A Novel Vulnerable Smart Contracts Profiling Method Based on Advanced Genetic Algorithm Using Penalty Fitness Function

Sepideh HajiHosseinKhani

A Thesis Submitted to the Faculty of Graduate Studies
In Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science

Graduate Program in Electrical Engineering and Computer Science

Supervisor: Arash Habibi Lashkari, Ph.D.

York University
Toronto, Ontario

July 25, 2024

Acknowledgement

I would like to thank my family for all their support throughout my Master's journey. I am also deeply grateful to my supervisor, Prof. Arash Habibi Lashkari, for his invaluable help and advice.

Abstract

With the advent of blockchain networks, there has been a transition from traditional contracts to Smart Contracts (SCs), which are crucial for maintaining trust within these networks. Previous methods for analyzing SCs vulnerabilities typically suffer from a lack of accuracy and effectiveness. Many of them, such as rule-based methods, machine learning techniques, and Neural Networks, also struggled to detect complex vulnerabilities due to limited data availability.

This study introduces a novel approach to detecting, identifying, and profiling SCs vulnerabilities, comprising two key components: a new analyzer named BCCC-SCsVulLyzer and an advanced Genetic Algorithm (GA) profiling method. The BCCC-SCsVulLyzer extracts 240 features across different categories, while the enhanced GA, explicitly designed for profiling SCs, employs techniques such as Penalty Fitness Function, Retention of Elites, and Adaptive Mutation Rate to create a detailed profile for each vulnerability.

Furthermore, due to the lack of comprehensive validation and evaluation datasets with sufficient samples and diverse vulnerabilities, this work introduces a new dataset named BCCC-SCsVul-2024. This dataset consists of 111,897 Solidity source code samples, ensuring the practical validation of the proposed approach.

Additionally, three types of taxonomies are established, covering SCs literature review, profiling techniques, and feature extraction. These taxonomies offer a systematic classification and analysis of information, enhancing the efficiency of the proposed profiling technique.

Our proposed approach demonstrated superior capabilities with higher precision and accuracy through rigorous testing and experimentation. It not only showed excellent results for evaluation parameters but also proved highly efficient in terms of time and space complexity. Moreover, the concept of the profiling technique makes our model highly transparent and explainable. These promising results highlight the potential of GA-based profiling to improve the detection and identification of SCs vulnerabilities, contributing to enhanced security in blockchain networks.

Contents

List of Tables

List of Figures

1 Introduction

Blockchain technology has revolutionized how we conceive of and implement digital transactions, enhancing the security and transparency of data exchange across decentralized networks [39, 40, 41]. Blockchain's utility has evolved beyond cryptocurrency, mainly originating from its original application, Bitcoin, to include finance, supply chain management, and other fields [40, 42]. Blockchain is a distributed ledger technology where data is stored in linked and secured blocks using cryptographic principles [43, 44]. This architecture assures that information recorded into the ledger is immutable and publicly available, depending on the network's configuration [45, 46, 47].

One of the most exciting applications of blockchain technology is the development of SC [4, 48]. With their terms directly written into lines of code, these contracts are self-executing and eliminate the need for intermediaries. The concept, first proposed by Nick Szabo [49] in the 1990s, has come to fruition with blockchain platforms like Ethereum. SCs ensure that all parties know the outcome without any intermediary's involvement [50]. Their deterministic nature means they execute predictably, without the possibility of downtime, censorship, fraud, or third-party interference [45].

The benefits of SCs are multifaceted. They reduce the need for trust among parties, decrease transaction costs by eliminating intermediaries, and increase the speed of transaction processing [51, 52]. Furthermore, in domains such as supply chain management, they enhance traceability and accountability, enabling a transparent view of product journeys from manufacturer to consumer. In finance, they introduce efficiencies in payments and settlements at a fraction of the time and cost of traditional banking systems [53, 54, 55].

Despite their advantages, deploying SCs has challenges [56, 57]. One of the most critical concerns is the security vulnerabilities inherent in deploying and executing SCs. Since they are code-based, SCs can contain vulnerabilities that malicious actors could exploit [58, 59]. The immutable nature of SC means that once an SC is deployed, it cannot be altered, making any vulnerabilities permanent and potentially leading to significant financial losses or operational disruptions [60, 61].

As underscored by past research, SC vulnerability detection presents numerous challenges. These challenges include imperfect detection due to false positives and negatives [4, 48], stability [50], limited coverage of detecting vulnerable SCs [40], and dependency on the availability of source code [40]. In addition, inadequate consideration of evolving threats, diverse vulnerabilities [62], and the complexity and usability challenges associated with specific detection techniques [48] are also noteworthy.

This study introduces a novel profiling model designed to address the issues of false positives, stability, dependency on source code availability, and complexity in detecting SCs vulnerabilities. Furthermore, the proposed GA profiling stands out in handling complex structures compared to other statistical, graph-based, and fuzzy profiling techniques. Drawing inspiration from natural selection and genetics enables developers to simulate program behavior identify performance bottlenecks and optimize code quickly. It provides precise data on execution time at every function, method, and line of code level. The accompanying feature selection method identifies relevant data attributes within SCs while discarding irrelevant ones, thereby boosting the genetic algorithm's efficiency and vulnerability detection accuracy. This integrated approach promises to significantly improve the speed and precision of vulnerability detection in SCs, enhancing the security of

SCs by profiling vulnerabilities.
Furthermore, this study creates a new dataset, namely BCCC-SCsVul-2024, for experiments and evaluates an innovative profiling approach. The BCCC-SCsVul-2024 dataset boasts a substantial sample size of 111,897 instances of Solidity SCs, encompassing a wide range of potential vulnerabilities. The findings from experiments include metrics such as precision, recall, accuracy, F1-score, execution time, macro average, and weighted average. These metrics highlight the effectiveness of the proposed GA techniques and emphasize the advantages of the proposed method for superior performance in SC analysis.
Our contributions in this study are multi-faceted, providing significant advancements in the realm of detecting and profiling vulnerabilities in SCs:

- **CONT1:** To design and implement an enhanced GA tailored for profiling, conducting experiments for effective feature selection from SCs to detect vulnerabilities, and profiling to improve the performance of SC analysis.
- **CONT2:** To propose a chronological overview of work development taxonomy, drawing from previous works, incorporating year-by-year evolution analysis detection methods to understand the subject matter comprehensively.
- **CONT3:** To propose a taxonomy of previous models, categorizing them by their approaches and providing detailed descriptions of each category's characteristics.
- **CONT4:** To propose a new feature taxonomy, showcasing extracted features from SCs, aiding in the creation of a more detailed and effective profile.
- **CONT5:** To propose a new profiling taxonomy, visualizing previous profiling techniques and categories, to create a comprehensive and structured framework.
- **CONT6:** To create a new SCs vulnerability Dataset, BCCC-SCsVul-2024, by combining Smart Bugs, Ethereum SCs (ESCs), slither-audited-smart-contracts, and the SmartScan-Dataset. The primary goal is to test and evaluate the proposed model on a substantial dataset comprising 111,897 samples.
- **CONT7:** To design and implement a new SCs analyzer, BCCC-SCsVulLyzer (V2.0), for experiments demonstrating effective feature extraction and optimization techniques, improving SC analysis performance.
- **CONT8:** To conduct a comprehensive analysis of smart contract vulnerabilities and propose two common vulnerability profiles.

The structure of the remainder of this thesis is as follows: Chapter 2 delves into the existing literature within this research field and identifies the limitations inherent in current solutions. Chapter 3 delves into the background of feature extraction within SCs and presents a detailed account of the profiling background. Furthermore, it provides an in-depth discussion of our proposed model. Chapter 4 outlines the extensive experiments, results, and implementation of our model. Chapter 5 thoroughly analyzes and discusses the research questions. Finally, Chapter 6 presents our conclusions and suggests directions for future work.

2 Literature Review

This chapter provides a comprehensive review of existing research in SC vulnerability detection, spanning from 2016 to 2023. Examining studies over the past eight years chronologically aims to illustrate how understanding and methodologies have progressed and how newer research builds upon or challenges earlier findings. This review synthesizes critical studies and critically examines the limitations and shortcomings in existing literature, paving the way for future research directions.

2.1 Previous Works

The initial research in SC vulnerabilities and security flaws emerged in 2016 with a pivotal study by Luu *et al.* [1] in 2016 initiated a deep dive into the security of Ethereum's SC system, which manages substantial amounts of virtual currency. This study describes the blockchain structure common in well-known cryptocurrencies such as Bitcoin and Ethereum. Each block contains a collection of transactions in this structure. Their investigation revealed previously unknown security flaws that posed potential profit-making exploits, highlighting a gap in comprehending Ethereum's distributed semantics.

To mitigate these vulnerabilities, Luu *et al.* [1] proposed enhancements to the operational semantics of Ethereum to strengthen contract security. A significant contribution of this study was the development of the Oyente tool, a symbolic execution instrument. Upon evaluating 19,366 Ethereum contracts using Oyente, it flagged 8,833 contracts as vulnerable.

Oyente took two main inputs: the bytecode of an Ethereum contract for analysis and the current Ethereum global state. It checked for any security issues in the contract, such as time-of-day dependence or mishandled exceptions, and identified problematic paths. Additionally, it generated a CFG of the contract's bytecode. Oyente interpreted the bytecode, which is accessible on the blockchain, using the EVM instruction set. EVM is a decentralized computing platform that executes smart contracts on the Ethereum blockchain, ensuring that transactions and computations are carried out exactly as intended. This allowed for accurate mapping of instructions to constraints. The Ethereum global state provided contract variables' initial or current values, aiding in more precise analysis. Other variables, like message call data, were considered symbolic input values. Oyente was designed modularly with four key components: CFGBuilder, Explorer, CoreAnalysis, and Validator. CFGBuilder created the CFG, where nodes were basic execution blocks, and edges were the execution jumps between these blocks. Explorer, the main module, performed the symbolic execution of the contract. The architecture of the Oyente tool is depicted in Fig. 1.

Continuing the exploration of vulnerabilities in SCs, Chen *et al.* [63] in 2017 focused on the issue of under-optimized SCs on Ethereum, a leading blockchain platform. In this environment, "gas" (measured in Ether, akin to Bitcoin) was essential for compensating miners for their computational efforts in executing SCs. A critical issue identified was that under-optimized contracts used more gas than necessary, resulting in excessive charges for creators or users. This paper pioneered the investigation of Solidity, Ethereum's preferred programming language, revealing its inability to optimize specific gas-intensive programming patterns. The authors categorized seven such patterns into two distinct groups.

They introduced GASPER, an innovative tool for automatically detecting gas-costly patterns by analyzing

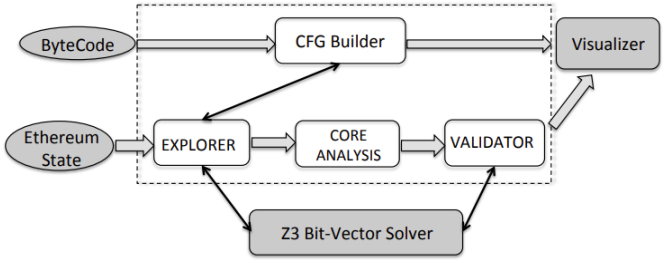


Figure 1: Overview of Oyente's Architecture [1].

SCs' bytecodes to address the inability to optimize specific gas-intensive programming patterns. GASPER performed symbolic execution on bytecode, covering all accessible code blocks. These blocks were sequences of code with only one entry and exit point. First, GASPER disassembled the SC's bytecode using Ethereum's disasm tool. Then, it created the CFG, which was updated as new control flows were discovered during the execution process. Starting from the CFG's root node, GASPER traversed the graph. When it encountered a conditional jump, it used the Z3 solver to determine which paths—true or false—were possible. If both paths were viable, GASPER chose one based on a depth-first search strategy.

In their initial analysis of 4,240 real SCs, they found that a significant majority—93.5%, 90.1%, and 80%—suffered from three widespread gas-inefficient patterns. This underscores the common issue of gas inefficiency in SC programming.

In 2018, research on SC vulnerability detection experienced a significant diversification. Several researchers employed familiar methodologies, such as interacting with and extracting features from the source code by converting it into more commonly known programming languages, as evidenced in works like [64, 65, 8, 66, 67]. Meanwhile, others, including [68, 4, 69, 70], ventured into developing novel approaches.

Initially, Liu *et al.* [2] in 2018 concentrated on Re-entrancy bug, a prominent security challenge in SCs on platforms like Ethereum. This issue gained notoriety following the DAO attack, which caused a loss of $60 million. The team introduced ReGuard, a fuzzing-based analyzer for detecting re-entrancy bugs in Ethereum SCs. ReGuard conducted fuzz testing by generating random transactions and analyzing the resulting runtime traces to pinpoint re-entrancy vulnerabilities dynamically.

For better visualization, Fig. 2 depicts the design of ReGuard. At a basic level, ReGuard processed the code of a SC, accepting both source and binary code as input. Its output was a bug report listing all the re-entrancy bugs in the SC. For every bug it detected, ReGuard pointed out where it was in the source code and provided an example of a transaction that could trigger the bug. SC execution greatly depended on transactions. Different transactions could result in various contract states where re-entrancy bugs might appear. To thoroughly examine the SC's state space, ReGuard used advanced fuzzing engines. These engines created random transactions to explore diverse execution scenarios.

They conducted an initial test of ReGuard on five altered SCs from Etherscan. Oyente was used to compare the differences between fuzzing-based approaches (ReGuard) and symbolic execution methods (Oyente). Their focus was on identifying re-entrancy bugs.

Similarly, Tikhomirov *et al.* [64] addressed the critical need for secure programming practices in Ethereum

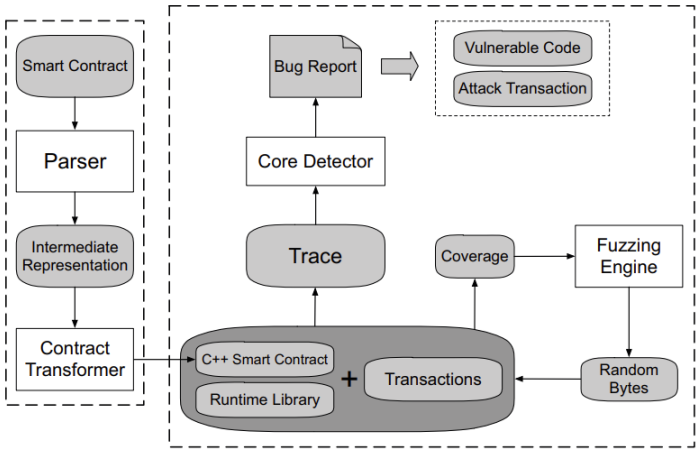


Figure 2: Overview of ReGuard's Architecture [2].

SC development, spurred by the "The DAO" hack in 2016. They developed SmartCheck, an innovative static analysis tool for Solidity, Ethereum's primary SC language. SmartCheck converted Solidity code into an XML-based intermediate representation and employed XPath patterns to detect vulnerabilities, providing an extensive classification of code issues. Despite potential limitations in identifying some complex bugs, SmartCheck's utility was affirmed by its performance in evaluating a substantial dataset of real-world contracts. It established its indispensability for SC developers through comparative manual audits on three contracts.

They provided a side-by-side evaluation of SC analysis tools—Oyente, Securify, and SmartCheck—over a selection of projects. The count of true positives, false positives, and false negatives pinpointed by each application was recorded, along with their respective False Discovery Rates (FDR) and False Negative Rates (FNR), all in percentage. This comprehensive data allowed for a detailed analysis of the performance of each application in terms of accuracy and reliability. It enabled a detailed review of how each tool performed regarding accuracy and reliability.

They provided detailed insights, including individual analyses of the "Genesis Vision," "Hive," and "Populous" projects and a comprehensive summary in the 'Overall' section. While SmartCheck displayed a high FDR of 69%, this rate was not especially alarming in this context. A FNR of 47% was acceptable, given that numerous SC vulnerabilities were specific to business logic and often escaped automated detection. Most false negatives detected by SmartCheck were identified through manual methods, rather than by other automated tools.

Continuing in 2018, the landscape of SC vulnerability detection witnessed a pivotal shift towards leveraging bytecode for more comprehensive feature extraction and analysis. Significant contributions from various researchers marked this trend. Grishchenko *et al.* [68] introduced EtherTrust, a novel tool for the automated and sound static analysis of EVM bytecode. EtherTrust addressed the critical need for secure SCs by offering a scalable solution to the security challenges on the Ethereum platform. This was especially emphasized in response to high-profile incidents like the DAO and Parity attacks.

EtherTrust streamlined the process of ensuring single-entrancy and managing dependency properties for SCs.

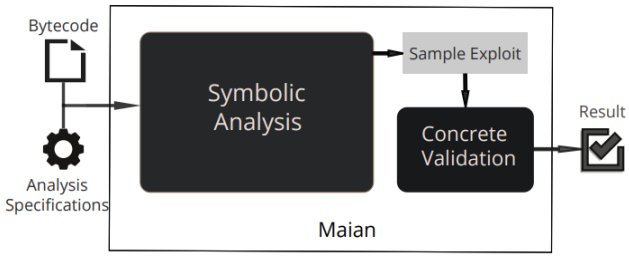


Figure 3: Overview of MAIAN's Architecture [3].

To simplify single-entrancy, EtherTrust used a method called 'call unreachability,' which meant a contract was designed to avoid calling itself during execution, thus preventing re-entries. This was based on the rule that a contract must not initiate an internal call after starting. Since all contracts started from a fresh state, the verification focused only on those initial executions.

For dependency properties, EtherTrust used a reachability analysis. It monitored how data moved through the contract and identified vital points where specific values influenced its path. If a conditional action was based on one of these critical values, EtherTrust noted all possible outcomes from that condition. This helped determine whether any particular value could affect the contract's call behavior during its lifecycle. EtherTrust's focus on reachability properties and EVM bytecode's complete, mechanized semantics marked a significant advancement in SC security analysis.

In another study, Nikolic *et al.* [3] introduced MAIAN, which represented a significant advancement in the systematic examination of trace vulnerabilities in SCs. This tool began its analysis with the contract's bytecode and an initial concrete context that reflected the state of the blockchain. Using interprocedural symbolic analysis, MAIAN examined transaction sequences and their properties, particularly those involving multiple interactions with a contract. It employed static symbolic analysis to extend its reasoning across contract calls and over several blockchain blocks. MAIAN symbolized the input variables from transaction fields and blockchain parameters, denoting a set of all possible values each variable could assume. Furthermore, it symbolically formulated the relationships between other contract variables, resulting in expressions that revealed their interdependencies. Fig. 3 presents a detailed view of MAIAN's structural design.

MAIAN's symbolic execution engine identified 34,200 contracts as potentially vulnerable. Concrete validation and manual checks confirmed that 97% of the extravagant and suicidal contracts and 69% of greedy contracts were true positives. This highlighted the importance of analyzing contract bytecode over Solidity source code, as only 1% of contracts had accessible source code. Among those flagged, just 181 had verified source codes on Etherscan, which was a small percentage across the three categories.

The study also discovered that nearly 4,905 Ether, valued at about 5.9 million US dollars then, could have been taken from prodigal and suicidal contracts before a specific block height. Additionally, 6,239 Ether (7.5 million US dollars) was trapped in posthumous contracts on the blockchain, with 313 Ether (379,940 US dollars) sent to these contracts after they were terminated.

Torres *et al.* [4] introduced Osiris, a framework designed to detect integer bugs in Ethereum SCs by combining symbolic execution and taint analysis. Fig. 4 illustrates the structure of Osiris. Osiris accepted either the SC's bytecode or the Solidity source code, which was internally converted to EVM bytecode. It determined

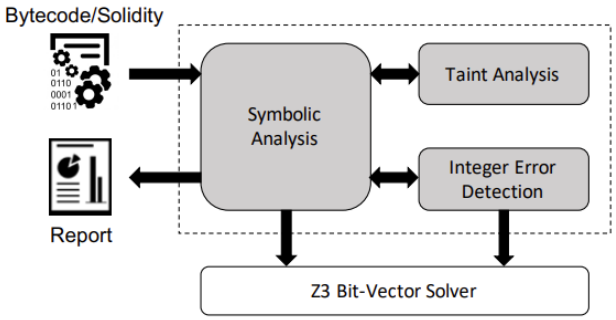


Figure 4: Overview of Osiris's Architecture [4].

whether a contract was affected by integer-related issues, such as overflows, underflows, or truncations.
Osiris comprised three key components: symbolic analysis, taint analysis, and integer error detection. The symbolic analysis part created a CFG and carried out symbolic execution along the contract's various paths. It then fed the outcomes of each instruction to both the taint analysis and integer error detection components. The taint analysis chapter introduced, propagated, and checked taints within the contract's stack, memory, and storage. Meanwhile, the integer error detection segment assessed each executed instruction to identify potential integer bugs.
They [4] evaluated the effectiveness of Osiris in identifying integer overflows and underflows compared to Zeus which is a framework to verify the correctness and validate the fairness of smart contracts [50]. Osiris tended to classify most contracts as "safe," meaning no overflow or underflow was detected. In contrast, Zeus often labeled them as "unsafe," indicating the detection of either overflow or underflow. This discrepancy arose because Osiris targeted overflows and underflows that could realistically be exploited by an attacker, resulting in fewer reported issues. In contrast, Zeus aimed for comprehensiveness in its reporting. Regarding handling errors, Zeus reported no result for 22 contracts, where "no result" indicated either an error or a timeout situation. Zeus experienced fewer timeouts compared to Osiris. However, Osiris consistently provided a result for every case it analyzed.
Rodler *et al.* [69] developed Sereum (Secure Ethereum) to protect deployed SCs against re-entrancy exploits. Sereum's backward-compatible protection efficiently detected and prevented re-entrancy attacks. It demonstrated its effectiveness with minimal false positives and runtime overhead. Sereum enhanced the EVM by adding two innovative components: a taint engine and an attack detector.
First, the taint engine employed dynamic taint-tracking, where labels were assigned to data at specific points. The movement and impact of this labeled data through the program's execution were monitored. Sereum was noted for being the first implementation of dynamic taint-tracking specifically for SCs. Second, the attack detector worked with the taint engine to identify potential signs of a re-entrancy attack during a transaction. It was integrated with the EVM's transaction manager and was designed to halt transactions immediately upon detection of an attack.
The experiment demonstrated that Sereum exhibited an impressively low false positive rate of just 0.06% for all rested transactions. The results included data on transactions that Sereum flagged as matching the pattern of a re-entrancy attack.

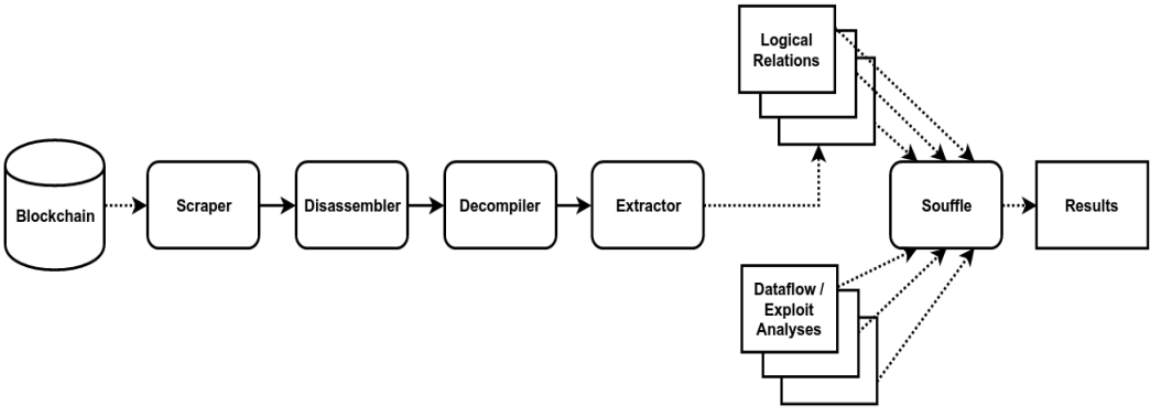


Figure 5: Soufflé Analysis Pipeline Overview [5].

Brent *et al.* [70] created Vandal, a security analysis framework that stood out for converting EVM bytecode into semantic logic relations. This allowed users to conduct security analyses using the Soufflé language, significantly enhancing the analysis of SC vulnerabilities. To understand the complex semantics of Ethereum's low-level bytecode, they developed a step-by-step analysis process, as illustrated in Fig. 5. This process simplified the conversion of bytecode into a format that was easier to analyze by dividing it into several distinct steps:

- Scraping: First, the Ethereum bytecode was collected from the blockchain using the Scraper tool.
- Disassembling: Next, the Disassembler broke down this bytecode.
- Decompiling: The Decompiler then transformed the bytecode into a register transfer language, effectively reconstructing the original SC's higher-level control and data flows.
- Extracting: After decompilation, the program, now in register-transfer language, was processed by the Extractor. This tool converted the program into logic relations, capturing the SC's semantics in a straightforward, tab-separated value format.
- Analyzing Security: Finally, Soufflé [5] used these logic relations to generate executable programs. These programs conducted a security analysis to detect potential vulnerabilities in the SC, reporting any found issues.

This pipeline made the complex task of detecting security vulnerabilities in Ethereum SCs more manageable and systematic.

Grech *et al.* [6] presented MadMax, an automated tool for detecting gas-focused vulnerabilities in Ethereum SCs. The MadMax analysis pipeline, extending from the Vandal decompiler to its final analysis output, is illustrated in Fig. 6. MadMax primarily utilized logic-based specifications to analyze Vandal's output. This analysis was structured in multiple layers, each aiming to infer progressively more complex aspects of the SC under examination. The MadMax analysis pipeline began with the Vandal decompiler and progressed to a comprehensive final analysis output. This process leveraged logic-based specifications to analyze the

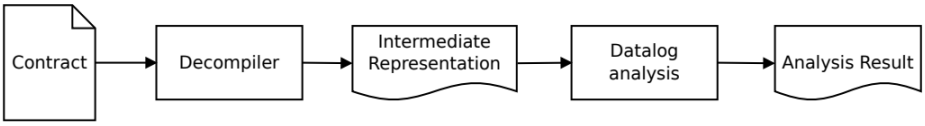


Figure 6: Overview of MadMax's Architecture [6].

Vandal output and was characterized by multiple layers, each designed to infer increasingly complex aspects of the SC.

The first step in MadMax's analysis involved deriving information about loops and data flow, which was crucial for expressing deeper semantics. This initial phase resulted in the generation of several relations, forming the foundation for subsequent analytical steps. While the Datalog rules for these relations were not provided, their implementation was standard, although not always straightforward. Following the initial phase, MadMax conducted a deeper analysis, focusing on memory and dynamic data structures. This step, necessary to model the EVM's dynamic data representation accurately, identified key concepts such as dynamic data structures, contracts with storage that increased upon re-entry, nested arrays, etc.

The analysis peaked in identifying potential gas-focused vulnerabilities, like loops with unbounded mass storage. This layered approach ensured a thorough examination and identification of vulnerabilities in SCs, contributing significantly to understanding their security and efficiency. MadMax efficiently analyzed all Ethereum SCs, identifying over $2.8 billion worth of vulnerable contracts. A manual inspection confirmed that 81% of the flagged contracts contained vulnerabilities, highlighting MadMax's precision in identifying and addressing gas-related security issues.

Albert *et al.* [71] discussed the importance of analyzing Ethereum bytecode. This is particularly crucial when the source code is unavailable or when analysis requires bytecode-level information, especially in cases where compiler optimizations might impact results. They introduced ETHIR, a framework for analyzing Ethereum bytecode to address these needs. ETHIR extended OYENTE [1], producing a rule-based representation of the bytecode, enabling high-level analyses, and bridging the gap between low-level bytecode and high-level insights.

When applied to EVM code, the Oyente [1] tool created blocks that encapsulated information necessary for constructing the CFG of that code. A limitation arose when the jump address within a block was variable, depending on the program's flow. Oyente typically recorded only the final jump address value, which sufficed for its style of symbolic execution. To overcome this limitation and reconstruct the entire CFG, modifications were made to Oyente's block structure to include all potential jump addresses.

In this enhanced version of Oyente [1], specific EVM instructions like SSTORE or SLOAD were marked with identifiers indicating the contract field they interacted with. For instance, an SSTORE operation acting on contract field 0 was labeled as "SSTORE 0". The same annotation logic applied to MSTORE and MLOAD instructions, where they were tagged with the memory addresses they worked with, transforming these addresses into variables whenever feasible.

However, in situations where the memory address wasn't statically determined (like in array accesses within loops with variable indices), the tool marked these instructions with a "?". Additionally, when Solidity code

was available, function names could be deduced from hash codes, enhancing the CFG with more context and clarity, as exemplified in Block 152 of the figure. This enhancement allowed for a more precise understanding of the CFG's continuation block.

Applying this method to a test example, the study successfully demonstrated the termination of six loops in the example and established linear bounds for these loops. Additionally, other SCs, along with their loop bounds inferred by SACO [72], were provided on their GitHub page. The study also noted that other high-level analyzers, which worked with intermediate forms like Integer transition systems or Horn clauses, could be adapted to work with the translated programs in their Resource-Bound Representation (RBR) format. These analyzers included AproVe [73], VeryMax [74], and CoFloCo [75].

Adding to these advancements, Tsankov *et al.* [65] developed Securify, a scalable, fully automated security analyzer for Ethereum SCs. Securify's approach of symbolic analysis of contract dependency graphs and assessment of compliance and violation patterns significantly enhanced the trustworthiness of Ethereum SCs. They established seven fundamental security properties about the EVM semantics. Given that the EVM is Turing-complete, it was impossible to check these properties accurately. Therefore, they approached this by defining patterns of compliance and violation in their language. These patterns were rough estimates of whether a property was being adhered to or violated.

Moreover, a match with a compliance pattern suggested that the property was being upheld, whereas a match with a violation pattern indicated the opposite. However, if neither pattern matched, it was unclear whether the property was being observed. They delved into each security property, explaining its importance, providing a formal definition, and then breaking it down into specific patterns of compliance and violation.

In their study, the researchers assessed the effectiveness of Securify in proving security properties (by matching a compliance pattern) and identifying violations (through matching a violation pattern) in real-world SCs. They conducted this evaluation by running Securify on all the SCs in their EVM dataset and tracked the occurrences of violations, warnings, and compliances as reported by Securify.

The results displayed a separate bar for each security property. Each bar was divided into three segments: (i) violations, indicating the percentage of instructions matching a violation pattern for that property; (ii) warnings, showing the percentage of instructions that did not match any pattern (neither violation nor compliance); and (iii) compliance, representing the percentage of instructions matching a compliance pattern. Notably, the sum of these three segments equaled 100%. Furthermore, they presented a further breakdown of these results. Similar to the previous figure, it showed a bar for each security property. However, the segment for warnings was further divided into true warnings and false warnings, in addition to the violation and compliance segments.

Similarly in 2018, another paradigm shift in research methodology emerged, focusing on the CFG version of code. This approach led to the development of innovative tools by several researchers [66, 67, 1, 6, 7], significantly enhancing the analysis and security of Ethereum SCs.

Norvill *et al.* [66] introduced E-EVM, a tool designed to emulate and visualize SC execution on the EVM. E-EVM operated by analyzing SC bytecode, providing in-depth insights into the contract's CFG, opcodes, and stack. A CFG is a graphical representation of all paths that might be traversed through a program during its execution. The tool's primary goal was to improve understanding of the EVM and assist in analyzing specific

SCs. It enabled users to examine code in each block, trace control flow paths, identify loops, and suggest optimizations, thereby functioning as both an analytical tool and a learning aid. E-EVM's effectiveness was demonstrated by achieving an average of 85.6% code coverage in testing.

