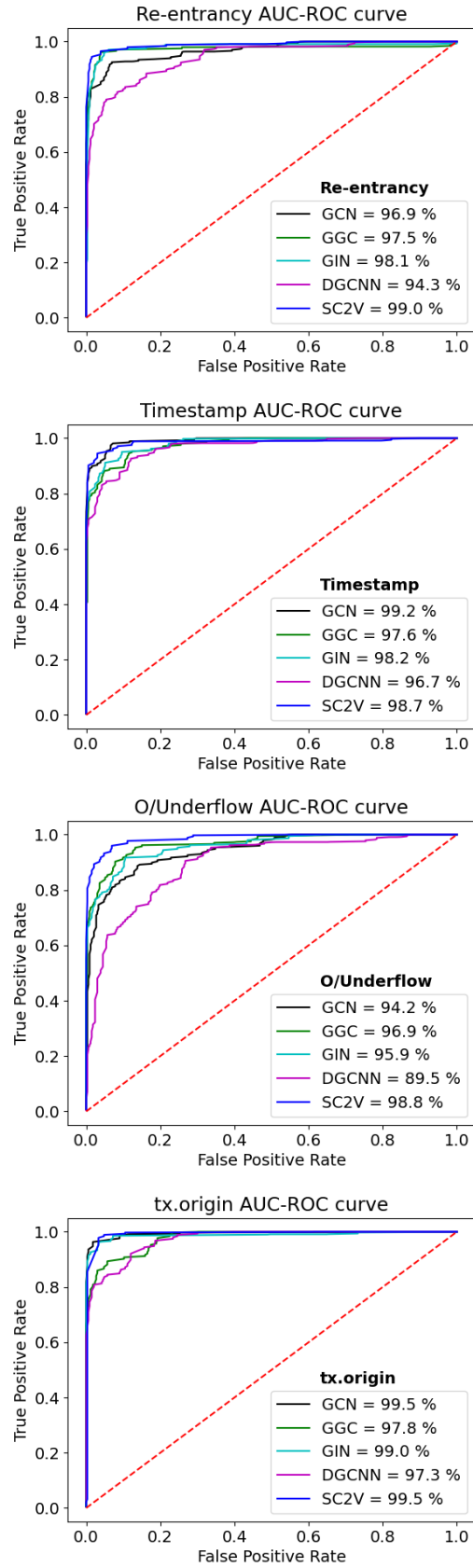


Figure 4.7: Evaluating SC2V vs. state-of-the-art GNNs

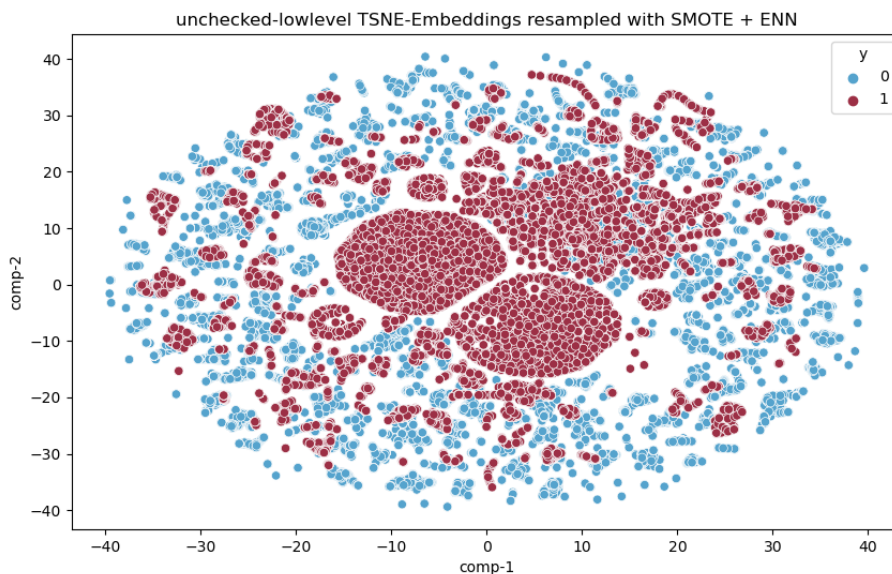


Figure 4.8: t-SNE-Embeddings for the “unchecked-lowlevel” Vulnerability.

Stochastic Neighbor Embedding (t-SNE) to compress into two dimensions for visualization purposes. Figure 4.8 does just this for the “unchecked-lowlevel” vulnerability, with Slither-labeled vulnerable contracts in red and safe contracts in blue. Even given this compression, the clustering effect is apparent.

Core Classifier (CC) In Figure 4.9 we benchmark CC against ten (well-trained) commonly used machine learning algorithms and one voting “meta-competitor.” We trained all competitors on the *EthereumSC_{large}* training/validation sets and tested using the associated test set (*cf.* §4.2.1). We graph accuracy (higher is better), the True Positive Rate (higher is better); and the False Positive Rate (lower is better).

The ten established competitors have average accuracy of 68.7% (MLP’s 71.1% is the highest). The voting meta-competitor reaches 71.6%. The CC’s average accuracy of 80.0% crushes the competition by 11.3% and 8.4%.

In fact, DLVA’s CC is more accurate than every other model, for every test. Moreover, the CC usually enjoys the highest/best TPR (or close), and the lowest/best FPR (or close). On a few tests the CC’s TPR is uninspiring: uninitialized-state and write-after-write are the most challenging. Fortunately, on those difficult vulnerabilities, the CC’s excellent FPR comes to the rescue. Conversely, the CC’s FPR is

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

uninspiring for constant-function-state; happily, it leads the pack on TPR.

The voting meta-competitor also performs well: well above average accuracy, for every test. But DLVA’s CC is better: it reduces the size of the error set—the set of contracts for which a given classifier goofs—by an average of 36%.

We put Table 4.3, which examines the competition from a “post-hoc” viewpoint. That is, while the meta-competitor does not know which of the 10 machine learning models will be the most accurate for a given vulnerability, in Table 4.3 we get to bet on the winners after the race has been run! Despite this advantage, DLVA’s CC reduces the average size of the error set by 30%.

DLVA-CC achieved an average accuracy of 80.0%, surpassing the average competitors accuracy of 68.94%. This indicates that DLVA’s CC effectively reduces the average size of the error set by approximately 36% $\approx (1 - \frac{100 - 80.0}{100 - 68.94}) * 100$. In addition, DLVA-CC even outperforms the highest competitor’s accuracy of 71.6%. This indicates that DLVA’s CC effectively reduces the typical size of the error set by approximately 30% $\approx (1 - \frac{100 - 80.0}{100 - 71.6}) * 100$.

Table 4.3: Best of the ten commonly used machine learning supervised binary classifiers results

Vulnerability	Classifier	Test size	TP	FP	TN	FN	Accuracy	TPR	TNR	FPR	FNR	AUC
shadowing-state	MLP	10037	290	1525	8038	184	72.6%	61.2%	84.1%	15.9%	38.8%	78.6%
suicidal	SVM	10037	58	3487	6482	10	75.2%	85.3%	65.0%	35.0%	14.7%	78.3%
uninitialized-state	MLP	10037	198	3851	5854	134	60.0%	59.6%	60.3%	39.7%	40.4%	62.9%
arbitrary-send	MLP	10037	653	2394	6755	235	73.7%	73.5%	73.8%	26.2%	26.5%	80.4%
controlled-array-length	XGB	10037	526	2688	6653	170	73.4%	75.6%	71.2%	28.8%	24.4%	80.1%
controlled-delegatecall	SVM	10037	54	3059	6909	15	73.8%	78.3%	69.3%	30.7%	21.7%	76.1%
reentrancy-eth	MLP	10037	377	2701	6842	117	74.0%	76.3%	71.7%	28.3%	23.7%	81.1%
reentrancy-no-eth	ET	10037	1373	2438	5812	414	73.6%	76.8%	70.4%	29.6%	23.2%	79.6%
unchecked-transfer	SVM	10037	1428	2672	5647	290	75.5%	83.1%	67.9%	32.1%	16.9%	81.2%
erc20-interface	MLP	10037	522	3138	6157	220	68.3%	70.4%	66.2%	33.8%	29.6%	73.5%
incorrect-equality	SVM	10037	932	3161	5614	330	68.9%	73.9%	64.0%	36.0%	26.1%	73.6%
locked-ether	ET	10037	513	3881	5445	198	65.3%	72.2%	58.4%	41.6%	27.8%	69.3%
mapping-deletion	ET	10037	22	1954	8052	9	75.7%	71.0%	80.5%	19.5%	29.0%	79.5%
shadowing-abstract	XGB	10037	192	1997	7776	72	76.1%	72.7%	79.6%	20.4%	27.3%	82.6%
tautology	MLP	10037	223	3098	6620	96	69.0%	69.9%	68.1%	31.9%	30.1%	74.6%
write-after-write	KNN	10037	62	4004	5949	22	66.8%	73.8%	59.8%	40.2%	26.2%	67.5%
constant-function-asm	MLP	10037	423	2826	6723	65	78.5%	86.7%	70.4%	29.6%	13.3%	84.5%
constant-function-state	AB	10037	24	1906	8100	7	79.2%	77.4%	81.0%	19.0%	22.6%	84.2%
divide-before-multiply	SVM	10037	1361	2909	5268	499	68.8%	73.2%	64.4%	35.6%	26.8%	73.0%
tx-origin	MLP	10037	37	3487	6506	7	74.6%	84.1%	65.1%	34.9%	15.9%	77.3%
unchecked-lowlevel	LR	10037	53	2498	7469	17	75.3%	75.7%	74.9%	25.1%	24.3%	80.3%
unchecked-send	XGB	10037	69	3602	6356	10	75.6%	87.3%	63.8%	36.2%	12.7%	79.5%
uninitialized-local	SVM	10037	734	4161	4894	248	64.4%	74.7%	54.0%	46.0%	25.3%	68.1%
unused-return	XGB	10037	1378	2884	5494	281	74.3%	83.1%	65.6%	34.4%	16.9%	80.1%
incorrect-modifier	MLP	10037	104	3353	6533	47	67.5%	68.9%	66.1%	33.9%	31.1%	71.9%
shadowing-builtin	MLP	10037	115	2706	7172	44	72.5%	72.3%	72.6%	27.4%	27.7%	77.6%
shadowing-local	MLP	10037	1534	3025	4624	854	62.3%	64.2%	60.5%	39.5%	35.8%	68.1%
variable-scope	MLP	10037	194	3413	6382	48	72.7%	80.2%	65.2%	34.8%	19.8%	79.0%
void-cst	AB	10037	35	3747	6243	12	68.5%	74.5%	62.5%	37.5%	25.5%	73.2%

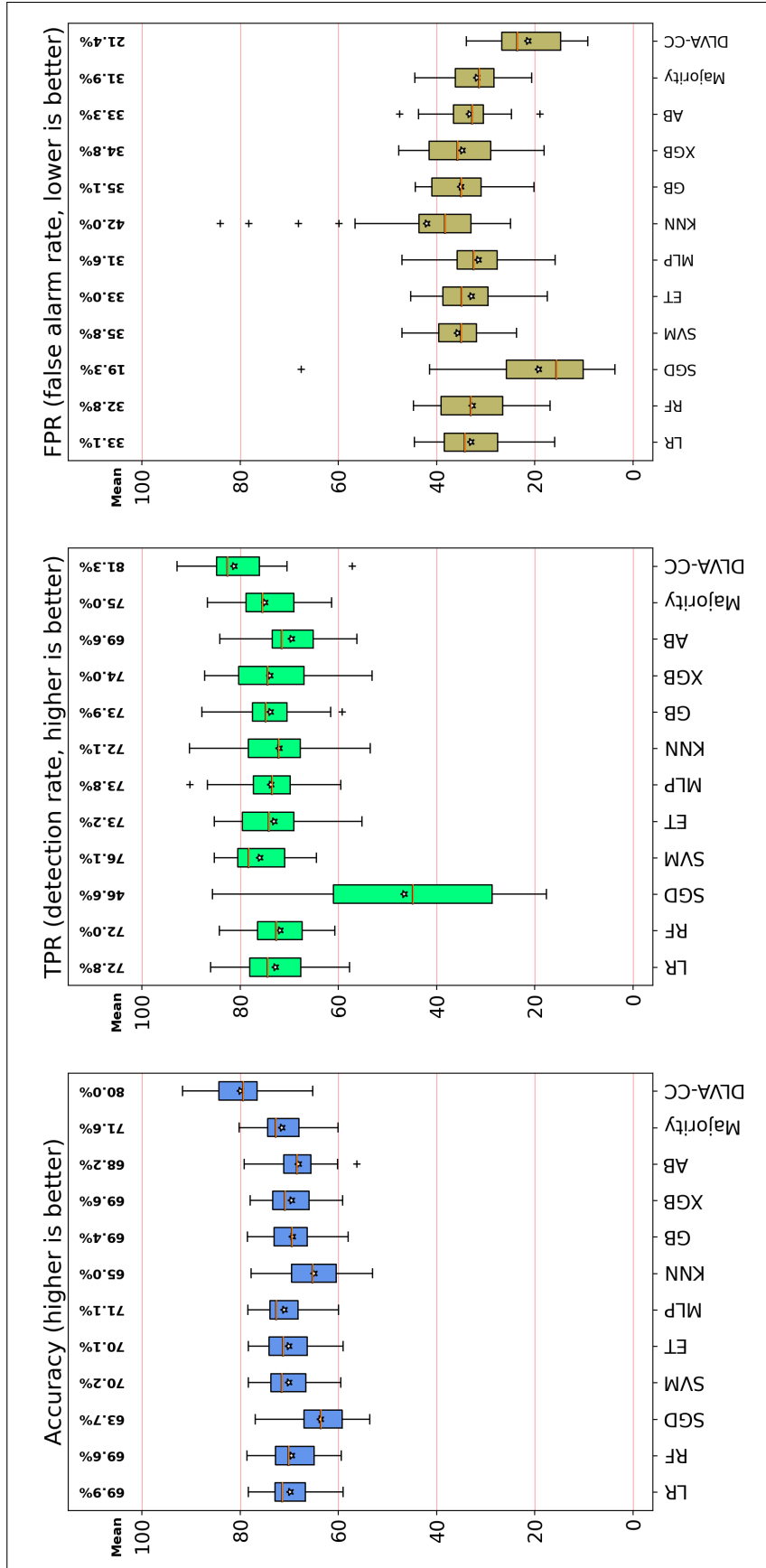


Figure 4.9: DLVA-CC vs. ten “off-the-shelf” ML classifiers and a majority voting strategy (★ is the average; + are outliers)

4.3.3 Evaluating DLVA’s models against Slither

Slither requires source code, whereas DLVA needs only bytecode. Only 32.6% of distinct contracts have source code available (§4.2.1); if DLVA accurately predicts Slither on those contracts, then it is probably accurately predicting how Slither would label the remaining 67.4%, where source code is unavailable.

Recall from §4.2.1 that we used the tool Slither to label two different datasets: *EthereumSC_{large}* and *EthereumSC_{small}*. We used 60% of both data sets for training, and a further 20% for validation/tuning. The final 20% were not used in the development of DLVA and are thus suitable for evaluation (recall that the data sets contain distinct contracts, so no contract in the test set has been seen during training/validation).

4.3.3.1 *EthereumSC_{large}* results

Figure 4.10 summarizes the evaluation of 29 vulnerabilities with labels in the dataset *EthereumSC_{large}*. We measure three key statistics: on the left, accuracy (higher is better); in the middle, the True Positive Rate (higher is better); and at the right, the False Positive Rate (lower is better).

Each subgraph shows four distinct tasks, labeled CC-only for the Core Classifier on the entire test set, SD-easy for the Sibling Detector on 55.7% of the test set, CC-hard for the Core Classifier on the remaining 44.3%, and DLVA (SD+CC) for DLVA as a whole.

Task CC-only: Core Classifier on entire dataset We first measure the Core Classifier (CC) against the entire 22,634-contract test set. Average accuracy is 86.0%, TPR is 86.1%, and FPR is 14.1%. Table 4.4 contains the details for each vulnerability. Our next goal is to show that we can do better by incorporating our Sibling Detector.

Task SD-easy: Sibling Detector The Sibling Detector looks for smart contracts in the test set that are “very close” to contracts in the training set. Our distance threshold of 0.1 balances applicability and accuracy. At 0.1, a healthy 55.7% of the contracts in the test are close to a training set contract. To study accuracy, we ran the experiments reported in Table 4.5. For the 12,597 test contracts (55.7%)

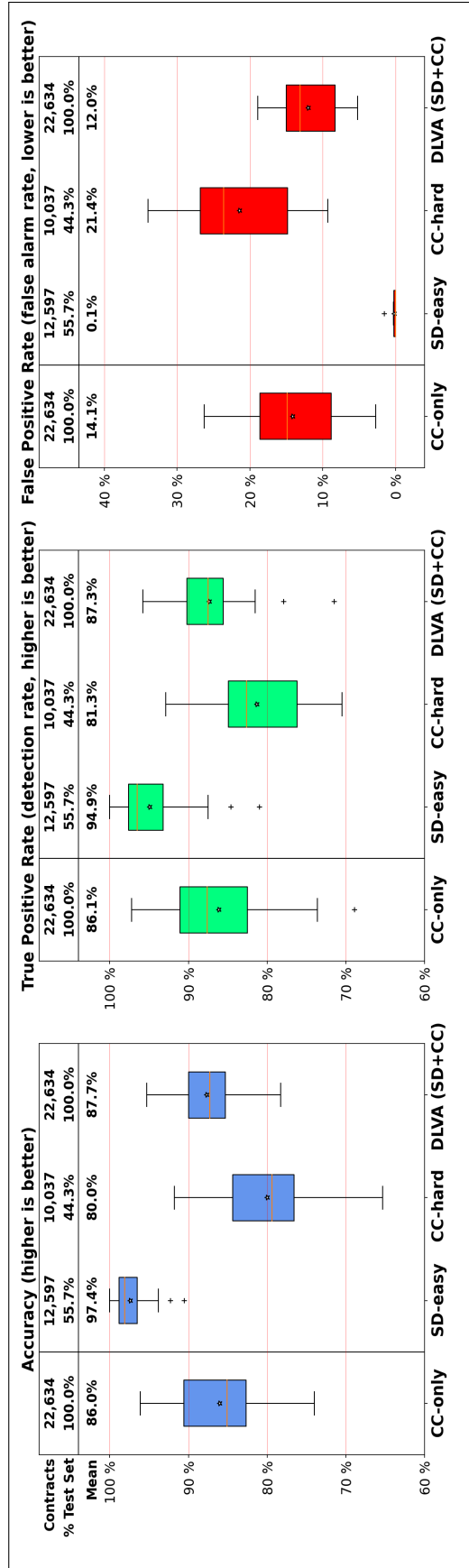


Figure 4.10: Deep Learning Vulnerability Analysis Tool Score Summary for 29 Vulnerabilities of *EthereumSC_{large}* Dataset (The star symbol * represents the average, while the plus + represents outliers)

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

 Table 4.4: DLVA Trained on *EthereumSC_{large}* (use DLVA’s core classifier only for the entire test set)

Vulnerability	Test size	TP	FP	TN	FN	Accuracy	TPR	TNR	FPR	FNR
shadowing-state	22634	537	2715	19193	193	80.6%	73.6%	87.6%	12.4%	26.4%
suicidal	22634	77	1456	21098	7	92.6%	91.7%	93.5%	6.5%	8.3%
uninitialized-state	22634	439	4599	17402	198	74.0%	68.9%	79.1%	20.9%	31.1%
arbitrary-send	22634	1227	3715	17577	119	86.9%	91.2%	82.6%	17.4%	8.8%
controlled-array-length	22634	939	2481	19121	97	89.6%	90.6%	88.5%	11.5%	9.4%
controlled-delegatecall	22634	266	1148	21195	29	92.5%	90.2%	94.9%	5.1%	9.8%
reentrancy-eth	22634	702	3816	18021	99	85.1%	87.6%	82.5%	17.5%	12.4%
reentrancy-no-eth	22634	2509	3153	16532	444	84.5%	85.0%	84.0%	16.0%	15.0%
unchecked-transfer	22634	2577	1948	17860	253	90.6%	91.1%	90.2%	9.8%	8.9%
erc20-interface	22634	1598	1845	19047	148	91.3%	91.5%	91.2%	8.8%	8.5%
incorrect-equality	22634	1561	5489	15378	210	80.9%	88.1%	73.7%	26.3%	11.9%
locked-ether	22634	2092	2366	17722	458	85.1%	82.0%	88.2%	11.8%	18.0%
mapping-deletion	22634	33	1869	20728	8	86.1%	80.5%	91.7%	8.3%	19.5%
shadowing-abstract	22634	570	1126	20889	53	93.2%	91.5%	94.9%	5.1%	8.5%
tautology	22634	391	4701	17455	91	80.0%	81.1%	78.8%	21.2%	18.9%
write-after-write	22634	73	5309	17232	24	75.9%	75.3%	76.4%	23.6%	24.7%
constant-function-asm	22634	794	1889	19932	23	94.3%	97.2%	91.3%	8.7%	2.8%
constant-function-state	22634	33	2864	19730	11	81.2%	75.0%	87.3%	12.7%	25.0%
divide-before-multiply	22634	2468	2935	16797	438	85.0%	84.9%	85.1%	14.9%	15.1%
tx-origin	22634	55	4191	18385	7	85.1%	88.7%	81.4%	18.6%	11.3%
unchecked-lowlevel	22634	276	611	21736	15	96.1%	94.8%	97.3%	2.7%	5.2%
unchecked-send	22634	116	3718	18796	8	88.5%	93.5%	83.5%	16.5%	6.5%
uninitialized-local	22634	1162	5170	16128	178	81.2%	86.7%	75.7%	24.3%	13.3%
unused-return	22634	1947	3991	16358	342	82.7%	85.1%	80.4%	19.6%	14.9%
incorrect-modifier	22634	219	2875	19513	31	87.4%	87.6%	87.2%	12.8%	12.4%
shadowing-builtin	22634	274	1174	21149	41	90.9%	87.0%	94.7%	5.3%	13.0%
shadowing-local	22634	4595	3416	13991	636	84.1%	87.8%	80.4%	19.6%	12.2%
variable-scope	22634	250	3457	18878	53	83.5%	82.5%	84.5%	15.5%	17.5%
void-cst	22634	59	3589	18981	9	85.4%	86.8%	84.1%	15.9%	13.2%

within 0.1 distance of a training contract, SD achieved an accuracy of 97.4% with an FPR of only 0.1%. Accuracy was never lower than 90.5% and the FPR was never higher than 1.5%. The most challenging metric was TPR. Although average TPR was 94.9%, variance was higher. On the 5 most challenging vulnerabilities, TPR was 81.0%–89.5%.

Task CC-hard: Core Classifier We plan to use the CC only when the SD reports “unknown,” *i.e.* the 10,037 contracts more than 0.1 away from any contract in the training set. The CC’s job here is harder than in CC-only, the contracts are less similar to those seen during training. Despite this restriction, the CC had an average accuracy of 80.0% with an average FPR of 21.4% and TPR of 81.3%. The results for individual vulnerabilities are in Table 4.6.

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

Table 4.5: DLVA Trained on *EthereumSC_{large}* (Sibling Detector Results)

Vulnerability	Test size ($d \leq 0.1$)	TP	FP	TN	FN	Accuracy	TPR	TNR	FPR	FNR	Test size ($d > 0.1$)
shadowing-state	12597	244	6	12335	12	97.6%	95.3%	100.0%	0.0%	4.7%	10037
suicidal	12597	14	0	12581	2	93.8%	87.5%	100.0%	0.0%	12.5%	10037
uninitialized-state	12597	273	6	12286	32	94.7%	89.5%	100.0%	0.0%	10.5%	10037
arbitrary-send	12597	444	12	12127	14	98.4%	96.9%	99.9%	0.1%	3.1%	10037
controlled-array-length	12597	316	22	12236	23	96.5%	93.2%	99.8%	0.2%	6.8%	10037
controlled-delegatecall	12597	219	2	12369	7	98.4%	96.9%	100.0%	0.0%	3.1%	10037
reentrancy-eth	12597	290	11	12279	17	97.2%	94.5%	99.9%	0.1%	5.5%	10037
reentrancy-no-eth	12597	1126	35	11396	40	98.1%	96.6%	99.7%	0.3%	3.4%	10037
unchecked-transfer	12597	1081	39	11446	31	98.4%	97.2%	99.7%	0.3%	2.8%	10037
erc20-interface	12597	1000	9	11584	4	99.8%	99.6%	99.9%	0.1%	0.4%	10037
incorrect-equality	12597	483	37	12052	25	97.4%	95.1%	99.7%	0.3%	4.9%	10037
locked-ether	12597	1810	24	10735	28	99.1%	98.5%	99.8%	0.2%	1.5%	10037
mapping-deletion	12597	10	0	12587	0	100.0%	100.0%	100.0%	0.0%	0.0%	10037
shadowing-abstract	12597	357	5	12233	2	99.7%	99.4%	100.0%	0.0%	0.6%	10037
tautology	12597	158	3	12432	4	98.8%	97.5%	100.0%	0.0%	2.5%	10037
write-after-write	12597	11	3	12581	2	92.3%	84.6%	100.0%	0.0%	15.4%	10037
constant-function-asm	12597	320	6	12263	8	98.8%	97.6%	100.0%	0.0%	2.4%	10037
constant-function-state	12597	12	0	12584	1	96.2%	92.3%	100.0%	0.0%	7.7%	10037
divide-before-multiply	12597	1008	26	11526	37	98.1%	96.5%	99.8%	0.2%	3.5%	10037
tx-origin	12597	18	0	12579	0	100.0%	100.0%	100.0%	0.0%	0.0%	10037
unchecked-lowlevel	12597	220	0	12376	1	99.8%	99.5%	100.0%	0.0%	0.5%	10037
unchecked-send	12597	42	0	12553	2	97.7%	95.5%	100.0%	0.0%	4.5%	10037
uninitialized-local	12597	343	9	12230	15	97.9%	95.8%	99.9%	0.1%	4.2%	10037
unused-return	12597	608	24	11943	22	98.2%	96.5%	99.8%	0.2%	3.5%	10037
incorrect-modifier	12597	88	9	12489	11	94.4%	88.9%	99.9%	0.1%	11.1%	10037
shadowing-builtin	12597	151	5	12436	5	98.4%	96.8%	100.0%	0.0%	3.2%	10037
shadowing-local	12597	2780	149	9607	61	98.2%	97.9%	98.5%	1.5%	2.1%	10037
variable-scope	12597	55	6	12531	5	95.8%	91.7%	100.0%	0.0%	8.3%	10037
void-cst	12597	17	2	12574	4	90.5%	81.0%	100.0%	0.0%	19.0%	10037

Task SD+CC: DLVA as a whole DLVA as a whole combines the SD and CC. If the SD can judge a contract, it does so. If not, DLVA uses the CC to make its best guess. DLVA has average accuracy of 87.7% with an associated FPR of only 12.0% and TPR of 87.3%. Table 4.7 reports per-vulnerability results.

Therefore, incorporating the SD into DLVA improves the statistics across the board: DLVA’s accuracy goes up by 1.7%, its FPR goes down 2.1%, and its TPR goes up by 1.2%.

4.3.3.2 *EthereumSC_{small}* results

We also evaluated DLVA on 21 vulnerabilities with labels in *EthereumSC_{small}*. As shown in Table 4.8, for such contracts DLVA has an average accuracy of 97.6% with a TPR of 95.4% and an associated FPR of only 2.3%.

4.3.3.3 Overall fidelity to Slither

Averaging the separately-evaluated performance on both sizes of contract, DLVA has an overall average accuracy (to Slither) of 92.7%, a TPR of 91.4%, and a FPR

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

Table 4.6: DLVA Trained on *EthereumSC_{large}* (Core Classifier Results)

Vulnerability	Test Size	TP	FP	TN	FN	Accuracy	TPR	TNR	FPR	FNR	AUC
shadowing-state	10037	334	1660	7903	140	76.6%	70.5%	82.6%	17.4%	29.5%	83.0%
suicidal	10037	62	1063	8906	6	90.3%	91.2%	89.3%	10.7%	8.8%	94.6%
uninitialized-state	10037	190	2586	7119	142	65.3%	57.2%	73.4%	26.6%	42.8%	68.4%
arbitrary-send	10037	759	2449	6700	129	79.4%	85.5%	73.2%	26.8%	14.5%	86.5%
controlled-array-length	10037	589	1378	7963	107	84.9%	84.6%	85.2%	14.8%	15.4%	91.8%
controlled-delegatecall	10037	57	2085	7883	12	80.8%	82.6%	79.1%	20.9%	17.4%	88.4%
reentrancy-eth	10037	413	2366	7177	81	79.4%	83.6%	75.2%	24.8%	16.4%	86.1%
reentrancy-no-eth	10037	1430	2071	6179	357	77.5%	80.0%	74.9%	25.1%	20.0%	85.7%
unchecked-transfer	10037	1516	1361	6958	202	85.9%	88.2%	83.6%	16.4%	11.8%	91.8%
erc20-interface	10037	613	1287	8008	129	84.4%	82.6%	86.2%	13.8%	17.4%	92.3%
incorrect-equality	10037	990	2685	6090	272	73.9%	78.4%	69.4%	30.6%	21.6%	81.5%
locked-ether	10037	541	2572	6754	170	74.3%	76.1%	72.4%	27.6%	23.9%	81.4%
mapping-deletion	10037	23	1336	8670	8	80.4%	74.2%	86.6%	13.4%	25.8%	82.8%
shadowing-abstract	10037	224	917	8856	40	87.7%	84.8%	90.6%	9.4%	15.2%	94.1%
tautology	10037	238	2925	6793	81	72.3%	74.6%	69.9%	30.1%	25.4%	80.0%
write-after-write	10037	61	3388	6565	23	69.3%	72.6%	66.0%	34.0%	27.4%	75.8%
constant-function-asm	10037	448	902	8647	40	91.2%	91.8%	90.6%	9.4%	8.2%	97.1%
constant-function-state	10037	27	2711	7295	4	80.0%	87.1%	72.9%	27.1%	12.9%	87.2%
divide-before-multiply	10037	1496	1800	6377	364	79.2%	80.4%	78.0%	22.0%	19.6%	87.0%
tx-origin	10037	37	2606	7387	7	79.0%	84.1%	73.9%	26.1%	15.9%	83.7%
unchecked-lowlevel	10037	65	925	9042	5	91.8%	92.9%	90.7%	9.3%	7.1%	96.3%
unchecked-send	10037	72	2387	7571	7	83.6%	91.1%	76.0%	24.0%	8.9%	89.2%
uninitialized-local	10037	736	2700	6355	246	72.6%	74.9%	70.2%	29.8%	25.1%	79.6%
unused-return	10037	1332	2205	6173	327	77.0%	80.3%	73.7%	26.3%	19.7%	84.4%
incorrect-modifier	10037	125	1912	7974	26	81.7%	82.8%	80.7%	19.3%	17.2%	88.7%
shadowing-builtin	10037	135	1178	8700	24	86.5%	84.9%	88.1%	11.9%	15.1%	93.9%
shadowing-local	10037	1819	2052	5597	569	74.7%	76.2%	73.2%	26.8%	23.8%	81.9%
variable-scope	10037	198	2308	7487	44	79.1%	81.8%	76.4%	23.6%	18.2%	87.5%
void-cst	10037	39	2303	7687	8	80.0%	83.0%	76.9%	23.1%	17.0%	85.4%

of 7.2%.

4.3.4 DLVA vs. state-of-the-art tools

We selected the 11 competitors given in Table 4.9 to benchmark DLVA. We selected competitors based on a number of factors. We selected tools that require Source or those that can handle Bytecode; three competitors can handle both, but prefer source. Most competitors use some form of Static Analysis. We also included SaferSC [128], SMARTEMBED [50, 49, 51], and SoliAudit [89], the only three publicly-available competitor tools using any form of Machine Learning. Two competitor tools use Fuzzing to augment their underlying analysis. DLVA is the first publicly available smart contract vulnerability analyzer using Deep Learning (graph neural nets). On average the competitors recognize 18 vulnerabilities, with significant variance. Table 4.9 also includes the year the version of the tool we used was released and a citation count for the underlying publication as a *very rough*

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

 Table 4.7: DLVA Trained on *EthereumSC_{large}* (Results when SD and CC are working together)

Vulnerability	Test size	TP	FP	TN	FN	Accuracy	TPR	TNR	FPR	FNR
shadowing-state	22634	578	1666	20238	152	85.9%	81.5%	90.3%	9.7%	18.5%
suicidal	22634	76	1063	21487	8	91.9%	89.6%	94.0%	6.0%	10.4%
uninitialized-state	22634	463	2592	19405	174	78.3%	71.5%	85.2%	14.8%	28.5%
arbitrary-send	22634	1203	2461	18827	143	87.8%	90.6%	85.0%	15.0%	9.4%
controlled-array-length	22634	905	1400	20199	130	90.0%	88.4%	91.7%	8.3%	11.6%
controlled-delegatecall	22634	276	2087	20252	19	88.6%	88.9%	88.4%	11.6%	11.1%
reentrancy-eth	22634	703	2377	19456	98	87.3%	88.4%	86.2%	13.8%	11.6%
reentrancy-no-eth	22634	2556	2106	17575	397	86.6%	87.4%	85.9%	14.1%	12.6%
unchecked-transfer	22634	2597	1400	18404	233	91.4%	92.2%	90.7%	9.3%	7.8%
erc20-interface	22634	1613	1296	19592	133	91.2%	90.1%	92.3%	7.7%	9.9%
incorrect-equality	22634	1473	2722	18142	297	84.3%	85.8%	82.8%	17.2%	14.2%
locked-ether	22634	2351	2596	17489	198	85.3%	86.0%	84.6%	15.4%	14.0%
mapping-deletion	22634	33	1336	21257	8	89.1%	85.6%	92.5%	7.5%	14.4%
shadowing-abstract	22634	581	922	21089	42	93.0%	91.3%	94.8%	5.2%	8.7%
tautology	22634	396	2928	19225	85	84.1%	84.8%	83.2%	16.8%	15.2%
write-after-write	22634	72	3391	19146	25	79.5%	77.9%	81.1%	18.9%	22.1%
constant-function-asm	22634	768	908	20910	48	94.6%	94.4%	94.8%	5.2%	5.6%
constant-function-state	22634	39	2711	19879	5	87.2%	89.4%	84.9%	15.1%	10.6%
divide-before-multiply	22634	2504	1826	17903	401	87.6%	87.5%	87.7%	12.3%	12.5%
tx-origin	22634	55	2606	19966	7	88.3%	91.2%	85.5%	14.5%	8.8%
unchecked-lowlevel	22634	285	925	21418	6	95.3%	95.8%	94.8%	5.2%	4.2%
unchecked-send	22634	114	2387	20124	9	89.9%	93.1%	86.6%	13.4%	6.9%
uninitialized-local	22634	1079	2709	18585	261	83.8%	84.2%	83.4%	16.6%	15.8%
unused-return	22634	1940	2229	18116	349	86.4%	87.5%	85.3%	14.7%	12.5%
incorrect-modifier	22634	213	1921	20463	37	87.3%	85.5%	89.2%	10.8%	14.5%
shadowing-builtin	22634	286	1183	21136	29	91.8%	90.2%	93.4%	6.6%	9.8%
shadowing-local	22634	4599	2201	15204	630	85.1%	85.8%	84.4%	15.6%	14.2%
variable-scope	22634	253	2314	20018	49	86.5%	86.2%	86.9%	13.1%	13.8%
void-cst	22634	56	2305	20261	12	84.7%	82.1%	87.1%	12.9%	17.9%

measure of significance.

Benchmarking against multiple tools is inherently challenging. Many tools do not recognize the same vulnerabilities. *More seriously, even for the vulnerabilities that are recognized in common, the tools can define them differently.* Consider reentrancy, perhaps the most-studied vulnerability, and one recognized by most of the competitors. Recall from Table 4.1 that reentrancy actually comes in two flavours (reentrancy-eth and reentrancy-no-eth); this supported by the associated Solidity documentation [124]. However, only Slither (and, thus, DLVA) recognizes the -no-eth variety. If we include -no-eth examples in our benchmarks, other tools suffer many false negatives. Accordingly, -no-eth examples are not in our test sets.

To give another example, eThor [116] provides a formal definition for their notion of reentrancy (“single-entrancy”), and is the only competitor focused on soundness

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

 Table 4.8: DLVA Trained on *EthereumSC_{small}*(use DLVA’s core classifier for the entire test set)

Vulnerability	Test Size	TP	FP	TN	FN	Accuracy	TPR	TNR	FPR	FNR	AUC
shadowing-state	1381	8	18	1355	0	98.7%	100.0%	98.7%	1.3%	0.0%	94.7%
suicidal	1381	9	143	1229	0	89.6%	100.0%	89.6%	10.4%	0.0%	89.2%
uninitialized-state	1381	10	32	1337	2	97.5%	83.3%	97.7%	2.3%	16.7%	94.0%
arbitrary-send	1381	54	16	1306	5	98.5%	91.5%	98.8%	1.2%	8.5%	94.9%
controlled-array-length	1381	8	20	1352	1	98.5%	88.9%	98.5%	1.5%	11.1%	91.4%
controlled-delegatecall	1381	5	25	1351	0	98.2%	100.0%	98.2%	1.8%	0.0%	99.6%
reentrancy-eth	1381	6	56	1319	0	95.9%	100.0%	95.9%	4.1%	0.0%	99.6%
reentrancy-no-eth	1381	18	2	1359	2	99.7%	90.0%	99.9%	0.1%	10.0%	97.6%
unchecked-transfer	1381	42	17	1318	4	98.5%	91.3%	98.7%	1.3%	8.7%	96.4%
erc20-interface	1381	22	64	1294	1	95.3%	95.7%	95.3%	4.7%	4.3%	96.9%
incorrect-equality	1381	19	26	1336	0	98.1%	100.0%	98.1%	1.9%	0.0%	92.3%
locked-ether	1381	71	37	1269	4	97.0%	94.7%	97.2%	2.8%	5.3%	95.3%
constant-function-asm	1381	5	11	1365	0	99.2%	100.0%	99.2%	0.8%	0.0%	99.7%
divide-before-multiply	1381	32	10	1336	3	99.1%	91.4%	99.3%	0.7%	8.6%	95.5%
unchecked-lowlevel	1381	11	4	1366	0	99.7%	100.0%	99.7%	0.3%	0.0%	98.5%
unchecked-send	1381	10	0	1371	0	100.0%	100.0%	100.0%	0.0%	0.0%	100.0%
uninitialized-local	1381	17	72	1290	2	94.6%	89.5%	94.7%	5.3%	10.5%	95.6%
unused-return	1381	473	3	894	11	99.0%	97.7%	99.7%	0.3%	2.3%	99.2%
incorrect-modifier	1381	36	6	1339	0	99.6%	100.0%	99.6%	0.4%	0.0%	99.9%
shadowing-builtin	1381	7	11	1363	0	99.2%	100.0%	99.2%	0.8%	0.0%	99.7%
shadowing-local	1381	27	90	1261	3	93.3%	90.0%	93.3%	6.7%	10.0%	93.7%

(*i.e.*, a 100% detection rate) above all else. However, single-entrancy considers unsafe some “litmus test” contracts that the SWC-107 description [101] labels safe³. Accordingly, eThor produces a lot of false positives when compared against a ground truth based on SWC-107. Moreover, eThor considers any contract containing a `DELEGATECALL` or `CALLCODE` opcode to be out of scope; in practice, this eliminates a *many* important examples.

Summary of results The high-level results of our competitor benchmarking was already given in Figure 4.1. Along the bottom we put the competitors, and in parenthesis the number of tests we include in the benchmark for that competitor (not every tool can handle every vulnerability).

We present five measures of performance. Overall, DLVA performed extremely well. The Completion Rate measures what percentage of contracts in our benchmarks terminate with a yes-or-no answer (rather than, *e.g.*, raising an exception, timing out, running out of memory). Most suffered from the occasional timeout or etc. Many of the source code analyzers were not able to analyze some contracts since

³For example, single-entrancy considers *both* the `simple_dao.sol` and `simple_dao_fixed.sol` litmus tests to be unsafe [114], whereas the SWC-107 description considers the first to be unsafe and the second to be safe [101].

CHAPTER 4. SUPERVISED DEEP LEARNING: DLVA

Table 4.9: Comparison of DLVA vs. state-of-the-art tools; Input: (S: Source code, B:Bytecode, S/B⁻: Source preferred, bytecode possible); Method: (SA/DA:Static/Dynamic Analysis, ML/DL:Machine/Deep Learning); Vul: # of vulnerability detectors; Year: year of release of the used version; Cits: number of citations from Google Scholar on 01/12/2023.

Analyzer	Input	Method	Vul	Year	Cits
Oyente 0.2.7 [93]	S/B ⁻	SA/DA	4	2017	2,219
Osiris [134]	S/B ⁻	SA/DA	5	2018	287
SaferSC [128]	S	ML/DL	1*	2018	97
Mythril 0.21.20 [100]	S/B ⁻	SA/DA	13	2019	153
SmartCheck 2.0 [132]	S	SA/DA	43	2019	636
SMARTEMBED [50, 49, 51]	S	ML/DL	10	2019	97
SoliAudit [89]	S	ML/DL	13	2019	98
eThor [116]	B	SA/DA	1	2020	107
Slither 0.8.0 [45]	S	SA/DA	74	2021	411
ConFuzzius [133]	S	SA/DA	10	2022	46
SAILFISH [22]	S	SA/DA	2	2022	46
DLVA [3, 4, 5]	B	ML/DL	29	2023	NA

the Solidity version was too old or new⁴. eThor refused to analyze many contracts due to `DELEGATECALL` or `CALLCODE` opcodes. Only DLVA, SaferSC, SMARTEMBED and SmartCheck answered every query.