The team conducted tests on their program by applying it to various contracts and monitored the code coverage attained. They calculated code coverage as a percentage for each contract using the formula: $\text{nodes}/\text{blocks} \times 100$. Here, 'nodes' refers to the number of nodes in the CFG, and 'blocks' refers to the total number of blocks in the contract. The results were presented in the format (Sym/Concrete) for the two versions of E-EVM, with the note that constant synthesis was only relevant to the Sym version.

Zhou *et al.* [67] tackled the challenge of auditing Ethereum SCs, particularly those without accessible source code. They introduced Erays, a reverse engineering tool for SCs on the Ethereum blockchain. Erays could convert bytecode into high-level pseudocode, thereby facilitating manual analysis. The analysis tools provided insights into the complexity of Ethereum contracts. A key finding was that most contracts were relatively small, with a median of 100 blocks and 15 instructions per block. Yet, there were exceptions, like one unusually large contract with 13,045 blocks, primarily filled with STOP instructions.

Beyond just the number of blocks, the study also examined contract complexity through cyclomatic complexity, a metric for measuring the number of linearly independent paths in a CFG. They presented a Cumulative Distribution Function (CDF) of McCabe complexity, showing that 79% of unique contracts had no function with complexity greater than 10, indicating simplicity in many contracts. However, there were still some highly complex contracts.

An evolution of code complexity over time was explored in [67]. It revealed that while contracts had been growing in size, with a significant increase in the number of blocks since late 2015, McCabe complexity hadn't seen a similar rise. This trend suggested improved coding practices over time. Notably, the increase in ERC20 Tokens, which were larger but not overly complex, might have influenced these trends.

Furthermore, Krupp *et al.* [7] addressed Ethereum's unique security challenges as a platform supporting Turing-complete SCs. They introduced teEther, a tool capable of automatically creating exploits for SCs using binary bytecode. Fig. 7 illustrates the main architecture of TEETHER.

The process started with the CFG-recovery module, which took EVM bytecode, disassembled it, and then reconstructed a CFG. Following this, the CFG was examined for both critical instructions and state-changing instructions. The path generation module then identified paths within the CFG that led to these instructions. Building on this, the constraint generation module used symbolic execution to develop a set of path constraints. The final step involved the exploit generation module, which resolved the combined constraints from both critical and state-changing paths to create an exploit. TeEther's large-scale analysis of 38,757 unique Ethereum contracts successfully found working exploits for 815 contracts, underscoring the critical need for robust security measures in the SC domain.

In 2018, a new wave of research employing diverse approaches and integrating various sources of information from source code led to the development of innovative tools for enhancing the security of Ethereum SCs. Notable contributions in this area included ContractFuzzer by Jiang *et al.* [8], SASC by Zhou *et al.* [9], and ZEUS by Kalra *et al.* [50].

ContractFuzzer [8] was an innovative fuzzer tool designed to assess the security vulnerabilities of Ethereum

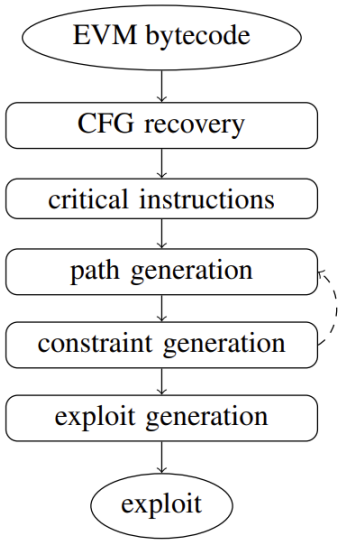


Figure 7: Overview of TEETHER's Architecture [7].

SCs. In the decentralized cryptocurrency sphere, where blockchain technology enabled peer-to-peer transactions without central oversight, the security of SCs was paramount. ContractFuzzer met this challenge by generating fuzzing inputs tailored to the ABI specifications, which is a set of conventions that dictate how different program modules or applications interact at the binary level of SCs. Additionally, it defined test oracles to detect vulnerabilities. ContractFuzzer is depicted in Fig. 8, which outlines its workflow through steps 0 to 5. This tool comprised both an offline EVM instrumentation tool and an online fuzzing tool.
They had also developed a web crawler to gather deployed SCs from the Etherscan website. This crawler collected contract creation codes, ABI interfaces, and constructor arguments of the contracts. These contracts were then deployed on their Ethereum testnet, serving dual purposes: as targets for fuzzing and as input for contract calls using their addresses. In step 0, the offline instrumentation process equipped the EVM to monitor SC execution, aiding in vulnerability analysis. The online fuzzing began at Step 1, where ContractFuzzer analyzed the ABI interface and bytecode of the contract under test. It extracted data types of ABI function arguments and the function signatures. Step 2 involved ABI signature analysis of all crawled contracts, indexing them based on the function signatures they supported, essential for testing contract interactions.
In Step 3, the tool generated valid fuzzing inputs adhering to the ABI specification and also mutated inputs. The indexed contracts from Step 2 were used to create inputs for ABIs requiring contract addresses. Step 4 initiated the fuzzing process, where the tool bombarded the ABI interfaces with these inputs through random function invocations. Finally, in Step 5, ContractFuzzer detected security vulnerabilities by analyzing execution logs from the fuzzing process. The fuzzing continued until the allotted testing time was exhausted. The campaign concluded once all targeted SCs had been fuzzed. Its effectiveness was proven through its application to 6,991 SCs, identifying over 459 vulnerabilities, including those in the DAO and Parity Wallet contracts, which had led to significant financial losses.
Zhou *et al.* [9] addressed the crucial need for ensuring the execution correctness of Ethereum SCs, which managed millions of dollars and were susceptible to asset theft attacks. They proposed a security assurance

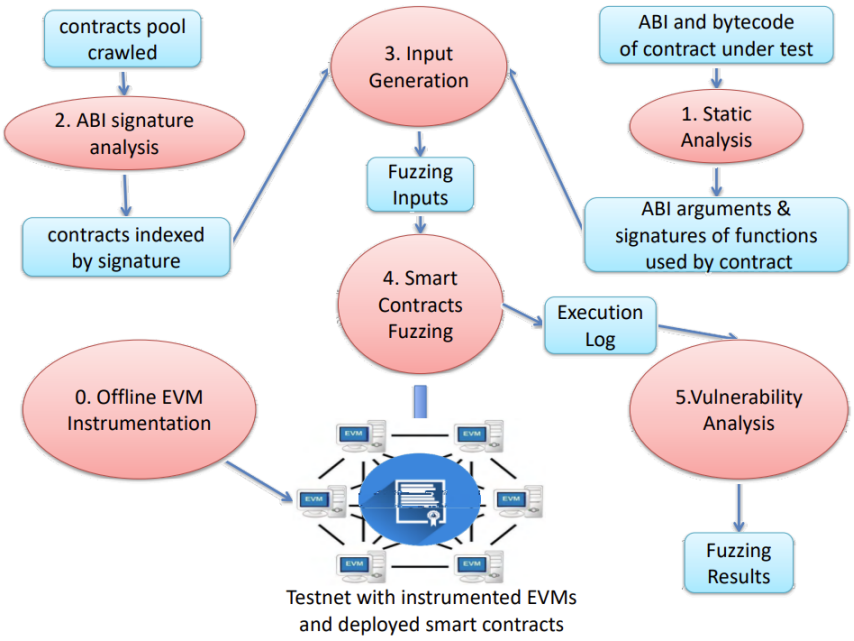


Figure 8: Overview of ContractFuzzer's Architecture [8].

method involving a syntax topological analysis of the SC invocation relationship, aiding developers in understanding their code structure. Building on Luu's research [1], they first incorporated detection rules for expanded logic risks, as previously mentioned, into the symbolic execution module. The aim was to pinpoint the detected logic risks to specific functions in a contract's source code. Fig. 9 illustrates the entire structure of their method.

Their approach dealt with the challenge that the opcodes generated from the bytecode of SCs lacked location information, making it difficult to directly map detected risk opcodes to the source code. On the other hand, opcodes compiled from the Solidity compiler (solc) did contain location info. However, they included some invalid opcodes, making them not entirely consistent with the bytecode-converted opcodes and unsuitable for direct symbolic execution. To resolve this, they developed a mapping mechanism involving a set of rules to identify invalid opcodes compiled from solc. This allowed them to add location information to the bytecode-converted opcodes, as detailed in Fig. 10. Finally, to locate risks to specific functions, they first gathered all location information of functions during the topological analysis stage. By combining this with the location info of risk opcodes, they could accurately locate detected risks to specific functions in the SC.

To validate their hypotheses, they analyzed the relationships between various factors and execution time. Due to the significant fluctuations in execution times between adjacent contracts, they employed the Moving Average technique to represent each contract's execution time more smoothly. This approach was referred to as Moving Average Execution Time (MAET). Their observations revealed that the execution time was nearly linearly dependent on the number of executed paths, blocks, or opcodes in the contracts. This linear relationship was essential in understanding how different elements within a contract affected its overall execution time.

Kalra *et al.* [50] introduced ZEUS in 2018, a comprehensive framework for verifying the correctness and

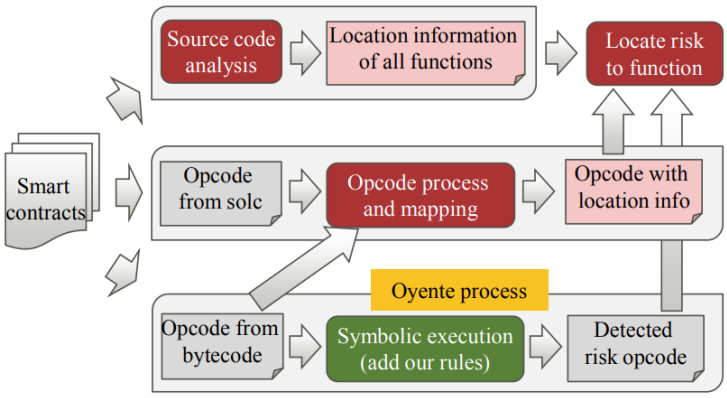


Figure 9: Logic risk detection and location structure in security assurance method [9].

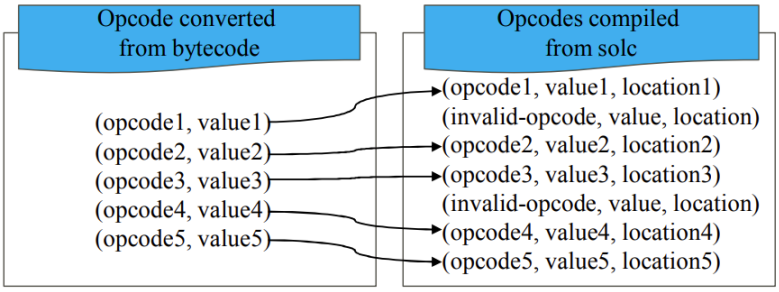


Figure 10: Opcode mapping mechanism in security assurance method [9].

validating fairness of SCs. Given the difficulty in patching bugs post-deployment, ZEUS's focus on correctness and fairness was critical, especially considering the substantial financial losses due to SC bugs in recent incidents. ZEUS employed abstract interpretation, symbolic model checking, and constrained horn clauses for safety verification.

ZEUS comprised three main components: (a) a policy builder, (b) a source code translator, and (c) a verifier. The process began with ZEUS taking a SC and a policy written in a specific specification language as input. This policy outlined the criteria that the SC needed to meet. ZEUS then conducted static analysis on the SC code, integrating the policy predicates as assert statements at the appropriate points in the program. Following this, the source code translator component of ZEUS came into play. It accurately converted the SC, now enhanced with policy assertions, into LLVM bytecode. The final step involved the verifier component of ZEUS, which was tasked with identifying any assertion violations. These violations were crucial as they indicated instances where the SC did not comply with the specified policy. This comprehensive approach ensured that SCs adhered to the set standards and policies.

In the study presented data on the frequency of duplicate contracts within their dataset. It was noted that less than 5% of the contracts had more than 10 duplicates. Further, they provided insights into the source Lines Of Code (LOC) in Solidity for unique contracts. The total source LOC exceeded 111,000, with an average (mean) of 74 LOC and a median of 54 LOC. The largest contract analyzed contained 1,405 LOC, but over 90% of the contracts comprised 200 LOC or less. For a significant portion of these contracts, specifically around 30%, ZEUS generated over 1,000 LOC of LLVM bytecode, with the maximum reaching

up to 91,338 LOC. The average bytecode per contract was 1,354 LOC, and the median was 439 LOC. In total, ZEUS verified more than 2 million lines of bytecode. It is important to note that these LOC measurements excluded blank lines and comments. Lastly, the study examined the most frequently occurring operations in the LLVM bytecode. They highlighted the top four classes of operations, aside from memory load, store, and GetElementPtr. This analysis revealed that ”arithmetic” operations occurred as frequently as ”comparison,” ”call,” and ”alloc” operations in the bytecode.

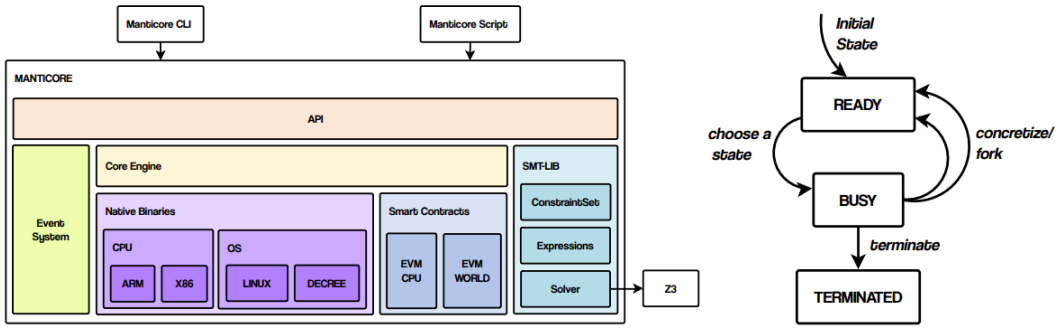
ZEUS introduced a certain level of instruction overhead when translating Solidity to LLVM bytecode. This overhead was less than 50 LOC for 97% of contracts in five of the seven bug categories. In the case of integer overflow/underflow, this overhead was under 200 LOC for 95% of contracts, although transaction order dependence checks could require over 500 LOC in 20% of contracts. In terms of analysis complexity, 75% of contracts generated fewer than 50 rules with a depth of around 700, with integer overflow leading to the most constraint rules and depth. Finally, the analysis time revealed that ZEUS could verify 97% of contracts within a minute, outperforming Oyente, which only returned results within a minute for 40% of contracts and failed to provide results or timed out for about 43% of them after 30 minutes.

In 2019, while the focus on SC vulnerability detection lessened, a group of researchers continued to delve into this area, exploring various features that could be gathered from compiled source code. Mossberg *et al.* [10] discussed Manticore, an open-source framework for dynamic symbolic execution. The Core Engine in this system-managed program states as outlined in the State Life Cycle shown in Fig. 11(a). Program states were abstract entities representing the state of a program at a specific execution point. These states provided an execution interface, which the Core Engine used to execute one atomic unit of the program; for native binaries and Ethereum, this was typically one instruction. During execution, states could interrupt and return control to the Core Engine to manage life cycle events.

The State Life Cycle, detailed in Fig. 11(b), comprised three states: Ready, Busy, and Terminated, and involved two main events: Termination and Concretization. The Core Engine continually picked a Ready state for execution, which then became Busy. A Busy state could either revert to Ready or trigger a Life Cycle event for the Core Engine to address. Termination occurs when a state reaches its endpoint, often due to a program exit or a memory access violation, leading to its transition to the Terminated state.

Moreover, Concretization was initiated when a state required a symbolic object to be transformed into one or more concrete values, influenced by the current constraints on that State. This process generated new child States for each concrete value, each marked as Ready. A common scenario of Concretization was forking, which happened when a program counter register turned symbolic and was concretized into various potential values, thereby creating new states for each possible program path. State exploration was fine-tuned through different policies, employing various heuristics for Ready state selection and Concretization. Designed with parallelism, the Core Engine supported multiple processes for managing the state queue, enhancing its efficiency and effectiveness.

This technique was vital for maximizing code coverage in software testing by navigating a program’s state space using constraint solving. Manticore was uniquely designed to analyze both binaries and Ethereum SCs, boasting a flexible architecture that adapted to diverse execution environments. Its customizable API further allowed users to tailor their analysis to specific needs. The paper highlighted Manticore’s capabilities



(a) A high-level architecture diagram.

(b) The state life cycle.

Figure 11: Overview of Manticore's Architecture [10].

in bug detection and code correctness verification, demonstrating its practical applications in commercial settings.

Building on previous ideas, Feist *et al.* [76] and Wang *et al.* [11] utilized bytecode for their research. In 2019, Feist *et al.* introduced Slither, a static analysis framework specifically developed for Ethereum SCs. Slither converted Solidity SCs into an intermediate representation called SlithIR, which employed Static Single Assignment (SSA) form and a simplified instruction set. Slither was a static analysis framework tailored for SCs, offering detailed insights into the contract code and the versatility to support various applications. It was currently employed for automated vulnerability detection, identifying a wide range of SC bugs without user input.

Additionally, it detected code optimizations overlooked by compilers, assisted in code understanding by summarizing and displaying contact information, and aided in assisted code reviews through its user-interactive API. The analysis in Slither was a multi-stage static process. Initially, it took the Solidity AST, generated by the Solidity compiler from the source code, as its input. The first stage involved Slither extracting crucial data like the contract's inheritance graph, the CFG, and a list of expressions.

Subsequently, the entire contract code was converted into SlithIR, Slither's internal representation language, which employed static single assessment (SSA) to simplify various code analyses. In the third stage, Slither performed actual code analysis, executing a set of predefined analyses that provided enriched information to its other modules. Slither was versatile, facilitating automated vulnerability detection, code optimization identification, enhancing user understanding of contracts, and aiding in code review. The framework's effectiveness was underscored by its speed and accuracy in identifying issues in Ethereum SCs, outperforming other static analysis tools.

In the same year, Wang *et al.* [11] presented NPChecker, a novel methodological approach for addressing the impact of non-determinism in Ethereum blockchain SC payments. This tool went beyond traditional patterns by modeling contract execution components and exposing nondeterministic factors through information flow tracking.

The workflow of NPChecker is summarized in Fig. 12. It started with the EVM binary code collected from the Ethereum mainnet. This code was first disassembled, and the EVM bytecode was then lifted to LLVM IR using an off-the-shelf LLVM IR lifter (EVMJIT). In this process, each EVM instruction was translated

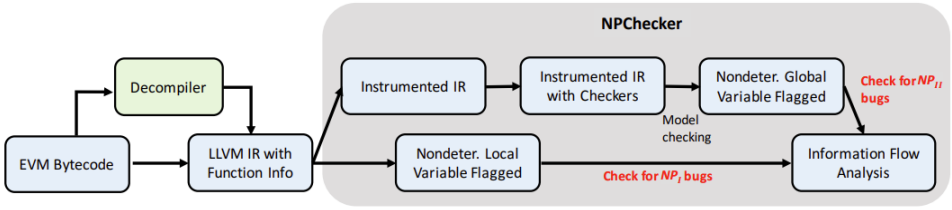


Figure 12: Overview of NPChecker's Architecture [11].

into one or multiple lines of LLVM IR statements, with specific instructions for SCs, like external contract calls, being translated into LLVM external calls.

Moreover, NPChecker recovered control flow structures using a commercial decompiler, augmenting the LLVM IR with functions. This step provided valuable high-level program information. The notable aspects of the LLVM IR code for SCs, along with the technical details of the reverse engineering process, were extensively covered in later sections of the research. NPChecker's testing on 30,000 Ethereum contracts effectively detected nondeterministic payments in 1,111 contracts, proving its high precision and efficiency. This approach also revealed new vulnerabilities and variants overlooked by conventional research reliant on vulnerability checklists.

Additionally, Torres *et al.* [12] introduced a systematic analysis of honeypot SCs on the Ethereum blockchain. These contracts, where attackers deployed traps instead of exploiting vulnerabilities, became increasingly common as SCs grew in popularity and value. A honeypot in the context of SCs was defined as a contract that appeared to inadvertently allow an arbitrary user (the victim) to access its funds, on the condition that the user first sent additional funds to it. However, this was a trap: the funds sent by the user became inaccessible to them, and only the creator of the honeypot (the attacker) could retrieve them. Fig. 13 illustrates the various actors involved and the phases of a honeypot operation, which typically included:

1. The attacker deploys a contract that appears vulnerable and seeds it with some funds as bait.
2. The victim, attempting to exploit what they perceive as a vulnerability, transfers the required amount to the contract but fails to retrieve any funds.
3. The attacker then withdraws both the bait and the funds the victim lost during their attempted exploitation.

Setting up a honeypot didn't require any special skills or capabilities beyond those of a regular Ethereum user. The attacker only needed sufficient funds to deploy the SC and place the bait. Fig. 14 presents the architecture and analysis pipeline of HONEYBADGER, a tool designed to detect honeypot techniques in SCs.

Moving through details, HONEYBADGER accepted EVM bytecode as input and produced a report on the various honeypot techniques it identified. The system comprised three main components: symbolic analysis, cash flow analysis, and honeypot analysis. The symbolic analysis component was responsible for constructing the CFG and performing symbolic execution on its different paths. The outcomes from this

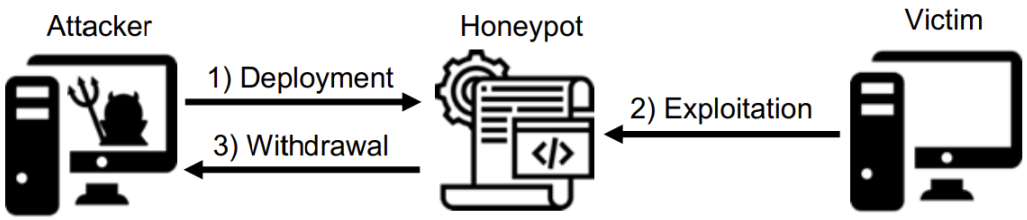


Figure 13: Actors and phases of a HONEYPOT [12].

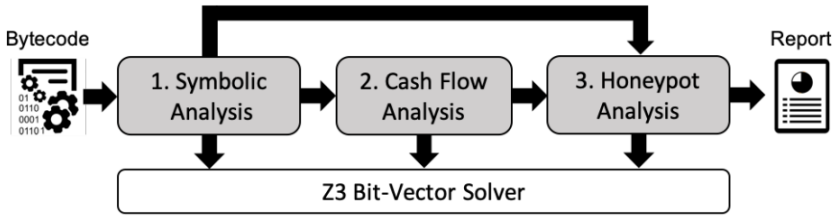


Figure 14: Overview of HONEYBADGER's Architecture [12].

symbolic analysis were then fed into both the cash flow and honeypot analysis components. The cash flow analysis used these results to determine whether the contract could receive and transfer funds. Lastly, the honeypot analysis component focused on identifying the various honeypot techniques discussed in the paper, utilizing a mix of heuristics and the insights gained from the symbolic analysis.

The evaluation proposed HONEYBADGER's effectiveness in identifying honeypots, as evidenced by the number of contracts it flagged for each specific honeypot technique. In total, HONEYBADGER detected 460 unique honeypots. Notably, 24 of these honeypots were already included in the initial dataset, which meant that the tool successfully discovered 436 new honeypots. This highlighted HONEYBADGER's capability to uncover a significant number of previously unrecognized honeypots in SCs.

The intensity of research in SC vulnerability detection slightly decreased between 2020 and 2021. However, several researchers continued to make significant contributions in this area, employing new methodologies and combining different sources of information.

Wang *et al.* [77] addressed the urgent need for efficient vulnerability detection in Ethereum SCs. They introduced ContractWard, a novel approach that utilized machine learning techniques. By extracting bigram features from simplified operation codes and combining various machine learning algorithms, ContractWard demonstrated high predictive scores and an impressive average detection time of just 4 seconds per contract. This work highlighted that classifiers trained on training sets balanced with SMOTE or SMOTETomek yielded higher Micro-F1 and Macro-F1 values than those trained on original training sets. Specifically, SMOTETomek outperformed SMOTE in five classifiers for balancing the data, achieving Micro-F1 and Macro-F1 scores above 96% with the XGBoost classifier. This suggested that both SMOTE and SMOTETomek effectively addressed the issue of classifiers' weak generalization ability due to class imbalance.

In the context of False Positive Rate and True Positive Rate, an ideal classification result would be at the point (0,1) on the graph, indicating perfect classification of both positive and negative samples. A point

closer to the top left corner implied better classification results. The Receiver Operating Characteristic curve of ContractWard with the XGBoost classifier. They also presented a comparison titled 'Overflow vs. The Rest', with 'The Rest' indicating invulnerable cases and 'Overflow' referring to Integer Overflow vulnerability. The True Positive Rates were over 94% and the True Negative Rates were over 97% for each type of vulnerability. These high TPR and TNR values demonstrated ContractWard's effectiveness in classification. Particularly in detecting vulnerabilities in BEC and DAO contracts, which had previously led to significant economic losses.

In 2020, Almakhour *et al.* [78] underscored the vital importance of verifying SCs within blockchain-based business processes. The immutable characteristic of blockchain technology meant that bugs in SCs could have serious economic repercussions. Their paper offered an extensive review of formal verification methods. It emphasized the need for thorough verification of both correctness and security in SCs. This was crucial for maintaining trust and continuity in blockchain applications. The paper served as a comprehensive survey, focusing on the capability of different methods to detect vulnerabilities and associated attacks in SCs.

Nguyen *et al.* [79] presented sFuzz in 2020, an advanced adaptive fuzzer for Ethereum SCs. sFuzz boasted a flexible architecture that enabled easy integration of various oracles. Presently, it supports eight oracles. The oracles operated by analyzing the logs of test cases. For example, to detect the Gasless Send vulnerability, sFuzz checked whether a test case executed a CALL instruction with non-zero data when the available gas was exactly 2300. Test cases revealing vulnerabilities were compiled into a separate test suite and reported to users, along with the specifics of the vulnerabilities they uncovered.

By design, sFuzz was calibrated to report only true positives based on their defined criteria for vulnerabilities, except in the case of Freezing Ether. However, a reported vulnerability might still have been a false positive if it aligned with the user's intentions, indicating that the definition might have been overly stringent. For the Freezing Ether vulnerability, a warning could have been falsely flagged if some test cases involved "send()" or "transfer()" functions, but such cases were not generated by the tool. The challenge in accurately identifying Freezing Ether vulnerabilities lay in covering all feasible opcodes, which was often impractical.

They presented a comparison of the efficiency of sFuzz with two other tools, Oyente and ContractFuzzer, revealing that sFuzz was significantly more efficient. On average, ContractFuzzer and Oyente generated and executed only 0.1 and 16 test cases per second, respectively. The reasons for sFuzz's superior speed were multifaceted. ContractFuzzer simulated the entire network and managed blockchain activities, such as committing state changes to storage and appending new blocks after function calls. Meanwhile, sFuzz focused solely on simulating aspects relevant to vulnerabilities in SCs. This targeted approach enhanced its speed. Furthermore, sFuzz's implementation in C++ was highly optimized, contributing to its efficiency, compared to ContractFuzzer, which was built using Node.js and Go language. As for Oyente's slower performance was expected, given that it was a symbolic execution tool, inherently slower than a fuzzer like sFuzz.

They provided a comparative analysis between sFuzz and ContractFuzzer, focusing on the number of distinct branches each tool covered. This comparison was graphically represented, with the y-axis indicating the differential in the number of branches covered by sFuzz compared to ContractFuzzer. Each SC was represented along the x-axis and sorted based on its y-axis value. Similarly, they compared sFuzz and Oyente in the

same context. In most cases, specifically in 4077 out of 4112 contracts, sFuzz succeeded in covering more branches than ContractFuzzer.
Moreover, for 35 contracts, ContractFuzzer covered more branches, which prompted further investigation. It was found that the higher branch coverage by ContractFuzzer could be attributed to certain factors. Firstly, sFuzz did not execute view functions for efficiency, leading to branches in these functions not being counted. Since view functions did not alter the state of a SC, they were deemed irrelevant for vulnerability considerations. Secondly, ContractFuzzer sometimes created invalid test cases that failed to meet mandatory constraints but covered additional branches. These mandatory constraints were set by the compiler (such as the Solidity compiler) and were incorporated into the bytecode to ensure the logical correctness of function calls or data types.
Continuing in 2021, Narayana *et al.* [80] focused on applying deep learning techniques to identify vulnerabilities such as Re-entrancy, Denial of Service, and Tx.origin in SCs. The proposed model for vulnerability detection in SCs leveraged direct feature extraction from SC source code to create a training dataset. This dataset was derived from the AST generated by the Solidity parser, an open-source tool known for its ease of installation and use. The parser facilitated navigating the AST using built-in functions and dictionary methods. Once the training dataset was prepared, deep learning models could be trained to classify new data, enabling efficient vulnerability detection. The architecture of this model involved six main steps:

1. Data Collection: SC source code was obtained from various online sources, including platforms like Etherscan. Etherscan provided access to information on SCs, transactions, and other Ethereum blockchain activities.

2. AST Generation: The collected SCs were processed using the Solidity parser to generate their ASTs. The parser allowed easy inspection of ASTs, including examining variable values, keywords (such as 'if', 'while', 'for', etc.), and function names.

3. Token Sequences: This step involved breaking the AST into sequences of tokens representing different SC code elements.

4. Dataset Generation: The token sequences generated a dataset instrumental in training the deep learning models.

5. Applying Deep Learning Models: These models were then trained on the generated dataset to learn how to classify and identify vulnerabilities in new SC code.

6. Experimental Results: The final step involved evaluating the model's performance and analyzing the results.

7. The datasets and programs utilized in this research, including the logic for detecting specific SC vulnerabilities like Re-entrancy, Denial of Service, and Tx.origin issues, were thoroughly discussed and made available on GitHub.

In this work, they introduced various deep learning models specifically designed to detect security-related vulnerabilities in SCs, focusing on re-entrancy, Tx.origin, and Denial of Service. These vulnerabilities were

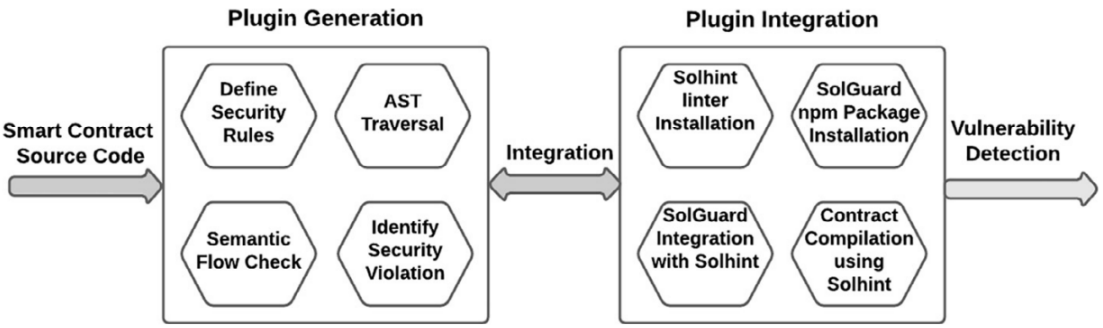


Figure 15: Overview of SolGuard's Architecture [13].

identified using three classification approaches: binary, multi-class, and multi-label classifications. The proposed deep learning techniques were practically implemented and tested on datasets, yielding satisfactory results with accuracy exceeding 97% in a relatively short time. The research utilized high-quality datasets containing approximately a thousand rows. The same dataset was used for training and testing, contributing to the high accuracy of the results.