Arguably the most important metrics are Accuracy, the True Positive Rate (TPR), and the False Positive Rate (FPR) (see §2.6 for definitions). We exclude any contract that failed to complete from these metrics (*i.e.*, we do not double count failures). Figure 4.1 shows the True Positive Rate (*i.e.*, detection rate; the higher the better), eThor had a 100.0% TPR; SaferSC followed with 99.8% TPR, Slither with 99.4% TPR, and DLVA came in fourth with 98.7% TPR. Figure 4.1 also shows the False Positive Rate (*i.e.*, false alarm rate; the lower the better), SAILFISH boasts an impressive 0.1% FPR, followed by SMARTEMBED at 0.4% FPR, DLVA at 0.6%, and SmartCheck at 2.4% FPR.

DLVA led the pack in accuracy at 99.7%, Slither came in second at 97.2%, SmartCheck came in third at 93.2%, and SaferSC came in fourth at 91.9%. (Moreover, recall that DLVA judges bytecode whereas Slither and Smartcheck require source

⁴As mentioned in §4.3.1, we made a good-faith effort to lightly clean source code to help them, but in many cases it was not enough. We did exclude any contract for which source code was unavailable; Completion Rates would have been far worse for source-only competitors otherwise.

code!)

DLVA led the pack in average analysis time per contract (the graph is in log scale, lower better) at only 0.2 seconds, SaferSC came in second at 1.0 seconds, Slither came in third at 1.3 seconds, and SMARTEMBED came in fourth at 2.6 seconds.

Tables 4.10, 4.11, and 4.12 contain the data behind Figure 4.1. We benchmark with *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11], and *SolidiFI*_{benchmark} [13] since we have high confidence in their labelling of ground truth (§4.3.1).

Table 4.10 presents the results of the five bytecode analyzers on *Elysium*_{benchmark} [7], *Reentrancy*_{benchmark} [11]. *Elysium*_{benchmark} [7] contains contracts with two vulnerabilities: REentrancy (with 75 Genuine/ground Positives and 825 Genuine/ground Negatives) and Parity Bug (with 823 GP and 77 GN). *Reentrancy*_{benchmark} only contains reentrancies (with 53 GP and 420 GN). For each benchmark, we document five statistics for each tool. ‘Exp’ gives the number of contracts for which analysis failed to complete (*e.g.*, timeouts). False Negatives gives the number of ground positives that were incorrectly labeled as negative; conversely, False Positives gives the number of ground negatives that were incorrectly labeled as positive. The Σum of Failures is just Exp + FN + FP, and the Average Failure is the number of failures as compared to the size of the test set (averaging the failure rate for RE and PB for the two tools that can handle both). Table 4.11 gives the same data for source code analyzers. Only 6.6% of contracts in *Elysium*_{benchmark} have available source (from 900 contracts to 59) since suicided PB contracts no longer have source code on Etherscan. We considered marking the 841 missing contracts as ‘Exp’ to emphasize the importance analyzing bytecode, but ultimately decided to simply exclude them. Table 4.12 gives the associated data for *SolidiFI*_{benchmark}.

Slither does not recognize Overflow/Underflow and *EthereumSC*_{large} had too few occurrences of Timestamp-Dependency to be included in the 29 vulnerabilities in Table 4.1. We retrained DLVA to handle these vulnerabilities with the training and validation portions of the SolidiFI dataset.

In our benchmarking, we found that DLVA, Slither, and SmartCheck had the best overall performance: their high accuracy (99.7%, 97.2%, and 93.2%, respectively) reflects a good balance between a high TPR (98.7%, 99.4%, and 78.1%) and a low FPR (0.6%, 15.3%, and 2.4%). Generally speaking, other tools with good TPR

suffered with poor FPR, and vice versa.

A few of these competitors deserve special attention to contextualize their results. eThor’s focus is entirely on soundness, and indeed we were never able to produce a false negative with it; the authors are to be commended. However, the cost to the other metrics is severe: their 34.5% completion rate and 79.8% false positive rate are pitiful; moreover, their analysis time is three orders of magnitude slower than DLVA.

SAILFISH leads the pack with 0.1% FPR, but their TPR is a mediocre 72.7%, explaining their unexceptional 87.5% overall accuracy.

SaferSC’s results seem to have benefited greatly from the nature of our test suite. Of all the vulnerabilities we test, it is only sensitive to “suicidal,” which is why we have only 1 relevant benchmark (Elysium_{benchmark} [7]); all other tools have at least 3 relevant benchmarks. We suspect that SaferSC is strongly biased to report a vulnerability. This naturally leads to a fantastic true positive rate (99.8% tested), but also results in a very poor false positive rate as well (92.2%). However, since the number of suicidal contracts in the Elysium test set is much higher than the number of non-suicidal contracts, SaferSC’s benchmarked accuracy of 91.9% looks better than we think it would, if benchmarked against a test suite that had a more realistic balance of suicidal and non-suicidal contracts. Nevertheless, since SaferSC was one of the few available ML/DL tools, we included it in our benchmark.

SMARTEMBED suffers from the opposite problem: due to a small predefined bug dataset, it is strongly biased to consider contracts as safe. This naturally leads to a fantastic false positive rate (0.4%), at the cost of a very poor true positive rate (0.2%). Accuracy (62.5%) is not as bad as might be anticipated since most contracts in the 4 test datasets are in fact safe for the “reentrancy” and “over/underflow” vulnerabilities. Like SaferSC, we included SMARTEMBED primarily since it was one of the few available ML/DL tools.

SoliAudit, the third easily available ML/DL tool, performed better than SaferSC or SMARTEMBED on balance, despite its unexceptional overall performance. Its middling TPR of 63.8% was in line with a number of other tools; its FPR of 28.1% was markedly higher than almost every other tool. Still, unlike SaferSC or SMARTEMBED, these results indicate that SoliAudit is not very strongly biased positively or negatively. Accordingly, we think that SoliAudit’s benchmarked

accuracy of 81.9% is a fair measure of its performance.

The tools that seem to be most widely used in the community at the moment are Mythril (including its commercial version MythX) and Slither. ConFuzzius is seeing increasing use in the fuzzing community.

4.3.5 Discussion

Overall we are pleased with DLVA’s performance as presented in this section: the machine learning models in DLVA are not trivial to best; DLVA is accurately predicting Slither’s labels; and DLVA performs well compared to competitors. What remains is to highlight a few points and observations.

Detecting vulnerabilities that caused heavy losses Two smart contract losses loom large in the popular understanding: the DAO hack and the Parity bug. DLVA’s performance on *Elysium_{benchmark}* [7] showed that for real-world contracts with these vulnerabilities, DLVA’s accuracy was 99.4%.

70% of contracts-with-money are bytecode-only Many existing tools—including Slither—require source code to analyze. In contrast, DLVA judges bytecode: essentially, extending Slither’s “analysis style” from source- to bytecode.

We have identified approximately 12,000 contracts that hold at least 1 ETH each; the combined value is approximately 25,700,000 ETH (1 ETH is about 1,750 USD on June 12, 2023). Only 30% of these 12k contracts have source code available and are thus analyzable by Slither. In contrast, DLVA can judge all of them. We suggest that a user of a DLVA-flagged contract that lacks source code proceed with caution.

In our data set, we used DLVA to detect vulnerabilities in 248,073 contracts that were not labelled by Slither in §4.2.1 due to unavailability of source code on Etherscan, with total balance 540,928 ETH. DLVA flags about 6% of contracts for at least one high severity vulnerability.

Speed matters for surveys and monitoring DLVA is much faster than other tools. Running in “batch mode”—where all of the models are loaded into memory and then large numbers of contracts are analyzed—DLVA judges the average contract in 0.2 seconds. Slither takes 1.3 seconds per contract (6.5x slower), for the 32.6% of

Table 4.10: Small contracts, bytecode analyzers; Exp: Exceptions; Vulnerability: {RE:Reentrancy, PB:Parity Bug}; GP: Ground Positives; GN: Ground Negatives; FN: False Negatives; FP: False Positives; ΣF : Sum of Failures

Analyzer	Benchmark Data Sets (entire benchmark)												
	Elysium [7]						Reentrancy [11]						
	RE		PB		Average Failure		RE		GP + 420 GN		Average Failure		
Exp	75 GP + 825 GN	FN	FP	ΣF	823 GP + 77 GN	FN	FP	ΣF	Exp	53 GP + 420 GN	FN	FP	ΣF
Oyente [93]	1	28	0	29	-	-	-	-	12	27	0	39	8.2%
Osiris [134]	0	6	0	6	-	-	-	-	12	2	3	17	3.6%
SaferSC [128]	0	-	-	-	2	71	73	-	-	-	-	-	-
Mythril [100]	37	3	0	40	819	0	856	49.8%	39	0	3	42	8.9%
SMARTEMBED [50]	0	52	0	52	-	-	-	5.7%	0	52	0	52	11.0%
eThor [116]	830	0	1	831	-	-	-	92.3%	287	0	130	417	88.1%
DLVA	0	1	4	5	1	3	4	0.5%	0	3	0	3	0.6%

Table 4.11: Small contracts, source code analyzers; Exp: Exceptions; Vulnerability: {RE:Reentrancy, PB:Parity Bug}; GP: Ground Positives; GN: Ground Negatives; FN: False Negatives; FP: False Positives; ΣF : Sum of Failures

Analyzer	Benchmark Data Sets (subset of benchmark with available source code)												
	Elysium [7]						Reentrancy [11]						
	RE			PB			RE			RE			
	Exp	52 GP	7 GN	ΣF	7 GP	52 GN	52 GP	420 GN	Exp	52 GP	420 GN	Average Failure	
SmartCheck [132]	0	5	1	6	-	-	-	0	0	1	1	0.2%	
SoliAudit [89]	7	11	1	18	0	1	8	88	10	5	103	21.8%	
Slither [45]	1	1	6	8	0	0	1	21	1	0	22	4.7%	
ConFuzzius [133]	7	7	1	15	0	0	7	111	2	0	113	23.9%	
SAILFISH [22]	6	16	0	22	-	-	-	125	24	1	150	31.8%	

Table 4.12: Large contracts; Exp: Exceptions; Vulnerability: (RE:Reentrancy, TS:Timestamp-Dependency, OU:Over/Underflow, TX:tx.origin); GP: Ground Positives; GN: Ground Negatives; FN: False Negatives; FP: False Positives; ΣF : Sum of Failures

Analyzer	SolidiFI [13] (entire benchmark)														
	Exp	RE			TS			OU			TX			Average Failure	
		111 GP + 333 GN	FN	FP	ΣF	111 GP + 333 GN	FN	FP	ΣF	111 GP + 333 GN	FN	FP	ΣF		
Oyente [93]	0	0	0	0	110	19	129	279	12	267	279	-	-	-	30.6%
Osiris [134]	0	10	0	10	111	17	128	260	20	240	260	-	-	-	29.9%
Mythril [100]	0	68	23	91	43	15	58	166	81	85	166	23	7	30	19.4%
SmartCheck [132]	0	0	0	0	52	0	52	83	83	0	83	0	0	0	7.6%
SMARTEMBED [50]	0	111	0	111	-	-	-	115	110	5	115	-	-	-	25.5%
SoliAudit [89]	0	111	0	111	1	21	22	295	13	282	295	0	7	7	24.5%
eThor [116]	194	0	135	329	-	-	-	-	-	-	-	-	-	-	74.1%
Slither [45]	0	0	0	0	0	20	20	-	-	-	-	0	0	0	1.5%
ConFuzzius [133]	7	54	2	63	-	-	-	129	64	58	129	-	-	-	21.6%
SAILFISH [22]	0	0	0	0	-	-	-	-	-	-	-	-	-	-	0.0%
DLVA	0	0	0	0	0	0	0	2	2	0	2	0	0	0	0.1%

the contracts it can judge.

Some tools such as Mythril or eThor can analyze bytecode like DLVA. However, completion rates are much lower and the time required for analysis is 2-3 orders of magnitude greater. It is not practicable to scan large numbers of contracts for vulnerabilities with these tools, whether to survey the current state of the chain or to monitor new contracts in real time.

Ongoing surveillance Since DLVA is fast, accurate, and handles bytecode, we periodically run it over new contracts. When DLVA flags we “get a second opinion” from Slither/Oyente/Mythril, depending on whether source is available.

For example, DLVA recently flagged 95 contracts for the reentrancy vulnerability. These contracts jointly hold ≈ 85 ETH / 148k USD. Oyente and Slither confirmed the vulnerability for the 17 contracts that had source available, and Oyente and Mythril for the 78 that had only bytecode.

Stability of vulnerability detection tools A vulnerability classifier X should give stable results: each time X runs over a contract c it should give the same answer. DLVA has this desirable property. We relabeled *EthereumSC_{large}* with Slither and discovered that 1,328 labels changed from “vulnerable” to “non-vulnerable,” and a further 172 labels changed from “non-vulnerable” to “vulnerable.” Clearly Slither is not deterministic, perhaps due to timeouts or randomised algorithms. We estimate that Slither is mislabeling approximately 1.25% of contracts due to these issues. Clearly, DLVA’s training algorithm is robust enough to cope with some mislabeling.

Discovering label contradictions We used the Sibling Detector to discover pairs of very similar contracts that Slither nonetheless labels differently. SD flagged them as potential label contradictions, reasoning that very similar contracts should have the same classification label. For example, for the “divide-before multiply” vulnerability, Slither labels the contract at address 0xaa3a2ae9 [35] as non-vulnerable and the contract at address 0xa8d8feeb [36] as vulnerable.

To resolve this apparent contradiction, we first asked DLVA’s CC for its opinion (both considered non-vulnerable), and then manually examined the Solidity source code. Happily, DLVA’s CC is right, whereas Slither’s analysis of the contract

at address 0xa8d8feeb is wrong. Further experiments with SD found 596 more “contradiction pairs.” We manually reviewed 70 further pairs. For each pair reviewed, we found that both contracts had nearly identical solidity source code, differing only in initial values or whitespace/comments. We were pleased when the CC always assigned the same label to both contracts in a pair. After further manual inspection, we discovered that the CC was right 39 out of 71 times (55.0%).

This experiment indicates that machine learning techniques can help debug and improve static analyzers.

4.4 Key Comparative Studies

The security analysis of Ethereum smart contracts is of utmost importance, and a range of analysis tools have been developed to ensure the creation of safe and secure smart contracts. Smart contracts pose a significant challenge as they cannot be modified or patched once they have been published to the blockchain network. Unlike traditional software applications, if there are changes in users’ requirements or bugs are discovered after deployment, smart contracts cannot be updated. As smart contracts contain significant value in crypto assets, attackers are highly motivated to exploit vulnerabilities in them to steal funds from the contracts. However, the community has extensively developed automated tools and methods to detect such vulnerabilities. The security analysis methods for smart contracts can be categorized into three types: static analyzers, fuzzing, and machine learning methods.

The community has developed a variety of static analysis and dynamic analysis techniques to identify vulnerabilities in smart contracts. Static analyzers such as Oyente [93], Mythril [100], Osiris [134], SmartCheck [132], Slither [45], Maian [105], Securify [136], and Manticore [99] rely on hand-crafted expert rules and manually engineered features.

- 1) The static analysis tool, OYENTE, as described in [93], is utilized for detecting vulnerabilities in smart contracts. It utilizes symbolic execution to identify vulnerabilities in smart contract functions based on simple patterns. The vulnerabilities are classified into several groups, including transaction-ordering dependent, timestamp dependence, re-entrance handling, and mishandled

exceptions. However, it is known to produce a high false-positive rate.

Symbolic execution is the foundation of OYENTE analysis tool and is defined in [79]. The symbolic execution technique represents smart contract variables as expressions of symbolic values, and each symbolic path has a path condition that is an expression over the symbolic inputs. Path conditions are built by accumulating constraints that must be satisfied in order to follow the path. A path is deemed infeasible if its path condition is unsatisfiable, and feasible if it is. This approach allows the tool to statically reason about a program path-by-path, by reasoning about one path at a time.

- 2) The Securify analysis tool, as discussed in [136], takes EVM bytecode and the security properties of a smart contract as inputs. Security properties include compliance and violation patterns. The tool utilizes the decompilation analysis method and represents the code as Data Log facts. If a pattern is detected, the tool infers that the code possesses the corresponding security vulnerability. Securify’s analysis consists of two steps. First, it symbolically analyzes the smart contract’s dependency graph to extract semantic information from the contract code. Second, it checks compliance and violation patterns that capture sufficient conditions to prove if a property is satisfied or not satisfied.
- 3) Mythril, as discussed in [100], is a static analysis tool used to detect security vulnerabilities in smart contracts. It works by analyzing the bytecode of the smart contract and using symbolic execution techniques to explore all possible execution paths. Mythril performs several analyses on the contract, including checking for vulnerabilities such as reentrancy attacks, transaction order dependence, and integer overflows/underflows. Additionally, it can detect issues related to uninitialized variables and access control. Mythril is an open-source tool and is actively maintained by a team of developers. It has been used extensively by the smart contract development community to ensure the security of their contracts.
- 4) SmartCheck, as discussed in [132], is designed to identify possible vulnerabilities in Solidity contracts by examining specific syntactic patterns in the source code. The tool achieves this goal by converting the code into an XML syntax

tree, which provides a structured representation of the code. SmartCheck then employs XQuery path expressions to specify the vulnerabilities and search for the relevant patterns within the XML tree. By using this approach, SmartCheck is able to effectively identify potential security issues in Solidity contracts and provide developers with the information they need to address these risks.

- 5) Manticore [99] is a dynamic binary analysis tool developed by TrailOfBits that has been adapted for use in analyzing smart contracts. It is designed to analyze the bytecode of Ethereum smart contracts and identify vulnerabilities that may not be apparent from simply examining the code.

Manticore works by executing the bytecode in a simulated Ethereum environment and exploring all possible execution paths. This allows it to identify potential vulnerabilities such as reentrancy attacks, arithmetic errors, and other types of bugs that may not be easily identified through static analysis alone. One of the key features of Manticore is its support for symbolic execution, which allows it to analyze code paths that may be difficult or impossible to reach through normal execution. This allows it to identify potential vulnerabilities that may be missed by other analysis tools.

- 6) Slither [45] is a static analysis tool for smart contracts that was developed by TrailOfBits. It is designed to detect security vulnerabilities in Solidity contracts, which is the most popular language used for writing smart contracts on the Ethereum platform. Slither analyzes the contract code by using a variety of techniques such as taint analysis, control flow analysis, and data flow analysis. These techniques are used to identify potential vulnerabilities such as reentrancy attacks, uninitialized storage pointers, and gas limit vulnerabilities. Slither also provides a range of output formats and integrations with popular development tools. This makes it easy for developers to integrate the tool into their development workflow and quickly identify and fix any security issues in their code. Overall, Slither is a powerful and flexible tool for analyzing smart contracts and is widely used by the Ethereum development community to ensure the security of their contracts.

Although such tools are very impressive, and indeed we ourselves use Slither, this reliance on expert rules can make these tools difficult to maintain and update. We are unaware of any detection tool that detects all known vulnerabilities; or that is easily extendable for future bugs without human developers carefully crafting subtle expert rules and/or hardcoding additional features. Most smart contract vulnerability analyzers use symbolic execution to reason about all execution paths of a program. However, symbolic execution can suffer from “path explosion” when the size and complexity of the code increases, leading to significant time and space requirements. Practical limits on time and space can lead to difficulties analyzing smart contracts at scale. Moreover, empirical evaluation of 9 static analysis tools [41] classified 93% of contracts as vulnerable, thus indicating a considerable number of false positives. In addition, only a few vulnerabilities were detected simultaneously that got consensus from four or more tools.

Fuzzing is a dynamic analysis technique, that has the advantage of scaling well to larger programs. Contractfuzzer [76], and Echidna [59] are two notable examples applied to smart contracts. Rather than relying on a fixed set of pre-defined bug oracles to detect vulnerabilities, fuzzing technique uses sophisticated grammar-based fuzzing campaigns based on a contract ABI to falsify user-defined predicates or Solidity assertions. However, generating meaningful inputs for fuzzing typically requires annotating the source code of a contract.

Machine learning has developed models to handle complex program data, and the training process has a high degree of automation. These advantages can help overcome one of the primary limitations of tools based on formal techniques: training a new model may not rely on the development of subtle expert rules. Of course, ML models bring their own challenges, such as major computational requirements for training, and the necessity of an “experimental science” approach to find a “good” model (*e.g.* by tuning hyperparameters).

There is an ongoing trend of using machine learning (ML) for source and binary analysis of security concerns [94]. Recent surveys of machine learning techniques for source code analysis [120], malware analysis [137], and vulnerability detection [65] explored multiple categories for utilizing machine learning in code analysis. These categories include code representation [40], program synthesis [24], program repair [39], code clone detection [144], code completion [33], code summarization [141],

84], code review [83], code search [60, 26], and vulnerability analysis [53]. The breadth of this work shows that machine learning highlights the potential of these techniques for software development and security.

One of the closest pieces of related work is Momeni et al. [98], which proposed a machine learning model to detect security vulnerabilities in smart contracts, achieving a lower miss rate and faster processing time than the Mythril and Slither static analyzers. Momeni et al.’s model is more handcrafted than DLVA, *e.g.* extracting 17 human-defined features from ASTs to measure the complexity of a small data set of source code. DLVA uses the Universal Sentence Encoder to extract features without human-provided hints.

Wesley et al. [128] adapted a long short-term memory (LSTM) neural network to analyze smart contract opcodes to learn vulnerabilities sequentially, which considerably improved accuracy and outperformed the symbolic analysis tool Maian. Compared with DLVA, Wesley et al.’s model learned from opcode sequences without considering the control flow of the smart contract, so it could not handle control-flow vulnerabilities. DLVA’s choice to represent contracts as CFGs lets it understand more subtle vulnerabilities.

Sun et al. [126] added an attention mechanism to (non-graph) convolutional neural networks to analyze smart contract opcodes. They achieved a lower miss rate and faster processing time as compared to the Oyente and Mythril static analyzers. Liao et al. [89] developed SoliAudit, which combined machine learning and a dynamic fuzzer to strengthen the vulnerability detection capabilities. Liao et al. used word2vec to obtain a vector representation for each opcode and concatenated these vectors row-by-row to form the feature matrix. They did not consider the control-flow of the smart contract. In contrast, DLVA uses graph convolutional neural networks to extract contract embeddings, resulting in a more sophisticated understanding of program structure. Rather than combining with a fuzzer, we added our sibling detector SD.

SMARTEMBED [50] introduced the idea of clone detection for bug detection in Solidity code. SMARTEMBED used AST syntactical tokens to encode bug patterns into numerical vectors via techniques from word embeddings. Contracts are judged clones if they are Euclidian-close. The authors manually validated some reported bugs and showed that SMARTEMBED significantly improved accuracy as compared

to SmartCheck. Our Sibling Detector was inspired by SMARTEMBED, although we work in control flow graphs rather than syntactic tokens. Moreover, SMARTEMBED was given predefined vulnerability matrices, rather than learning from labeled data as DLVA does.

Luca Massarelli et al. [94] investigated graph embedding networks to learn binary functions. Massarelli et al. proposed a deep neural network called structure2vec for graph embedding to measure the binary similarity of assembly code. In contrast, our SC2V engine leverages unsupervised feature extraction from a CFG, which has a much higher level of abstraction than syntactical tokens.

4.5 Summary

We have designed, trained, and benchmarked our novel Deep Learning Vulnerability Analyzer (DLVA) which is an efficient, easy-to-use, and fast tool for detecting vulnerabilities in Ethereum smart contracts. DLVA analyzes smart contract bytecode, meaning that almost all smart contracts can be targeted. DLVA transforms contract bytecode to an N-dimensional floating-point vector as a contract summary using DLVA’s SC2V, and then this vector is given to DLVA’s Sibling Detector to check whether it is close to the contracts seen before. If not, the vector is passed to DLVA’s Core Classifier to predict the 29 vulnerabilities learned during training.

DLVA has a generic design, rather than one customized for each vulnerability. Accordingly, given bytecodes and suitable labeling oracles, training DLVA to recognize future smart contract vulnerabilities should be straightforward without the need for expert rules and/or hardcoding additional features.

Overall, DLVA performed extremely well, and outperformed state-of-the-art alternatives. DLVA produced the results using about 0.2 seconds per contract, i.e. 5-1,000x faster than competitors. DLVA predicts Slither’s labels with an overall accuracy of 92.7% and associated false positive rate of 7.2%. We benchmark DLVA against eleven well-known smart contract analysis tools. Despite using much less analysis time, DLVA completed every query, leading the pack with an average accuracy of 99.7%, pleasingly balancing high true positive rates with low false positive rates.

We wish to explore several directions in the future. We hope to improve the

N2V module using the transformer architecture. We plan to experiment with using multiple oracles to label the same data set to increase the accuracy of our models. We plan to experiment with our sibling detector to understand why some vulnerabilities are hard to catch this way.

4.6 Availability

DLVA is available for download from <https://secartifacts.github.io/use-nixsec2023/appendix-files/sec23winterae-final67.pdf> [5]. The data sets we use in this research are available as well [8, 9, 13, 7, 11].

Chapter 5

Semi-Supervised Learning: SCooLS

In this chapter, we will discuss a critical challenge in detecting vulnerabilities using deep learning models, namely the lack of large, labeled datasets suitable for model training. We will demonstrate how this limitation poses a significant obstacle to the effective identification of vulnerabilities in smart contracts. To address this challenge, we propose leveraging the principles of semi-supervised learning, which can generate more accurate models than unsupervised learning, while not requiring the extensive, oracle-labeled training sets required by supervised learning. We will discuss the advantages of using semi-supervised learning in the context of vulnerability detection and provide insights into how this approach can be optimized for various software systems.

We introduce SCooLS, our Smart Contract Learning (Semi-supervised) engine. SCooLS uses neural networks to analyze Ethereum contract bytecode and identifies specific vulnerable functions. The recent use of deep learning techniques for vulnerability detection in smart contracts has demonstrated the ability to achieve comparable results to static analysis techniques while offering faster and more efficient analysis at scale. However, the lack of large, labeled data sets for training deep learning models presents a significant challenge for effective vulnerability detection.

SCooLS incorporates two key elements: semi-supervised learning and graph neural networks (GNNs). Semi-supervised learning produces more accurate models than unsupervised learning, while not requiring the large oracle-labeled training set that supervised learning requires. GNNs enable direct analysis of smart contract bytecode without any manual feature engineering, predefined patterns, or expert rules.

SCooLS represents a pioneering application of semi-supervised learning techniques in the realm of smart contracts vulnerability analysis. It uniquely enables the precise detection and exploitation of specific vulnerable functions. Significantly, it’s the first tool to not only identify these vulnerable functions but also to generate authentic attack demonstrations for end-users and developers. This approach diverges from the traditional method of simply labeling the entire contract as vulnerable, providing developers with a tangible method to test the exploitability of their contracts. SCooLS’s performance is better than existing tools, with an accuracy level of 98.4%, an F1 score of 90.5%, and an exceptionally low false positive rate of only 0.8%. Furthermore, SCooLS is fast, analyzing a typical function in 0.05 seconds.

We leverage SCooLS’s ability to identify specific vulnerable functions to build an auto-exploit generator, which was successful in stealing Ether from 76.9% of the true positives.

5.1 Introduction

SCooLS borrows two key ideas from DLVA: deep learning (neural nets) and, more specifically, the use of graph neural networks (GNNs). However, SCooLS differs in several critical ways: SCooLS uses *semi-supervised learning* rather than *supervised learning*, and targets *functions* rather than *contracts*. Our framework design and use of committees during training are also innovations. Accordingly, our resulting models are novel. Moreover, our focus on functions allowed the development of an *auto-exploit generator*. SCooLS is the first tool to utilize these elements in a smart contract vulnerability analyzer.

SCooLS focuses on a single well-known smart contract vulnerability, “reentrancy-eth” (specifically, SWC-107 according to the Smart contract Weakness Classification system [111]). We focus on reentrancy because it is a well-studied and serious vulnerability [95, 121]. This means that detecting reentrancy has value in practice; moreover, its popularity allows for a good comparison with previous work. Although most famous for the DAO attack in 2016, reentrancy continues to be a persistent issue in Ethereum smart contracts. Notable recent examples include the Uniswap/Lendf.Me hacks in April 2020, the SURGEBNB hack in August 2021, the CREAM FINANCE hack in August 2021, the Siren protocol hack in September

2021, and the Omni attack in July 2022 [28]. The occurrence of attacks can be attributed to the intricate nature of the semantics of the Ethereum Virtual Machine (EVM) bytecode. Developers may not possess a complete understanding of the fact that sending ether to a contract can trigger a function call, or that a non-recursive function can be re-entered before termination. In a reentrancy attack, a malicious actor takes advantage of these two subtleties to repeatedly call a vulnerable withdrawal-type function. If the contract has been naïvely coded, this can result in the same withdrawal being executed multiple times, despite the programmer’s informal intention to authorize only one withdrawal. It took two weeks to find and hand-verify our core *ReentrancyBook* data set (described in §5.3). Extending SCooLS to other vulnerabilities would require similar incremental effort.

§5.2 We illustrate the differences between deep learning styles.

§5.3 We assemble and hand-verify the *ReentrancyBook* data set, containing 22 vulnerable and 480 non-vulnerable functions. We also collect and preprocess our large unlabeled *BigBook* data set.

§5.4 We explain the design of our smart contract vulnerability analyzer SCooLS. SCooLS represents a pioneering application of semi-supervised learning techniques in the realm of smart contracts vulnerability analysis. It uniquely enables the precise detection and exploitation of specific vulnerable functions. Significantly, it’s the first tool to not only identify these vulnerable functions but also to generate authentic attack demonstrations for end-users and developers. This approach diverges from the traditional method of simply labeling the entire contract as vulnerable, providing developers with a tangible method to test the exploitability of their contracts. We use semi-supervised learning to train SCooLS. Semi-supervised learning helps address the scarcity of high-confidence labeled code data in practical smart contract vulnerability classification tasks. In total we train 120 distinct models derived from applying a variety of hyperparameters to five state-of-the-art graph neural networks, using a voting system to smooth out the variance in individual models during training.

§5.5 We implement an auto-exploit generator to prove that the detected vulnerabilities can be exploited by attackers to steal contract funds.

§5.6 We measure the performance of SCooLS and compare it with three state-of-the-art tools. SCooLS dominates the competition, obtaining a higher accuracy level of 98.4%, a higher F1 score of 90.5%, and the lowest false positive rate of just 0.8%. Moreover, the analysis is fast, requiring only 0.05 seconds per function. We also showed that the auto-exploit generator was able to attack 76.9% of the true positive instances for which an ABI was available.

§5.7, §5.8 We discuss related work and conclude.

SCooLS availability and ethical considerations. Any vulnerability analyzer can be used with ill intent. Blockchains are tricky for responsible disclosure [20]. Attackers are incentivized to find and attack weak contracts, and due to the pseudonymous nature of the blockchain, it is hard to quietly inform participants of vulnerabilities. Concerningly, SCooLS allows attackers to target weak contracts relatively precisely, and our Exploit Generator enables nearly automatic theft.

On the other hand, reentrancy has long been studied and many available tools already flag it (*e.g.*, [45, 100, 133, 3]). Other exploit generators for smart contracts have also been published and made publicly available (*e.g.*, [82, 77]). Moreover, honest actors benefit from SCooLS too: everyone wants to know if the contracts they use are vulnerable.

To balance these considerations, we released SCooLS *without the Exploit Generator* 60 days after publication and the Exploit Generator a further 60 days after that: publication at BCCA on 24-26 October, 2023; SCooLS release on 25 December (Christmas) 2023; and Exploit Generator release on 23 February 2024. SCooLS is available for download from <https://bit.ly/SCooLS-Tool> (see “README.md”). The instructions contain explanation for how to analyze batch of contract bytecodes at function-level, each function is judged extremely quickly (≈ 0.05 seconds per function).

5.2 Deep learning styles

The two most popular deep learning techniques are *supervised* and *unsupervised*. Supervised learning involves training a model on a *labeled* dataset, attempting to induce relationships between the elements and their labels. The main disadvantage of supervised learning is the need to source a large amount of labeled data for training, which can be expensive and time-consuming to collect. Moreover, labeling data can be subjective and error-prone, and mislabeled data can affect the accuracy of the trained model. For example, if we use a static analyzer to label a large training data set, training must cope with the false positives and negatives produced by said analyzer.

In contrast, unsupervised learning involves analyzing *unlabeled* data to identify patterns within the data, such as clusters, anomalies, or associations. Avoiding the necessity oracle-labeled data is a major positive, but without a target output for comparing predictions, it can be difficult to develop a model to solve a particular decision problem of interest. There are numerous algorithms available for unsupervised learning, but selecting the most appropriate one for a given problem can be challenging. Identifying which algorithm will produce the most optimal results for a specific dataset may be difficult.

In our research endeavor, we sought to employ the Fuzzy C-Means (FCM) clustering algorithm [17]. FCM is a commonly used algorithm in the domain of data analysis and pattern recognition that is capable of classifying input data into multiple clusters based on their degree of similarity. Unlike traditional clustering algorithms that assign a data point to a single cluster, FCM assigns each point to every cluster with a certain degree of membership, ranging from 0 to 1. This degree of membership represents the degree of similarity of the data point to the centroid of that cluster.

In Figure 5.1 the FCM algorithm works by randomly initializing the centroids for the given number of clusters. Then, for each data point, it calculates the degree of membership to each cluster based on the Euclidean distance between the data point and the centroid of that cluster. These membership degrees are used to update the centroids iteratively until convergence is reached, which occurs when the membership degrees stabilize and the centroids stop moving. The FCM algorithm is sensitive to

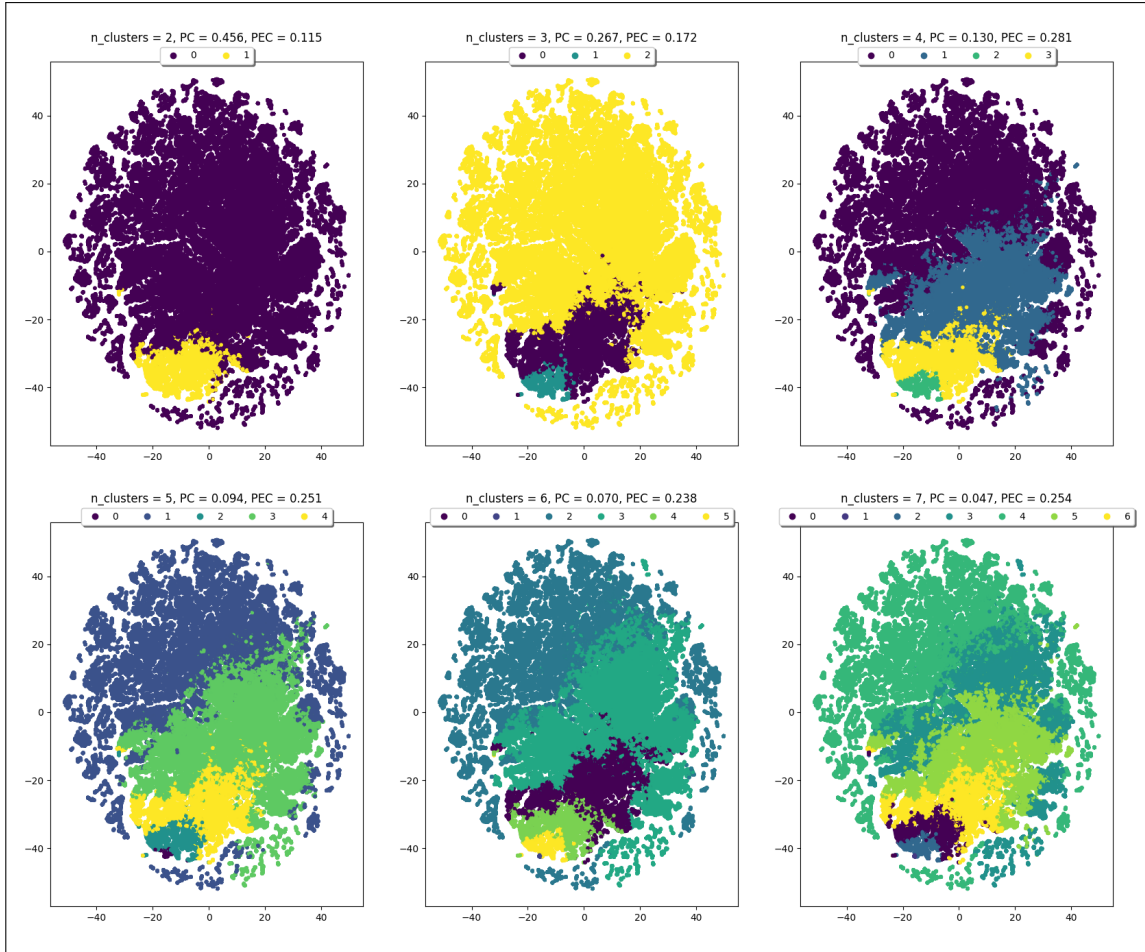


Figure 5.1: Fuzzy C-Means clustering algorithm for reentrancy.

the initial centroid positions, and therefore multiple runs with different initializations are typically performed to find the optimal solution. Additionally, the algorithm requires the specification of the number of clusters in advance. The performance of the FCM algorithm depends on the selection of the initial cluster center and/or the initial membership value.

Semi-supervised learning strikes a balance between supervised and unsupervised learning. An initial set of models is trained on a small *labeled* data set, and then used to label a large *unlabeled* data set. The resulting labels are sorted by confidence, and the high-confidence labels are then used to train the next generation of models. The process then iterates.

Key idea: In many domains, labeled data is scarce or expensive to obtain. Semi-

supervised learning uses one to three orders of magnitude less data than supervised learning, and it is easier to gain confidence in the labels of a small data set.

Semi-supervised learning leverages the small labeled training data set to orient its models (as compared to unsupervised learning). The large unlabeled data set is used to increase generality and improve robustness to noise/outliers (as compared to supervised learning on a small data set). By leveraging both, the model can learn to recognize patterns and make accurate predictions on new data, despite having only a small amount of genuinely high-confidence labeled data available.

5.3 Design of data sets

We assemble two data sets: a small high-confidence manually-labeled *ReentrancyBook*, and a large unlabeled *BigBook*. We publish both data sets [12, 6].

ReentrancyBook We collected 932 smart contracts, containing a total of 11,587 functions, from prior work [46, 54, 44]. We then removed redundant functions to yield 502 distinct functions. All of the 932 contracts had source available, enabling manual labeling of these 502 functions as *vulnerable* or *non-vulnerable* for the *reentrancy* bug.

Labelling functions requires judgment calls, and our labelling did not always agree with previous work. Not every vulnerable function is actually exploitable [107], and some functions are exploitable only under unusual circumstances. For example, some functions can only be run by the contract’s owner; or will only send Ether to specific hardcoded addresses; or can only work at specific times or block numbers; or some piece of contract state is updated before reentrancy, preventing exploitation. *We only label a function vulnerable if it is exploitable by the general public without such restrictions.* In total, we consider only 22 functions to be vulnerable, with the remaining 480 considered non-vulnerable. We dub the unique labelled functions the *ReentrancyBook* data set.

We use the *stratified sampling* of *scikit-learn* [106] to divide *ReentrancyBook* into halves, with one half being the training/validation set *ReentrancyStudyBook* (250 functions, of which 11 are vulnerable), and the other as the test set *ReentrancyTestBook* (252 functions, of which 11 are vulnerable).

BigBook Semi-supervised learning requires both a small trusted core data set (*ReentrancyBook*) and large secondary data set. We downloaded the latter on Feb 24, 2023 from Google BigQuery [57], yielding 17,806,779 contracts containing 76,024,596 functions¹. 99.3% of functions returned from BigQuery are duplicates; removing redundant functions left us with 554,111 distinct functions. We further removed the 446 functions already contained in *ReentrancyBook* to leave us with 553,665 distinct functions, which we dub the *BigBook* data set. The *BigBook* functions are unlabeled, and by construction *BigBook* and *ReentrancyBook* are disjoint.

5.4 Designing SCooLS

The design framework of SCooLS is sketched in Figure 5.2. The design divides into two parts: A) Preprocessing and B) Graph Neural Networks. Preprocessing is done at the beginning and once. The Graph Neural Networks (GNNs) are where the training occurs. We will discuss each part in turn.

5.4.1 Preprocessing

Preprocessing begins with the data collection and manual labeling discussed above in §5.3, and then proceeds to turn a collection of contracts into a collection of vector-labeled graphs representing individual contract functions. SCooLS is the first machine learning-based technique judging *individual functions* rather than *whole contracts* and provides an auto-exploit demonstrations for the end-users/developers.

Conventional NLP pretraining techniques treat code as a sequence of tokens, just as they would a natural language. However, this approach neglects the valuable structural information present in code that can aid the understanding of its behaviour. Control-flow graphs (CFGs), directed graphs whose vertices are basic blocks and whose edges represent execution flow, are more useful for analysis because they capture important semantic structures within the contract [3].