Moving forward, Praitheeshan *et al.* [13] explored the application of SCs in blockchain-based multi-agent robotic systems in 2021. They developed SolGuard, a plugin that extended the 'solhint' linter to check for vulnerabilities arising from low-level external calls in Solidity SCs. They focused on addressing three external call problems in SCs with their SolGuard plugin, designed to analyze these contracts during compilation time. The default Solidity compiler, solc, checks for basic issues related to best practices in Solidity programming. However, additional types of vulnerabilities in Ethereum SCs necessitated the integration of certain linters with solc for enhanced issue detection.

In addressing this issue, an open-source linter named Solhint was used. Solhint categorized its validation rules into security, style guides, and best practices. However, it's important to note that Solhint did not validate issues related to external calls. SolGuard was integrated with Solhint to bridge this gap. The architecture of SolGuard, as depicted in Fig. 15, comprised two primary components: plugin generation and compiler configuration. This integration enhanced the ability to effectively identify and address external call vulnerabilities in SCs. SolGuard's superior efficiency and accuracy made it a promising tool for enhancing security in autonomous multi-robot systems.

Chen *et al.* [81] introduced DefectChecker in 2021, a symbolic execution-based tool for detecting defects in Ethereum SCs. Focusing on identifying errors and flaws leading to incorrect behaviors, DefectChecker targeted eight defects associated with severe unwanted behaviors. They presented the architecture of the DefectChecker approach, which comprises four key components: the Inputter, CFG Builder, Feature Detector, and Defect Identifier. The Inputter, located on the left side of the figure, is accepted bytecode as input. While Solidity source code was also permissible, it required compilation into bytecode. This bytecode was then disassembled into opcodes. DefectChecker processed these opcodes, dividing them into basic blocks and symbolically executing the instructions within each block. This process led to generating a CFG for the SC and recording all stack events.

In addition, the Feature Detector, during the symbolic execution, identified three specific features: Money

Call, Loop Block, and Payable Function. Utilizing this information, the Defect Identifier employed eight rules to pinpoint contract defects in SCs. Bytecode analysis was crucial for identifying defects in Ethereum SCs, considering that all bytecode is stored on the blockchain, but less than 1% of SCs had publicly available source code. This aspect was essential when SCs interacted with other contracts that might not have open-source code for verification. In such cases, the caller SCs had to rely on bytecode analysis to determine the security of the callee contracts. The scores for their tool were higher than Oyente, Mythril, and Security.

Lastly in 2021, Ashizawa *et al.* [14] presented Eth2Vec, a machine-learning-based static analysis tool for detecting vulnerabilities in Ethereum SCs. Eth2Vec, unique in its robustness against code rewrites, used a neural network designed for natural language processing to learn the features of vulnerable EVM bytecodes. Fig. 16 provides an overview of Eth2Vec, a system that extended the PV-DM model, commonly used for natural language processing, to handle EVM bytecode for unsupervised learning in neural networks. Eth2Vec used JSON files generated from EVM bytecode as input and calculated code similarity based on the learning outcomes from this extended PV-DM model.

To support this model, the EVM Extractor was developed. This extractor created JSON files at various levels—instruction, block, function, and contract—by syntactically analyzing EVM bytecode. Eth2Vec processed EVM bytecode and outputted a list of code clones and their associated vulnerabilities at the contract level. For training purposes, vulnerabilities within contracts had to be pre-identified, which could be done using existing analysis tools. The vulnerabilities in the test data were then assessed based on the code similarity between the identified vulnerable contracts and the training data. Ultimately, Eth2Vec outputted identified vulnerabilities at the function level of the analyzed code, pinpointing the vulnerable functions.

They presented Eth2Vec's inference throughput, focusing on the time taken for vulnerability detection and displaying the results. The detection time was primarily influenced by the number of functions in the contract, while the time to display results depended on the number of detected code clones. Current measurements showed that Eth2Vec's detection throughput was generally faster than an SVM-based scheme, except for the EVM Extractor's processing.

Eth2Vec analyzed contracts in approximately 0.371 seconds per contract, which was notably faster compared to the SVM-based scheme, which took about 0.660 seconds per contract. Additionally, compared to an NN-based scheme, which took about 0.748 seconds per contract, Eth2Vec maintained its efficiency advantage. However, the time required to display detection results tended to be longer than the detection process itself. For instance, in a more extensive analysis scenario involving 9 contracts and 94 functions, Eth2Vec required 5.6 seconds for detection, 0.015 seconds for saving, and 3.5 seconds for summarizing the results.

In 2022, there was a renewed focus on detecting vulnerabilities in SCs. Sharma *et al.* [**?**] delved into the significant role of blockchain technology, particularly spotlighting the influences of Bitcoin and Ethereum in popularizing decentralized transactional records. The paper introduced Mythril, a tool designed to assess vulnerabilities in Ethereum SCs. Mythril's core functionality was based on a symbolic execution engine, enabling it to meticulously examine all possible states within a contract through function calls. This approach allowed Mythril to identify vulnerabilities, though it did not extend to detecting exploits.

Mythril operated by decompiling the bytecode of an SC back into EVM opcode instructions and exploring all possible program states over a series of transactions. It employed LASER, a symbolic virtual machine, to

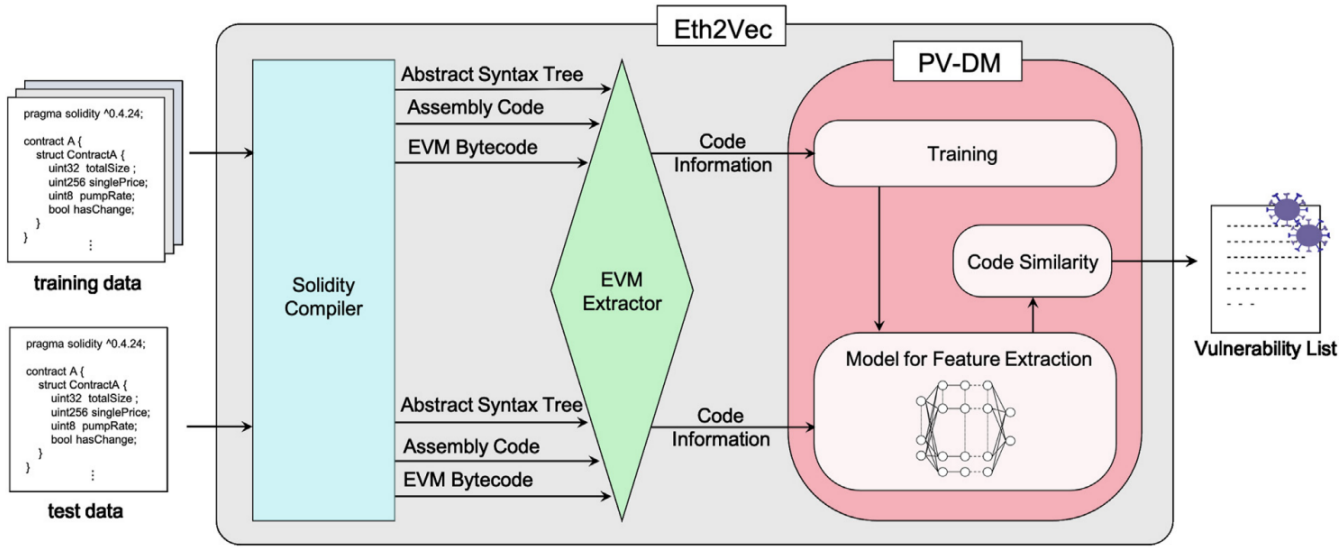


Figure 16: Overview of Eth2Vec's Architecture [14].

simulate opcode execution and uncover vulnerabilities. LASER was instrumental in computing all possible program states. To determine the existence of vulnerabilities, Mythril used various analysis modules and employed Z3, an automated theorem prover from Microsoft Research, to confirm or refute the feasibility of a compromised state. One such tool was Karl, which leveraged Mythril's capabilities to gain insights into new contracts on the blockchain. Mythril's capabilities highlight its crucial role in ensuring the security and integrity of SCs, especially in the context of the burgeoning NFT market and the broader Ethereum ecosystem.

Zhang *et al.* [15] introduced the Multiple-Objective Detection Neural Network (MODNN) in 2022, a groundbreaking and scalable tool for detecting vulnerabilities in SCs on blockchains. MODNN, capable of identifying 12 types of vulnerabilities, including new ones, was notable for its advanced Multi-Objective detection algorithms and implicit feature recognition capabilities. Fig. 17 outlines the architecture of a proposed method, which was divided into two key sections:

1. Feature Extraction: This was the first study to leverage a pre-trained Bert model for transforming the content of SCs into explicit features, aiming to enhance SC vulnerability detection. It also pioneered the use of opcodes to create co-occurrence matrices for learning implicit features.

2. MOD (Model Output Detection): Here, the features extracted earlier were utilized to deliver detection results. The model was designed to expand its output layer automatically to accommodate newly identified vulnerability types. The paper promised a detailed examination of these components and the associated technological challenges.

To train and test the multi-objective recognizer, the method employed a Convolutional Neural Network (CNN) with an attention mechanism and backpropagation process, distinct from typical CNNs as it used a feed-forward neural network. The training process was multi-faceted, beginning with feature extraction using the CNN's embedding layer and feature enhancement through the attention mechanism. Subsequently,

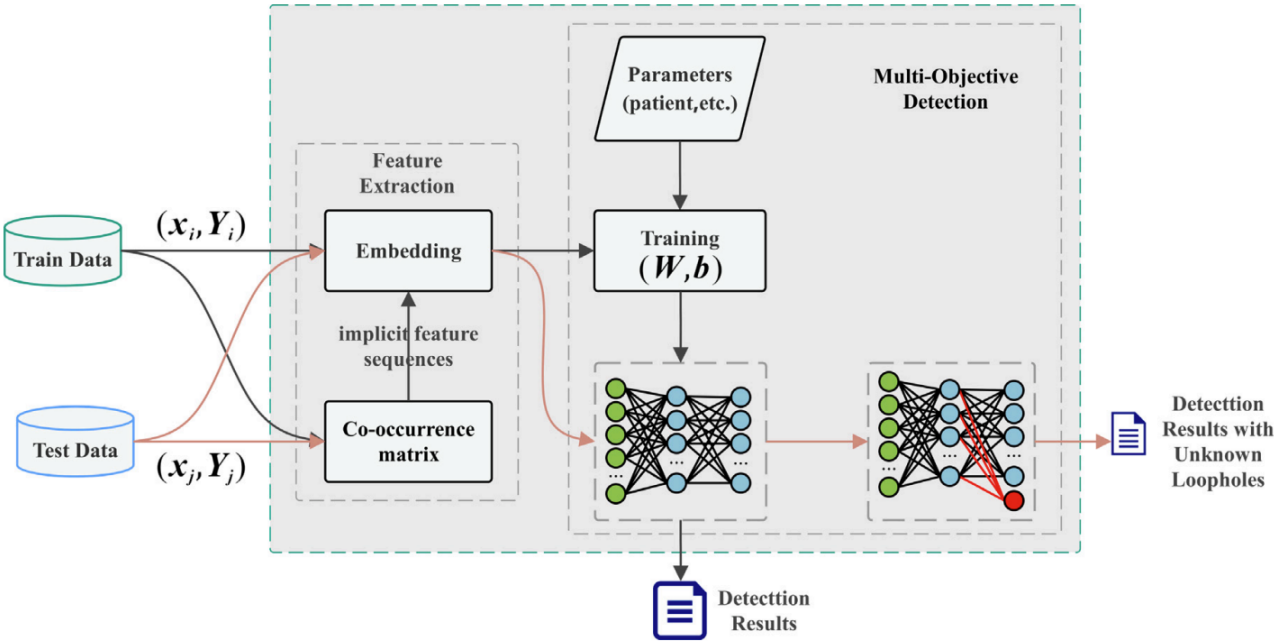


Figure 17: Overview of MODNN's Architecture [15].

further feature extraction occurred from the sequence using convolution and pooling layers with ReLU activation, followed by a multiple output structure in the dense layer, where each branch outputted a separate vulnerability probability. Fig. 18 displays the complete MODNN process.

The experiments and evaluations conducted demonstrated the high effectiveness of the work, achieving impressive metrics with an average precision of 96.51%, recall of 96.05%, and an F1 score of 96.27% across nine identified types of vulnerabilities. For three additional vulnerabilities, the scalability of MODNN, transformer, LSTM, and Bi-LSTM was evaluated at a default confidence level of 0.5. Without pre-assigned labels for unknown vulnerabilities, the method assumed all three remaining vulnerability labels in the test sequence to be set to 1 for prediction purposes. The prediction results were cross-verified with static analysis tools to ensure reliability.

In the same year, Ye *et al.* [16] presented Vulpedia, a novel method for detecting vulnerabilities in SCs. Vulpedia addressed the shortcomings of existing static and dynamic detection techniques, which often struggled with high rates of false positives and negatives. They provided details on the workflow for Vulpedia's vulnerability signature abstraction, encompassing four sequential steps. These steps included pre-detection using existing tools, manual inspection of vulnerability reports, AST clustering with signature abstraction, and finally, the composition of detection rules.

Delving further into the details, the process started by evaluating the accuracy and limitations of current tools in identifying vulnerabilities in SCs. For this, reports on a dataset of 76,354 contracts were collected and manually verified by experienced developers to differentiate between actual and false vulnerabilities. Next, the workflow involved clustering contracts based on similarity using tree edit distances calculated from their ASTs, particularly for contracts within the same vulnerability category. This step included summarizing common nodes from the contracts' program dependency graphs to form vulnerable and benign signatures,

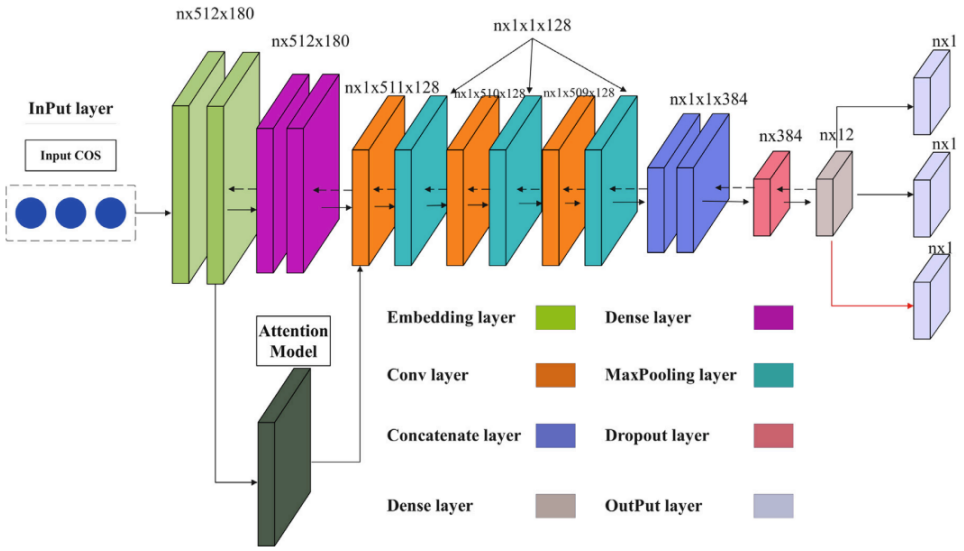


Figure 18: The entire process of the MODNN [15].

as demonstrated in Fig. 20.

Finally, the last step was manually integrating these signatures into comprehensive vulnerability detection rules. With these rules in place, as depicted in Fig. 19, Vulpedia's detector took unknown contracts as input and generated reports. It preprocessed the input contract code, extracted a normalized AST, performed PDG extraction, and compared these elements with the signatures in the vulnerability database. The outcome was determining the contract's vulnerability status based on whether its PDG aligned with vulnerable or benign signatures.

By mining expressive vulnerability signatures from both vulnerable and benign contracts and extracting structural program features, Vulpedia forms more accurate and comprehensive detection rules. In a comparative evaluation with established tools on a dataset of 17,770 contracts, Vulpedia demonstrated superior precision in four vulnerability types and leading recall in three, excelling in efficiency as well.

Lastly in 2022, Mohajerani *et al.* [82] presented a method for verifying security vulnerabilities in SCs, focusing on a casino SC case study. This method modeled the SC as interacting extended finite state machines. They used Solidity code for a casino, which featured three explicit states: IDLE, GAME_AVAILABLE, and BET_PLACED. In the IDLE state, the operator could create a game using the 'createGame' function, where a hashed number was assigned to decide the game's outcome. Upon creating a game, the state transitioned to GAME_AVAILABLE, allowing a player to place a bet. Once a bet was placed, the state changed to BET_PLACED.

The operator then used 'decideBet' to submit the original secret number and resolve the bet, determining the game's result based on whether the secret number was even or odd. The bet's outcome dictated whether the winnings were transferred to the player or added to the casino's pot. The casino operator had the flexibility to add or remove money from the pot at any state using 'addToPot' and 'removeFromPot' functions, the latter being subject to the condition that no active bet was placed. This was ensured by the 'noActiveBet' modifier. By framing security vulnerabilities as specific conditions within the extended finite state machine system and

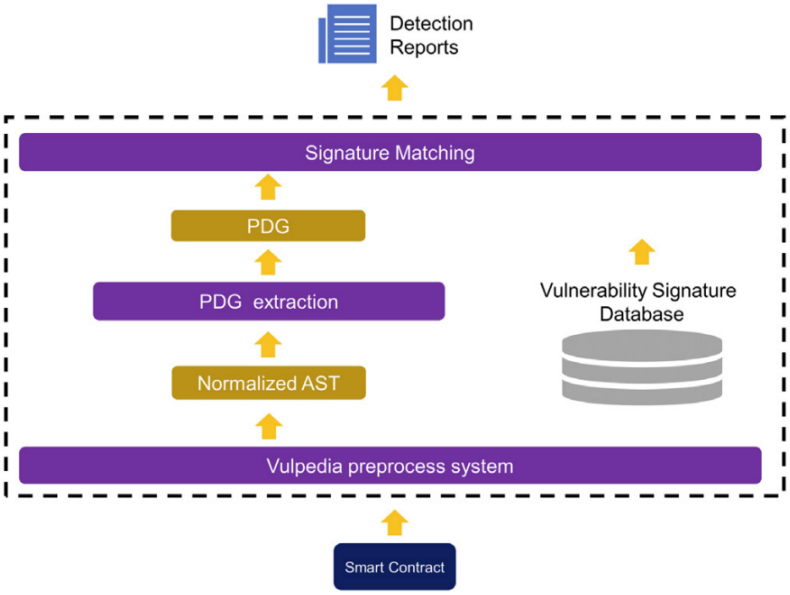


Figure 19: Overview of Vulpedia's Architecture [16].

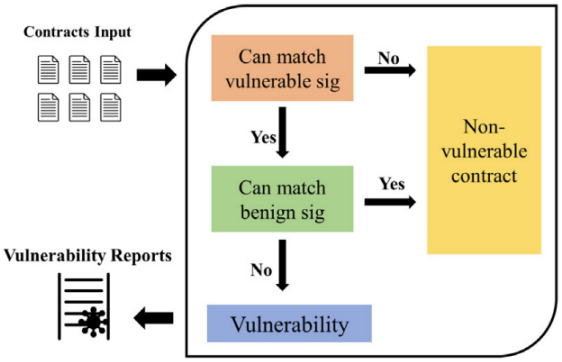


Figure 20: The detection workflow of Vulpedia [16].

applying non-blocking verification techniques, the authors successfully identified and resolved a significant vulnerability that could have locked funds permanently in the casino contract. The effectiveness of this approach was further validated by confirming the absence of such vulnerabilities in an improved version of the contract.

In 2023, there was a significant resurgence in the interest and development of tools for SC vulnerability detection, echoing the attention it received back in 2018. This revival was fueled by advancements in deep learning and machine learning, leading to a range of innovative solutions.

Xie *et al.* [17] introduced Block-gram, a novel feature extraction model for efficient vulnerability detection in SCs. They outlined the process of SC development and deployment. Developers initially wrote the source code in a high-level language like Solidity. This code was then compiled into byte arrays encoded as hexadecimal bytecodes. These bytecodes were uploaded to the EVM using an Ethereum client, where they could be translated into EVM instructions or opcodes.

Continuing the process, after compilation into EVM bytecode and ABI, the contract was deployed through the Web3.js interface. The deployment involved executing a transaction without a target address but with EVM bytecode as the data field. Once processed, the EVM executed this data as code, dividing the bytecode into deployment code, runtime code, and metadata. The runtime code and metadata were then stored on the blockchain, with their storage addresses matched to the contract account to complete the deployment.

In Fig. 21, the feature extraction model Block-gram for SC bytecode is detailed. It began by extracting 4-dimensional opcode block features from the opcode sequence flowchart, following EVM disassembly rules. Opcodes were then categorized into eight types based on six vulnerability analyses by expert knowledge, and their proportions were counted as eight-dimensional attribute features. The model analyzed individual contracts, test datasets, and training datasets for detailed feature contributions, calculating SHAP values to enhance interpretability.

Moreover, Fig. 22 illustrates the process of constructing an opcode sequence flow graph once the blocks and edges were identified. This flow graph, being a directed graph, was represented through an adjacency matrix. From this matrix, sequence features of the opcode were extracted, providing crucial insights. Following this, four-dimensional features specific to SC blocks were derived from the same adjacency matrix. These features effectively captured the sequential characteristics of the opcode, offering a detailed understanding of the SC's structure and behavior.

In their work, they proposed the feature contribution ratio. Significant feature contributions came from binary arithmetic opcodes ratio (bin), unary arithmetic opcodes ratio (una), and system opcodes ratio (sys), indicating their substantial impact on model training. These experimental results confirmed the features' interpretability in the model training process. As a result, the Block-gram features, in comparison to the traditional thousand-dimensional feature space, enhanced detection efficiency. An evaluation using seven advanced, learning-based methods demonstrated the effectiveness of these features in significantly improving the efficiency of detection.

Simultaneously, Sun *et al.* [18] and Ndiaye et al. [83] presented new frameworks, ASSBert and ADEFGuard, respectively.

ASSBert [18] combined active and semi-supervised learning with transformer networks to address chal-

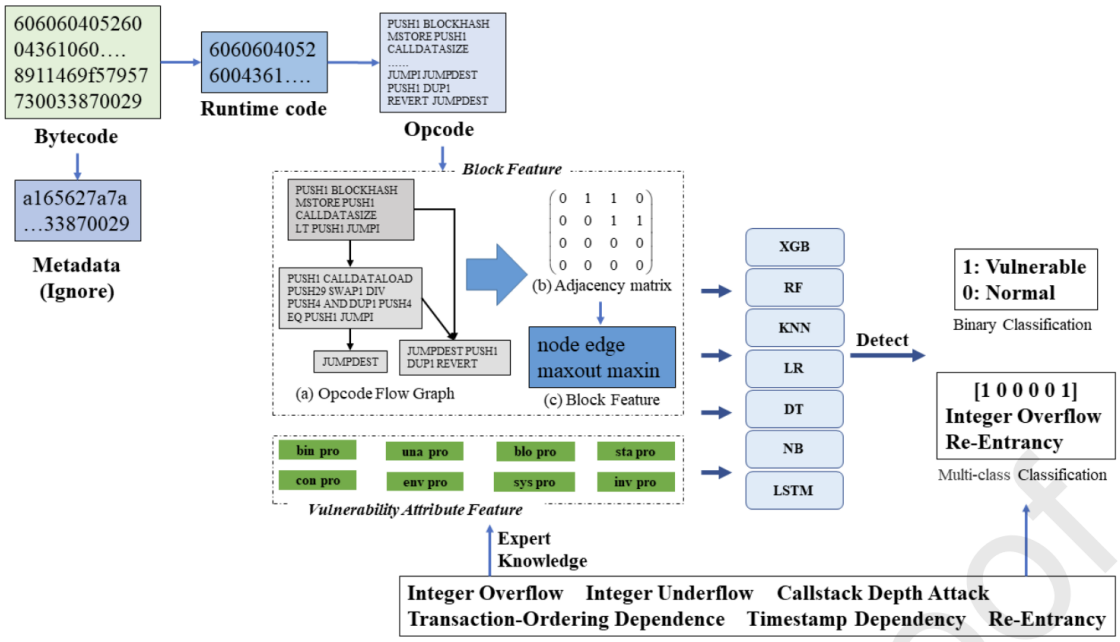


Figure 21: Overview of Block-gram Model's Architecture [17].

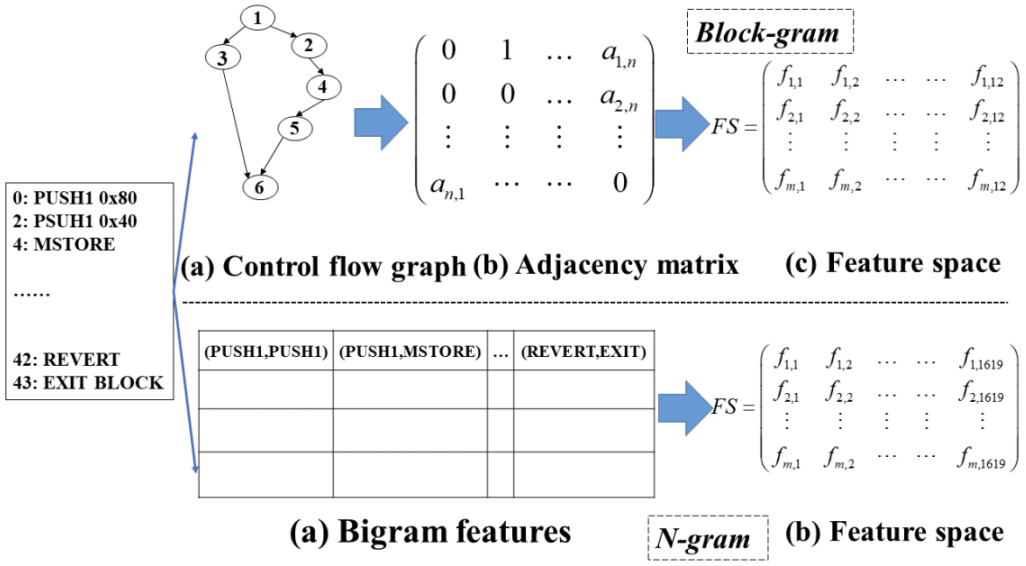


Figure 22: Extracting Block Features in Block-gram [17].

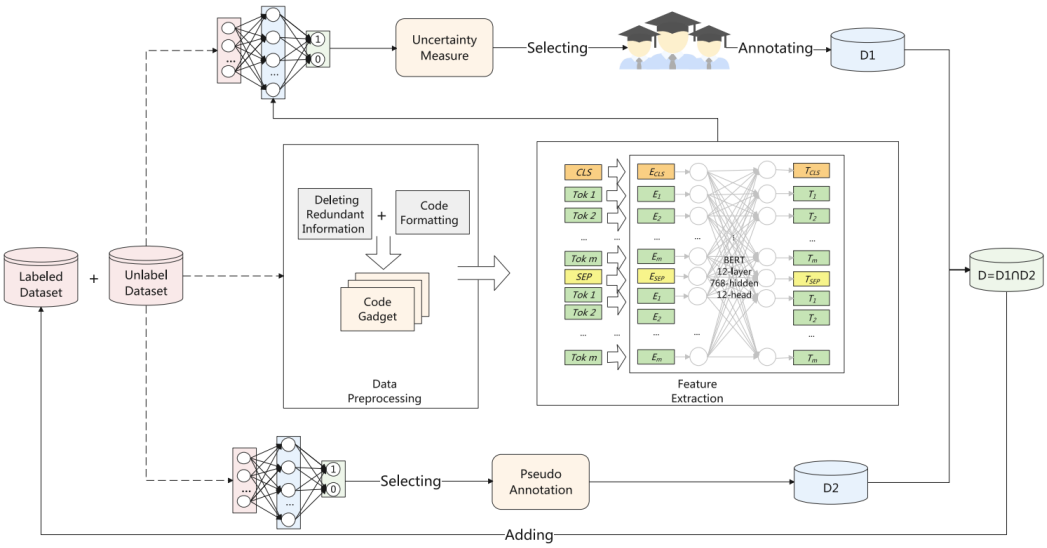


Figure 23: Overview of ASSBert's Architecture [18].

lenges in classifying SC vulnerabilities with limited labeled data. Fig. 23 illustrates the method's process as a circular one involving multiple rounds of data queries. In each round, the training set was formed from two sources: active learning queries and semi-supervised learning screenings. Active learning involved selecting a subset of unlabeled samples for manual marking, focusing on those with the highest uncertainty. Meanwhile, semi-supervised learning aimed to identify samples most likely belonging to a specific category, assigning them predictive pseudo tags based on the lowest uncertainty.

The method started with preprocessing the original SC source code, which included cleaning the data by removing irrelevant lines and comments, eliminating redundant information, and formatting the cleaned code. The vocabulary from the official Ethereum language description was then added to the Bert model's vocabulary. In preparing the code sequences, a [cls] token was added at the beginning and a [sep] token at the end of each code line. Following this, SC code gadgets were analyzed and transformed into eigenvector representations. The training set was iteratively expanded, with the active learning module selecting highly informative samples from the unlabeled set for expert labeling. Concurrently, the semi-supervised module picked the most confident sample subset from the unlabeled set using pseudo tags for the next training round. This iterative process continued until the desired outcome was achieved.

They provided the classification accuracy of ASSBert under various data preprocessing levels, with labeling ranging from 5% to 20%. To present the experimental results more clearly, a line chart was used to depict accuracy changes across different data preprocessing methods at these labeling ratios. As the labeling proportion increased, there was a general trend of significant growth in the prediction accuracy of classification models for six vulnerabilities under different data processing levels.

For the Timestamp vulnerability, up to a 15% labeling ratio, the joint processing method showed no distinct advantage over the contract-level processing method but outperformed the other two methods. Beyond a 15% labeling ratio, the combined processing method demonstrated higher detection accuracy than the other three models. For the CallDepth vulnerability, at labeling ratios below 15%, the contract level processing

method showed better model improvement than the other methods. However, at a 20% labeling ratio, the combined processing method achieved higher accuracy.

Regarding re-entrancy vulnerability, the combined processing method and function level processing method significantly enhanced model accuracy, particularly when the labeling ratio exceeded 10%. In contrast, for TransactionOrderDependence and Flow vulnerabilities, the combined processing method did not perform as well. Below a 15% labeling ratio, its improvement in model performance was not markedly better than the other methods, and at a 20% labeling ratio, it didn't show superior performance.

Ndiaye *et al.* [83] proposed ADEFGuard, which is an anomaly detection framework that utilizes SC behavior as a key feature for detecting scams and malicious contracts. The goal of anomaly detection was to identify states that differed from the norm, a concept that addressed the limitations of misuse detection by focusing on typical system behaviors instead of attack patterns. This detection process involved a training phase where the system was observed without any attacks, and a calculation method was established to define what constitutes a normal profile. The system's final state was then compared against this normal profile during the detection phase to identify any deviations, which were flagged as potential attacks. However, anomaly detection often resulted in a high number of false alarms due to legitimate behaviors that were not previously observed, and its effectiveness was highly dependent on the specific aspects of system behavior that were monitored.

Their model comprised two modules: an invariant calculation algorithm for determining a SC's normal profile, and an algorithm to monitor the contract's execution. This approach offered the advantage of detecting anomalies without needing pre-trained data or signatures and could identify unknown intrusions with relatively few false alarms. The initial state represented the system's state immediately after deploying the SC, before its execution. The normal profile reflected the desired state post-execution, characterized by a series of coherent system states. The final stage was the system's condition after the SC's execution, which, under normal circumstances, should have aligned with the normal profile.

They presented various performance metrics of their model. They defined state mutations and the concept of causal dependence between states and the alerts generated by threshold-crossing processes. Drawing inspiration from the causal dependence relations proposed by Lamport and Augsburg, they introduced the concept of contextual action and its related causal dependence relation. This method has shown promising results in detecting several well-known scams, allowing for the attainment of high-performance measurements in their evaluations.

Zhang *et al.* [19] developed SVScanner, a method that utilized heterogeneous patterns for vulnerability detection. Fig. 24 depicts the architecture of their approach, which was divided into three main parts: data processing, deep semantic extraction, and vulnerability detection.

Initially, the Solidity source code was processed into token and structural AST sequences. These sequences were then used to extract global and structural-semantic information through a sequence model and an attention mechanism. For the vulnerability detection phase, their approach employed TextCNN as the classifier to identify vulnerabilities in SCs. To generate structural AST sequences (SAS), the AST was constructed from the contract source code using Solidity's syntax. SCs, being collections of functions, had important elements extracted from each function, such as function names, variables, operators, and control flow state-

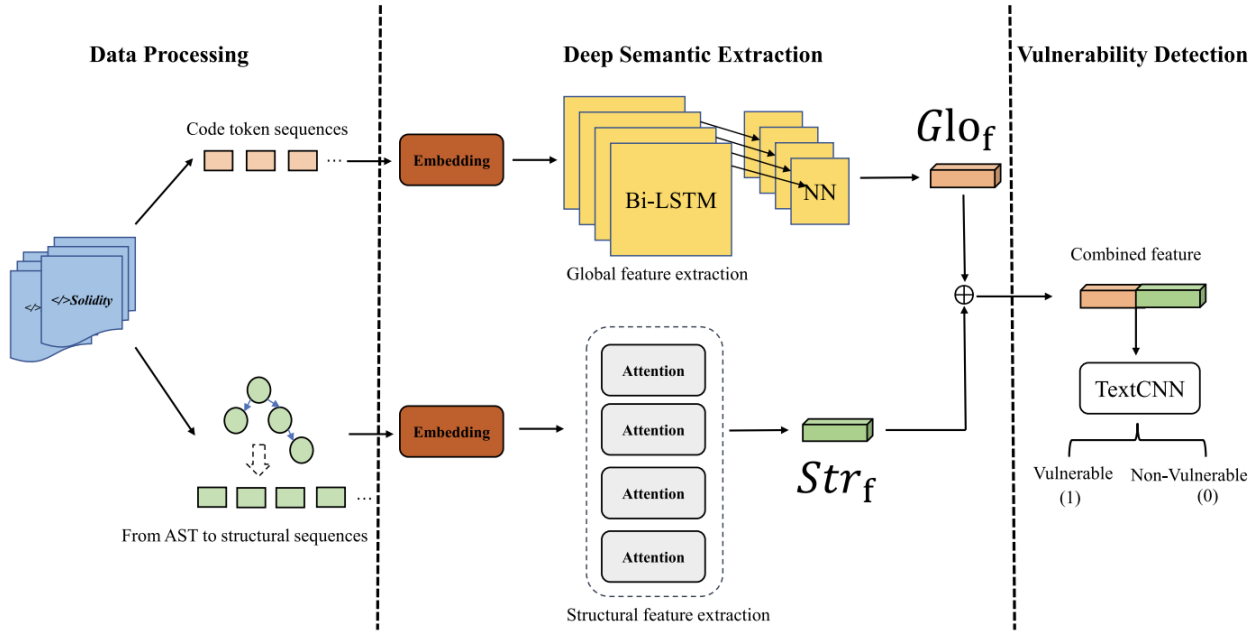


Figure 24: Overview of SVScanner's Architecture [19].

ments. Each element was represented by an AST node, with various attributes like type and value to convey the semantic and syntactic information of the contracts. The AST was built according to the grammatical relationships between different elements.

Continuing the procedure, The solidity-parser-antler tool was utilized to get the AST of SCs, and another tool was used to convert all ASTs into SAS, with each SAS token assigned a unique index. They provided a detailed result of converting an SC into SAS, showing the rich structural and semantic information it provides. TextCNN was chosen as the classifier due to its ability to extract deeper semantic information from raw features, proven effectiveness in text classification tasks, and simple, lightweight structure suitable for fast vulnerability detection.

He *et al.* [84] explored the realm of SC security, analyzing a variety of tools designed for vulnerability detection. The article initially delved into the prevalent security issues found in blockchain SCs. It then organized these tools into six distinct categories based on their approach to detection: formal verification, symbolic execution, fuzzy testing, intermediate representation, stain analysis, and deep learning. The effectiveness of 27 such tools was scrutinized, particularly in terms of their ability to identify vulnerabilities in various versions of SCs.

The results of their study revealed a limitation in most tools, as they primarily detected vulnerabilities in older and singular versions of SCs. Among the methods, deep learning, though less diverse in vulnerability coverage, was noted for its superior accuracy and efficiency in detection. The article concluded by recommending a future research direction that combines static methods like deep learning with dynamic approaches, including fuzzy testing. This integration aims to identify a wider range of vulnerabilities in different versions of SCs, thus improving both accuracy and comprehensiveness.

Liu *et al.* [20] and Tian *et al.* [21] both focused on enhancing the accuracy of vulnerability detection. Liu

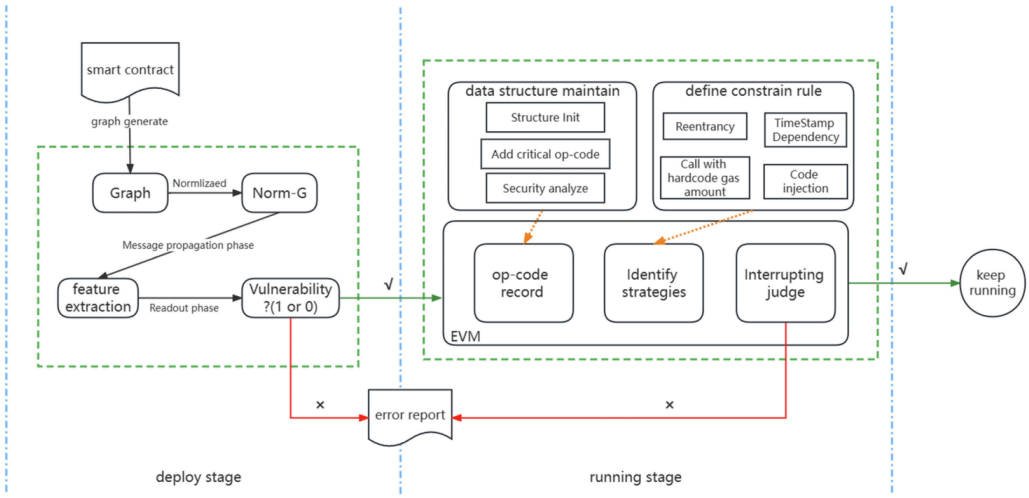


Figure 25: Overview of GNN-Vul-Detection's Architecture [20].

et al. [20] proposed a mechanism that integrated graph neural networks with expert patterns within a deep learning framework. Fig. 25 demonstrates the two-phase approach to managing SCs, which includes steps both before and after contract deployment. In the pre-deployment phase, SCs underwent vulnerability testing using a component equipped with a Graph Neural Network (GNN) model (GNN-Vul-Detection). Contracts deemed unqualified were prevented from being deployed, and a bug report was created and filed.

As per the model's specifications, during the runtime phase, the user established a value for the contract. For contracts of high value, expert rules were applied to scrutinize sensitive opcodes related to risky operations. If any violations were detected, the contract transaction was interrupted, and an error report was generated and submitted. A SC that successfully passed this second level of vulnerability testing without any interruptions in its opcode execution proceeded to execute its code. This ensured that transactions of SCs identified with vulnerabilities could continue to operate normally, maintaining both security and functionality.

They revealed that the deep learning model significantly outperformed other network models in identifying vulnerabilities, with the Graph Convolutional Network being the most effective among them. This superiority was attributed to the tendency of other networks to lose essential information from the SC code during processing, as they often overlooked structural elements of the contract program like data flow and call linkages. This suggested that merely treating source programs as code sequences was inadequate for vulnerability detection tasks. In contrast, representing source code as graphs and employing graph neural networks was a more effective approach. The graph also indicated that DM benefited from the added advantages of deep learning, surpassing traditional detection tools in performance.

Similarly, Tian *et al.* [21] introduced TrVD, which used tree decomposition to capture subtle semantic features of code fragments. The overall architecture of TrVD, as shown in Fig. 26, consisting of a training phase where a classifier learns from vulnerability datasets with well-labeled ground truths and a detection phase where the trained classifier outputs predictions. In both phases, the target source code fragment underwent processing through five main modules:

1. AST Generator: This module normalized the code fragment and constructed its AST using a parser.

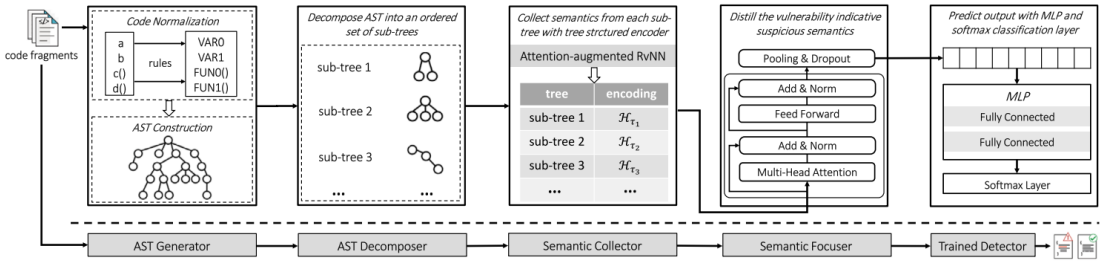


Figure 26: Overview of TrVD's Architecture [21].

2. AST Decomposer: It converted the AST into an ordered set of subtrees using a decomposition algorithm.

3. Comprehensive Semantic Collector: This collected semantics from each subtree into a real-valued vector with a tree-structured neural encoder.

4. Suspicious Semantic Focuser: It aggregated all collected subtree semantics into a contextualized embedding vector, focusing on vulnerability-specific semantics using a Transformer-based model.

5. Vulnerability Detector: This module, implementing a multi-layer perceptron (MLP) with a softmax layer, was trained to predict an output using cross-entropy loss.

Extensive experiments on five large datasets, comprising both synthetic and real-world samples, demonstrated TrVD's superiority over state-of-the-art deep learning-based methods in detection performance and runtime overhead. An ablation study also thoroughly evaluated TrVD's core designs.

In the realm of testing and verification, Driessen *et al.* [22] presented SolAR, an automated test suite generation tool for Solidity SCs, focusing on branch coverage. The tool SOLAR, as shown in Fig. 27, automated the generation of test suites for SCs. It initiated this process by scraping the provided ABI to identify all methods capable of altering the contract's internal state, excluding pure functions like views, and collected any hardcoded values useful for test generation. Subsequently, a Control Dependency Graph (CDG) was generated, with nodes representing coherent blocks of code and edges indicating transitions between these blocks. By designing a test suite that traversed each edge at least once, SOLAR ensured comprehensive coverage of all logical paths through the code.

Fig. 28 displays the various components of SolAR, which were also available in a pre-packaged Docker file for ease of use. Users interacted with the SolAR back-end through a command-line interface. This back-end accessed the SC ABI, employed a CFG-creation module, and refined the CFG into a CDG using Lengauer and Tarjan's algorithm. The edges of this CDG were covered by the test suite created by SOLAR. SOLAR generated test cases and executed them in a blockchain (ganache) environment, which returned the results of the method calls from each test suite. Based on their evaluation results, SolAR proved to be a valuable asset for both SC developers seeking to test contract behavior and researchers looking to evaluate new algorithms and tools for SC testing.

Moving forward, Jie *et al.* [23], Ji *et al.* [24], and Ali *et al.* [25] introduced novel approaches for vulnerability

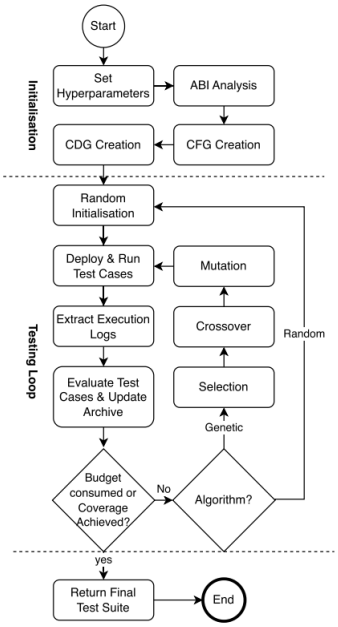


Figure 27: A step-by-step overview of the SolAR tool [22].

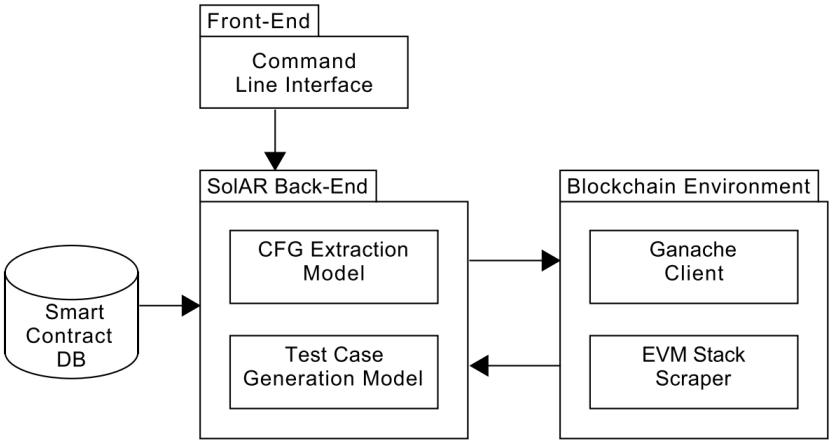


Figure 28: Overview of SolAR's Architecture [22].

detection and testing, each leveraging different techniques to enhance the robustness and effectiveness of their solutions.

First, Jie *et al.* [23] introduced a method that treated each strategy as a task in supervised vulnerability detection. This method involved several steps: selecting features, unifying feature dimensions, fusing features, training the model, and making decisions. Their approach utilized features from single-mode settings (SC, BB, and EVMB), two-mode combinations (SC+BB, SC+EVMB, and BB+EVMB), and a three-mode combination (SC+BB+EVMB), as shown in Fig. 29 Within this multimodal learning framework, techniques such as max-pooling, spatial pyramid pooling, and dense layers were used to standardize feature dimensions. Additionally, the method incorporated either horizontal or vertical concatenation for combining features.

The framework was designed to function optimally even if one or two modalities were absent and could analyze contracts published without source code. Empirical findings suggested that under intramodal settings, BB yielded the best performance, followed by SC and then EVMB. In two-by-two intermodal settings, SC+BB was the most effective, followed by BB+EVMB and SC+EVMB. The three-by-three intermodal setting outperformed the two-by-two intermodal setting, which, in turn, outdid the intramodal settings. Despite limitations in the number of AI models used, their framework represented a significant advancement in SC vulnerability detection. Future work would address the out-of-vocabulary issue, explore multi-class classification, and delve into feature importance.

Second, Ji *et al.* [24] proposed Effuzz, a method that combined an input parameter selection strategy with an accelerated search strategy to enhance the efficiency and branch coverage in fuzzing Ethereum SCs. The design is illustrated in Fig. 30 Effuzz operated in three main steps: Effuzz started by analyzing the dependencies of branch constraints. Based on this analysis, it identified that only a subset of the input parameters effectively satisfied the branch constraint.

Consequently, selectively mutating the involved parameters was more efficient than considering all input parameters. However, for on-chain SCs, constructing a CFG, crucial for classic dependency analysis, was challenging. To overcome this, they introduced an approximate dependency analysis technique that extracted dependency information at the bytecode level, aiding in subsequent parameter selection. The selection process was designed to maintain reachability and satisfiability, reducing invalid mutations. Initially, Effuzz selected parameters that ensured reachability, as reaching the target statement was vital. Once such parameters were identified, Effuzz proceeded to the next step with this subset of input parameters.

The approach tackled two main challenges. Firstly, it involved selecting the right subset of input parameters for mutation, considering the unique characteristics of Ethereum SCs. Secondly, it focused on accelerating the search to satisfy difficult branch constraints without generating excessive ineffective test cases that would otherwise waste resources. Effuzz was implemented and tested on 3600 Ethereum SCs, with experimental results indicating its superiority over current state-of-the-art SC fuzzers. Notably, Effuzz was able to find more vulnerabilities while also improving efficiency.

Finally, Ali *et al.* [25] proposed an efficient SC Runtime Protection framework, called SRP, that minimized the on-chain burden of runtime verification by integrating an off-chain mechanism with on-chain contract execution. Fig. 31 displays the architecture of SRP. Alongside the EVM, it included three primary components, each highlighted in gray, and each component played a distinct role.

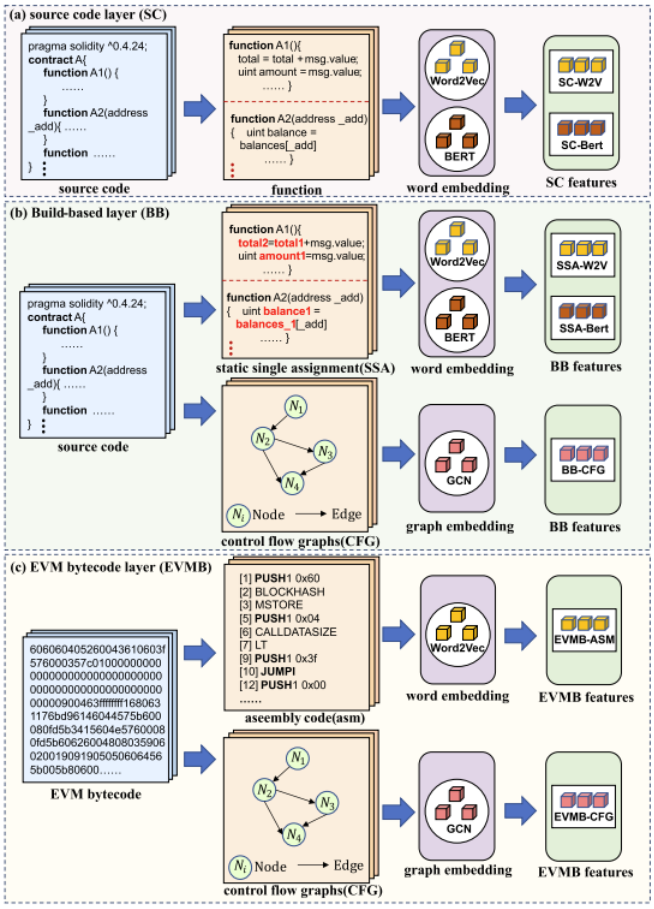


Figure 29: An Overview of the Three Key Modalities and Their Associated Features [23].

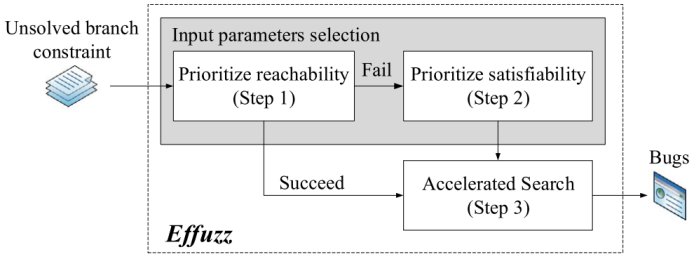


Figure 30: Overview of Effuzz's Architecture [24].

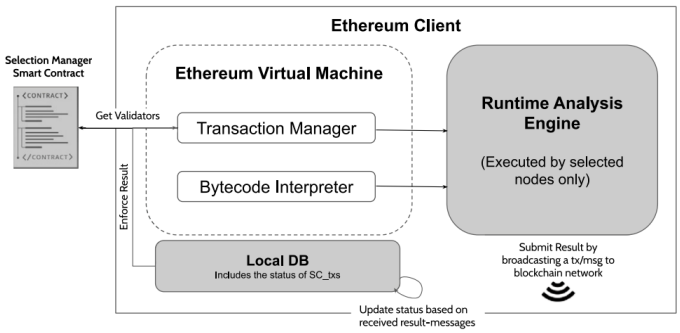


Figure 31: Overview of SRP's Architecture [25].

- Selection Manager: This was an SC responsible for handling the registration of vulnerable SCs and off-chain validators. It also managed the selection process for these entities.
- Local Database: This component stored transaction records and the results of Runtime Verification (RV).
- Runtime Analysis Engine: This engine was activated only by selected validators to apply Runtime Verification off-chain, ensuring a focused and efficient analysis process.

The researchers tackled the challenge of overhead in on-chain Runtime Verification (RV) for SCs on blockchain platforms. They developed SRP, a novel framework that integrated a decentralized off-chain service for SC RV into a permissionless blockchain environment. The main innovation of SRP was its ability to enhance RV performance by offloading the verification process off-chain. This shift allowed for the concurrent verification of multiple SCs and facilitated the on-chain execution of other types of transactions.
Delving into specifics, the RV process strategically occurred during block construction rather than validation, resulting in heightened efficiency by minimizing redundant executions and expediting block construction times. The effectiveness of SRP was empirically validated using data from Etherscan and through a proof-of-concept implementation on the Ethereum platform. The results demonstrated that SRP outperformed traditional on-chain RV methods, achieving higher throughput and reduced block construction time and confirmation latency as the workload increased. This study not only proved the feasibility of SRP but also opened up new avenues for research in developing efficient and secure off-chain RV solutions for SCs.
Continuing into 2023, Ren *et al.* [26], Yuan *et al.* [27], Cai *et al.* [28], and Zhou *et al.* [29] individually presented distinct approaches to tackle SC vulnerabilities.
Ren *et al.* [26] proposed a novel approach, namely, Blass, based on a semantic code structure and a self-designed neural network. As depicted in Fig. 32, Blass was structured into three phases. The first phase involved constructing Complete Program Slices (CPSs) with full semantic structure information through program slicing and semantic code structure fusion. The second phase encoded these CPSs into vector sets, and the third phase focused on designing and training the Bi-LSTM-Att model.
Moreover, they created an SC dataset using code sourced from Etherscan and GitHub, comprising 25,103 SCs. To address the initial insufficiency of SC vulnerability samples, they manually modified the code to add defect samples to the dataset. This process resulted in a total of 43,766 program slices extracted by sensitive

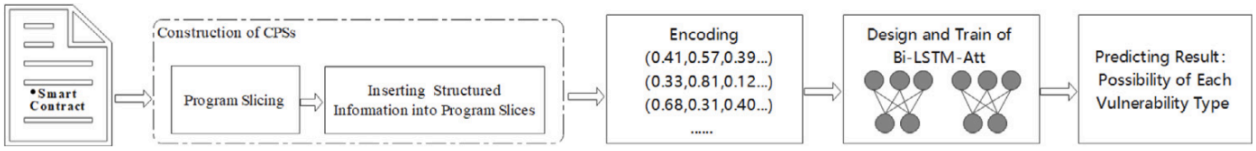


Figure 32: Overview of Blass's Architecture [26].

operational elements, including 18,559 vulnerable and 25,197 nonvulnerable slices. The vulnerable slices consisted of 5317 with integer overflows, 3785 with re-entrancy attacks, 3991 with dangerous delegate calls, and 5466 with timestamp dependencies.

Extensive experiments demonstrated that Blass achieved impressive recall, accuracy, and F1 scores in detecting the four vulnerabilities, outperforming five traditional vulnerability detection methods. The semantic code structure significantly improved Blass's detection effectiveness, and the Bi-LSTM-Att model was confirmed as a state-of-the-art classification model. While Blass showed promising results in detecting SC vulnerabilities, it had some limitations. It did not consider binary code and only focused on four types of potential vulnerabilities.

Advancing, Yuan *et al.* [27] employed the Bug Injection framework with transfer learning techniques. Their approach involved pre-training the Transformer encoder using source code, intermediate representation, and assembly code. Initially, they translated the source and byte codes into intermediate representation and assembly code, respectively. These multi-modality code snippets were then concatenated into sequences and fed into the Transformer encoder to learn code features. The approach was further fine-tuned and evaluated using the Bug Injection framework and a classifier for detecting SC vulnerabilities. Fig. 33 illustrates an overview of this working procedure.

The approach comprised a multi-layer Transformer encoder, augmented by the Bug Injection framework and a Softmax classifier, designed to fine-tune the model and identify new SC vulnerabilities. It operated on three different code modalities: the SC's source code, the intermediate representation from the source code, and the assembly code derived from byte code. To enhance feature extraction, they considered these as multi-modalities code snippets, transmitting them into the Transformer encoder for pre-training.

The process entailed three main steps. First, the preprocessing step combined and tokenized the input code snippets. Then, the Transformer encoder processed these as multi-modality code sequences for pre-training. Finally, the approach was fine-tuned and evaluated through the Bug Injection framework and Softmax classifier. The default embedding consisted of three parts: token, segment, and position embedding. To improve this, they introduced entropy embedding, using entropy values to determine the weights of the code sequence, thus influencing the weight distribution of the multi-modality code sequence.

They outlined the process of calculating the code's entropy value and the derived entropy embedding. They translated the Solidity source code into an intermediate representation using the EVM compiler, while the assembly code was converted from byte code obtained by parsing transaction data with an EVM decompiler. Following this, they counted the code token frequency of the multi-modality code sequence as per their vocabulary. Treating the code sequence as a random variable, they calculated its entropy value H_x, which was then used to generate the entropy embedding.

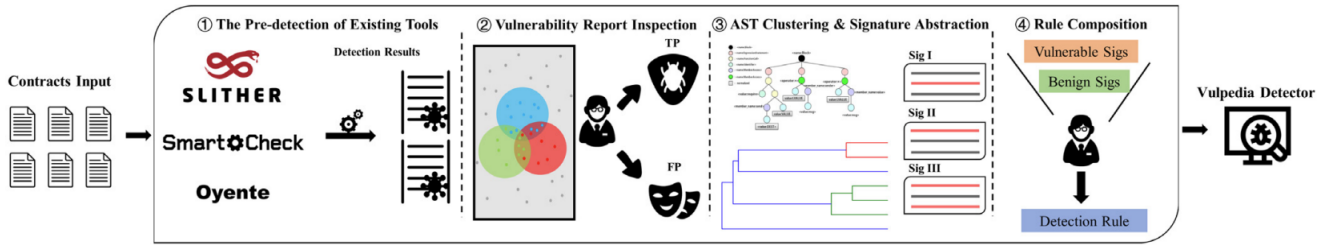


Figure 33: Overview of multi-layer Transformer encoder's Architecture [27].

As a result, by employing the Bug Injection framework to automatically generate buggy code, they facilitated the fine-tuning of the classifier for detecting vulnerabilities in SCs. The experimental results indicated that their proposed approach, leveraging multi-modality and entropy embedding techniques, enhanced the accuracy of vulnerability detection compared to methods relying on uni-modality code sequences.

Following, Cai *et al.* [28] proposed a novel approach for detecting vulnerabilities in SCs at the slice level, focusing on capturing critical syntax and semantic features from the source code through a code graph representation and employing a Graph Neural Network (GNN) model to identify vulnerable patterns.

As illustrated in Fig. 34, their approach began by taking source code files of SCs as input. To effectively extract syntax and semantic information, they generated graph representations for each function of the target SC by combining different code graphs, such as AST or CFG. Program slicing was then applied to remove nodes irrelevant to vulnerabilities from these graphs. The sliced graphs were fed into a bidirectional Gated Graph Neural Network (GGNN) model for feature learning, thereby obtaining hidden features for each node. A hybrid attention graph pooling, integrating self-attention and global attention pooling, was used to construct the graph-level feature from individual node features. A Multilayer Perceptron then served as the classifier to detect vulnerabilities based on these graph-level features.

They proposed that combining different graphs through their Joint Code Graph (JCG) approach enhanced feature representation capabilities. The top-k nodes, determined by their attention scores and a hyperparameter named pooling rate, were selected to form a more task-related subgraph. This subgraph was then processed through average pooling to generate a subgraph-level feature. This method of constructing a sliced joint graph representation for an SC's function retained ample syntactic and semantic information from the source code by merging various graph representations and purifying noise nodes through program slicing. The bidirectional graph representation aided in contextual feature learning during model training, and the hybrid attention pooling layer extracted graph-level features for effective vulnerability detection.

In the next study, Zhou *et al.* [29] leveraged adversarial multi-task learning. They implemented a multi-task learning model for SC vulnerability detection, defined by two distinct tasks: identifying the presence of vulnerabilities and classifying specific types of vulnerabilities. The first task was addressed using a binary classification network, and the second through a multi-classification network. A general classification network was used for simplicity, with a shared layer feature vector serving as the input.

As depicted in Fig. 35, the left branch, tower A, was dedicated to detecting vulnerabilities in SCs. This branch consisted of a convolutional layer, a pooling layer, a dropout layer, two fully connected layers, and an activation layer. Building on their previous research, they further enhanced vulnerability detection in SCs

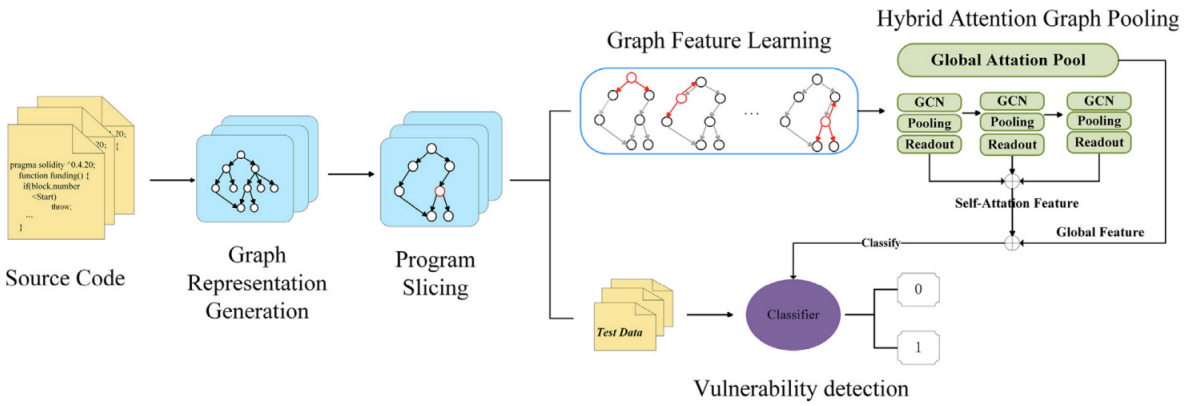


Figure 34: Overview of GGNN's Architecture [28].

using multi-task learning, adopting a parameter-sharing framework to facilitate knowledge sharing. Experimental results showed that their model, incorporating adversarial transfer learning and orthogonality constraints, outperformed the previous simple multi-task learning model. They also compared their model with deep learning-based methods and other multi-task learning models. The findings revealed that their model surpassed deep learning-based methods in detection accuracy due to the knowledge-sharing structure of multi-task learning, which effectively extracted relevant task features to enhance performance.

Additionally, their model achieved notable detection precision compared to most multi-task learning models. They noted that these differences were primarily related to the structures of other multi-task learning models, such as MT-CNN and MT-DNN, which were based on classic CNNs with varying degrees of optimization for feature learning. Other models like MMoE and CGC also aimed to optimize the structure of multi-task models to improve performance. In contrast, their proposed method presented a mixed parameter-sharing architecture, introducing adversarial transfer learning to mitigate interference between private and common features and adding regularization constraints to reduce feature redundancy.

In 2023, advancements continued as the complexity of EVM bytecode posed challenges, especially in resolving jump addresses. State-of-the-art static analysis-based solutions faced with accurately detecting programming defects and vulnerabilities in Ethereum SCs.

To address this issue, Pasqua *et al.* [85] introduced a new method, named EESAC-CFGEB, that employed symbolic execution of the EVM operands stack. This method effectively resolved jump addresses in EVM bytecode and constructed a precise Control-Flow Graph for compiled SCs. Many static analysis techniques that relied on a CFG-based representation of SCs could benefit from this approach. The CFG reconstruction algorithm was implemented in a tool called EtherSolve. This tool was validated on a large dataset of real-world Ethereum SCs, proving its ability to extract more precise CFGs compared to existing approaches. Additionally, EtherSolve was enhanced with detectors for two major Ethereum SC vulnerabilities: re-entrancy and Tx.origin.