We use *evm-cfg-builder* (v0.3.1) [19] to extract control flow graphs (CFGs) from directly EVM bytecode. The average function in our data sets has 14 basic blocks

¹The query is available here: <https://console.cloud.google.com/bigquery?sq=814627022739:e4a5075f5a7141078f0e170ced82ffa0>

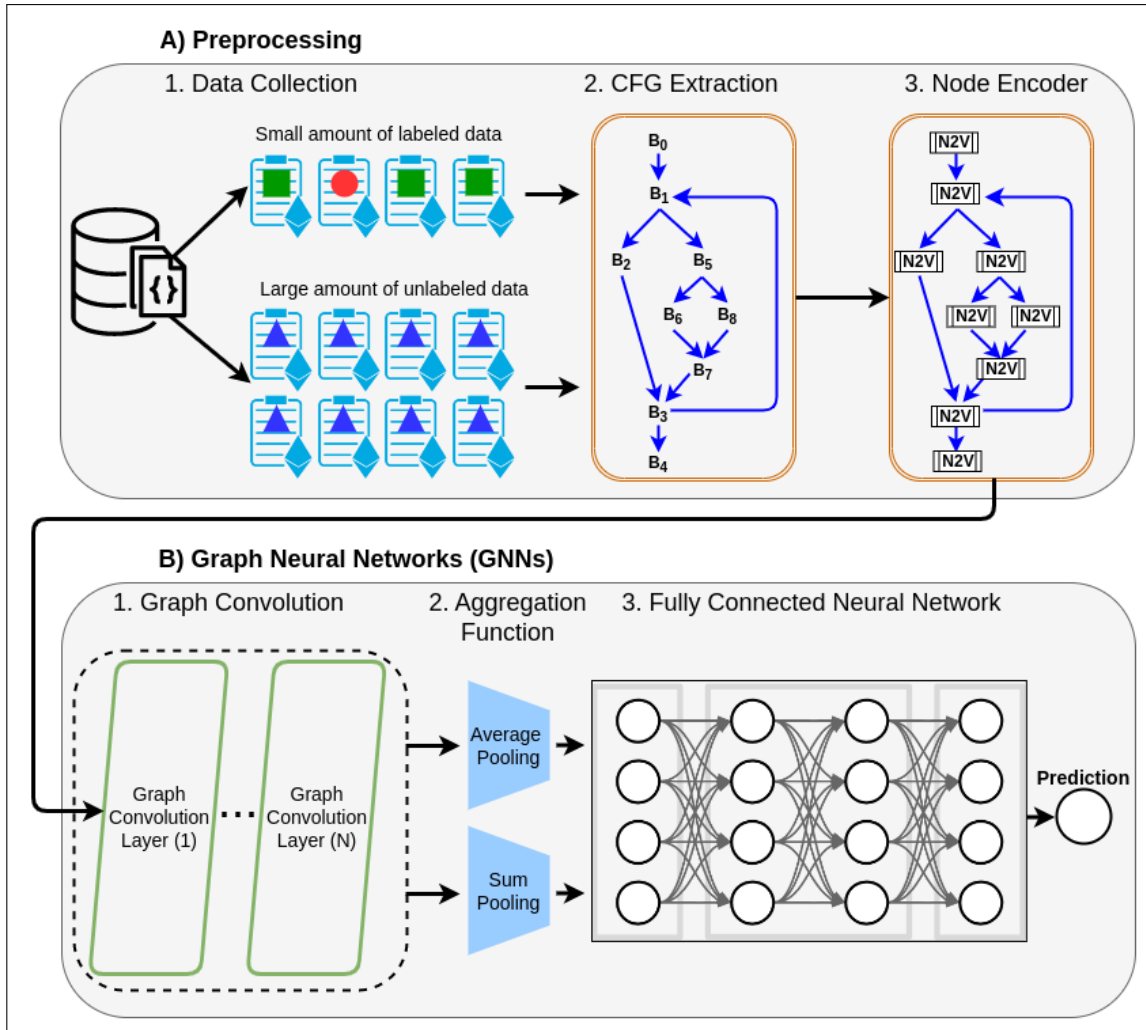


Figure 5.2: Data Preprocessing and Graph Neural Networks (GNNs) Design.

(nodes), each containing a textual sentence of opcodes (*e.g.*, “PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE...”).

Most machine learning techniques prefer to work on vectors rather than sentences, so to encode a node into fixed-length 512-dimensional vector (N2V), we use the transformer architecture [138] in the Universal Sentence Encoder [29].

Transformers rely on the self-attention mechanism, processing the whole sequence all at once (no sequential processing like in RNNs), and assigning a weight to each opcode to indicate how much “attention” the model should pay to said opcode. The model takes opcode order and the larger surrounding context into account when generating an opcode representation. The transformer encoder is composed of 6

stacked transformer layers. Each layer has two sub-layers: a multi-head self-attention mechanism followed by a fully connected feed-forward network. Transformer uses a residual connection around each of the two sub-layers, followed by layer normalization to produce its 512-dimensional outputs, as shown in Figure 5.3.

5.4.2 Graph Neural Networks (GNNs)

A graph neural network (GNN) is designed to perform inference on graph data structures. In other words, it is a neural network that can classify based on graph features. In a GNN, each node in the graph is represented by a feature vector containing information about the node and its neighbors. The GNN uses a series of graph convolutional layers based on *message passing* [55] to update each node’s feature vectors by aggregating the information from neighboring nodes. This process is iterated to allow the network to learn more complex relationships between nodes (*e.g.*, after three graph convolutional layers, a node has information about the nodes three steps away from it).

Graph neural networks (GNNs) have been successfully applied in a wide range of domains across various learning settings. *Key idea:* GNNs are particularly effective when dealing with datasets that can be represented as graphs, where traditional machine learning algorithms may not be suitable.

SCoolS’s neural nets are based on five state-of-the-art graph convolution methods [153, 130, 149, 64, 80]. As shown in Figure 5.2.B, our design consists of a series of graph convolutional layers, followed by an aggregation, which in turn is followed by a fully connected feed-forward network with a sigmoid activation function for classification. The design space of our models involves several hyperparameters, including:

- 1) the number of graph convolutional layers (1, 2, or 3)
- 2) the number of neurons (32, 64, 128, or 256) updated by each layer using a rectified linear unit (ReLU) activation.
- 3) the pooling aggregation function (average or sum).
- 4) the number of neurons in two Dense layers with a ReLU activation in the fully connected feed-forward network, which is the same as the number of

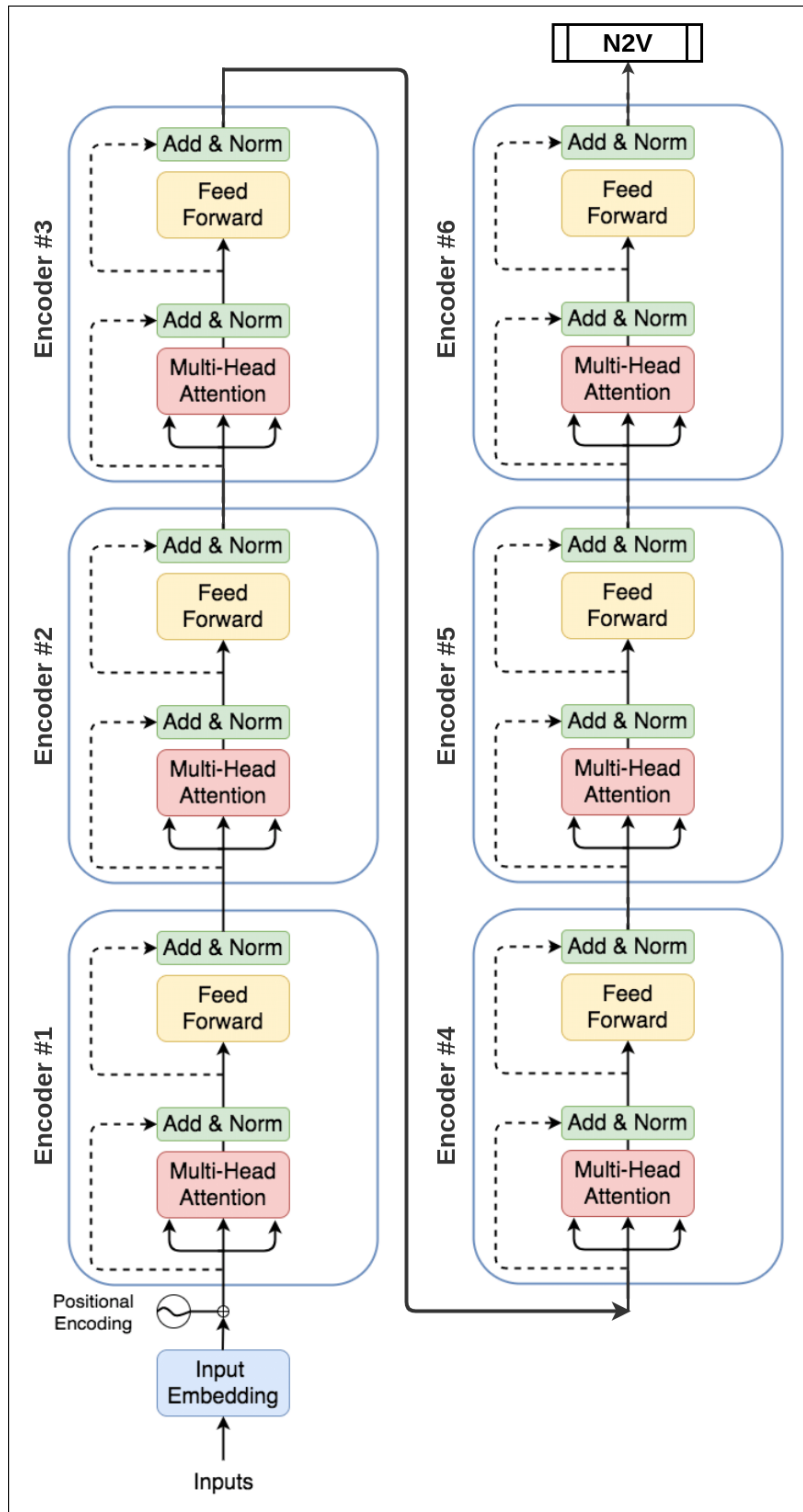


Figure 5.3: Architecture of a Transformer with six encoder layers.

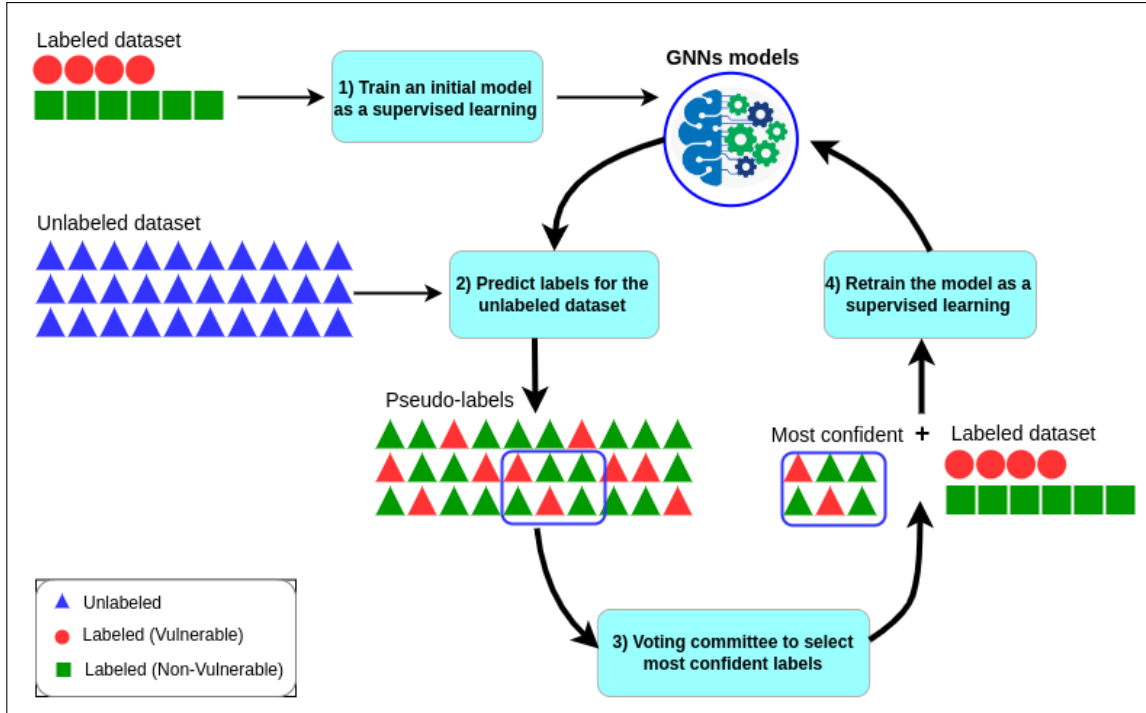


Figure 5.4: The Smart Contracts Semi-Supervised Learning (SCoolS).

neurons in the convolutional layer. Each Dense layer is surrounded by BatchNormalization and Dropout layers to enhance the model’s generalization and prevent overfitting.

Key idea: our framework thus generates $5 \times 3 \times 4 \times 2 = 120$ models that can learn from different perspectives, resulting in a significant improvement in accuracy.

5.4.3 Semi-Supervised Self-Training

With the basic design framework in place, we must now explain how the neural nets are trained. One important consideration when using the self-training approach is that the model may make errors when labeling the unlabeled data, which can lead to the propagation of these errors in subsequent iterations of the training process. To mitigate this risk, it is important to carefully monitor the quality of the newly labeled data and use techniques such as active learning or voting committee to select the most informative samples for labeling.

We overview SCoolS’s training cycle in Figure 5.4, illustrating the semi-supervised

learning process. The process begins by using our small labeled dataset *ReentrancyStudyBook* to train an initial set of models (the training engineering is in §5.6).

After the initial models are trained, we enter an iteration loop in which we ask the current set of models to judge the enormous *BigBook* of 553,665 functions. The output for each of the 120 models is a number between 0 (certainly non-vulnerable) and 1 (certainly vulnerable). *Key idea:* we now run a voting committee to determine which contract labels have sufficient model support. We discard any model with confidence $0.1 < x < 0.9$, leaving only the high confidence models, which then vote for 0 or 1. If a function gets 80+ votes (two-thirds of the models), then that label is accepted. A function without such a supermajority is considered *unknown*.

We take the newly-labeled functions, add the trusted *ReentrancyStudyBook*, and retrain the models for the next iteration. We continue until a termination condition is met (e.g. we label all the data, there are no more unlabeled functions that meet our voting committee criteria, we reach the specified max number of iterations, or the effect of the classification models is no longer improved).

5.4.4 Final trained models

After training, analyzing a fresh contract is straightforward. The bytecode is converted to a CFG, whose nodes are transformed into vectors. Subsequently, the trained GNNs models from the final self-training iteration predict labels for each function, producing a total of 120 predictions. Finally, the voting committee applies a voting mechanism to determine the final classification label based on whether at least two-thirds of the predictions with a confidence level of 90% or higher classify the function as *vulnerable* or *non-vulnerable*.

5.4.5 Discussion

Key idea: semi-supervised training is particularly useful in situations where labeled data is scarce or expensive to obtain, but a large amount of unlabeled data is available. To use *supervised training*, we would need to assemble vulnerability data sets with hundreds (or thousands) of positive examples, and tens (or hundreds) of thousands of negative examples.

This size is prohibitive for manual analysis. On the other hand, every automated

analysis exhibits false positives and negatives, which can easily corrupt the learning process (garbage in, garbage out). Semi-supervised training sidesteps both of these concerns. It can use a training set 1-3 orders of magnitude smaller, so developing a high-confidence labeled data set is practicable. Recall that *ReentrancyStudyBook* has only 250 functions, of which merely 11 are vulnerable.

By leveraging the unlabeled data, the model can learn to recognize patterns and make accurate predictions on new data, despite having only a small amount of genuinely high-confidence labeled data available.

5.5 Auto-Exploit Generator Design

Much of the effort in a hack is *finding* the needle vulnerability in the blockchain haystack; as we will see in §5.6, SCooLS does an admirable job at this. Step two, *exploiting* the discovered vulnerability, is then often straightforward.

A *smart contract auto-exploit generator* automatically generates the malicious code needed to exploit a vulnerable smart contract. We implement a simple yet effective auto-exploit generator (combination of DL and Program Synthesis) to prove that attackers can exploit the detected reentrancy vulnerabilities to steal contract funds.

Generating the exploit The generator is given the Application Binary Interface (ABI) of the victim contract C , together with the name of the function \mathbf{V} that SCooLS has flagged as vulnerable. The generator examines the ABI to identify all functions $\mathbf{P}_1, \dots, \mathbf{P}_n$ marked as **payable**. Next, for each \mathbf{P}_i , the generator proceeds as follows:

- 1) It generates an **Itarget** interface to facilitate interaction with the victim contract C by combining the signature of the target payable function \mathbf{P}_i with the signature of the flagged-vulnerable function \mathbf{V} .
- 2) It generates the contract **TheAttacker**, containing the functions **attack_step1**, **attack_step2**, **receive**, and **steal**, together with contract boilerplate. The details are generated by template as follows.

- 3) In `attack_step1`, the attacker contract invokes the chosen `payable` function P_i . Appropriate-typed arguments are selected at random from predefined dictionary values. The result is to transfer some Ether to the victim contract, emulating an honest user's deposit.
- 4) In `attack_step2`, the attacker contract calls the vulnerable function V , again providing it with suitably-typed and randomly-chosen arguments.
- 5) The key to the hack is the `receive` function, which is automatically triggered if V transfers Ether to `TheAttacker` during `attack_step2`. `receive` checks to see if C has sufficient remaining funds to justify continuing the attack, and if so calls V again².
- 6) Lastly, the `steal` function transfers the money out of `TheAttacker` contract and into the hacker's wallet.

```

1  interface Itarget{
2  /* signature of the payable function P */
3  /* signature of the vulnerable function V */
4  }
5  contract TheAttacker{
6      ...
7      function attack_step1() external payable {
8          /* call the payable function P */
9      }
10     function attack_step2() external {
11         /* call the vulnerable function V */
12     }
13     receive() external payable {
14         if ( has_funds ) {
15             /* call the vulnerable function V */
16         }
17     }
18     function steal() public payable{
19         attacker.transfer(address(this).balance);
20     }

```

²A more sophisticated version would also track the recursion depth and would halt the theft before hitting the 1,024 function call depth.

```
21 }
```

Here is the victim contract **TheBank** using payable function **deposit** and vulnerable function **withdrawal**:

```
1 // SPDX - License - Identifier : MIT
2 pragma solidity ^0.8.17;
3 contract TheBank {
4     mapping(address => uint) theBalances;
5     function deposit() public payable {
6         theBalances[msg.sender] += msg.value;
7     }
8     function withdrawal() public {
9         uint bal = theBalances[msg.sender];
10        require ( bal > 0);
11        (bool success, ) = msg.sender.call{value: bal}("");
12        require(success, "transaction failed");
13        theBalances[msg.sender] = 0;
14    }
15 }
```

The aim of the reentrancy attack is to drain the funds of a victim contract and steal its money. In the case of **TheBank** contract, the reentrancy attack can be taken advantage of due to the fact that the balance of the sender is updated after the money transfer has occurred . At this point, the attacker makes another call to withdraw while the balance has not yet been updated, so the conditions for a new transfer are still satisfied. This process can be repeated in a loop of reentrant calls until all the funds of **TheBank** are transferred.