They proposed a novel approach to extract a precise CFG from EVM bytecode, which could serve as a foundation for new static analysis tools aimed at detecting defects and vulnerabilities in Ethereum SCs. EtherSolve was evaluated against state-of-the-art analysis tools using a CFG-based code representation. Moreover, EtherSolve's enhanced capability to detect re-entrancy and Tx.origin vulnerabilities was compared with ex-

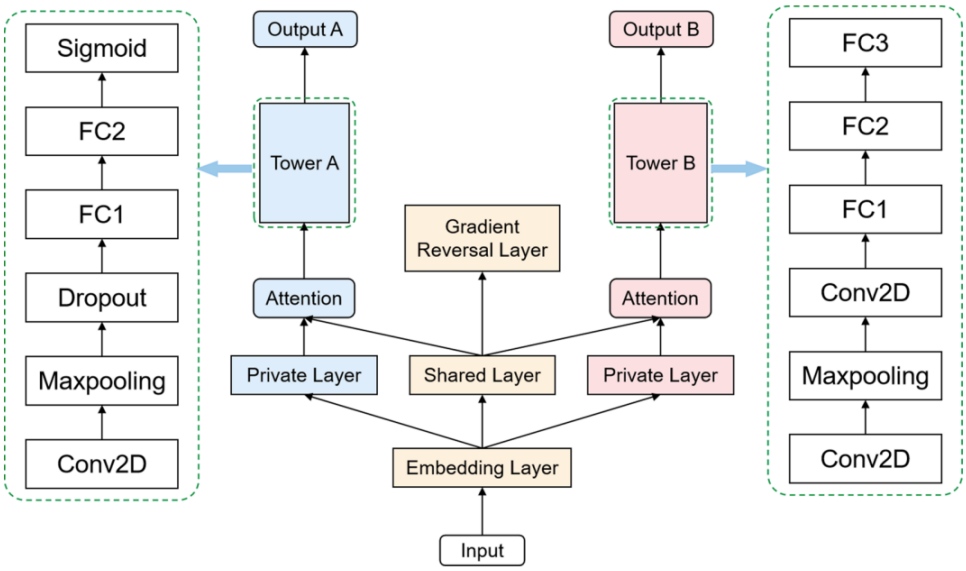


Figure 35: Diagram of the Multi-Task Network Branch [29].

isting Ethereum analyzers, yielding excellent results.

Subsequently, Wu *et al.* [30] developed TaintGuard, a new cross-contract static analysis tool designed to enhance the robustness of SC privilege control mechanisms at the source code level. TaintGuard, based on the AST of Solidity code, utilized taint analysis and instrumentation monitoring. It employed the solc compiler to create an AST of contract source code, filtering call relations that used 'delegatecall' for cross-contract calls. TaintGuard then generated a CFG within the target function to identify potential problematic paths through static taint analysis and inserted monitoring code at appropriate program positions. This process monitored the contract status at runtime to prevent tampering with contract privileges by malicious callers.

TaintGuard's system architecture, as depicted in Fig. 36, involved a five-step process of Solidity contract analysis. The first step compiled the source code to generate an AST, which was then used to filter 'delegate-call' function calls and record nodes. The second step checked for unknown contracts and conducted code instrumentation as needed.

Following the steps, the third step involved generating an internal CFG and applying taint analysis to identify problematic paths. The fourth step involved transforming problem paths into a branch of the AST, inserting monitoring code, and then converting it back into Solidity code to create a security-enhanced SC. The final step returned the analyzed SC to the developer. TaintGuard was validated on a large dataset of real-world Ethereum SCs and was found to have better analysis efficiency than current tools. It accurately provided privilege protection for contracts at risk of implicit permission leakage and enhanced contract permission monitoring.

In another study, Li *et al.* [31] introduced ExpenGas, a new learning-based method utilizing evolutionary computation-based machine learning, to detect expensive operation patterns in Ethereum SCs. This approach was developed in response to the limitations of most static analysis methods, which typically relied on spe-

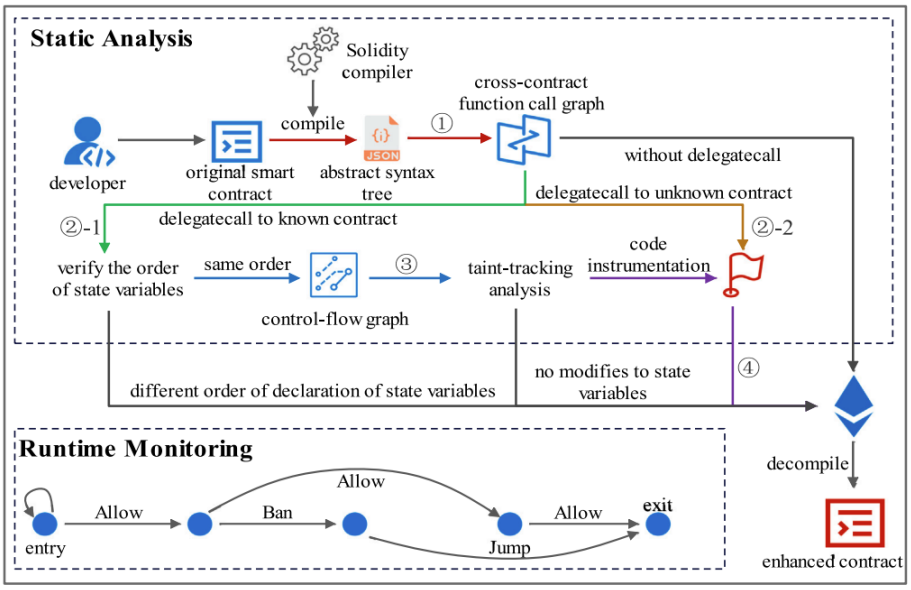


Figure 36: Overview of TaintGuard's Architecture [30].

cialist knowledge and manually summarized patterns for detecting gas-expensive operations. Additionally, the rapid proliferation of SCs, generating vast amounts of data, posed challenges in reapplying these methods across varying patterns.

ExpenGas, showcased in Fig. 37, was grounded in the neural Transformer architecture of the PL and employed linear layers for outputting results. Its design emphasized a low-complexity multi-crucial data flow graph, allowing the model to concentrate on essential features. Tested on 21,981 SC files, ExpenGas achieved an accuracy of 83.05% and a recall of 91.96% in detecting expensive operation patterns, outperforming current state-of-the-art methods significantly.

Despite its achievements, ExpenGas faced some limitations. Subjectivity in pre-training was an issue, as they manually labeled the Smartbugs-wild dataset due to the absence of a ready-labeled dataset. Furthermore, ExpenGas was only tested for the expensive operation mode, not other gas-expensive patterns, primarily due to dataset constraints and the time-consuming nature of labeling extensive datasets. Additionally, the less apparent keywords in some gas-expensive patterns could potentially impact the detection effectiveness.

In [32], Wang *et al.* focused on minimizing transaction risks in DeFi systems, particularly within Ethereum's DeFi ecosystem, which relied heavily on SCs. Due to the pseudo-anonymity of Ethereum, users faced significant fraud threats, especially from Ponzi schemes. Previous studies have utilized machine learning to build detection models for Ponzi schemes based on static SC samples. However, these methods often fell short in the early detection stages, as Ponzi schemes tended to reveal their characteristics over time. Current approaches struggled with capturing the temporal features of SCs in big data environments, leading to inadequate recognition rates.

To address this, they proposed TTPS, a Long Short-Term Memory (LSTM) based method for detecting Ponzi schemes, considering the time series transaction information of SCs. TTPS analyzed both temporal account features and code features of SCs. They employed Adaptive Synthetic Sampling (ADASYN) to effectively

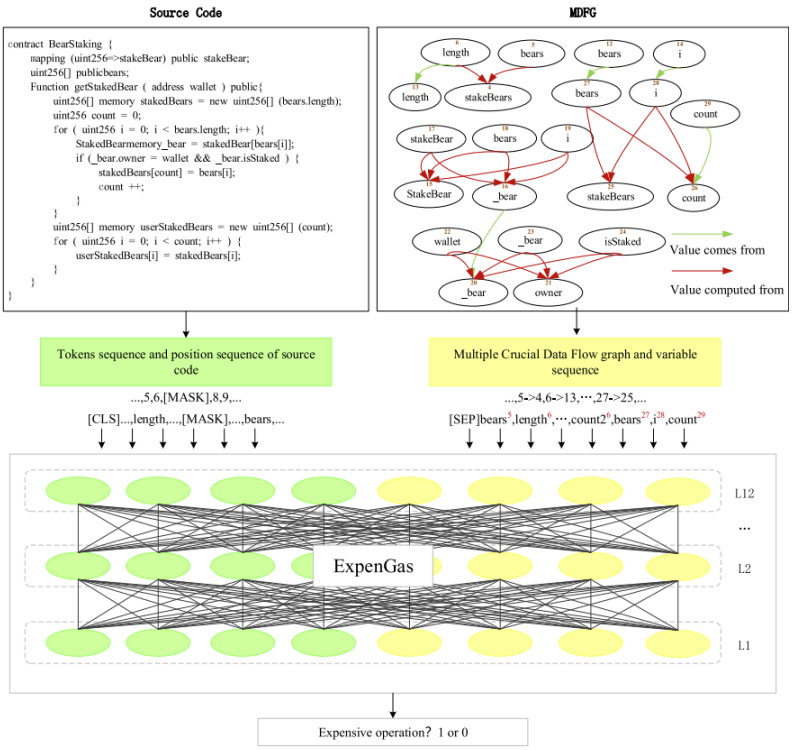


Figure 37: Overview of ExpenGas's Architecture [31].

enhance the feature data of minority-class Ponzi scheme samples. The TTPS approach, shown in Fig. 38, comprised of four stages:

1. Data Acquisition: Transaction information and SC bytecode were collected using Ethereum's API.
2. Temporal Feature Extraction: TTPS combined account and code features of SCs. SC bytecode was converted into opcode, and its frequency was calculated. They also monitored the entire transaction process, determining seven account features based on the Ponzi scheme's capital flow characteristics.
3. Adaptive Synthetic Sampling: To address few-shot learning issues in the dataset, Ponzi scheme samples were expanded using ADASYN, with min-max normalization being used to compress the eigenvalue range.
4. LSTM Learning: TTPS used the LSTM algorithm to build a Ponzi scheme detection model, training it with processed data and optimizing parameters.

They provided details on the model training flow. Moreover, they proposed details on the average detection time of different methods. Although TTPS took longer for each detection than traditional machine learning methods, it ensured timely fraud detection in contracts. Their experiments demonstrated that TTPS could detect smart Ponzi schemes as soon as fraud characteristics appeared, ensuring timeliness in detection and reducing investor risks.

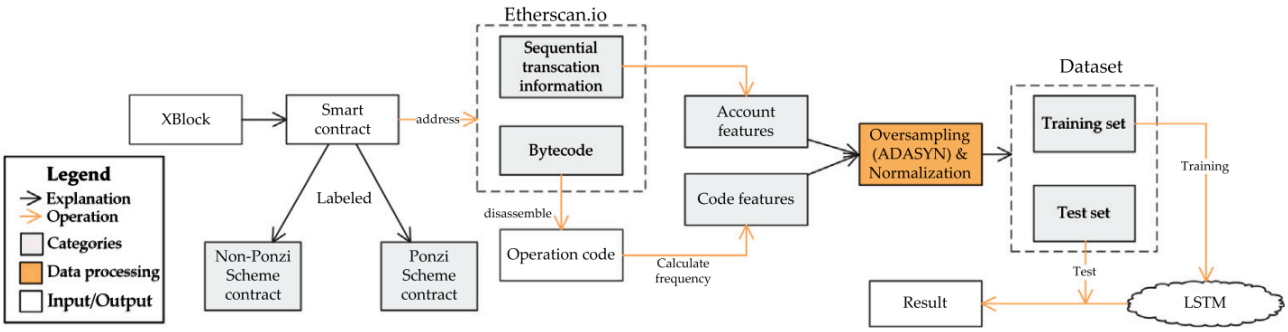


Figure 38: Overview of TTPS's Architecture [32].

Almakhour *et al.* [33] explored the security verification of composite SCs on the Ethereum blockchain. These contracts, which executed other contracts using external calls to accomplish tasks, presented unique security challenges due to their reliance on external contracts that could be owned by different entities. Their novel approach, depicted in Fig. 39, focused on verifying the security and correctness of composite SCs written in Solidity. It employed finite-state machine models and model-checking methods for modeling and verifying these contracts. The approach consisted of three phases: modeling, formal specification, and verification, and involved four principal components: Solidity source code, FSM, Properties, and a Symbolic Model checker.

They considered seven security properties and context-dependent security issues in composite SCs. Their verification method was two-pronged: the first type, referred to as 'standard properties', applied generic verification to all SCs, while the second type focused on 'specific properties' that varied from one contract to another. All properties were expressed using computation tree logic formulae, and the nuXmv symbolic model checker was employed for verification against these properties. This approach was validated using a variety of Solidity SCs. The results from implementing different sets of Solidity SCs demonstrated the efficacy of their proposed approach in verifying security properties within the context of composite SCs, especially in collaborative environments based on the Ethereum blockchain.

Advancing in 2023, Chen *et al.* [34] addressed the problem of SC vulnerability detection at the function level by constructing a novel semantic graph (SG) for each function and learning the SGs using graph convolutional networks (GCNs) with residual blocks and edge attention. Their method for SC vulnerability detection, as depicted in Fig. 40, involves three stages and focuses on creating a Semantic Graph (SG) for each SC function.

- Stage 1: Involved in the creation of an SG for each SC function.
- Stage 2: They introduced a model called EA-RGCN to learn both the content and semantic features of the code from the SG. EA-RGCN comprised three parts: node and edge feature representation using word2vec, code content feature learning through the Relational Graph Convolutional Network (RGCN) module, and semantic feature learning with the Edge Attention (EA) module.
- Stage 3: The security of the SC function was validated using a classifier, which took the concatenated

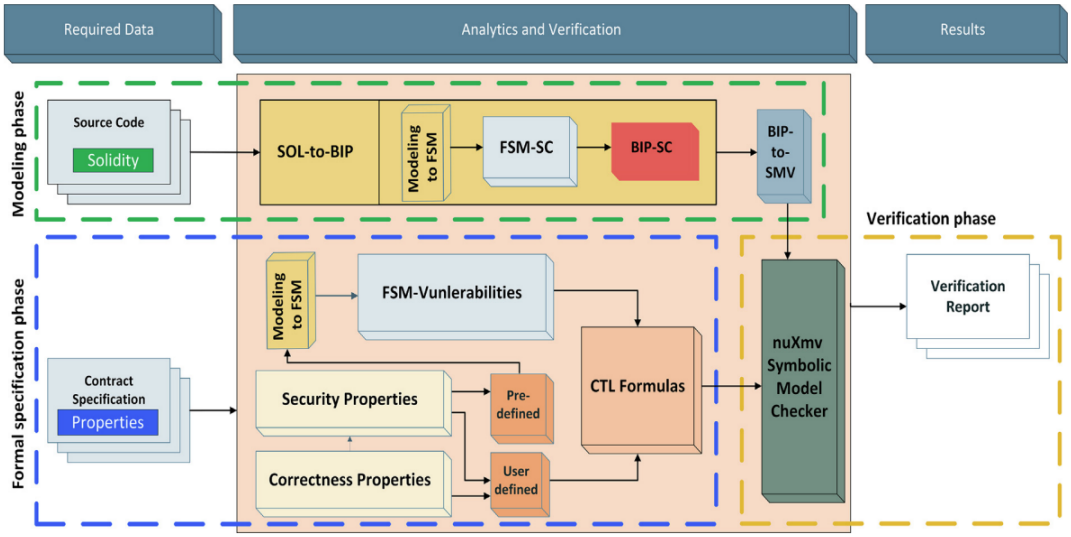


Figure 39: Overview of FSM's Architecture [33].

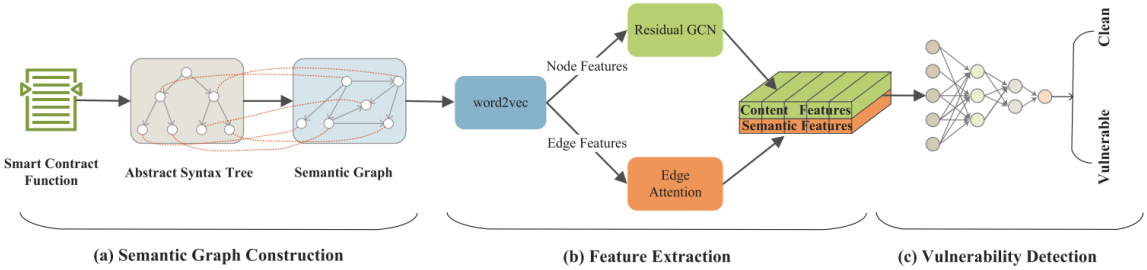


Figure 40: Overview of EA-RGCN's Architecture [34].

code content and semantic features as input.

The approach aimed to capture both local code content features and global semantic features of the SC code. The node features were represented as vectors using word2vec, and the edge features as the average feature of two adjacent nodes. RGCN learned code content features from the SG, covering syntax and some semantic information, while the EA module extracted semantic features by capturing graph structure features.

Fig. 41 illustrates the overall architecture of EA-RGCN. They conducted experiments on four significant vulnerabilities: arithmetic, re-entrancy, timestamp, and unchecked low calls. The results showed that representing SC code as a graph was more effective than sequence or tree representations. Their SG method characterized more code information than existing graph-based representations. Both RGCN and EA modules significantly contributed to vulnerability detection, leading to superior performance in terms of accuracy, precision, recall, F1-score, and ROC-AUC compared to state-of-the-art conventional methods and deep learning-based approaches.

Furthermore in 2023, Liu *et al.* [35] focused on the problem of locating vulnerable functions within SCs. To address this, they constructed a Multi-Relational Nested contract Graph (MRNG), effectively capturing the syntactic and semantic information in SC code, including data and instruction relationships. An MRNG represented a SC with nodes signifying functions and edges illustrating calling relationships between these

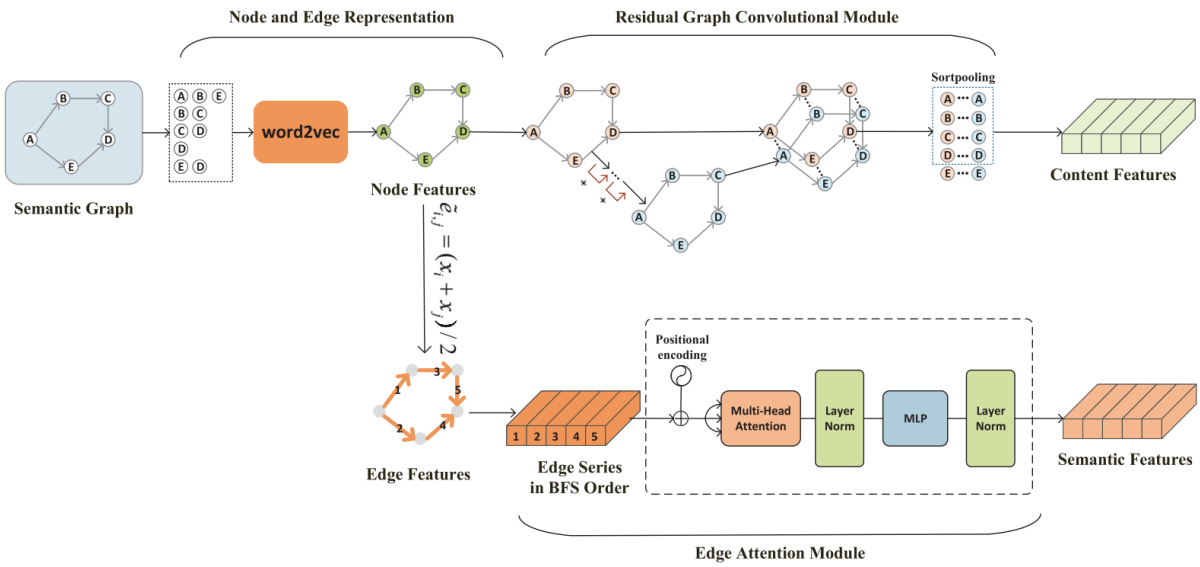


Figure 41: Detailed Overview of the EA-RGCN's Architecture [34].

functions. Additionally, for each function, a Multi-Relational Function Graph (MRFG) was created to characterize the function code, with various edge types denoting different control and data relationships within a function. Their method, outlines in Fig. 42, involves three stages:

- Stage 1: Generation of an MRNG for each SC, where each node represented a function, and each edge described the invocation relationship between functions. Correspondingly, each function was characterized as an MRFG within the MRNG.
- Stage 2: Introduction of the MRN-GCN model to learn semantic and structural features, first from each MRFG and then from the MRNG.
- Stage 3: Utilization of a classifier to classify each node in the MRNG, thereby determining the validity of the corresponding function.

They proposed the Multi-Relational Nested Graph Convolutional Network (MRN-GCN) to process the MRNG. MRN-GCN extracted and aggregated features from each MRFG using an edge-enhanced graph convolution network and a self-attention mechanism. The resultant feature vector was assigned to the corresponding node in the MRNG, creating a new Featured Contract Graph (FCG) for the SC. Further feature extraction from the FCG was achieved through graph convolution, and a feed-forward network with a Sigmoid function was used to locate vulnerable functions.

Experimental results on real-world SC datasets demonstrated that MRN-GCN significantly improved accuracy, precision, recall, and F1-score in locating vulnerable SC functions. Specifically, MRN-GCN achieved 88.96%, 96.43%, and 94.21% accuracy in tasks related to arithmetic, re-entrancy, and timestamp dependency vulnerabilities, respectively. It also attained an average of 94.92% in precision, 91.83% in recall, and 94.21% in F1-score. The MRNG effectively enabled neural networks to comprehend SC code and enhance

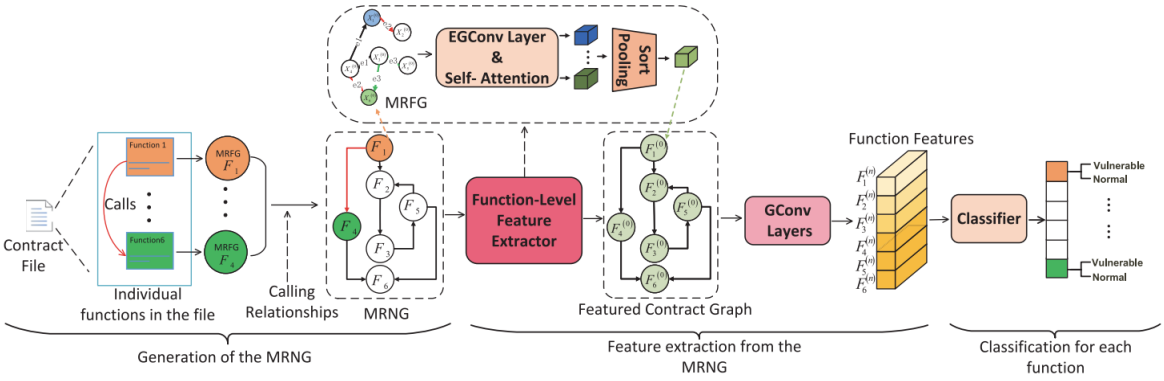


Figure 42: Overview of MRN-GCN's Architecture [35].

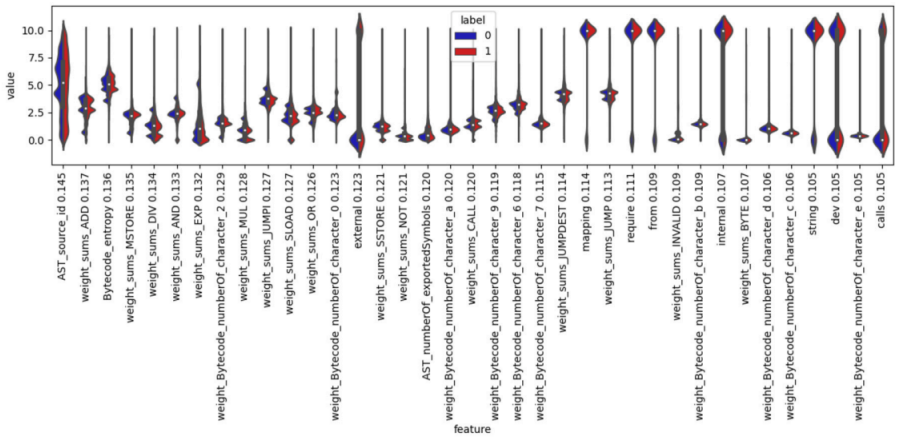


Figure 43: Profiling features [36].

the performance of vulnerable function locating tasks. Furthermore, both the self-attention module and the nested graph convolutional module played significant roles in the success of the MRN-GCN model.

Finally in 2023, Hajihosseinkhani *et al.* [36] introduced the innovative concept of profiling vulnerable SCs. They developed an analyzer named SCsVulLyzer V.1, grounded in their proposed feature taxonomies. Additionally, they presented an enhanced genetic algorithm (GA)-based profiling technique and a classifier. The GA-based profiling technique leverages computational methods inspired by natural evolution to optimize and enhance the profiling process, forming the core model of their work. Fig. 43 showcases the main contribution of their work, illustrating a profile for both vulnerable and secure SCs.

Moreover, they compiled an extensive benchmark dataset comprising 36,670 Solidity source code samples, representing a wide range of examples to validate their method effectively. Comprehensive testing and experimentation demonstrated that their model could accurately identify vulnerable SCs. Compared to traditional and deep neural network (DNN)-based methods, their approach achieved higher precision (0.78), recall (0.77), and F1-score (0.78). Furthermore, the GA-based profiling technique rendered the model highly transparent and explainable.

Starting in 2024, the rising number of SCs on the blockchain brings with it a heightened risk of security

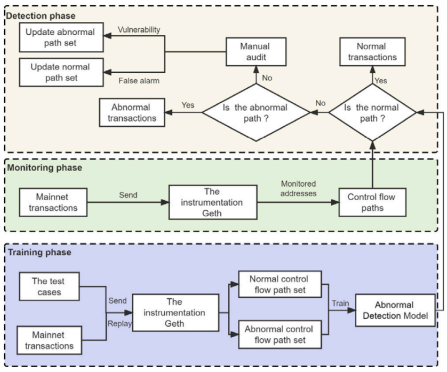


Figure 44: Abnormal detection framework [37].

breaches. Because SCs are immutable once deployed, vulnerabilities in their code become exploitable by hackers, leading to substantial economic losses for users [37].

In response to these challenges, Li *et al.* [37] presented a novel approach to identify abnormal behaviors indicative of SC vulnerabilities using opcode sequences. Normal transactions followed a typical control flow path, while abnormal transactions deviated from this path. By analyzing and verifying these control flow paths, abnormal behaviors in SCs were detected and flagged for further investigation. The model's architecture comprised three phases: training, monitoring, and detection.

They implemented the method by modifying the Ethereum client Geth to capture the control flow paths of SC execution, represented by opcode sequences. These sequences were collected and stored in a database for analysis. For newly developed SCs, a private chain was created to generate normal and abnormal control flow paths. Meanwhile, contracts from the main Ethereum network were monitored using a full node setup. Subsequently, a machine learning model was trained using the collected data to identify abnormal control flow paths. The abnormal detection framework for SCs is presented in Fig. 44.

Based on the experimental results of this research, the abnormal detection framework aimed to improve SC security by identifying and addressing potential vulnerabilities. By capturing abnormal behaviors through control flow path analysis, the model enhanced the security posture of SCs on the blockchain. Despite introducing some additional time overhead, experimental results demonstrated the efficacy of the approach in achieving high detection accuracy with minimal impact on execution time.

Continuing in 2024, Zhen *et al.* [86] addressed the issues of low accuracy in traditional SCs vulnerability detection methods and the insufficient feature extraction in neural network-based approaches for SCs. They introduced an intelligent contract vulnerability identification method, Dual Attention Graph Neural Network (DA-GNN). DA-GNN transformed the operation code sequence of nodes in the SC CFG into a feature matrix of semantic features and relationships between nodes based on five types of proposed instructions.

The architecture of the DA-GNN started by generating attributes of the SC from its bytecode, including the CFG and opcode sequence. It extracted attribute features from both parts and combined them as input to the dual attention module to obtain graph features for identifying vulnerabilities in SCs. The method consisted of two main modules:

1. Attribute Generation Module: This module generated the attributes of the SC from its bytecode, in-

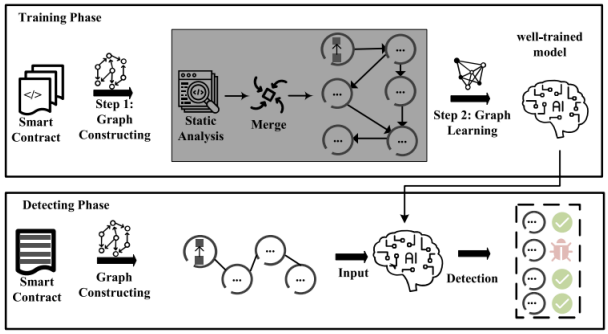


Figure 45: Overview of Fine-grained's Architecture [38].

cluding the CFG and opcode sequence. It separately extracted relationship features between CFG nodes and semantic features of nodes' opcodes.

2. Dual Attention Module: This module utilized the graph attention mechanism to update the attention between nodes. It further extracted attention coefficients from the updated CFG, generating graph features. These features were then fed into the classification module to accomplish the vulnerability detection task.

The dual attention mechanism introduced node semantic features and relationship features between nodes into the GAT to achieve node embedding updates. The updated graph node information was fused through a self-attention mechanism to obtain the graph features. The classification and prediction of vulnerabilities were achieved through the classification module. The method was evaluated on 17,670 real SCs. Experimental results showed that the precision in detecting integer overflow vulnerabilities self-destruct vulnerabilities, and transaction sequence dependency vulnerabilities reached 72.17%, 67.03%, and 73.66%, respectively.

In 2024, to utilize deep learning for fine-grained SC vulnerability detection, Cai *et al.* [38] proposed a security best practices (SBP) based dataset construction approach to address the scarcity of datasets. They also introduced a syntax-sensitive graph neural network to tackle the challenge of heterogeneous code feature learning. The dataset construction approach was motivated by the insight that SC code fragments guarded by SBP might contain vulnerabilities in their original unguarded form. Thus, they located and stripped SBP from the SC code to recover its original vulnerable form and performed sample labeling. Fig. 45 presents an overview of the Fine-grained Architecture.

To address the heterogeneity between tree-structured syntax features embodied in the AST and graph-structured semantic features reflected by relations between statements, they proposed a code graph whose nodes are each statement's AST subtree. This was enhanced by a syntax-sensitive graph neural network utilizing a child-sum tree-LSTM cell to learn these heterogeneous features for fine-grained SC vulnerability detection. Finally, They compared their approach with three state-of-the-art deep learning-based methods that only supported contract-level vulnerability detection and two popular static analysis-based methods that supported fine detection granularity. The experimental results showed that their approach outperformed the baselines at both coarse and fine granularities.

Moreover in 2024, Sendner *et al.* [87] investigated the gap between the effectiveness of existing security

scanners and the persistent vulnerabilities in practice. They compiled four distinct datasets for their analysis. The first dataset comprised 77,219 source codes extracted directly from the blockchain, while the second included over 4 million bytecodes obtained from Ethereum Mainnet and testnets. Additionally, the other two datasets consisted of nearly 14,000 manually annotated smart contracts and 373 smart contracts verified through audits, providing a solid foundation for a rigorous ground truth analysis on both bytecode and source code.