Example of automatic generation of **TheAttacker** contract:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.1.0 <0.9.0;
3 interface Itarget{
4     function deposit() external payable;
5     function withdrawal() external;
6 }
7 contract TheAttacker{
8     Itarget public theBank;
9     address payable public attacker;
```

```

10     uint256 public amount = 1 ether;
11     constructor(address _thebankAddress, address payable _attackerAddr) {
12         theBank = Itarget(_thebankAddress);
13         attacker = _attackerAddr;
14     }
15     function attack_step1() external payable {
16         theBank.deposit{value: msg.value}();
17     }
18     function attack_step2() external {
19         theBank.withdrawal();
20     }
21     receive() external payable {
22         if (address(theBank).balance >= amount) {
23             theBank.withdrawal();
24         }
25     }
26     function steal() public payable{
27         attacker.transfer(address(this).balance);
28     }
29 }

```

The attack starts with the `attack_step1` function, which initiates a payment to the victim contract using the `deposit` function. Then, the `attack_step2` function calls the `withdrawal` function, which transfers the funds back to the attacker. During this transfer, the `receive` function is triggered, and before it completes, `TheAttacker` calls the `withdrawal` function again to continue draining the victim's funds.

Testing the exploit We use Ganache [73], a local development and testing platform, to replicate the blockchain environment.

- 1) Deploy the bytecode and ABI of the victim contract to our local blockchain, with a balance of 0 Ether.
- 2) Allow normal users to perform standard transactions, invoking the *payable* function to send some Ether to the victim contract and increase its balance.
- 3) SCoolS's detector pinpoints exploitable functions within a victim contract.

- 4) For each exploitable function, SCooLS’s auto-exploit generator designs N attacker contracts corresponding to N payable functions.
- 5) These attacker contracts are deployed to a controlled local blockchain environment for safe and ethical testing.
- 6) Initiate the hack by executing the *attack_step1*, *attack_step2*, and *steal* functions in sequence using three transaction calls.
- 7) The attacker’s balance is monitored before and after each attack to see if the attack was successful:

```

Attacker Balance Before Attack : 1808.65 Ether(s)
Attacker Balance After Attack  : 1810.64 Ether(s)
This exploitation net profit is: 1.99 Ether(s)
The detected reentrancy can be exploited.

```

- 8) SCooLS provides concrete evidence of financial risks, enabling developers to prioritize and address vulnerabilities effectively.

5.6 Experiments and Evaluation

5.6.1 Evaluative Metrics

We evaluate the quality of our models from several complementary perspectives. We use metrics Accuracy (ACC), F1-Score (F1), and False Positives Rate (FPR). Accuracy is the number of correct predictions made by the model divided by the total number of predictions made. However, accuracy alone can be misleading if the dataset is imbalanced or if the cost of false positives and false negatives are not equal. An F-score (F1) is commonly used to benchmark deep learning for classification tasks, defined as the harmonic mean of *precision* and *recall*³. Precision measures the proportion of true positives out of all predicted positives, while recall measures the proportion of true positives out of all actual positives. The False Positive Rate

³The F1 score ranges from 0 to 1, with 1 being the best possible score indicating perfect precision and recall. In general, a higher F1 score indicates better model performance.

represents the proportion of actual negative instances that are incorrectly predicted as positive by the model. In other words, it measures the percentage of times that the model generates a false positive prediction out of all negative instances. We formally define metrics in §2.6.

5.6.2 Experimental setup

Our machine had 32 GB of memory and a 12-core 3.2 GHz Intel(R) Core(TM) i7-8700 CPU. We used 64-bit Ubuntu 20.04.6 LTS (Focal Fossa), tensorflow 2.12.0 [1], tensorflow_hub 0.13.0, spektral [58], evm-cfg-builder [19], and ganache-2.5.4-linux-x86_64.AppImage [73].

We train the initial models using *ReentrancyStudyBook*, and then begin the self-training cycle with a learning rate of 0.005, a batch size of 2,048, and 1,000 epochs. We use the Adam optimizer with a categorical cross-entropy loss function.

To prevent overfitting, we use a rolling 200-epoch window and measure the validation loss for each model against the *ReentrancyStudyBook*. If the model with the lowest validation loss was 200 epochs ago, then training stops and restores that model. Figure 5.5 displays the training and validation accuracy of an arbitrarily-chosen model. The validation loss (orange line) minimizes around epoch 57, so when training stops 200 epochs later, it restores the model from epoch 57.

Table 5.1: The *BigBook* data set.

Models	<i>Vulnerable</i>	<i>Non-vulnerable</i>	<i>Unknown</i>
<i>BigBook</i> size	0	0	553,665
<i>ReentrancyStudyBook</i> training	48	529,783	23,834
Self-training first iteration	69	553,132	464
Self-training second iteration	82	553,322	261

After training 120 models (one cycle depicted in Figure 5.4), we repeat again, for three iterations in total including the initial training. Table 5.1 shows how the training process shrinks the unknown set over time. In total we trained 360 models over a period of four days, *i.e.* approximately 16 minutes per model. The 120 models from the last iteration are used in SCoolS.

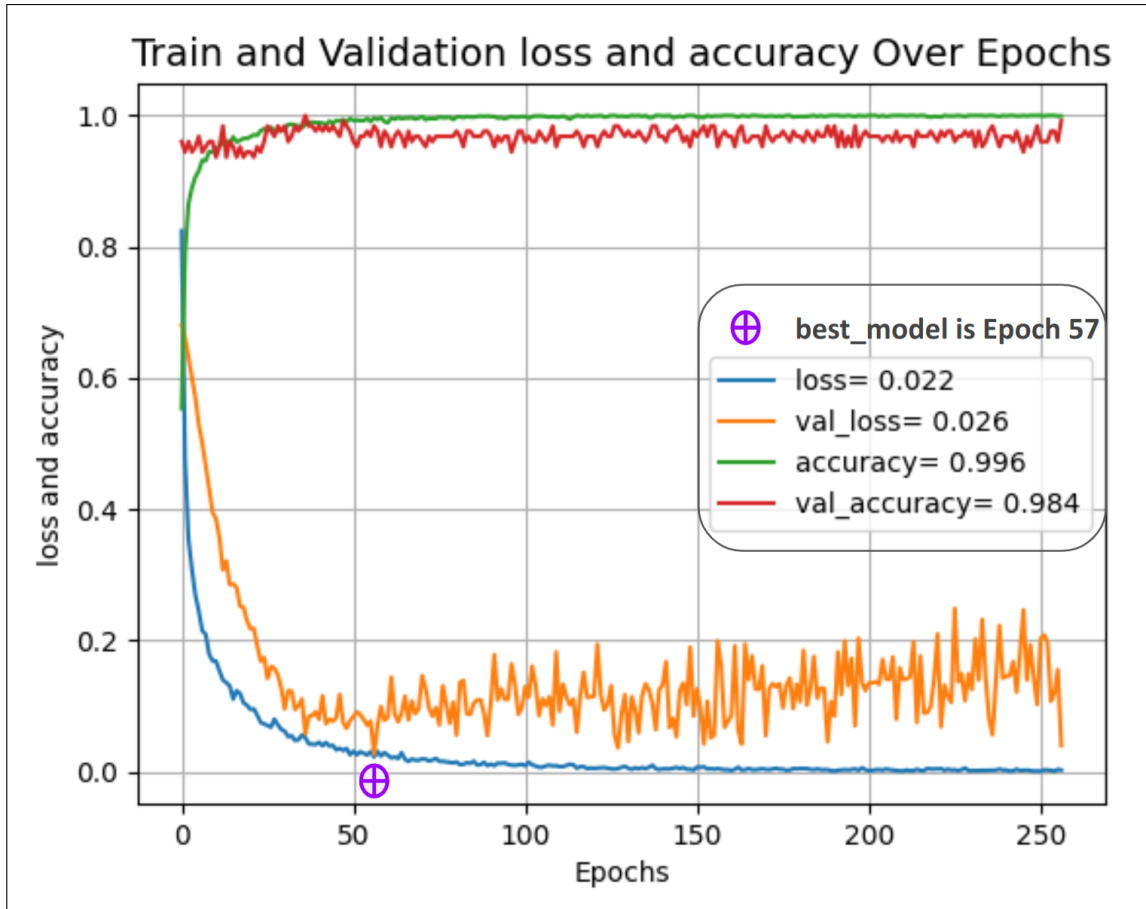


Figure 5.5: Training Accuracy and Loss, and Validation Accuracy and Loss

5.6.3 SCoolS vs. state-of-the-art tools

To test the performance of our approach, we evaluated it on the *ReentrancyTestBook*, which by construction is disjoint from the *ReentrancyStudyBook* and *BigBook* data sets used during training. Moreover, we manually labelled each of its 252 functions, giving us high confidence their labels.

We also wish to compare SCoolS against the state-of-the-art actively-used alternatives ConFuzzius v0.0.1 [133], Slither 0.9.0 [45], and Mythril v0.23.17 [100].

The results are shown in Table 5.2. The primary data are True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). The derived statistics are Accuracy (ACC), F-score (F1), and False Positive Rate (FPR).

SCoolS has an overall accuracy 98.4% and F1 score 90.4% with an associated false positive rate of only 0.8%. It enjoys the highest accuracy among the tools, the

highest F1 score, and the lowest false positive rate. Moreover, the average time to analyze a function was only 0.05 seconds, tied for first place.

ConFuzzius [133] uses a hybrid fuzzer that uses data dependency analysis to generate effective test cases for smart contracts. Hybrid fuzzing involves an initial stage of traditional fuzzing that continues until reaching a saturation point where no new code coverage is achieved after executing a predefined number of steps. Upon reaching this point, the hybrid fuzzer automatically switches to the process of symbolic execution, which performs an exhaustive search for unexplored branching conditions. If a branching condition is found, the symbolic execution process solves it, and the hybrid fuzzer reverts to the traditional fuzzing stage. ConFuzzius attains an accuracy level of 97.2%, accompanied by an F1 score of 85.3%, while maintaining a remarkably low false positive rate of merely 2.1%. The entire process takes 0.48 seconds per function.

Slither [45] is a static analysis framework that analyzes smart contracts source code through the integration of data flow analysis and taint analysis. Slither incorporates a large number of detectors that can identify specific issues such as reentrancy, integer overflow, and uninitialized variables. Slither achieves an accuracy level of 95.6%, accompanied by an F1 score of 82.2%, while retaining a notably low false positive rate of 4.6%. The complete analysis process for each function takes a mere 0.05 seconds, also tied for first place.

Mythril [100] is a powerful tool for identifying potential security issues in smart contracts, and it is widely used by developers, auditors, and researchers. Mythril works by using symbolic execution to explore all possible execution paths through the smart contract’s code to detect potential vulnerabilities. Mythril demonstrates an accuracy level of 92.1%, along with an F1 score of 72.8%, while maintaining a relatively higher false positive rate of 7.9%. The complete analysis process for each function is relatively slower, taking around 12.5 seconds per function.

Table 5.2: SCooLS vs. state-of-the-art tools.

Tool	TP	FP	TN	FN	ACC	F1	FPR	Time
SCooLS	9	2	239	2	98.4	90.5	0.8	0.05
ConFuzzius [133]	9	5	236	2	97.2	85.3	2.1	0.48
Slither [45]	11	11	230	0	95.6	82.2	4.6	0.05
Mythril [100]	10	19	222	1	92.1	72.8	7.9	12.5

We evaluate each self-training iteration (Initial, Second, and Third). We use three committee sizes [70, 80, 90] out of 120 trained models, and three confidences (Con) [0.85, 0.90, 0.95] in each iteration. As shown in Table 5.3,

Table 5.3: Results on *ReentrancyTestBook*.

	Con	<i>Committee 70/120</i>			<i>Committee 80/120</i>			<i>Committee 90/120</i>		
		ACC	F1	FPR	ACC	F1	FPR	ACC	F1	FPR
Initial	0.85	97.6	87.8	1.7	97.6	85.7	0.8	98.4	89.6	0.0
	0.90	96.8	82.5	1.7	98.4	89.6	0.0	98.4	89.6	0.0
	0.95	97.6	85.7	0.8	98.4	89.6	0.0	98.4	89.6	0.0
First	0.85	98.4	90.5	0.8	98.4	90.5	0.8	98.4	90.5	0.8
	0.90	98.4	90.5	0.8	98.4	90.5	0.8	98.4	90.5	0.8
	0.95	98.4	90.5	0.8	98.4	90.5	0.8	98.0	87.6	0.8
Second	0.85	98.4	90.5	0.8	98.4	90.5	0.8	98.0	87.6	0.8
	0.90	98.4	90.5	0.8	98.4	90.5	0.8	98.0	87.6	0.8
	0.95	98.4	90.5	0.8	98.0	87.6	0.8	98.0	87.6	0.8

5.6.4 Auto-exploit generator results

To test our auto-exploit generator, we took the 82 positives SCoolS found in *BigBook* (see Table 5.1). These 82 positives have a further 89 duplicate instances in our Google BigQuery, for a total of 171 vulnerable functions.

Unfortunately, not all of the contracts containing these functions offered an Application Binary Interface (ABI) that allows the general public to easily interact with their functions. Specifically, only 33 out of the 171 functions had an ABI available on Etherscan, yielding 14 unique functions. We were able to retrieve the source code for all 14 unique functions. Manual inspection resolved 4 false positives (7 including duplicates, 21.2%) and 10 true positives (26 including duplicates, 78.8%).

Most of the false positives are due to functions that are *nearly* exploitable, for example because:

- 1) the contract cannot receive ether, so nothing to steal;
- 2) the receiver is hardcoded into the contract, so ownership of a specific address is required to attack; or,
- 3) the function can only be called at certain timestamps or block numbers, which makes it challenging to exploit.

Very few of the flagged functions are not exploitable for the “right” reasons, *i.e.* because the contract manages its internal state shrewdly to avoid the exploit during reentrancy.

Of course, our auto-exploit generator cannot exploit a false positive. However, it was able to exploit 6 of the true positives (20 including duplicates, 76.9%) as shown in Table 5.4. The remaining 4 true positives (6 with duplicates) are exploitable, but our auto-exploit generator is not smart enough to do so.

Conclusively, the automated identification of multiple practical end-to-end exploits in smart contracts holds substantial importance.

Table 5.4: The Auto-Exploit Generator.

Address	Vulnerable function	Function duplicates	Exploit Gen
0x65e5909d665cb ...	CashOut(uint256)	11	✓
0xe610af01f92f1 ...	Collect(uint256)	1	✗
0x2ec17d1df257d ...	call()	1	✗
0x2a98d8fc14b31 ...	withdraw()	2	✓
0xa5d6accc56953 ...	CashOut(uint256)	4	✓
0x0ebe1a9cbf4e2 ...	settleEther()	2	✗
0xdd17afae8a3dd ...	Collect(uint256)	2	✗
0xb7c5c5aa4d429 ...	withdraw()	1	✓
0xf6dbe88ba55f1 ...	withdraw(uint256)	1	✓
0xaf905ab8dad7c ...	pullFunds()	1	✓

5.7 Key Comparative Studies

Static and dynamic analyzers The detection of vulnerabilities in software is crucial to ensure the security and reliability of the system. Traditional methods used for vulnerability detection, such as static and dynamic analysis, have also been applied to smart contracts [93, 100, 134, 132, 45, 105, 136, 99, 76, 59, 133, 105]. Several of these tools (especially ConFuzzius, Slither, and Mythril) are under active development and are widely used in the community. These tools use expert-crafted rules and manually engineered features, which can make it challenging to maintain and update them. Many require or at least benefit from source code (Mythril can analyze bytecode with some accuracy loss, and earlier tools such as Oyente worked

directly with bytecode). Bytecode analyzers tend to be less precise (and are often significantly slower), so there is a need for more advanced techniques for vulnerability detection at scale.

Machine learning Machine learning approaches for smart contract vulnerability detection has gained some attention as an alternative to traditional analyzers [128, 98, 89, 126, 109, 3]. Wesley et al. [128] improved vulnerability detection in smart contracts by using a customized LSTM neural network that sequentially examined opcodes, resulting in superior accuracy compared to Maian. Momeni et al. [98] proposed a machine learning model that used AST and CFG to analyze static source code, with 17 code complexity-based features for model training. This approach achieved a faster processing time and lower miss rate than the static analyzers Mythril and Slither. Liao et al. [89] created SoliAudit, an approach that combines machine learning and a dynamic fuzzer to enhance vulnerability detection capabilities. To construct the feature matrix, Liao et al. employed word2vec [97] to generate a vector representation for each opcode, which were then combined row-wise. The control-flow of the smart contract was not taken into account in their approach. Sun et al. [126] improved smart contract vulnerability detection by incorporating an attention mechanism into (non-graph) convolutional neural networks. Their approach outperformed Oyente and Mythril in terms of both miss rate and processing time. The proposed method utilizes a CNN model combined with self-attention and achieves swift detection of vulnerabilities with a reduced miss rate and average processing time. Qian et al. [109] proposed a deep machine learning approach to identify Reentrancy vulnerabilities in smart contracts, utilizing the BLSTM-ATT model. Their method divided the source code into snippets and employed word2vec to extract code features. The approach showed that deep learning techniques are suitable for smart contract vulnerability detection and can achieve high performance.

Sun et al. [125] proposed ASSBert, a new framework that utilizes BERT [38] for smart contract vulnerability classification, specifically designed to handle limited labels of solidity source files. The framework uses active and semi-supervised learning approaches to improve the model's performance. ASSBert outperformed baseline methods such as Bert, Bert-AL, and Bert-SSL in terms of performance. However, it

has not been compared with state-of-the-art tools and is not publicly available for comparison.

Overall, the application of deep learning approaches to smart contract vulnerability detection shows promising results, and the use of semi-supervised learning techniques can potentially improve the accuracy of the models while reducing the cost of obtaining labeled data.

Auto-exploit generators teEther by Krupp et al. [82] generates exploits for suicidal and call injection vulnerabilities on the Ethereum platform by analyzing the binary bytecode of a smart contract. However, it is not designed to identify or exploit other common vulnerabilities, such as reentrancy.

Jin et al. [77] developed EXGEN, a tool that generates attack contracts with multiple transactions and tests their exploitability on a private blockchain using public blockchain values. While the tool is publicly available, reproducing its results for comparison is challenging due to its complex environment setup.

5.8 Summary

In this research, we have proposed a deep learning approach for Smart Contracts Semi-Supervised Learning (SCooLS) to detect vulnerable functions in Ethereum smart contracts at the bytecode level. Our approach incorporates semi-supervised learning and deep graph neural networks (GNNs) to analyze smart contract bytecode without any manual feature engineering, predefined patterns, or expert rules.

The results show that our SCooLS approach outperforms existing state-of-the-art tools, achieving an accuracy level of 98.4% and an F1 score of 90.5%, while exhibiting an exceptionally low false positive rate of only 0.8%. Additionally, the analysis process for each function is also quicker than existing tools, requiring only 0.05 seconds.

We have also introduced a voting committee to ensure the integrity of newly labeled data during the self-training process and avoid the spread of errors to subsequent iterations. Moreover, we have implemented an auto-exploit generator to verify that the detected vulnerabilities are real and can be exploited by attackers to steal contract funds.

To the best of our knowledge, our work is the first to propose a semi-supervised self-training method for detecting vulnerabilities in smart contracts bytecode at the function level and providing developers with a tangible method to test the exploitability of their contracts.

Our results demonstrate the effectiveness of the proposed approach and its potential to enhance the security of smart contracts. In conclusion, our work contributes significantly to the field of smart contract security and provides a strong foundation for future research in this area.

We wish to explore several directions in the future. We aim to extend with training more vulnerabilities as much as we can get a small labeled functions for them. We hope to enhance the auto-exploit generator by using Fuzzing to help in creating inputs for functions, and may train a ML model to guide the search for which input should be selected instead of randomly selection, this will make it more effective.

5.9 Availability

SCoolS is available for download from <https://bit.ly/SCoolS-Tool> (see “README.md”). The data sets we use in this research are available as well [12, 6].

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In conclusion, this thesis proposes two deep learning approaches for detecting vulnerabilities in Ethereum smart contracts: the Deep Learning Vulnerability Analyzer (DLVA) and Smart Contracts Semi-Supervised Learning (SCoolS). Both approaches incorporate deep learning techniques to analyze smart contract bytecode without the need for manual feature engineering, predefined patterns, or expert rules.

DLVA uses a generic design of supervised deep neural networks to detect 29 different types of vulnerabilities in smart contracts. DLVA predicts Slither’s labels with an overall accuracy of 92.7% and associated false positive rate of 7.2%. We benchmark DLVA against eleven well-known smart contract analysis tools. Despite using much less analysis time, DLVA completed every query, leading the pack with an average accuracy of 99.7%, pleasingly balancing high true positive rates with low false positive rates. The DLVA is efficient, easy-to-use, and fast, making it a valuable tool for detecting vulnerabilities in smart contracts.

SCoolS, on the other hand, uses semi-supervised learning and deep graph neural networks (GNNs) to analyze smart contract bytecode at the function-level. The SCoolS approach achieved an accuracy level of 98.4%, an F1 score of 90.5%, and a false positive rate of only 0.8%, outperforming existing state-of-the-art tools. The approach also includes a voting committee to ensure the integrity of newly labeled data during the self-training process and an auto-exploit generator to verify the detected vulnerabilities.

Overall, the thesis presents two novel approaches for detecting vulnerabilities in Ethereum smart contracts using deep learning techniques, both of which outperform

existing state-of-the-art tools. These approaches have the potential to enhance the security of smart contracts and provide a strong foundation for future research in this area.

6.2 Future Research Directions

While this thesis focuses on utilizing our approach for smart contract vulnerability analysis, the underlying principles and insights gained are transferable to a broader range of secure program analysis tasks. This opens exciting possibilities for further research and development in diverse domains.

In the following, we propose prospective avenues for further research and expansion of the contributions made in this thesis.

- 1) Exploring hybrid approaches that leverage both deep learning and classical program analysis techniques to develop a robust infrastructure for vulnerability analysis. Such hybrid approaches can potentially improve the accuracy of detection by combining the strengths of both techniques, such as the ability of deep learning to identify complex patterns and the ability of classical program analysis to provide precise and interpretable results. Additionally, the investigation of novel techniques for optimizing the integration of these approaches can be an interesting avenue of exploration.
- 2) The integration of more data sources into vulnerability detection models, such as user behavior data, can help in building an anomaly detection models that could be able to catch effectively zero-day vulnerabilities in smart contracts without relying solely on labeled data, and to enhance the accuracy of detection and better protect smart contract users.
- 3) We aim to investigate the potential of utilizing smart contract transactions in the development of a robust exploit generator. Specifically, we propose to explore the effectiveness of employing fuzzing techniques to create a precise input generator for the exploit based on transaction data.
- 4) Furthermore, there exists another intriguing research avenue that involves exploring a scalable methodology for the automatic repair of vulnerable smart

CHAPTER 6. CONCLUSION AND FUTURE WORK

contracts. This approach would involve integrating deep learning techniques with both template-based and semantic-based patching methods, with the additional step of inferring context information from the bytecode.

We believe the above (but not limited to) future research directions will advance the technology presented in this thesis and contribute to academia and industry.

Bibliography

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”, *arXiv preprint arXiv:1603.04467*, 2016.
- [2] T. Abdelaziz and A. Hobor, “Schooling to exploit foolish contracts”, in *2023 Fifth International Conference on Blockchain Computing and Applications (BCCA)*, 2023, pp. 388–395. [Online]. Available: <https://ieeexplore.ieee.org/document/10338924>.
- [3] T. Abdelaziz and A. Hobor, “Smart learning to find dumb contracts”, in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 1775–1792, ISBN: 978-1-939133-37-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity23/presentation/abdelaziz>.
- [4] T. Abdelaziz and A. Hobor, “Smart learning to find dumb contracts (extended version)”, in *arXiv.org*, 2023. [Online]. Available: <https://arxiv.org/abs/2304.10726>.
- [5] T. Abdelaziz and A. Hobor, “Usenix’23 artifact appendix: Smart learning to find dumb contracts”, in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, ISBN: 978-1-939133-37-3. [Online]. Available: <https://www.usenix.org/system/files/usenixsecurity23-appendix-abdelaziz.pdf>.
- [6] T. Abdelaziz and A. Hobor, “BigBook dataset.” <https://bit.ly/BigQueryUnlabeledDataset>, [Online; accessed: April-2023].
- [7] T. Abdelaziz and A. Hobor, “Elysium_{benchmark}.” https://bit.ly/Elysium_benchmark, [Online; accessed December-2022].

BIBLIOGRAPHY

- [8] T. Abdelaziz and A. Hobor, “EthereumSC_{large} dataset.” https://bit.ly/EthereumSC_Dataset_Large, [Online; accessed October-2022].
- [9] T. Abdelaziz and A. Hobor, “EthereumSC_{small} dataset.” https://bit.ly/EthereumSC_Dataset_Small, [Online; accessed October-2022].
- [10] T. Abdelaziz and A. Hobor, “eThorZeus_{benchmark}.” https://bit.ly/eThor_Zeus_groundTruth1, [Online; accessed May-2023].
- [11] T. Abdelaziz and A. Hobor, “Reentrancy_{benchmark}.” https://bit.ly/Reentrancy_benchmark, [Online; accessed December-2022].
- [12] T. Abdelaziz and A. Hobor, “ReentrancyBook dataset.” <https://bit.ly/ManuallyLabeledDataset>, [Online; accessed: April-2023].
- [13] T. Abdelaziz and A. Hobor, “SolidiFI_{benchmark}.” https://bit.ly/SolidiFI_benchmark, [Online; accessed December-2022].
- [14] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, “Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts”, in *Proceedings of the 3rd ACM international symposium on blockchain and secure critical infrastructure*, 2021, pp. 47–59.
- [15] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok)”, in *International conference on principles of security and trust*, Springer, 2017, pp. 164–186.
- [16] E. Bertino, M. Kantarcioglu, C. G. Akcora, S. Samtani, S. Mittal, and M. Gupta, “Ai for security and security for ai”, in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021, pp. 333–334.
- [17] J. C. Bezdek, R. Ehrlich, and W. Full, “Fcm: The fuzzy c-means clustering algorithm”, *Computers & geosciences*, vol. 10, no. 2-3, pp. 191–203, 1984.
- [18] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, “Vulnerability prediction from source code using machine learning”, *IEEE Access*, vol. 8, pp. 150 672–150 684, 2020.
- [19] T. of Bits., “Evm-cfg-builder.” https://github.com/crytic/evm_cfg_builder/releases, [Online; accessed: April-2023].

BIBLIOGRAPHY

- [20] R. Böhme, L. Eckey, T. Moore, N. Narula, T. Ruffing, and A. Zohar, “Responsible vulnerability disclosure in cryptocurrencies”, *Commun. ACM*, vol. 63, no. 10, pp. 62–71, Sep. 2020, ISSN: 0001-0782.
- [21] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching word vectors with subword information”, *arXiv preprint arXiv:1607.04606*, 2016.
- [22] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, “Sailfish: Vetting smart contract state-inconsistency bugs in seconds”, in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 161–178.
- [23] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, “Vandal: A scalable security analysis framework for smart contracts”, *arXiv preprint arXiv:1809.03981*, 2018.
- [24] R. Bunel, M. Hausknecht, J. Devlin, R. Singh, and P. Kohli, “Leveraging grammar and reinforcement learning for neural program synthesis”, *arXiv preprint arXiv:1805.04276*, 2018.
- [25] V. Buterin *et al.*, “A next-generation smart contract and decentralized application platform”, *white paper*, vol. 3, no. 37, 2014.
- [26] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, “When deep learning met code search”, in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.
- [27] E. Capital, “Electric capital developer report”, <https://www.developerreport.com/developer-report>, [Online; accessed: Jan-2023].
- [28] P. M. Caversaccio, “A historical collection of reentrancy attacks.” <https://github.com/pcaversaccio/reentrancy-attacks>, [Online; accessed: April-2023].
- [29] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, *et al.*, “Universal sentence encoder”, *arXiv preprint arXiv:1803.11175*, 2018.
- [30] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet”, *IEEE Transactions on Software Engineering*, 2021.

BIBLIOGRAPHY

- [31] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, “Defining smart contract defects on ethereum”, *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 327–345, 2020.
- [32] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, “Defectchecker: Automated smart contract defect detection by analyzing evm bytecode”, *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2189–2207, 2021.
- [33] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, “Evaluating large language models trained on code”, *arXiv:2107.03374*, 2021.
- [34] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, “Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses”, in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 227–239.
- [35] D. S. Contract, Ethereum address 0xaa3a2ae9f85a337070cc8895da292ac373c17851.
- [36] K. S. Contract, Ethereum address 0xa8d8feeb169eeaa13957300a8c502d574bda2114.
- [37] F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda, “Ethersolve: Computing an accurate control-flow graph from ethereum bytecode”, in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, IEEE, 2021, pp. 127–137.
- [38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding”, *arXiv preprint arXiv:1810.04805*, 2018.
- [39] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, “Hoppity: Learning graph transformations to detect and fix bugs in programs”, in *International Conference on Learning Representations (ICLR)*, 2020.
- [40] S. H. Ding, B. C. Fung, and P. Charland, “Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization”, in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019.

BIBLIOGRAPHY

- [41] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, “Empirical review of automated analysis tools on 47,587 ethereum smart contracts”, in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.
- [42] M. Eshghie, C. Artho, and D. Gurov, “Dynamic vulnerability detection on smart contracts using machine learning”, in *Evaluation and assessment in software engineering*, 2021, pp. 305–312.
- [43] Ethereum, “Etherscan: The ethereum blockchain explorer”, <https://etherscan.io/>, [Online; accessed: Jan-2023].
- [44] Etherscan, “Etherscan Contracts Verified.” <https://etherscan.io/contractsVerified?filter=audit>, [Online; accessed February-2023].
- [45] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts”, in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, 2019, pp. 8–15.
- [46] J. F. Ferreira, P. Cruz, T. Durieux, and 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.
- [47] C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State, “The eye of horus: Spotting and analyzing attacks on ethereum smart contracts”, in *International Conference on Financial Cryptography and Data Security*, Springer, 2021, pp. 33–52.
- [48] C. Ferreira Torres, H. Jonker, and R. State, “Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts”, in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 115–128.
- [49] Z. Gao, “When deep learning meets smart contracts”, in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1400–1402.

BIBLIOGRAPHY

- [50] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, “Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding”, in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2019, pp. 394–397.
- [51] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, “Checking smart contracts with structural code embedding”, *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2874–2891, 2020.
- [52] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with lstm”, *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.
- [53] S. M. Ghaffarian and H. R. Shahriari, “Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey”, *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.
- [54] A. Ghaleb and 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.
- [55] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural message passing for quantum chemistry”, in *International conference on machine learning*, PMLR, 2017, pp. 1263–1272.
- [56] Google, “Bigquery - ethereum dataset”, https://console.cloud.google.com/bigquery?project=dataset-316302&ws=!1m5!1m4!4m3!1sbigquery-public-data!2scrypto_ethereum!3scontracts, [Retrieved on March 31, 2022].
- [57] Google, “Bigquery bigquery-public-data.ethereum_blockchain”, <https://console.cloud.google.com/bigquery>, [Retrieved on 25 October 2022].
- [58] D. Grattarola and C. Alippi, “Graph neural networks in tensorflow and keras with spektral [application notes]”, *IEEE Computational Intelligence Magazine*, vol. 16, no. 1, pp. 99–106, 2021.

BIBLIOGRAPHY

- [59] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, “Echidna: Effective, usable, and fast fuzzing for smart contracts”, in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.
- [60] X. Gu, H. Zhang, and S. Kim, “Deep code search”, in *Proceedings of the 40th International Conference on Software Engineering*, 2018.
- [61] R. Guo, W. Chen, L. Zhang, G. Wang, and H. Chen, “Smart contract vulnerability detection model based on siamese network (scvsn): A case study of reentrancy vulnerability”, *Energies*, vol. 15, no. 24, p. 9642, 2022.