Utilizing the unlabeled datasets, they conducted a quantitative evaluation of 18 vulnerability scanners. Their analysis, encompassing millions of smart contracts with both source codes and bytecodes, as well as manually labeled ones, highlighted a clear finding: there is significant room for improvement in the field of smart contract security. This study revealed the reasons for poor performance and emphasized that the current state of the art for smart contract security falls short in effectively addressing open problems. It underscores that the challenge of effectively detecting vulnerabilities remains a significant and unresolved issue.

2.2 Synthesis

This sub-chapter explores the evolution of SCs security from 2016 to 2024, a period characterized by significant innovation and varying pace of development. The field witnessed a transformative shift in 2018 with the introduction of bytecode analysis, which greatly improved security methods. This advancement, utilizing CFG-based approaches, marked a crucial step in ensuring the integrity of blockchain transactions.

Despite a slowdown in momentum from 2019 to 2021, this period still saw continued efforts in vulnerability detection, with the development of innovative techniques and tools that contributed to the security and stability of blockchain applications. The pace of innovation picked up again in 2022, signaling a resurgence in SCs security research. By 2023 and continuing to the present, the field has returned to a peak of innovation, characterized by the introduction of various new tools and methodologies that have further enhanced the safety and reliability of blockchain technology.

Figures 46 and 47 in this sub-chapter provide a visual representation of this journey. Fig. 46 offers a chronological overview of significant works in the field, while Fig. 47 presents an extensive taxonomy of previous models, detailing their development and interrelationships.

Alongside these advancements, we identify and address several shortcomings in the field, including:

- Lack of Complete Coverage: Research often targets specific vulnerability types or patterns in SCs, potentially overlooking the full array of vulnerabilities. This narrow scope might leave certain threats unaddressed.

- False Positives and Negatives: Tools and methods under discussion are prone to inaccuracies, such as false positives—erroneously marking safe contracts as vulnerable—and false negatives—overlooking actual vulnerabilities. Striking a balance between precision and recall is an ongoing challenge.

- Scalability Challenges: Some approaches struggle with scalability when applied to numerous SCs, a pressing concern as the Ethereum blockchain continues to expand.

- Limited Dataset Diversity: The success of certain detection methods is heavily contingent on the quality and variety of the datasets used for training and testing. Poor or biased datasets can undermine detection accuracy.

- Evolving SC Language: With the continuous evolution of the Solidity language and Ethereum platform, existing tools and methods risk becoming obsolete and less effective against vulnerabilities in newer Solidity versions or different programming languages.

- Unknown Vulnerabilities: Current research primarily targets known vulnerabilities, potentially neglecting emerging or novel threats that could compromise SCs.

- Complexity of the Ethereum Ecosystem: The dynamic nature of the Ethereum ecosystem poses a significant challenge in keeping detection tools and methodologies abreast of the latest trends, coding practices, and evolving attack strategies.

- Dependency on Accessible Source Code: Some research relies on the availability of SC source code for analysis, thereby limiting their applicability to contracts with public source code and excluding proprietary or closed-source contracts.

- Reliance on Historical Data: Emphasizing historical data and past vulnerabilities can lead to a focus on already resolved issues, while newer vulnerabilities might not receive adequate attention.

Through comprehensive analysis, the goal is not only to underscore these existing gaps but also to motivate enhanced efforts toward addressing critical issues such as the lack of complete coverage, the prevalence of false positives and negatives, scalability challenges, unknown vulnerabilities, the limited diversity of datasets, and reliance on historical data.

2.3 Concluding Remarks

This chapter provides a comprehensive overview of previous studies and methodologies, specifically focusing on the contributions referenced in Chapter 1 as CONT2 and CONT3. Throughout this analysis, the aim is to identify the limitations of prior approaches, enumerate these deficiencies, and initiate the process by selecting a few for examination in this work. In the subsequent chapter 3, we will delve into the details of the proposed model.

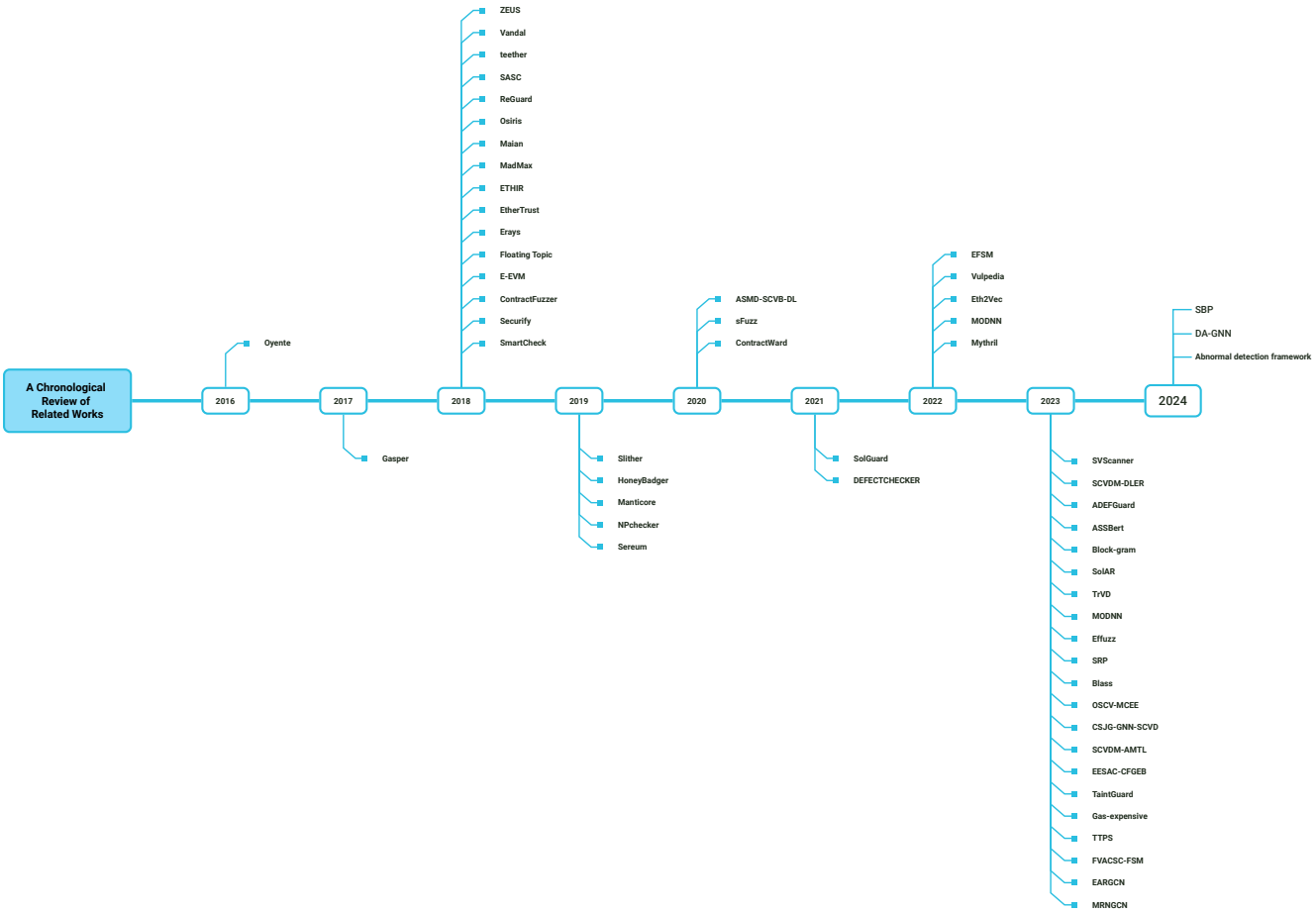


Figure 46: Chronological Overview of Work Development: A Year-by-Year Evolution Analysis

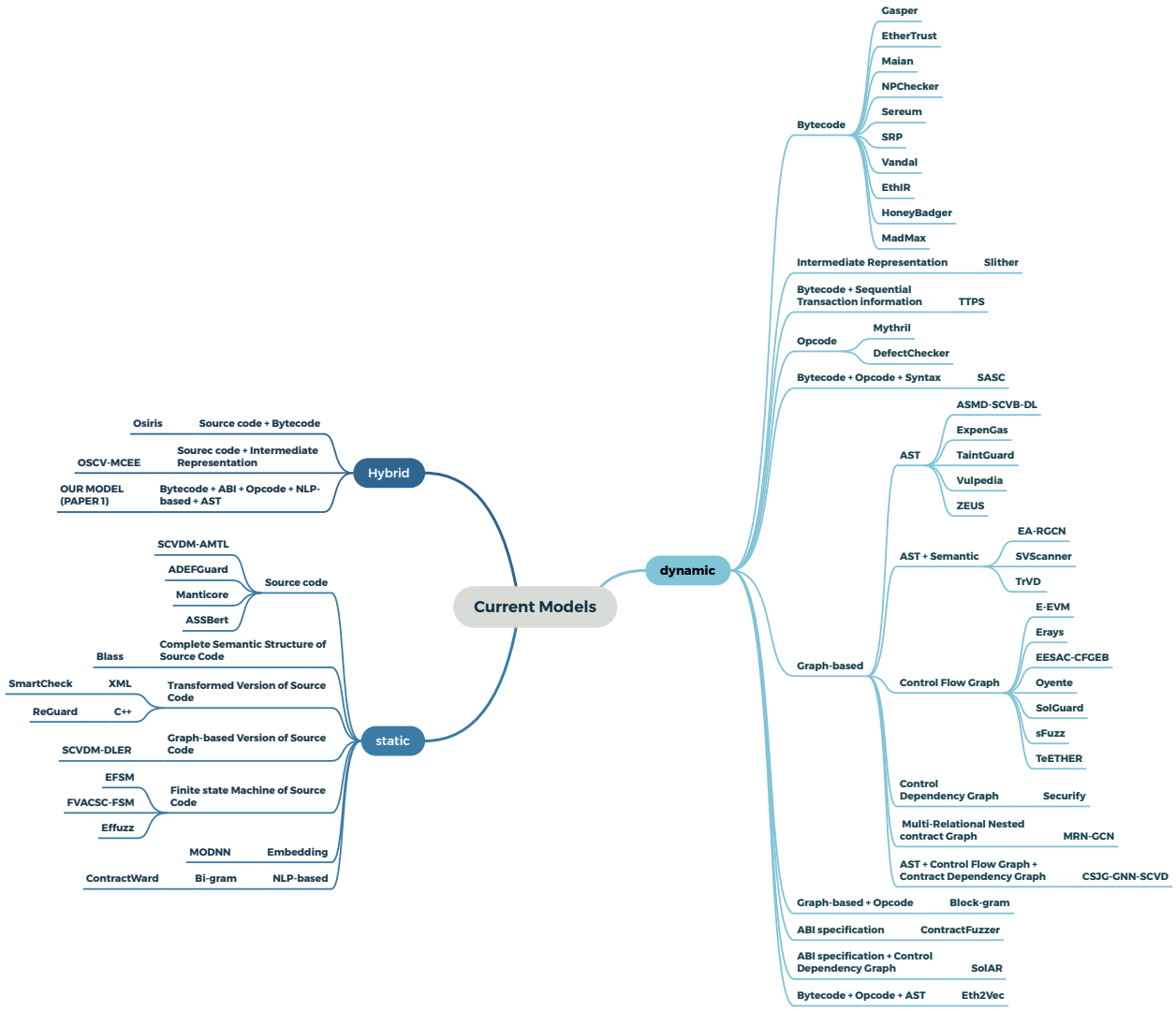


Figure 47: Taxonomy of Previous Models

3 Model Architecture

This chapter presents a comprehensive and innovative methodology for analyzing and detecting vulnerabilities in SCs, delineated into seven distinct phases. The methodology is initiated with 'Data Processing' (Chapter 3.1), where the initial groundwork for the analytical framework is set. Subsequently, the transition is made to the 'Best Compatible Compiler' (Chapter 3.2), where the gathered SCs are systematically compiled, preparing them for detailed scrutiny. The analysis progresses to 'Feature Extraction' (Chapter 3.3), where an analyzer is employed to identify and isolate critical attributes from the SCs.
Moving forward, 'Feature Selection' (Chapter 3.4) involves strategically filtering the dataset to focus solely on features with significant impact. The subsequent 'Profiling' phase (Chapter 3.5) involves a thorough examination of profiling techniques. This examination aims to assess the effectiveness of these techniques and explore potential improvements to propose an enhanced version for identifying SC vulnerabilities. This methodology extends to the 'Classification' phase (Chapter 3.6). Here, a basic classification algorithm is applied to prepare the output for evaluating our proposed profiling method through a classification process. The evaluation process is detailed in the 'Evaluation Criteria' (Chapter 3.7), which outlines the standards for assessing the proposed model.

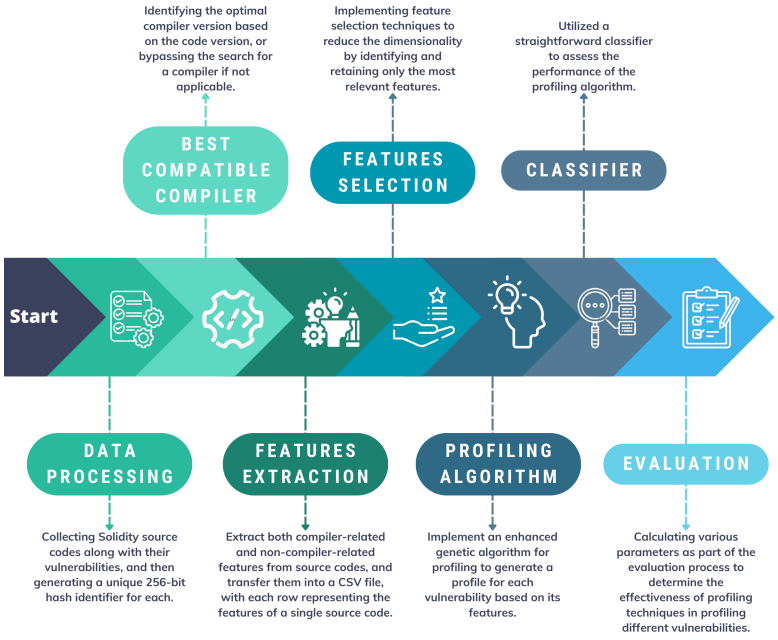


Figure 48: Proposed Model Architecture

3.1 Data Processing

The first step in the proposed model architecture, depicted in Fig. 48, involves collecting and processing data. In this step, we introduce a new dataset, BCCC-SCsVul-2024, and provide detailed information about it in chapter 4.2.

3.2 Best Compatible Compiler

A SC compiler plays a crucial role in developing and deploying SCs on blockchain platforms, such as Ethereum [60]. The compiler transforms high-level, human-readable code into low-level code or bytecode that can run on the EVM [44]. One critical aspect of this process is ensuring compatibility between the compiler version and the SC code. In Solidity, the programming language used for writing SCs for Ethereum, the version of the compiler intended for the code is usually specified at the beginning of the code [10, 80]. For instance, in the example provided 3.2, the directive 'pragma solidity `^0.4.24`;' explicitly states that the contract is written for version 0.4.24 of the Solidity compiler. This specification helps prevent inconsistencies and ensures that the code compiles and runs as expected, considering the changes and improvements that come with different compiler versions.

```
/**
 * @title ERC20Basic
 * @dev Simpler version of ERC20 interface
 * See https://github.com/ethereum/EIPs/issues/179
 */
contract ERC20Basic {
  function totalSupply() public view returns (uint256);
  function balanceOf(address _who) public view returns (uint256);
  function transfer(address _to, uint256 _value) public returns (bool);
  event Transfer(address indexed from, address indexed to, uint256 value);
}
...
contract IEtherToken is ERC20 {
    function deposit() public payable;
    function withdraw(uint256 amount) public;
}

contract MultiChanger {
    using SafeMath for uint256;
    using CheckedERC20 for ERC20;
    using ExternalCall for address;
}
contract check {
    uint validSender;
    constructor() public {owner = msg.sender;}
    function destroy() public {
        assert(msg.sender == owner);
        selfdestruct(this);
    }
}
```

Listing 1: Solidity code example

3.3 Feature Extraction

This sub-chapter explores the crucial feature extraction process as it applies to the analysis of SCs. Initially, it offers a thorough overview of the background and existing methodologies in feature extraction. Building upon this foundation, this sub-chapter introduces SCsVulLyzer-V2, an updated iteration of our previous analyzer proposed in [36].

3.3.1 Feature Extraction Background

Feature extraction is pivotal in analyzing SCs, pinpointing key attributes essential for predicting their behavior and identifying security vulnerabilities [36]. In this sub-chapter we explore various features, including

AST, ABI specifications, bytecode, opcode, and NLP-based features, along with their extraction methods. We also examine existing research on how these features contribute to detecting security vulnerabilities.

- **AST**

  The AST is a tree representation of the abstract syntactic structure of source code, where each node represents a construct occurring in the code [88]. It plays a pivotal role as a fundamental data structure for representing the structural syntax of source code in formal languages [89]. It works by translating each code element into a node within the tree-like structure. The connections between these nodes, known as edges, illustrate the relationships between different code elements [90]. ASTs find a broad range of applications, extending from the realms of compilers and interpreters to static code analysis tools, as highlighted in [88]. More recently, an innovative application of AST was introduced by Hu *et al.* in [91]. They developed a code generator that uses the AST of a SC to assign security types based on predetermined rules. This novel application of the AST allows the generator to detect security vulnerabilities. It also assigns appropriate security types to various code segments, thus enhancing the security of SCs. By identifying and addressing these vulnerabilities at the code level, this method underscores the vital role of ASTs in the development of secure systems.

  In conclusion, the use of ASTs for representing the structural syntax of source code is vital for improving security. This is exemplified by the recent advancements made by Hu *et al.* in [91], where ASTs were employed to enhance the security of SCs. Their approach facilitates the early detection and correction of potential security weaknesses. This emphasizes the importance of ASTs in the development of secure systems. The ongoing exploration of AST capabilities is set to play a significant role in establishing new standards for system security and integrity.

- **ABI**

  The ABI is a cornerstone within the Ethereum ecosystem, essential for enabling contracts to interact with external entities. It also facilitates contract-to-contract communication, as elaborated in [92]. Within Ethereum, the ABI ensures that data is correctly encoded according to its type, adhering to the specifications defined in the ABI [93]. This encoding process is critical for ensuring compatibility and interaction among various SCs and applications within the network [94].

  Samreen *et al.* [44] introduced a cutting-edge methodology for testing SCs in their recent work. Their approach involves generating the ABI for a contract post-compilation. This significantly aids in uncovering vulnerabilities, particularly focusing on Re-entrancy. Re-entrancy, if undetected and unaddressed, poses a significant threat to the integrity of the entire Ethereum ecosystem.

  In summary, the ABI is not just a fundamental part of the Ethereum structure; its correct implementation is crucial for the security and functionality of SCs. The testing technique introduced by Samreen *et al.* [44] represents a significant advancement in SC testing. It emphasizes the crucial role of thorough testing in identifying and mitigating potential security risks. This development further emphasizes the critical role of the ABI in preserving the stability and resilience of the Ethereum ecosystem.

- **Bytecode**

  Bytecode is a unique type of object code that virtual machines (VMs) can interpret and convert into

binary machine code, enabling it to be executed by a computer's hardware processor [95]. Its standout feature is the enhancement of code portability across different platforms, as the VM can read and adapt bytecode to suit the specific requirements of any target platform [96].

Recent academic advancements explored notable utilization of bytecode in the fields of deep learning and system security analysis, as demonstrated in studies by Gupta *et al.* [97] and Vivar *et al.* [98]. These studies leverage the inherent interpretability and adaptability of bytecode to develop new methods for analyzing and fortifying systems.

Gupta *et al.* [97] implemented three types of deep learning algorithms, including LSTM, GRU, and ANN. By comparing precision, recall, and F1 score, they found that LSTM, with bytecode input, performs well in the context of vulnerability detection in SCs. Furthermore, Vivar *et al.* [98] introduced ESAF, which uses the bytecode version of SC as input and passes it through multiple detection tools. Based on the majority of probabilities, it decides whether the SC is vulnerable or secure.

In essence, bytecode plays a pivotal role due to its ability to enhance portability and interpretability of code across various platforms. Its recent application in research highlights its significance as an important characteristic in ongoing system analysis and security techniques. This solidifies its value in advancing security systems.

- **Opcode**

  In the computing environment, an opcode, or operation code, is a crucial part of machine language instruction, determining the specific operation to be carried out by the system [99]. Each opcode represents a different command, usually occupying 1 byte. This byte-sized structure allows for 256 unique opcodes, each corresponding to a distinct operation [100, 101].

  A recent development in malware detection methodology by Gupta *et al.* [97] brought the importance of opcodes to the forefront. Their research employed opcodes as a crucial input for a deep learning algorithm, in order to extract patterns of opcodes in malware. It was proven that opcode is highly effective in identifying vulnerabilities, showcasing the significance of opcode analysis in security threat detection.

  Furthermore, Huang *et al.* [102] proposed a deep learning model-based approach that combines the characteristics of contract source codes and Opcodes for vulnerability detection. Specifically, the expert pattern features are retrieved from the source codes, and the contextual features are extracted based on Opcodes. Their experiment findings showed that their method beats the state-of-the-art methods and achieves impressive F1-Score and accuracy using the real-world dataset of Ethereum SCs addressing Re-entrancy vulnerability.

  In summary, opcode analysis offers valuable insights into the workings and potential risks associated with executing a SC. These opcode features can be a reliable measure of the contract's complexity and security aspects. Recent academic research highlights the crucial role of opcode analysis in understanding and securing SCs, as each opcode represents a different command effectively. This emphasizes the importance of opcode analysis in navigating the intricate aspects of SCs and enhancing their security.

- **Natural Language Processing (NLP)-based Features**
  NLP-based features, lying at the convergence of computer science and linguistics, play a pivotal role in handling anomaly detection in text-based systems such as SCs [103]. Key techniques in NLP, such as Word2Vec and FastText, are instrumental for generating word embeddings, which are vector representations of words, contributing significantly to the field as detailed in [104].

  Innovative applications of these NLP techniques demonstrated in the work of Qian *et al.* [42] and Zhang *et al.* [105]. Qian *et al.* [42] employed Word2Vec, a NN model, to effectively learn word embeddings. They utilized the output of Word2Vec to train a deep NN model for analyzing patterns of anomalies in SCs. Zhang *et al.* [105] introduced FastText, an open-source library that enables learning text representations and classifiers. It provides an efficient approach to analyze textual data in SCs. They developed a hybrid model combining FastText and Word2Vec, which showed an enhanced precision and efficiency in detecting vulnerabilities in SCs. This hybrid model underlines the capability of NLP techniques to interpret the relationships between different contract components, which is crucial for identifying vulnerabilities.

  In conclusion, applying techniques like Word2Vec and FastText has been instrumental in advancing NLP, providing innovative ways for automated vulnerability detection in SCs. Prior research highlights the potential advantages of combining these methods to improve vulnerability detection, indicating a viable path for SC and its ecosystem security.

3.3.2 Feature Extraction Synthesis

Feature extraction is crucial in analyzing and identifying vulnerabilities in SCs. Feature categories such as AST, ABI specifications, Bytecode, Opcodes, and NLP-based are vital for exploring SC characteristics.
Previous studies [89, 106, 92, 97, 104, 105] showed that these categories offer valuable insights into SCs' complex structures and behaviors. The AST visually maps the source code, ABI specifications manage data encoding and interoperability, and Bytecode and Opcodes reveal detailed information about within execution. Moreover, NLP-based features enhance the understanding of semantic relationships within the contract code, which is vital for spotting security vulnerabilities.
Studies suggested that for thorough analysis and precise vulnerability detection in SCs, it is essential to extract and analyze these feature sets [98, 107, 106]. Each type of feature provides unique and crucial insights, significantly aiding the vulnerability detection process [42, 87]. As the domain evolves, the systematic extraction and analysis of these features will continue to be vital in improving the security and dependability of SCs [44, 97].

3.3.3 Proposed Feature Extraction BCCC-SCsVulLyzer

The comprehensive analysis in sub-chapter 3.3.2 highlights the need for a thorough and precise approach to vulnerability detection in SCs. This requires extracting and examining features across a diverse range of categories. The extensive literature review identifies five predominant categories: AST, ABI, Bytecode, Opcode, and features derived using NLP techniques. Each of these categories contributes unique and vital

insights, significantly augmenting the effectiveness of the vulnerability detection process. For a concise overview, Table 1 outlines the feature categories reviewed in previous works.

In the bytecode category, this study introduces a new feature called bytecode entropy, which calculates the entropy of bytecode. This feature is proposed because the entropy formula effectively measures the randomness and information content of data, offering a quantifiable metric for uncertainty. This is crucial for applications in cryptography, data compression, and anomaly detection, where higher entropy indicates more randomness and less predictability.

Given a bytecode represented as a hexadecimal string, the function first converts it to a byte array. Then, it calculates the frequency of each unique byte in the array. Finally, it computes the Shannon entropy of the byte array. Mathematically, the Shannon entropy $H(X)$ of a discrete random variable X with possible values $x_1, x_2, \ldots, x_n$ and corresponding probabilities $p_1, p_2, \ldots, p_n$ is given by:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

In the context of the function:

-Let X be the set of bytes in the byte array.

-Let $p(x_i)$ be the probability of byte x_i occurring in the byte array.

-The probability $p(x_i)$ is calculated as the frequency of x_i divided by the total number of bytes in the array.

So, for a byte array $\mathbf{d}$ of length N, the entropy $H(\mathbf{d})$ is:

$$H(\mathbf{d}) = -\sum_{x \in \mathbf{d}} \left(\frac{\text{count}(x)}{N} \right) \log_2 \left(\frac{\text{count}(x)}{N} \right)$$

where: - count(x) is the number of occurrences of byte x in the byte array. - N is the total number of bytes in the byte array.

This formula quantifies the amount of uncertainty or randomness in the byte array.

Table 1: Summary of features

Paper	**Features**	**Description**
[91]	AST	Generating security type assignments based on proposed rules for SCs and providing a tree representation of the abstract syntactic structure of source code.
[44]	ABI specifications	Provides a standard way for outside interaction and contract-to-contract interaction in the Ethereum ecosystem by encoding data according to its type.
[97, 98]	Byte code	Form of computer object code that an interpreter converts into binary machine code.
[97]	Opcode	Represents a specific operation in the environment, with 256 possible opcodes.
[105, 42]	NLP-based	Involving the conversion of raw text into numerical representations that machine learning algorithms can more easily process.

To generally categorize these features, we assess their activation time—whether they are evaluated before or after compilation—and classify them into two main categories: compiler-based and non-compiler-based. This classification enables us to maximize the breadth of feature extraction. Compiler-based features pri-

marily include elements that are processed after the source code passes through the compiler, such as ABI and AST. On the other hand, non-compiler-based features are derived by synthesizing insights from existing NLP-based features, which have been explored and refined in prior research. We adapt these features to our specific context, focusing on keywords pertinent to SCs.

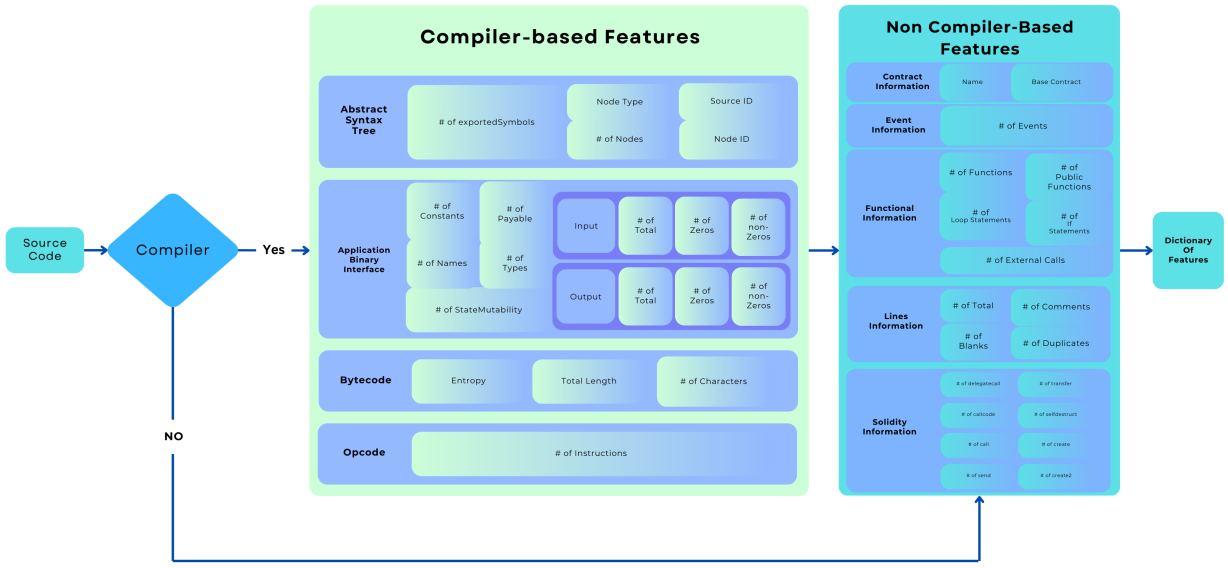


Figure 49: Architecture: BCCC-SCsVulLyzer

Moreover, this research introduces innovative features under three newly defined categories: Contract Information, Source Code Information, and Solidity Information. These categories are specifically designed to concentrate on the quantitative elements of the code, encompassing metrics like the count of diverse functions, statements, loops, and lines. The integration of these novel features is intended to enrich our analysis, offering a more detailed and thorough perspective. This enhancement is pivotal in advancing the effectiveness of the proposed SCs vulnerability detection framework. This study introduces BCCC-SCsVulLyzer, a new analyzer that combines both compiler-based and non-compiler-based features. This integration enhances the methodology's standing in SC vulnerability detection within the field.

Fig. 49 presents a detailed diagram illustrating the BCCC-SCsVulLyzer process. This comprehensive illustration outlines the steps from obtaining the source code to extracting and passing the dictionary of all features. Moreover Fig. 50 presents a detailed taxonomy of all extracted feature categories used in BCCC-SCsVulLyzer. Detailed descriptions of each feature, along with their categories, are provided in Tables 2 and 3.

3.4 Feature Selection

Through the features extracted by BCCC-SCsVulLyzer, a substantial set of 240 features is obtained, which is more than sufficient. To refine this extensive feature set and identify the most impactful ones, this subchapter proposes a feature selection process that employs an ensemble model integrating three distinct feature selection methods. This approach enables leveraging the strengths of each method, ensuring a comprehensive

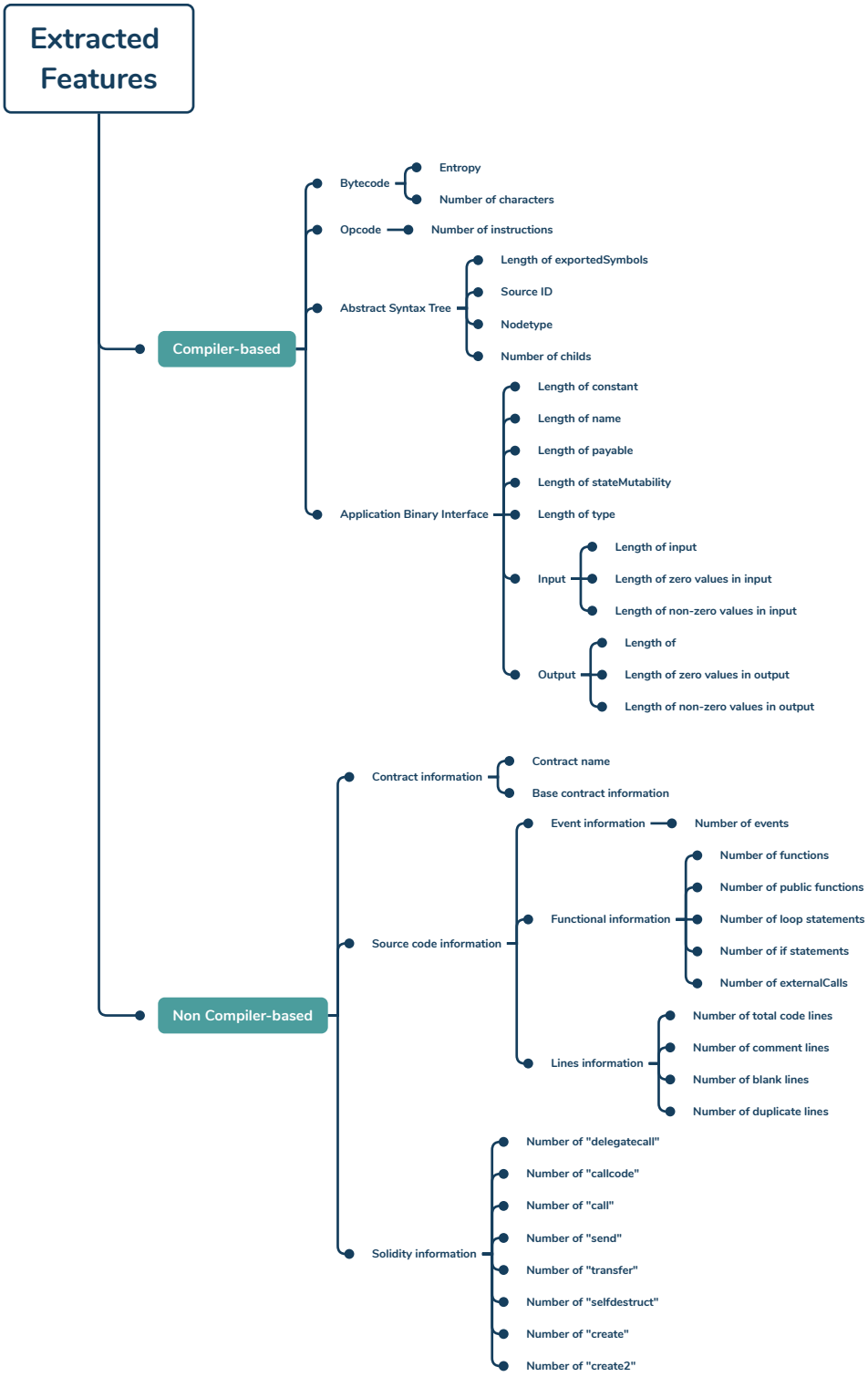


Figure 50: Taxonomy: Extracted Features Categories

Table 2: Details of Compiler-Based Features Extracted

Category	Measure	abbreviation	Description
Bytecode	Entropy	BC F1	Measure of randomness or complexity in the program's compiled bytecode.
	Number of Characters	BC F2 [Character name]	Count of unique character symbols in the program's compiled bytecode.
Opcode	Number of Instructions	OC F1 [Character name]	Total count of operational code (opcode) instructions, representing the execution steps.
AST	Length of exportedSymbols	AST F1	Size or complexity measure of the exported symbols in the AST.
	Source ID	AST F2	Unique identifier for specific source code elements within the AST.
	NodeType	AST F3	Category or class of a node, indicating the syntax construct it represents.
	Number of Children	AST F4	Count of immediate descendant nodes within the AST structure.
ABI	Length of Constant	ABI F1	Fixed size of a constant value in the ABI data representation.
	Length of Name	ABI F2	Number of characters or bytes for an identifier or name in the ABI.
	Length of Payable	ABI F3	Specifies if a function can receive Ether transactions (not a length measure).
	Length of StateMutability	ABI F4	Number of characters describing the mutability type of a contract's state.
	Length of Type	ABI F5	Number of characters or bytes specifying an ABI element's type.
	Length of Input	ABI F6	Number of parameters or size of data for a function or method call.
	Length of Zero Values in Input	ABI F7	Count or size of parameters set to zero in a function or method call.
	Length of Non-Zero Values in Input	ABI F8	Count or size of non-zero value parameters in a function or method call.
	Length of Output	ABI F9	Number of return values or data size returned by a function or method.
	Length of Zero Values in Output	ABI F10	Count or size of return values set to zero by a function or method.
	Length of Non-Zero Values in Output	ABI F11	Count or size of non-zero value return values by a function or method.

Table 3: Details of Non-Compiler-Based Features Extracted

Category	Measure	abbreviation	Description
Contract Information	Contract Name	CI F1	Identifier or title for a specific SC in a blockchain network.
	Base Contract Information	CI F2	Details of a parent or primary contract from which a SC inherits features.
Source Code Information	Number of Events	EI F1	Count of distinct occurrences or actions within a system or application.
	Number of Functions	FI F1	Count of distinct operations or procedures in a specific context.
	Number of Public Functions	FI F2	Count of externally accessible functions in a module or contract.
	Number of Loop Statements	FI F3, FI F4	Count of repetitive control structures (e.g., ”while”, ”for”) in a code block.
	Number of If Statements	FI F5	Count of conditional statements for decision-making in a code block.
	Number of External Calls	FI F6	Count of calls to external contracts or services in blockchain development.
	Total Code Lines	LI F1	Total lines of code, including comments and whitespace.
	Comment Lines	LI F2	Lines containing comments or explanations in a codebase.
	Blank Lines	LI F3	Empty lines without code or comments in a codebase.
	Duplicate Lines	LI F4	Identical lines of code appearing more than once in a codebase.
Solidity Information	”delegatecall” Count	SI F1	Occurrences of the ”delegatecall” instruction in SC code.
	”callcode” Count	SI F2	Occurrences of the ”callcode” instruction in SC code.
	”call” Count	SI F3	Occurrences of the ”call” instruction in SC code.
	”send” Count	SI F4	Uses of the ”send” function to transfer funds in SC code.
	”transfer” Count	SI F5	Uses of the ”transfer” function to transfer funds in SC code.
	”selfdestruct” Count	SI F6	Occurrences of ”selfdestruct” to remove contracts from the blockchain.
	”create” Count	SI F7	Uses of ”create” to deploy new contract instances on the blockchain.
	”create2” Count	SI F8	Uses of ”create2” to deploy new contract instances on the blockchain.

and effective selection of the best features for the proposed profiling model.

The first feature selection algorithm, the Fisher Score, detailed in 1, is utilized to pinpoint the most discriminative features among various classes [108]. The method involves computing the mean of each feature across all samples and then separating the features by class. For each class, it calculates the within-class scatter by measuring the squared differences between each class sample and the class mean [109]. It also calculates the between-class scatter by assessing the squared differences between the class means and the total mean, scaled by the class size. The Fisher Scores are then obtained by dividing the between-class scatter by the within-class scatter for each feature [110, 111].

The Fisher Score algorithm has several advantages, particularly its effectiveness in scenarios where the goal is to maximize class separability, making it ideal for classification tasks. By focusing on the ratio of between-class to within-class variance, the Fisher Score can identify features that contribute most to distinguishing between classes, thus facilitating more accurate and efficient classification models [110]. The method is also relatively simple to understand and apply, allowing practitioners to quickly identify relevant features without extensive preprocessing or parameter tuning [109, 111]. Moreover, by reducing the dimensionality of the data by selecting the most relevant features, the Fisher Score can help in alleviating issues related to high dimensionality, such as overfitting and computational inefficiency. Furthermore, the Fisher Score is model-independent, meaning it does not assume any specific model structure, making it versatile and broadly applicable across different types of data and classification algorithms. However, its effectiveness is particularly pronounced in linearly separable problems, where the separation between different classes can be captured through linear combinations of features [111].

Algorithm 1 Feature Selection: Fisher Score

Require: X: matrix of features, y: target vector
Ensure: *scores*: Fisher Scores for each feature
  $mean\_total \leftarrow \text{np.mean}(X, axis = 0)$
  $classes \leftarrow \text{np.unique}(y)$
  $S\_W \leftarrow 0$ ▷ Within class scatter
  $S\_B \leftarrow 0$ ▷ Between class scatter
  for c **in** *classes* **do**
    $X\_c \leftarrow X[y == c]$
    $mean\_c \leftarrow \text{np.mean}(X\_c, axis = 0)$
    $S\_W \leftarrow S\_W + \text{np.sum}((X\_c - mean\_c)^2, axis = 0)$
    $n\_c \leftarrow X\_c.shape[0]$
    $mean\_diff \leftarrow (mean\_c - mean\_total)^2$
    $S\_B \leftarrow S\_B + n\_c * mean\_diff$
  end for
  $scores \leftarrow S\_B / S\_W$
  return *scores*

The second algorithm, described in 2, is the Information Gain algorithm. It plays a crucial role in constructing decision trees, serving as an advanced method for feature selection [112]. It begins by evaluating the dataset's overall entropy, which measures the initial state of disorder or uncertainty [113]. As it cycles through each dataset attribute, the algorithm calculates the reduction in entropy, known as information gain. This reduction results from segmenting the data according to the distinct values of each attribute. This systematic approach

enables the identification of the attribute that most significantly decreases entropy [114]. By dividing the dataset into subsets based on the values of each attribute and then calculating each subset's entropy, the algorithm can ascertain the total remaining uncertainty, or conditional entropy, following the split. The difference between the original entropy and this conditional entropy yields the information gain associated with each attribute [113, 115].

The primary advantage of information gain is that it ensures, at each decision point, the systematic reduction of uncertainty with every split in the decision tree. This not only streamlines the decision-making process by focusing on the most pertinent features but also aids in constructing more efficient and interpretable trees [112, 115]. The Information Gain algorithm is highly esteemed for its capability to simplify intricate data, thereby lowering the risk of overfitting and improving the decision tree's performance on unfamiliar data. However, it may disregard feature interactions unrelated to entropy reduction [114].

Algorithm 2 Feature Selection: Information Gain

Require: D: Dataset with classes $C_1, C_2, \ldots, C_n$, F: Set of features
Ensure: A^*: The feature with the highest information gain
  $maxGain \leftarrow 0$ ▷ Maximum information gain
  $A^* \leftarrow NULL$ ▷ Best feature
  $H(D) \leftarrow -\sum_{i=1}^{n} p(C_i) \log_2 p(C_i)$ ▷ Entropy of Dataset
  for A **in** F **do**
    $conditionalEntropy \leftarrow 0$
    for each possible value v of feature A **do**
      Partition D into subsets D_v where $A = v$
      $H(D_v) \leftarrow -\sum_{i=1}^{n} p(C_i|D_v) \log_2 p(C_i|D_v)$
      $conditionalEntropy \leftarrow conditionalEntropy + \frac{|D_v|}{|D|} H(D_v)$
    end for
    $infoGain \leftarrow H(D) - conditionalEntropy$
    if $infoGain > maxGain$ **then**
      $maxGain \leftarrow infoGain$
      $A^* \leftarrow A$
    end if
  end for
  return A^*

The third feature selection algorithm, the Chi-squared (χ^2), discussed in 3, is a statistical method used to evaluate the independence between each feature and the target variable in a dataset [116]. It operates by calculating the Chi-squared statistic between each feature and the target variable, quantifying the degree of dependence between categorical variables. In practice, for each feature, the algorithm computes the expected frequencies of each category under the assumption that the feature and target are independent [117]. It then compares these expected frequencies with the actual observed frequencies, accumulating the discrepancies across all categories to form the Chi-squared score [118, 119]. Features with higher scores are considered more important as they indicate a stronger association with the target variable, suggesting that changes in the feature have significant impacts on the outcome.

The main advantages of the Chi-squared feature selection algorithm are its simplicity and effectiveness in identifying relevant categorical features, especially in classification problems [119, 120]. Focusing on the statistical significance of feature-target relationships helps in reducing the dimensionality of the dataset,

leading to improved model performance and reduced computational costs. Additionally, since it only requires categorical data, it is particularly useful in scenarios where data is naturally discrete or when dealing with count data [116, 117]. However, it is not suitable for continuous variables unless they are binned into categories, which can introduce additional considerations into the feature selection process.

Algorithm 3 Feature Selection: Chi-squared

Require: X: matrix of features, y: target vector
Ensure: $chi2\_scores$: Chi-squared scores for each feature
  $categories \leftarrow$ unique values in y
  $features \leftarrow$ number of features in X
  Initialize $chi2\_scores$ as zeros with length equal to $features$
  for $feature$ **in** X **do**
    for $category$ **in** $categories$ **do**
      $O \leftarrow$ observations of $feature$ in $category$
      $E \leftarrow$ expected observations of $feature$ if independent
      $chi2\_scores[feature] \leftarrow chi2\_scores[feature] + \frac{(O-E)^2}{E}$
    end for
  end for
  return $chi2\_scores$

After analyzing three feature selection algorithms, their advantages and disadvantages are strategically summarized in Table 4, guiding the decision to adopt an ensemble approach. This selection was driven by the goal of harnessing diverse methodologies that illuminate different facets of feature relevance and their interactions with the target variable, ensuring a thorough and multifaceted analytical approach. The Fisher Score algorithm is renowned for its ability to distinguish features based on their class separability, particularly beneficial for linearly separable data. This method enhances model performance in classification tasks by focusing on the variance ratios between and within classes, thus pinpointing features that are crucial for class differentiation [110, 111]. However, its primary limitation lies in its lessened effectiveness in capturing the complexities of non-linear class distributions, an area where it may fall short.

Complementing the Fisher Score, the Information Gain algorithm explores the reduction of entropy, offering a deeper understanding of how each feature reduces uncertainty within the dataset, an asset in crafting decision trees [112, 114, 115]. This approach shines in mapping out non-linear relationships and complex feature interactions, areas that may be overlooked by the Fisher Score's linear focus. Moreover, the ensemble incorporates the Chi-squared algorithm to enhance the feature selection strategy further by targeting categorical data and assessing feature independence from the target variable. This inclusion broadens the feature selection scope, addressing biases inherent in the other two algorithms, particularly their focus on linear separability and continuous variables [118, 119, 117].

The integration of these algorithms into an ensemble model leverages their individual strengths while counterbalancing their weaknesses. This comprehensive strategy ensures a well-rounded feature selection process, accommodating both continuous and categorical data types. By combining these approaches, a framework for feature selection is created that is both balanced and efficient, resulting in analytical results that are both reliable and perceptive. This framework simplifies and boosts overall model performance by taking into account the complex structure of data while also supporting the goal of extracting the most informative and

discriminative features from large collections. Table 5 displays the outcome of the feature selection framework. It showcases the top 80 features out of 240, selected based on their high scores obtained from three feature selection algorithms. These features were chosen because the remaining ones exhibited significantly lower scores, resulting in a noticeable decline in their values.

Table 4: Comparison of Feature Selection Algorithms

Algorithm	Advantages	Disadvantages
Fisher Score	Maximizes class separability, ideal for linearly separable data, simple to apply, and model-independent.	Less effective for non-linear class distributions.
Information Gain	Reduces dataset entropy, aids in constructing interpretable trees, simplifies complex data, and minimizes overfitting.	May overlook feature interactions not related to entropy reduction.
Chi-squared (χ^2)	Simple, effective for categorical features, aids in dimensionality reduction.	Not suitable for continuous variables without binning, may miss non-categorical relationships.

3.5 Profiling

This sub-chapter focuses on profiling, beginning with a thorough review of existing literature on profiling techniques. Next, a taxonomy, as shown in Fig. 51, is introduced to concisely summarize previous profiling methods. Following this, a synthesis of these methodologies is presented, detailing the criteria and reasoning behind the selection of the most appropriate techniques for the specific objectives of this work.

3.5.1 Profiling Background

Profiling is a technique pivotal for evaluating the performance of systems, algorithms, or applications, aimed at optimizing effectiveness and pinpointing areas for improvement [121]. This process involves the assessment of various operational aspects, such as execution time, resource usage, and the accuracy and reliability of machine learning models. It also examines the interplay between input features and their resulting predictions [122]. The primary objective of profiling is to boost the system's effectiveness, reliability, and overall quality [123]. This sub-chapter delves into various profiling approaches, offering a detailed exploration. A comprehensive categorization of existing profiling methods is presented in Figure 51, providing a clear overview of the field.

3.5.1.1 Statistical

Statistical profiling evaluates a system, algorithm, or application's performance by gathering and examining data about its state [121]. Statistical profiling is used to collect data on the system's CPU, memory, and I/O operations to pinpoint areas of inefficiency or subpar performance. The analytical techniques utilized in the profiling process include the Anderson-Darling test, Mahalanobis distance, Hotelling's t-squared, hypothesis testing, the Girvan-Newman algorithm, and the Louvain method. Additionally, prominent methodologies like the t-test, Kolmogorov-Smirnov, and Kullback-Leibler divergence are also employed, being widely recognized in statistical analysis.

Table 5: Result of Feature Selection Process

Feature name	Score	Feature name	Score
Bytecode Entropy	3681.07	Count_bytecode_character_d	164.69
AST Features_ast_nodetype	1917.06	ABI Features_len_non_zero_input	162.59
AST Features_ast_src	1137.47	Opcode Count Features_JUMPI	162.46
Opcode Count Features_STOP	505.08	Functional_Number of functions	160.66
Opcode Count Features_RETURN	399.32	Count_bytecode_character_7	158.04
Opcode Count Features_CALLVALUE	345.91	AST Features_ast_len_exportedSymbols	157.33
ABI Features_len_list_payable	277.00	Opcode Count Features_SLOAD	152.51
ABI Features_len_list_type	267.10	Lines_Code	147.80
Count_bytecode_character_3	244.91	Lines_total	147.40
ABI Features_len_list_constant	244.77	Opcode Count Features_DUP2	145.94
ABI Features_len_list_name	241.37	Opcode Count Features_JUMP	144.54
ABI Features_len_list_stateMut	241.02	Opcode Count Features_SSTORE	139.88
Count_bytecode_character_4	240.44	Opcode Count Features_MSTORE	135.80
Bytecode Length	237.99	Opcode Count Features_ISZERO	134.72
Count_bytecode_character_5	234.30	Opcode Count Features_CALLDATALOAD	131.59
Opcode Count Features_SWAP1	217.55	Opcode Count Features_DUP3	130.43
ABI Features_len_non_zero_output	217.00	Opcode Count Features_SUB	128.92
Count_bytecode_character_b	216.80	ABI Features_len_list_output	128.62
Opcode Count Features_DUP1	212.42	Opcode Count Features_ADD	128.54
Count_bytecode_character_9	210.31	Count_bytecode_character_a	127.08
Opcode Count Features_AND	204.80	ABI Features_len_zero_input	125.06
Count_bytecode_character_6	204.28	Opcode Count Features_DIV	124.07
Opcode Count Features_JUMPDEST	198.28	Opcode Count Features_MLOAD	124.00
Opcode Count Features_SWAP3	198.10	Opcode Count Features_REVERT	115.62
AST Features_ast_id	197.04	Opcode Count Features_DUP4	109.85
Opcode Count Features_LOG3	190.66	Opcode Count Features_CALLCODE	109.63
Opcode Count Features_SWAP2	188.27	Opcode Count Features_EXP	108.33
Count_bytecode_character_1	185.25	Count_bytecode_character_c	104.46
Count_bytecode_character_8	181.79	Opcode Count Features_PUSH4	103.91
Opcode Count Features_PUSH1	180.70	Opcode Count Features_DUP7	100.97
Opcode Count Features_POP	180.11	Opcode Count Features_DUP6	100.24
Count_bytecode_character_2	178.60	Opcode Count Features_DUP5	99.89
Count_bytecode_character_f	176.01	Solidity call	97.14
Opcode Count Features_CALLER	175.60	Opcode Count Features_EQ	95.84
ABI Features_len_list_input	175.08	Contract Name	95.70
Functional_Number of ”public” functions	173.30	Opcode Count Features_CALLDATASIZE	92.78
Opcode Count Features_PUSH20	172.13	Solidity send	91.64
AST Features_ast_len_nodes	170.75	Count_bytecode_character_0	87.02
Opcode Count Features_PUSH2	168.24	Opcode Count Features_EXTCODECOPY	86.88
Duplicate Lines Count	167.45	Opcode Count Features_SWAP4	81.81

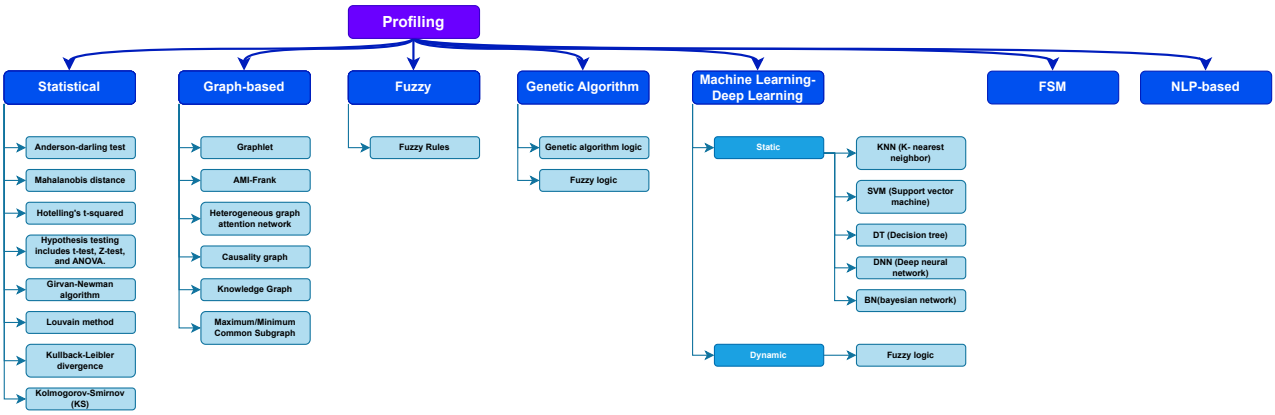


Figure 51: Taxonomy: profiling

- **Mahalanobis distance:** Mahalanobis distance was used by Hoffelder *et al.* [124] as an alternative multivariate measure to assess the similarity of groups. After initial testing, they focused on problems in the context of the dissolution. For instance, one may choose a precise approach, but it would be difficult to calculate a product-independent equivalency margin. They presented T2EQ, a useful technique built on the Mahalanobis distance to address these issues. This method offers an internal equivalence margin for comparing dissolution profiles, aligning with existing dissolution guidelines.

- **Girvan-Newman:** Iorio *et al.* [125] applied the Girvan-Newman algorithm to create a drug similarity network using a public dataset with genome-wide Gene Expression Profiles (GEPs) from treatments with over a thousand compounds. In this network, drugs with shared molecular targets are linked or fall within the same group. Their method calculated a unique similarity distance between two drugs by merging GEPs through the Girvan-Newman algorithm. The Girvan-Newman algorithm works by (1) Calculating the shortest paths between nodes. (2) Count the paths through each edge (edge betweenness or centrality). (3) Removing the edge with the highest betweenness. This process continues until no edges are left, resulting in a hierarchy of drug communities based on the segments that become disconnected as edges are removed.

- **Kolmogorov-Smirnov:** Rassokhin *et al.* [126] pioneered a novel approach for experimental design in medicinal chemistry, capitalizing on multiobjective optimization principles. This method balances traditional design objectives and additional selection criteria, thereby achieving designs that encompass diverse objectives. Significantly, their work introduced two design types—single and array—that integrated the Kolmogorov-Smirnov profiling technique to evaluate performance based on drug-likeness parameters. This profiling approach was powerful when developing an exploratory library, optimizing five distinct parameters. The design outcome exhibited diversity, reflected molecular weight and $\log(P)$ profiles akin to known drugs, required minimal reagents, and allowed facile synthesis using robotic hardware in an array format. Consequently, their study underscored the substantial potential of employing the Kolmogorov-Smirnov profiling technique in the optimization and synthesis of chemical compounds [126].

- **Kullback-Leibler divergence:** Zhang *et al.* [127] devised a method to effectively combine medium and low spatial resolution images, leveraging a conceptual model that classifies the study area into diverse pixel types at a MODIS 250m scale. A key component of their approach was using Kullback-Leibler (KL) divergence as a profiling technique. This method allowed for the examination of similarity between unknown pixels and pure winter wheat samples, achieved through studying temporal changes in NDVI. The KL divergence profiling approach demonstrated promising results, particularly in the principal agricultural region of the Yiluo River Basin [127].

In summary, this sub-chapter discusses four methodologies within the statistical category: Mahalanobis distance [124], Girvan-Newman algorithm [125], Kolmogorov-Smirnov [126], and Kullback-Leibler divergence [127]. As statistical profiling techniques, these methods analyze behaviors by gathering statistical data, such as calculating similarities or dissimilarities between parameters using various algorithms. Mahalanobis distance provides an internal equivalence margin for comparing dissolution profiles. The Girvan-Newman algorithm is effective in calculating unique similarity distances between entities.
However, both the Mahalanobis distance and Girvan-Newman methods can be complex, with their results being significantly influenced by the characteristics of the dataset. Kolmogorov-Smirnov can optimize multiple parameters simultaneously, while Kullback-Leibler divergence is proficient in assessing similarities between distinct entities. Nonetheless, both suffer from limited generalizability and applicability. Furthermore, the accuracy and effectiveness of the Kullback-Leibler divergence may depend on the quality and specificity of the examined data. In essence, each method offers specific advantages in its application domain but also has limitations that require careful consideration and selection based on the particular context.

3.5.1.2 Graph-Based

Graph-based profiling is a technique for evaluating the performance of a system, algorithm, or application using graph data structures [128]. This method collects performance data and represents nodes and edges in a graph, visually representing the relationships between various system elements [129]. Graph-based profiling aims to optimize performance by reducing the time and resources required to execute specific tasks and improve the efficiency, reliability, and overall quality of the system, algorithm, or application being evaluated [130]. This technique is widely used in software development, database optimization, and machine learning, providing a robust performance analysis, debugging, and optimization tool. As follows, graph-based techniques are divided into six categories.

- **Graphlet:** In their study on Graphlet profiling, Karagiannis *et al.* [128] developed comprehensive end-host profiles to capture and visualize diverse user activities and behavior. They introduced an innovative profiling approach based on graphs, enabling the extraction and representation of flow-level data at the transport layer. The authors introduced a graphlet consisting of six columns: source IP, protocol, destination IP, source port, destination port, and destination IP. Their initial findings highlighted the dynamic nature of user behavior, underscoring the importance of adaptive profiling methods.

- **AMI-Frank:** Han *et al.* [131] introduced an advanced graph-based model called AMI-Frank, designed explicitly for personalized search within folksonomies. The model incorporated the individual user's multiple interests and preferences [131]. To tackle challenges such as multiple user interests and shifts in user interest over time, the algorithm employed a community clustering technique and an ant algorithm-based evaporation method to update the graph [131]. The experimental results demonstrated enhanced personalization performance compared to existing state-of-the-art algorithms [131].

- **Heterogeneous graph attention network:** Graph-based methods have gained significant attention in recent research due to their effectiveness in various applications, such as enhancing user profiles and classifying apps and tweets [129]. Chen *et al.* [132] proposed a semi-supervised approach to enhance user profiles, leveraging heterogeneous graph learning. They introduced a universal solution using the heterogeneous graph attention networks (HGAT) technique. This method learned representations for each entity by leveraging the graph structure and assessing the relative importance of neighboring entities through the attention mechanism. The results underscored the effectiveness of this approach in enhancing user profiles.

  In a related study by Xue *et al.* [133], they introduced AppDNA, a framework that aimed to generate concise representations of an app's behavior using a combination of function-call graphs and heterogeneous graphs. This compact representation proved useful for multiple applications, such as malware detection, app categorization, and plagiarism detection. To overcome the challenge of converting large call graphs into fixed-size vectors, the authors proposed a graph-encoding technique. The results showcased a high accuracy rate of 93.07 percent in classifying 4024 apps as either secure or malware, achieved within a fast runtime of approximately 5.06 seconds.

  Labadie *et al.* [129] introduced a novel system that employed a graph-based model to encode tweets as individual nodes within a graph structure. This graph representation was subsequently fed into a NN for profile classification. The authors highlighted the advantages of their approach over previous methods, emphasizing its ability to overcome existing challenges and generate a valuable and reliable profile kernel for classification tasks. The effectiveness of these graph-based methods underscores their potential for diverse applications and paves the way for future research in this domain.

- **Causality graph:** Asai *et al.* [134] presented a comprehensive application identification framework comprising three key modules: feature vector space construction, supervised learning of feature vectors, and application identification based on similarity measures. To extract distinctive patterns for identification, the framework utilized a graph-mining algorithm. The authors evaluated the framework using packet traces from seven P2P applications, serving as ground truth and achieved an impressive overall accuracy of 90% [134].

- **Knowledge Graph:** In [135], Munir *et al.* delved into profiling within the context of Industry 4.0, focusing on the fourth industrial revolution. They proposed an approach that utilized a knowledge graph—a graph-based data structure that captures entities and their relationships. In addition, semantic modeling was employed to transform real-world concepts into a formal representation suitable for

computer analysis [135]. The study's findings showcased the efficacy of this method in enhancing the accuracy and efficiency of profiling within the industry. Furthermore, they [135] shed light on the potential applications of knowledge graphs and semantic modeling, offering valuable insights into their utilization in profiling tasks.

- **Maximum/Minimum Common Subgraph:** Daoud *et al.* [136] introduced a method for personalized document ranking by leveraging a semantic user profile represented as a graph. The proposed model utilized graph representation learning techniques to consider the user's preferences and interests across various topics. This information was then employed to rank documents based on the most relevant maximum or minimum common subgraph for the user [136]. Using a graph-based representation, the model effectively captured complex relationships between different concepts and topics, ultimately enhancing the quality of personalized document ranking [136].

Summing up, graph-based profiling presents a collection of potent techniques for enhancing performance and addressing complexities in areas such as software development, machine learning, and database optimization. These methods, characterized by their diversity and adaptability, leverage graph data structures to improve system operation. The wide-ranging applicability of these techniques, from personalization algorithms to semantic modeling, underscores the significance and promising future of graph-based profiling. As research continues, the use and impact of these strategies are expected to expand, catalyzing advancements in performance optimization.

3.5.1.3 Fuzzy

Utilizing fuzzy logic to assess system performance is called fuzzy profiling [137]. It uses fuzzy sets, which are collections of values with varying degrees of membership, to evaluate performance data while considering many variables and their interactions. Using less time and resources to complete particular tasks maximizes performance, efficiency, and dependability [138]. Fuzzy profiling is a helpful tool for performance analysis in complicated or dynamic systems. Following, we delve into the category of fuzzy profiling known as the fuzzy rule [139].

- **Fuzzy rules:** In[140], authors presented a method for developing fuzzy information retrieval (IR) systems. Their approach involved incorporating relevance feedback and fuzzy rule-based summarization techniques to construct a unified index by combining three relevancy profiles: a task profile, a user profile, and a document profile. The system's performance was evaluated using standard precision and recall metrics, and the results exhibited significant improvements in retrieving relevant documents in response to user queries. The proposed method demonstrated its effectiveness in enhancing the retrieval capabilities of IR systems.

  Xu *et al.* [130] presented an innovative system for student profiling that leverages the collective efforts of multiple agents. By diligently recording the learning activities and interaction history of each student in a dedicated profile database, the system generates a comprehensive student model. This model serves as the foundation for dynamically designing personalized learning plans tailored to individual

students, in conjunction with the content model. The system's adaptive nature ensures a customized and optimized learning experience for each student.