- [62] R. Gupta, M. M. Patel, A. Shukla, and S. Tanwar, “Deep learning-based malicious smart contract detection scheme for internet of things environment”, *Computers & Electrical Engineering*, vol. 97, p. 107583, 2022.
- [63] A. Hagberg, P. Swart, and D. S Chult, “Exploring network structure, dynamics, and function using networkx”, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.
- [64] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs”, *Advances in neural information processing systems*, vol. 30, 2017.
- [65] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, “The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches”, *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.
- [66] X. Hao, W. Ren, W. Zheng, and T. Zhu, “Scscan: A svm-based scanning system for vulnerabilities in blockchain smart contracts”, in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, IEEE, 2020, pp. 1598–1605.
- [67] K. Hara, T. Takahashi, M. Ishimaki, and K. Omote, “Machine-learning approach using solidity bytecode for smart-contract honeypot detection in the ethereum”, in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, 2021, pp. 652–659.

BIBLIOGRAPHY

- [68] S. Hochreiter and J. Schmidhuber, “Long short-term memory”, *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [69] H. Hu, Q. Bai, and Y. Xu, “Scsguard: Deep scam detection for ethereum smart contracts”, in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2022, pp. 1–6.
- [70] T. Hu, B. Li, Z. Pan, and C. Qian, “Detect defects of solidity smart contract based on the knowledge graph”, *IEEE Transactions on Reliability*, 2023.
- [71] T. Hu, J. Li, B. Li, and A. Storhaug, “Why smart contracts reported as vulnerable were not exploited?”, 2023.
- [72] S.-J. Hwang, S.-H. Choi, J. Shin, and Y.-H. Choi, “Codenet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection”, *IEEE Access*, vol. 10, pp. 32 595–32 607, 2022.
- [73] C. S. Inc., “Ganache.” <https://trufflesuite.com/ganache/>, [Online; accessed: April-2023].
- [74] M. Iyyer, V. Manjunatha, J. Boyd-Graber, and H. Daumé III, “Deep unordered composition rivals syntactic methods for text classification”, in *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)*, 2015, pp. 1681–1691.
- [75] S. Jeon, G. Lee, H. Kim, and S. S. Woo, “Smartcondetect: Highly accurate smart contract code vulnerability detection mechanism using bert”, in *KDD Workshop on Programming Language Processing*, 2021.
- [76] B. Jiang, Y. Liu, and 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.
- [77] L. Jin, Y. Cao, Y. Chen, D. Zhang, and S. Campanoni, “Exgen: Cross-platform, automated exploit generation for smart contract vulnerabilities”, *IEEE Transactions on Dependable and Secure Computing*, 2022.
- [78] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “Zeus: Analyzing safety of smart contracts.” In *Ndss*, 2018, pp. 1–12.

BIBLIOGRAPHY

- [79] J. C. King, “Symbolic execution and program testing”, *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [80] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks”, *arXiv preprint arXiv:1609.02907*, 2016.
- [81] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and 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.
- [82] J. Krupp and C. Rossow, “{Teether}: Gnawing at ethereum to automatically exploit smart contracts”, in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.
- [83] H. Lal and G. Pahwa, “Code review analysis of software system using machine learning techniques”, in *2017 11th International Conference on Intelligent Systems and Control (ISCO)*, IEEE, 2017, pp. 8–13.
- [84] A. LeClair, S. Haque, L. Wu, and C. McMillan, “Improved code summarization via a graph neural network”, in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 184–195.
- [85] N. Li, Y. Liu, L. Li, and Y. Wang, “Smart contract vulnerability detection based on deep and cross network”, in *2022 3rd International Conference on Computer Vision, Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA)*, IEEE, 2022, pp. 533–536.
- [86] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks”, *arXiv preprint arXiv:1511.05493*, 2015.
- [87] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “Sysevr: A framework for using deep learning to detect software vulnerabilities”, *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.
- [88] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection”, *arXiv preprint arXiv:1801.01681*, 2018.

BIBLIOGRAPHY

- [89] J.-W. Liao, T.-T. Tsai, C.-K. He, and 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.
- [90] L. Liu, W.-T. Tsai, M. Z. A. Bhuiyan, H. Peng, and M. Liu, “Blockchain-enabled fraud discovery through abnormal smart contract detection on ethereum”, *Future Generation Computer Systems*, vol. 128, pp. 158–166, 2022.
- [91] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, “Combining graph neural networks with expert knowledge for smart contract vulnerability detection”, *IEEE Transactions on Knowledge and Data Engineering*, 2021.
- [92] Y. Lou, Y. Zhang, and S. Chen, “Ponzi contracts detection based on improved convolutional neural network”, in *2020 IEEE International Conference on Services Computing (SCC)*, IEEE, 2020, pp. 353–360.
- [93] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter”, in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.
- [94] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, R. Baldoni, *et al.*, “Investigating graph embedding neural networks with unsupervised features extraction for binary analysis”, in *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019, pp. 1–11.
- [95] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski, “Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack”, *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.
- [96] F. Mi, Z. Wang, C. Zhao, J. Guo, F. Ahmed, and L. Khan, “Vsc1: Automating vulnerability detection in smart contracts with deep learning”, in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2021, pp. 1–9.
- [97] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space”, *arXiv preprint arXiv:1301.3781*, 2013.

BIBLIOGRAPHY

- [98] P. Momeni, Y. Wang, and R. Samavi, “Machine learning model for smart contracts security analysis”, in *2019 17th International Conference on Privacy, Security and Trust (PST)*, IEEE, 2019, pp. 1–6.
- [99] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts”, in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019, pp. 1186–1189.
- [100] B. Mueller, “Smashing ethereum smart contracts for fun and real profit”, *HITB SECCONF Amsterdam*, vol. 9, p. 54, 2018.
- [101] MythX, <https://swcregistry.io/docs/SWC-107>, Retrieved on 11 June 2023.
- [102] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system”, *Decentralized Business Review*, p. 21 260, 2008.
- [103] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, “Mando-guru: Vulnerability detection for smart contract source code by heterogeneous graph embeddings”, in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1736–1740.
- [104] H. H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan, and L. Jiang, “Mando: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities”, in *2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA)*, IEEE, 2022, pp. 1–10.
- [105] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale”, in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 653–663.
- [106] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn:

BIBLIOGRAPHY

- Machine learning in Python”, *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [107] D. Perez and B. Livshits, “Smart contract vulnerabilities: Vulnerable does not imply exploited”, in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.
- [108] A. V. Phan, M. Le Nguyen, and L. T. Bui, “Convolutional neural networks over control flow graphs for software defect prediction”, in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, IEEE, 2017, pp. 45–52.
- [109] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, “Towards automated reentrancy detection for smart contracts based on sequential models”, *IEEE Access*, vol. 8, pp. 19 685–19 695, 2020.
- [110] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, “Improving language understanding by generative pre-training”, 2018.
- [111] S. Registry, “Smart contract weakness classification and test cases”, <https://swcregistry.io/>, [Online; accessed: Jan-2023].
- [112] M. Rodler, W. Li, G. O. Karame, and L. Davi, “Sereum: Protecting existing smart contracts against re-entrancy attacks”, *arXiv preprint arXiv:1812.05934*, 2018.
- [113] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, “Automated vulnerability detection in source code using deep representation learning”, in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, IEEE, 2018, pp. 757–762.
- [114] C. Schneidewind, “Personal email”, May 2023.
- [115] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, “EThor: Practical and provably sound static analysis of ethereum smart contracts”, 2020. arXiv: 2005.06227. [Online]. Available: <https://arxiv.org/abs/2005.06227>.

BIBLIOGRAPHY

- [116] C. Schneidewind, I. Grishchenko, M. Scherer, and 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.
- [117] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks”, *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
- [118] I. Sergey and A. Hobor, “A concurrent perspective on smart contracts”, in *International Conference on Financial Cryptography and Data Security*, Springer, 2017, pp. 478–493.
- [119] S. Shakya, A. Mukherjee, R. Halder, A. Maiti, and A. Chaturvedi, “Smartmixmodel: Machine learning-based vulnerability detection of solidity smart contracts”, in *2022 IEEE international conference on blockchain (Blockchain)*, IEEE, 2022, pp. 37–44.
- [120] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, “A survey on machine learning techniques for source code analysis”, *arXiv preprint arXiv:2110.09610*, 2021.
- [121] D. Siegel, “Understanding the dao attack”, <https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/>, [Online; accessed: Jan-2023].
- [122] S. So, S. Hong, and H. Oh, “{Smartest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {model-guided} symbolic execution”, in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1361–1378.
- [123] S. So, M. Lee, J. Park, H. Lee, and H. Oh, “Verismart: A highly precise safety verifier for ethereum smart contracts”, in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1678–1694.
- [124] Solidity, “Security Considerations”, <https://docs.soliditylang.org/en/v0.8.20/security-considerations.html#reentrancy>, 2023.
- [125] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, and Y. Wang, “Assbert: Active and semi-supervised bert for smart contract vulnerability detection”, *Journal of Information Security and Applications*, vol. 73, p. 103 423, 2023.

BIBLIOGRAPHY

- [126] Y. Sun and L. Gu, “Attention-based machine learning model for smart contract vulnerability detection”, in *Journal of physics: conference series*, IOP Publishing, vol. 1820, 2021, p. 012 004.
- [127] X. Tang, Y. Du, A. Lai, Z. Zhang, and L. Shi, “Deep learning-based solution for smart contract vulnerabilities detection”, *Scientific Reports*, vol. 13, no. 1, p. 20 106, 2023.
- [128] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, “Towards safer smart contracts: A sequence learning approach to detecting security threats”, *arXiv preprint arXiv:1811.06632*, 2018.
- [129] P. Technologies, “A postmortem on the parity multi-sig library self-destruct”, <https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/>, [Online; accessed: Jan-2023].
- [130] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, “Attention-based graph neural network for semi-supervised learning”, *arXiv preprint arXiv:1803.03735*, 2018.
- [131] D. Tian, X. Jia, R. Ma, S. Liu, W. Liu, and C. Hu, “Bindeep: A deep learning approach to binary code similarity detection”, *Expert Systems with Applications*, vol. 168, p. 114 348, 2021.
- [132] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and 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.
- [133] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, “Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts”, in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2021, pp. 103–119.
- [134] C. F. Torres, J. Schütte, and R. State, “Osiris: Hunting for integer bugs in ethereum smart contracts”, in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 664–676.
- [135] C. F. Torres, M. Steichen, *et al.*, “The art of the scam: Demystifying honeypots in ethereum smart contracts”, in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1591–1607.

BIBLIOGRAPHY

- [136] P. Tsankov, A. Dan, D. Drachler-Cohen, A. Gervais, F. Buenzli, and 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.
- [137] D. Ucci, L. Aniello, and R. Baldoni, “Survey of machine learning techniques for malware analysis”, *Computers & Security*, vol. 81, 2019.
- [138] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need”, *Advances in neural information processing systems*, vol. 30, 2017.
- [139] B. Wang, H. Chu, P. Zhang, and H. Dong, “Smart contract vulnerability detection using code representation fusion”, in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2021, pp. 564–565.
- [140] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, “Contractward: Automated vulnerability detection models for ethereum smart contracts”, *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.
- [141] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, S. Y. Philip, and G. Xu, “Reinforcement-learning-guided source code summarization using hierarchical attention”, *IEEE Transactions on software Engineering*, vol. 48, no. 1, pp. 102–119, 2020.
- [142] Z. Wang, Q. Zheng, and Y. Sun, “Gvd-net: Graph embedding-based machine learning model for smart contract vulnerability detection”, in *2022 International Conference on Algorithms, Data Mining, and Information Technology (ADMIT)*, IEEE, 2022, pp. 99–103.
- [143] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, “Vudenc: Vulnerability detection with deep learning on a natural codebase for python”, *Information and Software Technology*, vol. 144, p. 106 809, 2022.
- [144] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection”, in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 87–98.

BIBLIOGRAPHY

- [145] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger”, *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [146] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, “Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques”, in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2021, pp. 378–389.
- [147] Z. Wu, S. Li, B. Wang, T. Liu, Y. Zhu, C. Zhu, and M. Hu, “Detecting vulnerabilities in ethereum smart contracts with deep learning”, in *2022 4th International Conference on Data Intelligence and Security (ICDIS)*, IEEE, 2022, pp. 55–60.
- [148] G. Xu, L. Liu, and Z. Zhou, “Reentrancy vulnerability detection of smart contract based on bidirectional sequential neural network with hierarchical attention mechanism”, in *2022 International Conference on Blockchain Technology and Information Security (ICBCTIS)*, IEEE, 2022, pp. 56–59.
- [149] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” *arXiv preprint arXiv:1810.00826*, 2018.
- [150] Y. Xu, G. Hu, L. You, and C. Cao, “A novel machine learning-based analysis model for smart contract vulnerability”, *Security and Communication Networks*, vol. 2021, pp. 1–12, 2021.
- [151] H. Yan, S. Luo, L. Pan, and Y. Zhang, “Han-bsvd: A hierarchical attention network for binary software vulnerability detection”, *Computers & Security*, vol. 108, p. 102 286, 2021.
- [152] C. S. Yashavant, S. Kumar, and A. Karkare, “Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities”, *arXiv preprint arXiv:2202.11409*, 2022.
- [153] J. You, Z. Ying, and J. Leskovec, “Design space for graph neural networks”, *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 009–17 021, 2020.

BIBLIOGRAPHY

- [154] X. Yu, H. Zhao, B. Hou, Z. Ying, and B. Wu, “Deescvhunter: A deep learning-based framework for smart contract vulnerability detection”, in *2021 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2021, pp. 1–8.
- [155] C. Zeng, C. Y. Zhou, S. K. Lv, P. He, and J. Huang, “Gcn2defect: Graph convolutional networks for smotetomek-based software defect prediction”, in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2021, pp. 69–79.
- [156] J. Zhang, L. Tu, J. Cai, X. Sun, B. Li, W. Chen, and Y. Wang, “Vulnerability detection for smart contract via backward bayesian active learning”, in *Applied Cryptography and Network Security Workshops: ACNS 2022 Satellite Workshops, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, SiMLA, Rome, Italy, June 20–23, 2022, Proceedings*, Springer, 2022, pp. 66–83.
- [157] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, “An end-to-end deep learning architecture for graph classification”, in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [158] Y. Zhang, S. Kang, W. Dai, S. Chen, and J. Zhu, “Code will speak: Early detection of ponzi smart contracts on ethereum”, in *2021 IEEE International Conference on Services Computing (SCC)*, IEEE, 2021, pp. 301–308.
- [159] Y. Zhang and D. Liu, “Toward vulnerability detection for ethereum smart contracts using graph-matching network”, *Future Internet*, vol. 14, no. 11, p. 326, 2022.
- [160] 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.
- [161] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks”, *Advances in neural information processing systems*, vol. 32, 2019.

BIBLIOGRAPHY

- [162] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, “Smart contract vulnerability detection using graph neural networks”, in *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, 2021, pp. 3283–3290.
- [163] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, “Smart contract development: Challenges and opportunities”, *IEEE Transactions on Software Engineering*, 2019.

Appendix A

Word vs. Sentence Embeddings

Word and sentence embeddings are two essential techniques used in natural language processing (NLP) for representing text data.

Word embeddings are a type of distributed representation of words in a language. They are essentially a way of representing words as numerical vectors that capture the semantic and syntactic meaning of the word in the context of a given text corpus. Word embeddings are typically generated using unsupervised learning techniques such as word2vec, GloVe, or FastText. These algorithms use large amounts of text data to learn the underlying relationships between words in the corpus, and then generate word vectors that can be used as features for downstream NLP tasks such as sentiment analysis, language modeling, or machine translation.

Sentence embeddings, on the other hand, are a way of representing entire sentences as numerical vectors. Unlike word embeddings, which represent individual words, sentence embeddings capture the semantic meaning of an entire sentence in a given context. There are different methods for generating sentence embeddings, including averaging the word embeddings of the sentence, using recurrent neural networks (RNNs) or convolutional neural networks (CNNs) to encode the sentence, or using pre-trained language models such as BERT, GPT-2, RoBERTa, or universal sentence encoder (USE).

One of the main advantages of using sentence embeddings over word embeddings is that they can capture the overall meaning and context of a sentence, including its syntax, structure, and discourse. This makes sentence embeddings particularly useful for tasks such as document classification, text clustering, or information retrieval, where the goal is to group or classify text documents based on their content.

In summary, both word and sentence embeddings are important techniques for

representing text data in NLP. Word embeddings capture the meaning of individual words, while sentence embeddings capture the meaning of entire sentences. Depending on the task and the nature of the text data, one or the other may be more appropriate or effective.

A.1 Word-level Embeddings

Word-level embeddings are crucial in several NLP tasks, including sentiment analysis, machine translation, and text classification. They enable machines to learn from textual data in a more efficient and effective manner, as compared to traditional methods that rely on hand-crafted features. By representing words as vectors, word embeddings capture the context and meaning of words, enabling machines to understand relationships between words.

Bytecode instructions as text data poses a challenge in machine learning, as models cannot directly interpret textual information and instead require numerical feature vectors. Binary vectors, such as one-hot encoding, are commonly used by mapping tokens to integer values. However, word embeddings provide a more effective alternative by using unsupervised machine learning algorithms like fastText [21] and word2vec [97] to learn semantic relationships between bytecode instructions. The resulting word vectors encode the meaning of the instructions and offer a more nuanced representation of the text data. To produce a single vector representing each basic block of instructions, an appropriate composition function, such as RNNs, LSTMs, or BiLSTMs, is necessary when using word embedding methods.

A.1.1 Recurrent Neural Networks (RNNs)

RNNs are powerful machine learning models adapted to sequence data and operate over sequences of vectors one by one. RNNs can use their internal state (memory) to process variable length sequences of inputs which makes them applicable to process variable length of bytecode instructions. RNNs, as shown in Figure A.1, combine the input vector x_i with previous hidden state h_{i-1} where $i \in \{1, 2, 3, \dots, m\}$ into one vector that has information on the current input and previous inputs, then it goes through the tanh activation function (tanh activation is used to help regulate the values flowing through the network to be always between -1 and 1.) to produce

APPENDIX A. WORD VS. SENTENCE EMBEDDINGS

a new hidden state (or representations) vector h_i in Equation A.1 for the next step x_{i+1} .

$$h_i = \tanh(W_h \cdot h_{i-1} + W_x \cdot x_i + b) \quad (\text{A.1})$$

where W represents the weight matrix, b represents the bias parameter and $\tanh(\cdot)$ represents the standard hyperbolic tangent function.

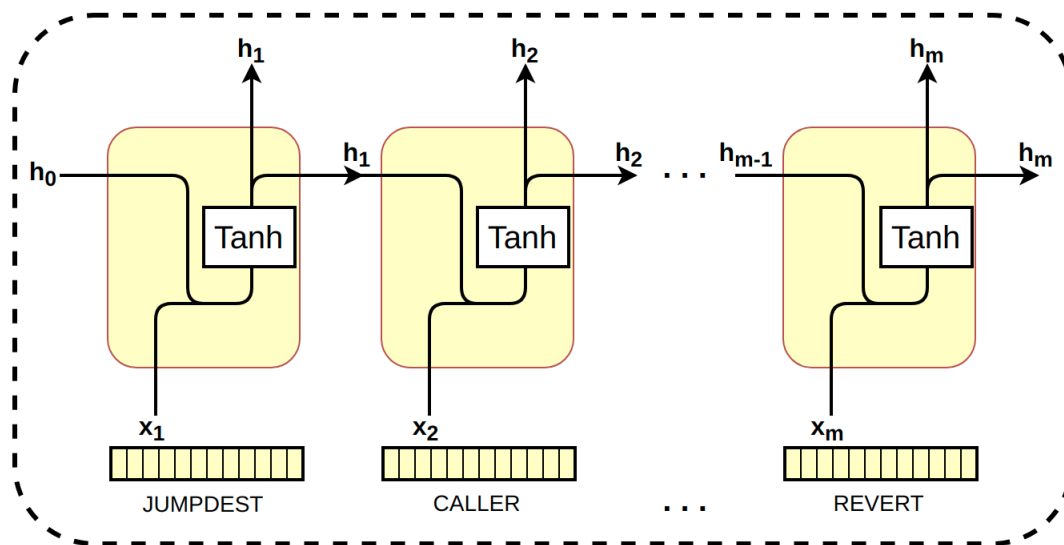


Figure A.1: Recurrent Neural Networks (RNNs).

RNNs use the hidden representations $\{ h_1, h_2, \dots, h_m \}$ for estimation and prediction. Simple RNNs are great when we are dealing with short-term dependencies, but fail to understand and remember the context behind long-term sequences. This is because RNNs suffer from the vanishing gradient problem that happens when gradient (values used to update a neural networks weights) shrinks as it back propagates through time of processing long sequences. RNNs layer that gets an extremely small gradient update doesn't contribute too much learning. So because some layers stop learning, RNNs can forget what it seen in longer sequences.

A.1.2 Long Short Term Memory Networks (LSTMs)

LSTMs [68, 52] have an edge over RNNs because of their property of selectively remembering patterns for long duration of time and the ability of learning long-term dependencies. LSTMs have a similar control flow as a recurrent neural networks, but

the repeating module has a different structure and an internal mechanisms called gates interacting in a very special way.

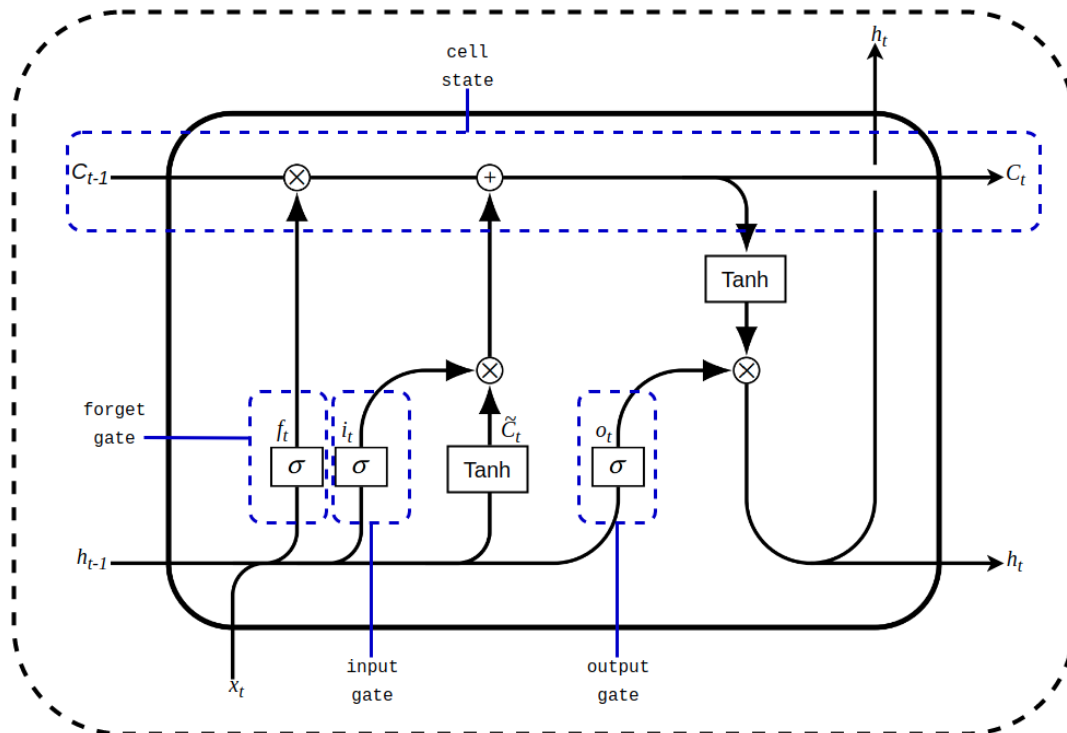


Figure A.2: Long Short Term Memory Networks (LSTMs).

The key concept of LSTMs, as shown in Figure A.2, is the cell state that works as the “memory” of the network. LSTMs first gate is called forget gate f_t in Equation A.2 that is to decide what information will be thrown away from the cell state or kept. Forget gate is passing the previous hidden state h_{t-1} and the current input x_t through sigmoid function, and outputs a number between 0 and 1 for each number in the cell state C_{t-1} . The closer to 0 means “completely get rid of this”, and the closer to 1 means “completely keep this”.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (\text{A.2})$$

LSTMs input gate i_t in Equation A.3 decides which values will be updated, and \tilde{C}_t in Equation A.4 creates a vector of new candidate values that could be added to the cell state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (\text{A.3})$$

APPENDIX A. WORD VS. SENTENCE EMBEDDINGS

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (\text{A.4})$$

In order to update the cell state C_t in Equation A.5, the old state is multiplied by f_t to forget information that forget gate decided to forget earlier. Then it's added to the new candidate values of \tilde{C}_t multiplied by the input gate i_t value.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \quad (\text{A.5})$$

Finally, the output gate o_t in Equation A.6 decides what the next hidden state should be in Equation A.7. The new cell state and the new hidden is then carried over to the next time step.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (\text{A.6})$$

$$h_t = o_t * \tanh(C_t) \quad (\text{A.7})$$

LSTMs have the ability to remove or add information to the cell state using gates. LSTMs gates can learn which data in a sequence is important to keep or throw away to keep the important features in long sequences and use them to make prediction.

A.1.3 Applying the word-level embeddings during research

In summary, word-level embeddings play a critical role in advancing the field of NLP and enabling machines to understand and process text more effectively. Their importance lies in their ability to represent words and sentences in a meaningful and efficient manner, allowing machines to learn from textual data and perform various NLP tasks with high accuracy and efficiency.

Our dataset consists of roughly 714 million instructions. To learn vector representations of bytecode instructions, we applied word embedding methods, including fastText and word2vec. We found fastText performed better than word2vec, reducing the loss to 2.3% compared to 30% in word2vec. Both methods require an appropriate composition function, we used a 2-layer of Bidirectional Long Short-Term Memory (BiLSTM) [117] to aggregate all instructions of each CFG node n_i into a single vector \vec{x}_i , as shown in Figure A.3. The BiLSTM model consists of two LSTMs, one taking input in a forward direction and the other in a backward direction. It can

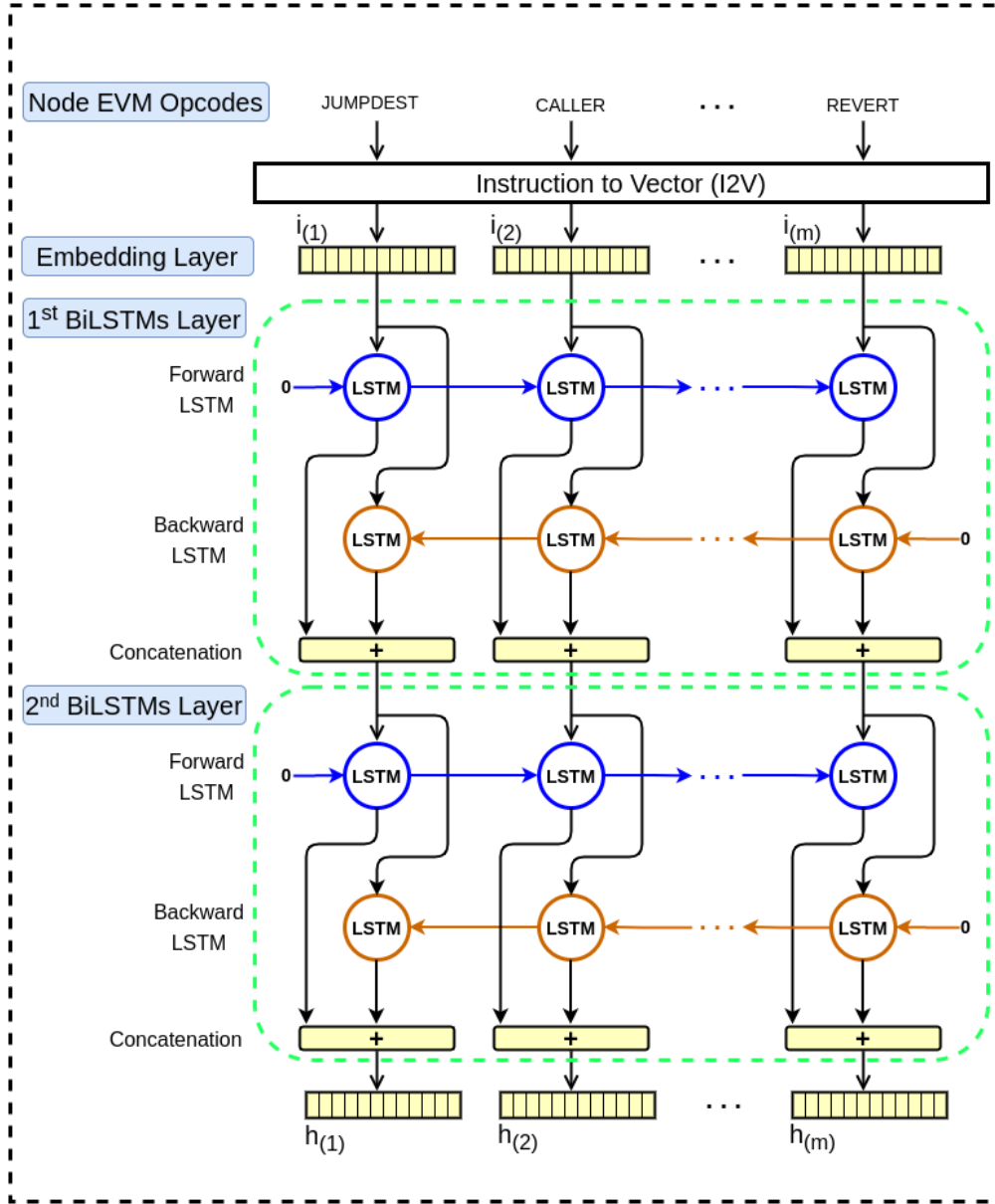


Figure A.3: Node Instructions Composition using Bidirectional Long Short-Term Memory.

learn and achieve high performance and accuracy on sequential learning tasks. The model is trained end-to-end, taking input of all instruction embedding vectors in order $(\vec{i}_1, \dots, \vec{i}_m)$, generating m outputs and m hidden states $(\vec{h}_{(1)}, \dots, \vec{h}_{(m)})$. The final feature vector \vec{x}_i for each node is obtained from the last hidden state $\vec{h}_{(m)}$. It took 6.22 seconds per contract to learn the representation of bytecode instructions using fastText and generate the final node feature vector using BiLSTM.

A.2 Sentence-level Embeddings

One popular method for generating sentence-level embeddings is to use pre-trained models such as BERT (Bidirectional Encoder Representations from Transformers) [38], GPT (Generative Pretrained Transformer) [110] or USE (Universal Sentence Encoder) [29]. These models are trained on large amounts of text data and can capture the contextual relationships between words in a sentence, allowing them to generate high-quality sentence embeddings.

The Universal Sentence Encoder (USE) is a pre-trained model developed by Google that generates high-quality sentence-level embeddings. These embeddings can be used to represent the meaning of a sentence in a high-dimensional vector space, allowing for efficient similarity calculations and comparisons between sentences.

There are two variants of the universal sentence encoder models: a) *Transformer Encoder* makes use of the transformer architecture [138], it consists of 6 stacked transformer layers, each layer has a self-attention module followed by a feed-forward network, it targets high accuracy at the expense of computing time and memory usage scaling dramatically with the length of the sentence. b) *Deep Averaging Network(DAN)* [74] is computationally less expensive with slightly reduced accuracy, DAN is much simpler where the embeddings for word and bi-grams present in a sentence are averaged together then passed through 4-layer feed-forward deep DNN to produce 512-dimensional sentence embedding as output, the embeddings for word and bi-grams are learned during training. The main advantage of the DAN encoder is that the computation time is linear in the length of the input sequence,

In our proposed architecture, DLVA, we utilized the Universal Sentence Encoder (USE), which has demonstrated superior performance compared to *fastText* followed by BiLSTMs. We first use *fastText* to learn the representation of EVM instructions, and then employ BiLSTMs to aggregate the instructions of each node, taking approximately 6.22 seconds per contract. Alternatively, by utilizing USE, we can transform these instructions in just 0.3 seconds per contract.

Publications during PhD Study

- [1] T. Abdelaziz and A. Hobor, “Smart learning to find dumb contracts”, in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 1775–1792, ISBN: 978-1-939133-37-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity23/presentation/abdelaziz>.
- [2] T. Abdelaziz and A. Hobor, “Smart learning to find dumb contracts (extended version)”, in *arXiv.org*, 2023. [Online]. Available: <https://arxiv.org/abs/2304.10726>.
- [3] T. Abdelaziz and A. Hobor, “Usenix’23 artifact appendix: Smart learning to find dumb contracts”, in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, ISBN: 978-1-939133-37-3. [Online]. Available: <https://www.usenix.org/system/files/usenixsecurity23-appendix-abdelaziz.pdf>.
- [4] T. Abdelaziz and A. Hobor, “Schooling to exploit foolish contracts”, in *2023 Fifth International Conference on Blockchain Computing and Applications (BCCA)*, 2023, pp. 388–395. [Online]. Available: <https://ieeexplore.ieee.org/document/10338924>.