Moreover, Mencar *et al.* [141] put forth a novel approach for describing preference profiles using fuzzy sets. Their methodology involves observing the sequential order in which users access resources and subsequently updating preference profiles to align with their evolving preferences. Through this iterative process, the system adeptly recommends relevant resources when users revisit them. The simulations conducted by the authors provided compelling evidence of the method's efficacy, showcasing its potential for delivering tailored recommendations based on user preferences.

Finally, in the realm of intrusion detection, Dickerson *et al.* [142] introduced a pioneering technique employing fuzzy network profiling. Their method entailed constructing a fuzzy profile of network behavior by meticulously analyzing network traffic data. Leveraging this profile, a sophisticated model was created to discriminate between normal and abnormal network behavior accurately. By harnessing the power of fuzzy logic, the approach demonstrated its ability to account for uncertainty and imprecision in real-world network data, thereby enabling robust intrusion detection capabilities.

To summarize, the sub-chapter explores fuses of fuzzy rules in different domains. It examines how they enhance information retrieval systems, enable personalized student profiling, improve recommendation systems with preference profiles, and strengthen intrusion detection through network profiling. These studies emphasize the effectiveness of fuzzy rules in addressing uncertainty and imprecision within these areas.

3.5.1.4 Genetic Algorithm (GA)

GA-based profiling techniques employ the principles of GAs or fuzzy logic to analyze and enhance system performance [143]. These strategies utilize iterative processes that involve evolving a population of potential solutions through mechanisms inspired by natural selection and genetics [144]. Operations like mutation, crossover, and choice are applied to iteratively improve the solutions and converge towards the optimal configuration for performance optimization [145]. The effectiveness of the solutions is evaluated using a fitness function, and over multiple generations, the techniques progressively refine the solutions toward the optimal outcome. By leveraging these approaches, complex performance analysis and optimization become flexible and powerful tools that can be utilized to achieve optimal system performance. The following describes the categories of GA-based profiling techniques, which form a comprehensive framework for analyzing and optimizing system performance.

- **GA logic:** In their study on machining parameter optimization, Asokan *et al.* [146] focused on achieving continuous finished profiles from cylindrical stock. Their objective was to select machining settings that would minimize production costs for continuous profile machining. Given the high complexity of the problem, the authors utilized a GA to address this machining optimization challenge effectively.

  Similarly, Gupta *et al.* [107] introduced and tested a GA-based profiling technique to tackle security-specific vulnerabilities. Their approach enabled enterprises to select the lowest-cost security profile

with the broadest vulnerability coverage. By leveraging GA, the technique facilitated the identification of an optimal security profile considering both cost-effectiveness and vulnerability coverage.

Furthermore, researchers have proposed anomaly-based intrusion detection techniques, where deviations or anomalies are identified as intrusions based on the description of "normal" behavior [147]. In the study by Resende *et al.* [147], a GA-based adaptive approach was presented for profiling features and determining parameters in anomaly-based intrusion detection. The reported tests conducted on the CICIDS2017 dataset showcased excellent results, achieving a detection rate of 92.85% and a low false positive rate of 0.69%.

- **Fuzzy logic:** Hamamoto *et al.* [148] introduced the concept of utilizing a GA for detecting network anomalies. They proposed employing the GA to establish a digital signature for a network segment through flow analysis. This involved utilizing data from network flows to predict the traffic patterns of the network over a specific period. Additionally, the authors incorporated a fuzzy logic scheme with GA to determine whether a particular case represents an anomaly. They suggested the use of an expert system that continuously monitors IP flows in the network, ensuring regular desired behaviors while promptly alerting potential issues. The proposed anomaly detection method demonstrated its capability to identify network problems autonomously. The results of the approach exhibited high accuracy (96.0%) and a low false positive rate (0.56%) when applied to actual network traffic flows.

In conclusion, GA-based profiling techniques demonstrate substantial versatility and adaptability in addressing complex challenges, from machining optimization to cybersecurity. Exemplified by Asokan *et al.* [146], Gupta *et al.* [107], and Resende *et al.* [147], GAs have been effectively used to minimize production costs, optimize security profiles, and improve intrusion detection. Furthermore, the integration of fuzzy logic with GA, as shown by Hamamoto *et al.* [148], augments GA's ability to detect network anomalies. These findings highlight the potential of GA-based profiling techniques in solving real-world problems and underscore the importance of further exploration in this field.

3.5.1.5 Machine Learning and Deep Learning

The primary objective of machine learning and deep learning profiling methodologies is to meticulously scrutinize and amplify the performance of these predictive models [149]. These innovative techniques aim to measure the performance, reliability, and precision of models, meticulously identifying areas that warrant improvements to augment their effectiveness and efficiency [150]. The end goal is not merely performance enhancement, but the elevation of the overall quality of the models [151]. Continuing, this sub-chapter outlines the two primary categories of profiling techniques: static and dynamic, specifically tailored for machine learning and deep learning profiling techniques.

1. **Static:** Static machine learning profiling falls under a category of machine learning strategies that formulate predictions or decisions based on a predetermined data set, without taking into account any new data introduced over time [151]. Static machine learning algorithms employ a model trained once to generate their predictions. In the following, various static machine learning profiling techniques

will be categorized and detailed, including methodologies such as K-nearest neighbor, support vector machine, decision tree, deep NN, and Bayesian network.

- **K nearest neighbor (KNN):** In [152], Haque *et al.* proposed a solution to the indoor localization challenge by determining the Cartesian coordinates of individuals or objects within a building. They introduced a localization technique based on profiling, utilizing the KNN approach. This technique demonstrated notable accuracy, outperforming state-of-the-art methods, while also leveraging affordable and low-power wireless devices. The KNN-based approach showcased superior localization accuracy compared to other sophisticated techniques commonly employed for indoor localization tasks.

  Tsalera *et al.* [153] focused on the qualitative identification of environmental noise by analyzing sound-specific characteristics in the temporal and spectral domains. They devised a mechanism for classifying ambient noise into distinct groups based on these characteristics. The researchers utilized the KNN profiling for the classification task, employing training and test data from the publicly available UrbanSound8K dataset. The method was configured to consider one to three nearest neighbors, resulting in the creation of nine different models. The performance of these models ranged from 0.70 to 0.85, depending on the specific distance measurement techniques employed.

  Moreover, Nagaraj *et al.* [154] presented a model that leveraged different data extraction techniques to scrape information from previous student profiles that were successful in securing admission. The authors combined the KNN profiling with Feature Weighted algorithms. By utilizing machine learning technologies, they determined weighted scores for the features and applied training and testing data. This framework aimed to assist students in applying for graduate admission at suitable colleges, providing them with valuable insights based on successful profiles from the past.

- **Support vector machine (SVM):** In their study on profiling, Bayot *et al.* [155] specifically targeted the task of performing profiling across multiple languages. They employed SVMs, a popular machine learning algorithm for profiling, in combination with word embedding averages, which represent words as vectors. By utilizing word embeddings with SVMs, the authors aimed to achieve a more accurate and flexible approach to author profiling, especially when dealing with multiple languages. This methodology holds the potential to improve the precision and adaptability of author profiling techniques across different linguistic contexts [155].

- **Decision tree (DT):** Decision trees, as a machine learning profiling technique, provide a valuable approach for identifying crucial factors and their interrelationships within large and intricate datasets [156]. This type of analysis holds the potential for developing early intervention strategies and identifying individuals at high risk who may require closer monitoring and support [157].

  In a study conducted by Batterham *et al.* [156], a decision tree profiling was utilized to explore the development of suicide risk in a population cohort consisting of 6656 Australian adults. The

findings revealed that various factors, including previous suicidality, anxiety symptoms, depression symptoms, neuroticism, and rumination, were robust predictors of suicidal ideation and behavior over a span of four years. Notably, substance use was found to be associated with suicidal thoughts and behaviors specifically among individuals with moderate levels of anxiety or depression. These results highlight the potential value of decision trees in developing personalized assessments of suicide risk and tailored interventions to address the individual needs of at-risk individuals [156].

In 2013, Duchess *et al.* [157] employed the decision tree profiling technique and survey data to profile the online and mobile technologies and services of ski resorts. Their aim was to develop effective promotional and advertising strategies. The study focused on two distinct segments, namely millennials and non-millennials, and examined various technologies including resort websites, microblogging services, and online coupon services. The findings indicated that ski resorts employed different strategies for each segment, tailoring their approaches accordingly. Furthermore, the study revealed that these technologies and services had a positive impact on resort sales, both in terms of immediate and sustained outcomes. This highlighted the significance of leveraging decision trees to identify effective strategies and optimize promotional efforts in the ski resort industry [157].

- **Deep NN (DNN):** In [158], Hawley *et al.* proposed a deep auto-encoder model as a profiling technique, where the model is conditioned using the target audio effect's control settings and time-domain samples. Experimental findings demonstrated the possibility of capturing the fundamental functional and auditory features of compressors using this approach [158]. However, further research is needed to address audible noise in audio processing processes [158].

  In 2020, several different profiling approaches were introduced by multiple researchers [159, 160, 161]. Yu *et al.* [159] presented SKYLINE, an interactive tool for DNN training that supports computational performance profiling, visualization, and debugging. A qualitative user study indicated the practicality and user-friendly nature of SKYLINE. Li *et al.* [160] proposed APPDNA, an automated approach for app profiling based on function-call graphs, achieving reliable app profiling with a high accuracy rate. Cura *et al.* [161] conducted a study comparing LSTM and CNN architectures for driver profiling, where the CNN architecture outperformed LSTM in detecting aggressive driving behavior.

- **Bayesian network (BN):** In the field of criminal profiling, Bayesian network models have the potential to play a significant role in analyzing offender behavior and drawing inferences about the characteristics of individuals responsible for crimes [162]. According to Baumgartner *et al.* [162], the application of the Bayesian network profiling technique in criminal profiling has the potential to offer valuable insights to law enforcement. However, the accuracy and validity of the model heavily rely on the quality of the data used to construct it and the appropriateness of the underlying assumptions [162].

  By leveraging appropriate algorithms and methodologies, the identification of deviations from

anticipated behavior within video behavior profiling for anomaly detection can prove advantageous. This approach aids in detecting potential security risks or unusual incidents [163]. Xiang *et al.* [163] proposed the utilization of Bayesian Networks in their research to evaluate the similarity of behavioral patterns. The effectiveness and resilience of this approach were demonstrated through studies conducted using noisy and sparse datasets obtained from both indoor and outdoor surveillance scenarios.

All in all, static machine learning profiling techniques provide robust solutions for various practical applications, ranging from indoor localization and environmental noise classification to personalized risk assessment and behavioral anomaly detection. Techniques such as KNN, SVM, DT, DNN, and BN profiling, each demonstrate their unique strengths in the various case studies explored. However, the effectiveness of each method remains contingent upon the quality of data, appropriateness of underlying assumptions, and the specific use case. As the field continues to evolve, further research and refinement of these techniques will undoubtedly enhance their potential, ultimately leading to increasingly sophisticated and accurate prediction models.

2. **Dynamic:** In the context of dynamic machine learning profiling, the term refers to the utilization of machine learning techniques capable of adjusting to evolving data or environments in real-time [164]. Dynamic machine learning encompasses various approaches, such as online learning, transfer learning, active learning, meta-learning, reinforcement learning, and fuzzy logic [165]. These approaches can be employed individually or in combination to construct a dynamic machine-learning profiling system that can effectively adapt to changing data and environments. Specifically, in this work, we concentrate on fuzzy logic, which is the final approach mentioned [164].

    - **Fuzzy deep Boltzmann:** Zheng *et al.* [164] introduced a deep learning approach for passenger profiling. The key element of this profiling method is the Pythagorean fuzzy deep Boltzmann machine (PFDBM), where the parameters are represented as Pythagorean fuzzy numbers. This representation enables each neuron to learn the influence of a feature on generating the correct output from both positive and negative perspectives. Experimental results on various datasets demonstrated that the proposed technique exhibits significantly improved learning capabilities and classification accuracy when compared to existing profilers.

In summary, dynamic machine learning profiling offers a powerful solution for managing real-time adjustments in evolving data and environments. The application of Fuzzy deep Boltzmann, as demonstrated by Zheng *et al.*'s [164] research, showcases its strength in improving learning capabilities and classification accuracy in the context of passenger profiling. This highlights the potential of dynamic machine learning profiling in developing sophisticated, adaptable systems for a wide range of applications.

3.5.1.6 Finite State Machine (FSM):

FSM profiling has emerged as a robust technique for understanding complex behaviors in data analysis, especially when fused with fuzzy logic [46]. This approach provides flexibility in handling the uncertainty and variability common in human activities. The applications range from enhancing worker productivity to visualizing human behavior, as demonstrated by Langensiepen *et al.* [45] and Fernandez *et al.* [46]. These researchers presented innovative FSM profiling frameworks for activity recognition and human behavior analysis.

The research conducted by Langensiepen *et al.* [45] focused on the application of fuzzy FSM (FFSM) for activity recognition and worker profiling in an intelligent office environment. The authors' primary objective was to enhance workers' efficiency and productivity by monitoring and analyzing their activities and behaviors using FFSM. The utilization of fuzzy logic in activity recognition and worker profiling gained attention in research due to its ability to handle uncertainty and variability inherent in human activities.

In the same research direction, Fernandez and *et al.* [46] introduced a novel framework called VISUVER, which aims to profile and visualize human behavior. VISUVER [46] integrates dynamic finite state machines (DFSM) and text-based similarities to effectively profile and analyze patterns of human behavior. By automating data manipulation related to profiling human behavior, VISUVER have the potential to significantly enhance understanding of individuals, including their motivations and challenges.

In conclusion, while FSM profiling offers valuable insights into managing uncertain and changeable behaviors, the introduction of fuzzy logic in frameworks like VISUVER represents a promising avenue. This approach addresses the challenges posed by non-deterministic FSMs, paving the way for a deeper understanding of complex behaviors and activities.

3.5.1.7 NLP-Based

NLP technologies offer transformative potential in diverse areas, one of which is the creation of semantically rich document profiles for document identification and retrieval. Guillén *et al.* [47] and Anrig *et al.* [166] highlighted the power of NLP in creating document profiles by harnessing semantic metadata extracted from text, ultimately catering to specific user requirements.

In [47], Guillén *et al.* conducted a study on the application of NLP technologies to aid in the accurate identification of documents based on user needs. To achieve this objective, they developed a document profile capable of representing semantic metadata extracted from documents through the use of NLP technologies. The document profile serves as a structured representation of relevant information derived from the documents, enabling efficient document identification and retrieval based on user requirements. By leveraging NLP techniques, the authors aimed to enhance the accuracy and effectiveness of document identification processes in order to meet the specific needs of users.

Furthermore, in [166], authors explored various NLP technologies combined with algorithms to facilitate the creation of a novel document profile proposal from semantic perspectives. Algorithms play two crucial roles in the data mining endeavor. First, some algorithms are designed to control the process of extracting information from the large quantities of data that have become readily available in our modern, data-rich society. In this situation, they can be tuned for data capture, verification, and validation. Second, they

dominantly mathematical procedures, can be used as the profiling engine to identify trends, relationships, and hidden patterns in disparate groups of data. Using NLP integrated with algorithms in this way often means that more effective profiles can be computed than manual ones.
In summary, leveraging NLP and algorithms, researchers significantly improved document identification processes. They [47, 166] crafted precise document profiles using semantic metadata extraction via NLP. Meanwhile, algorithms handle complex tasks such as data capture, validation, and unearthing hidden patterns. These advancements underscore the transformative potential of NLP and algorithmic approaches in creating highly accurate, user-centric document profiles.

3.5.2 Profiling Synthesis

This sub-chapter explores a range of profiling techniques, which are systematic approaches aimed at enhancing performance optimization. Among these techniques, GA profiling stands out due to its robustness and effectiveness. GA-based profiling is influenced by the principles of natural selection and genetics. It is a computational optimization method known for handling complex logic and interactions. This technique navigates through a variety of potential inputs, evolving them across numerous iterations to identify hidden weaknesses. It is particularly effective in detecting problems that arise from complex interactions or intricate logical structures. GA-based algorithms perform an extensive exploration of the search space and apply evolutionary principles to refine inputs. However, traditional GAs face several specific challenges, including:

- **Premature Convergence**: This problem occurs when the algorithm settles on a sub-optimal solution too early in the process, which can happen due to a lack of sufficient diversity or exploration in the early stages of the algorithm.
- **Genetic Diversity**: This issue emerges when there's a potential loss of diversity within the population over generations. A lack of genetic diversity can lead the algorithm towards local optima, limiting its ability to explore a broader range of potential solutions.
- **Feature Selection Balance**: This problem arises when there is an imbalance in the selection of features. It can manifest as either selecting too few features (underfitting) or too many (overfitting), leading to poor performance of the algorithm.
- **Retention of Elite Solutions**: The challenge occurs when there's a need to avoid losing high-quality solutions found in previous generations. Without a mechanism to preserve these solutions, the algorithm may discard valuable information that could guide it towards optimal solutions.
- **Adaptive Mutation Rate**: This issue arises due to the static nature of mutation rates. A fixed mutation rate may not be suitable throughout the entire run of the algorithm, as different phases require different levels of exploration and exploitation for efficient convergence.
- **Solution Diversity via Semantic Similarity**: This issue occurs when there's a risk of premature convergence due to a lack of diversity in the solutions. If the crossover process does not introduce

sufficient diversity, the population may become too homogeneous, reducing the algorithm's effectiveness.

- **Ensuring Non-zero Solutions**: This issue emerges when the algorithm converges on trivial solutions, where all features are turned off. This can happen if there is no mechanism to ensure that at least some features remain active in the solutions.

Conclusively, GA-based profiling emerges as a formidable method notably in detecting vulnerabilities in SCs. Its adaptive nature, rooted in natural selection and genetics, effectively manages complex structures and interactions. The algorithm's capability to reduce false negatives and positives significantly bolsters detection accuracy and reliability. Therefore, GA-based profiling represents a robust and innovative approach, fundamental to advancing computational system security. However, due to the limitations of traditional GA, the following sub-chapter provides details on Enhanced GA (EGA) to address these issues.

3.5.3 Enhanced GA-based Profiling technique

Profiling is the systematic process of analyzing a program's activity and performance to identify specific sections of code that significantly impact desired outcomes or metrics [166]. This technique is crucial for understanding how different parts of the program contribute to its overall behavior and performance [164]. In the context of smart contract vulnerability detection, profiling becomes even more essential [134]. By examining the execution patterns and behaviors within a smart contract, profiling can uncover unusual or unexpected patterns that may lead to vulnerabilities [135, 47].
This sub-chapter explores the proposed profiling algorithm, an advanced version of the traditional GA. This work introduces three additional enhancements beyond the improvements proposed in our previous work [36]. Fig. 52 illustrates the improvements of the EGA profiling technique. Detailed descriptions of each step are provided in the following discussion. The baseline is represented by "00", while "01", "02", "03", and "04" denote incremental improvements over the previous version proposed in [36]. The latest enhancements, labeled "05", "06", and "07", represent the most recent advancements.

- **"00" Regular Genetic Algorithm**: GA is a search and optimization technique inspired by the process of natural selection and genetics. It works by simulating the evolution of a population of candidate solutions to a problem. This population evolves over time through processes mirroring natural selection, including crossover, mutation, and survival of the fittest [146, 147]. Individuals in the population are evaluated based on a fitness function, which determines their suitability as solutions. Over successive generations, the population evolves towards an optimal or satisfactory solution [36]. GAs are widely used for solving complex problems where traditional optimization methods are not effective.

- **"01" Increased Population Size**: The GA employs a significantly enlarged population size of 10,000 individuals, markedly surpassing the sizes typically used in conventional GA implementations. This expansion in population size is strategic, aiming to encompass a broader diversity of potential solutions [36]. By incorporating a vast array of genetic variations, the algorithm enhances its ability to thoroughly explore the multidimensional solution space. This extensive search capability is crucial

Figure 52: Architecture: EGA

for mitigating the risk of premature convergence on suboptimal solutions, thereby augmenting the algorithm's proficiency in identifying and evolving toward the global optimum.

- **"02" Semantic Similarity Distance (SSD) Checks**: Incorporating semantic similarity checks prior to crossover operations in a GA is a sophisticated approach. It ensures that the genetic recombination of individuals is between those that are adequately diverse yet not excessively dissimilar, thereby facilitating a more meaningful and effective exploration of the solution space. By preventing the crossover of overly similar individuals, the algorithm maintains genetic diversity within the population. Simultaneously, it avoids the combination of drastically different individuals to prevent the generation of implausible or inefficient offspring [36].

  Supporting this mechanism, the SSD function quantifies the dissimilarity between two solutions, playing a crucial role in maintaining an optimal level of diversity by preferentially selecting parents for crossover based on their semantic distance. It ensures that selected parents are neither too similar (which would stifle diversity) nor completely unrelated (which could disrupt cohesive genetic structures), thus optimizing the evolutionary search for novel and efficient solutions [36].

- **"03" Initialization with Bias Towards Zeros**: Initiating the population with a bias towards zeros, especially in the context of feature selection, advocates for the inception of the evolutionary process with simpler, less complex models. This bias towards minimalism encourages the algorithm to commence with a conservative approach, gradually introducing complexity only when it demonstrably enhances

the solution's fitness [36].

- **"04" Handling Zero Features**: To avoid the generation of invalid solutions—specifically, those devoid of any selected features—the algorithm incorporates a safeguard mechanism. This mechanism ensures that at least one feature remains selected following the mutation process, thereby preserving the validity and potential utility of the offspring. This precautionary measure is essential for maintaining the integrity of the solution space, preventing the algorithm from exploring infeasible or meaningless solution territories [36].

- **"05" Penalty Fitness Function**: The integration of a penalty fitness function introduces a methodical approach to discourage the selection of solutions exhibiting certain undesirable traits. In the context of feature selection, solutions that incorporate too few features might be penalized. This penalization mechanism operates by reducing the fitness score of less desirable solutions, effectively guiding the evolutionary process toward more favorable and viable solutions. This method enhances the algorithm's efficiency by focusing the search on solution regions that are more likely to yield optimal outcomes.

- **"06" Retention of Elites**: By preserving a subset of elite individuals—the top performers within each generation—the algorithm ensures the continuity of superior genetic material across generations. This elite retention strategy is pivotal for accelerating the convergence of the population toward optimal solutions. By safeguarding the genetic integrity of the best solutions, the algorithm can build upon these high-quality foundations, thereby enhancing the overall rate of evolutionary progress and increasing the likelihood of achieving superior outcomes.

- **"07" Adaptive Mutation Rate**: The algorithm's adaptive mutation rate mechanism modulates the mutation frequency in response to the generational phase of the evolution. Initially, a higher mutation rate is employed to foster a broad exploration of the solution space, encouraging the discovery of diverse and potentially viable genetic configurations. As the evolution progresses and promising solutions begin to emerge, the mutation rate is strategically lowered to refine these solutions through more targeted mutations. This adaptive approach ensures a balanced dynamism between exploratory diversity and exploitative precision throughout the evolutionary process.

In summary, the EGA profiling technique marks a notable improvement over traditional GA. It integrates several critical enhancements designed to boost the algorithm's effectiveness, diversity, and capability to navigate toward optimal solutions. Originating from the conventional GA foundation, this approach expands the population size, incorporates SSD checks, and favors simpler initial models. These strategies refine the search process and prevent early convergence on subpar solutions, ensuring a broader exploration of possible solutions and a higher chance of finding optimal outcomes.
The introduction of a penalty fitness function, preservation of elite individuals, and an adaptive mutation rate further enhance the algorithm's performance. These enhancements work together to discourage suboptimal solutions, maintain high-quality genetic material, and achieve a balance between exploration and exploitation. Each enhancement, systematically numbered from "01" to "07", demonstrating a deliberate strategy

to surpass the limitations of traditional GAs and expand the capabilities of evolutionary computation. The careful design and targeted improvements of the EGA profiling technique showcase the power of advanced algorithms in solving complex optimization challenges with increased effectiveness and efficiency. The EGA algorithm, as presented in 4, methodically outlines the EGA profiling methodology. It presents the sequence and integration of each innovative component in detail.

Algorithm 4 Enhanced Genetic Algorithm for Profiling

Input: population size P, number of generations G, initial mutation rate μ, semantic similarity parameters α, β, feature space X, labels y.
Output: best_individuals_per_label after G generations.

```
Initialize population with size P randomly
Initialize best individuals and fitness per label in y
function PENALTY_FITNESS_FUNCTION(individual)
    Compute base_fitness
    Apply penalty if necessary
    return base_fitness - penalty
end function
function RETAIN_ELITES(population, fitness_scores, elite_count)
    Sort and select top elites
    return elites
end function
function ADJUST_MUTATION_RATE(generation)
    Adjust mutation rate based on generation
    return adjusted rate
end function
function SSD(st1, st2)
    Compute semantic similarity distance
    return distance
end function
function SEMANTIC_SIMILARITY(st1, st2)
    Check if α < ssd(st1, st2) < β
    return True/False
end function
for each generation in G do
    Adjust mutation rate
    Calculate fitness for each individual
    Retain elites
    for each individual to maintain size P do
        Perform crossover and mutation
    end for
    Update population
    Update best_individuals_per_label
    Print best fitness
end for
return best_individuals_per_label
```

3.6 Classification

This sub-chapter elaborates on the classification step, which utilizes a simple classifier during the initial training phase. This approach allows us to assess the effectiveness of the proposed feature selection and

profiling approach in identifying various types of vulnerabilities within SCs. The simplicity of the classifier aids in isolating the impact of the feature selection and profiling steps. This ensures that the results primarily reflect the quality of the preprocessed and structured data, rather than the complexity of the classification algorithm. In doing so, the aim is to validate the efficiency of the preprocessing techniques and set a benchmark for future improvement and comparison with more sophisticated classifiers.

3.7 Evaluation Criteria

The final stage of the proposed model architecture centers on its evaluation, utilizing the testing set from the dataset to assess the efficacy of the profiling model. This sub-chapter outlines the metrics employed for evaluation and explain their significance. Additionally, Table 6 offers a breakdown of each evaluation metric's definition, ensuring a clear understanding of the criteria used to measure the model's performance. A comprehensive analysis of the metric values and the insights gained from this evaluation are detailed in chapter 4.

Metric	**Definition**	**Formula**
Precision	Measures the accuracy of positive predictions.	$Precision = \frac{TP}{TP+FP}$
Recall	Measures the fraction of positives correctly identified.	$Recall = \frac{TP}{TP+FN}$
F1-Score	Harmonic mean of precision and recall.	$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$
Accuracy	Fraction of all correct predictions.	$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$
Macro Average	Average metrics were computed independently for each class.	$MacroAvg = \frac{1}{N}\sum_{i=1}^{N} Metric_i$
Weighted Average	Average metric weighted by class size.	$WeightedAvg = \sum_{i=1}^{N} w_i \cdot Metric_i$

Table 6: Details of Evaluation Metrics

3.8 Concluding Remarks

This chapter covers four main aspects of the project, corresponding to contributions CONT1, CONT4, CONT5, and CONT7. Initially, it introduces a general overview of the model architecture. Following this, it provides a detailed taxonomy of the features that can be extracted and the profiling techniques available. This information assisted in selecting the profiling method, EGA, and in developing a new feature analyzer, specifically the BCCC-SCsVulLyzer(V.2). The subsequent chapter will concentrate on the experimental validation processes for the analyzer and profiling method.

4 Experiments and Results

This chapter delves into the experimental setup and methodologies employed to validate the efficacy of the proposed profiling model for SCs vulnerability detection. A cornerstone of proposed approach involved the creation of a new, comprehensive dataset specifically tailored for the task. This dataset served as a rich testing ground, offering diverse vulnerabilities to evaluate the model's performance. A rigorous implementation and evaluation procedure was followed to gauge the model's proficiency and report the results. This procedure includes creating a new dataset, tuning hyper-parameters, and obtaining the results of experiments for different vulnerabilities. The subsequent sections explain the experiment's setup, execution, and results.

4.1 Experimental Setup

Our testing environment consists of a laptop running Windows 11, equipped with an Intel i5 8-core CPU and 32GB of memory. Both the analyzer and the EGA are implemented in Python 3.8. We conducted the experiment on a new dataset, as described in subsection 4.2.2, using the following data split: 75% for training, 15% for validation, and 10% for testing.

4.2 A New SCs Dataset

This sub-chapter starts by reviewing the publicly available datasets listed in Table 7. After that, it introduces the newly created dataset for this study, called BCCC-SCsVul-2024.

4.2.1 Available Datasets

In this subsection, we critically analyze the datasets utilized in the field of SC vulnerability detection research, categorizing them by their respective sizes.

- **Less than 100 samples:** Samreen *et al.* [44] utilized a dataset comprising six samples from Etherscan. Their work highlights the challenges associated with small datasets, such as limited scope and potential reliance on external tools for labeling. Similarly, Liu *et al.* [2] proposed a dataset of five SCs exclusively containing reentrancy vulnerabilities, formatted in Solidity.

  Furthermore, Vivar *et al.* [98] focused on a single bytecode sample, labeled as vulnerable without specifying the exact type of vulnerability. In another study, Aquilina *et al.* [167] used a dataset of six SCs in the Solidity language. These SCs were categorized into three different types of vulnerabilities: DoS, over/underflow, and Re-entrancy.

- **Between 100 and 10,000 samples:** In their study, Hu *et al.* [91] used 110 samples from Etherscan to evaluate their type verifier system. However, these samples were not labeled, which limits the system's usefulness in detecting security issues. To address the need for labeled datasets, several works have provided binary datasets that indicate whether SCs are secure or vulnerable.

  Liu *et al.* [41] provided a general dataset comprising 1,382 samples of Solidity SCs. Similarly, Gupta *et al.* [97] and Zhang *et al.* [105] utilized two different binary datasets containing 1,382 and 7,000

Solidity SCs, respectively. Despite the availability of these datasets, they are still insufficient for training robust models due to their limited size.

- **More than 10,000 samples:** Other studies have utilized larger datasets, providing a broader range of information on binary-classified SCs. For instance, Qian *et al.* [42] included 40,700 SCs in different formats (Solidity, Bytecode, and Binary code), focusing on four types of vulnerabilities: Delegatecall, Integer Overflow, Reentrancy, and Timestamp.

  Liu *et al.* [40] proposed two different datasets. The first dataset included 40,932 Solidity SC samples, covering three types of vulnerabilities: Reentrancy, Timestamp, and Infinite Loop. The second dataset contained 4,170 Solidity samples, addressing the same vulnerabilities as the first.

  Additionally, Ding *et al.* [43] provided a dataset of SCs with HF instances, although the number of instances was not specified. This dataset includes vulnerabilities such as logic loopholes and integer overflow. The most recent dataset, proposed by Hajihosseinkhani *et al.* [36], contains 36,670 instances of Solidity SCs, all of which are binary classified.

These datasets from prior research exhibit significant limitations. Smaller datasets often face issues of overfitting and lack broad applicability. Medium-sized datasets, while offering more samples, are still insufficient for comprehensive model training and typically provide limited binary classifications. The larger datasets, though extensive, are typically limited to specific vulnerabilities and formats. This underscores the need for a more expansive, detailed, and versatile dataset to effectively detect vulnerabilities in security controls.

4.2.2 New Vulnerable SCs datasets (BCCC-SCsVul-2024)

This research presents an innovative approach through the creation of a new dataset. We gathered Solidity source code for SCs from several sources, including Smart Bugs, Ethereum SCs (ESCs), slither-audited-smart-contracts, and the SmartScan-Dataset. These sources were selected due to the presence of numerous compromised SCs, each exhibiting distinct vulnerabilities. Following the collection process, we transformed the source codes into unique SHA-256 hashes to eliminate any duplicates. Table 8 proposes further details about the new dataset.

4.2.3 Hyper-parameter Tuning

This sub-chapter discusses the tuning of hyper-parameters that are pivotal for the performance optimization of the proposed EGA. The main focus is primarily on the number of generations, alpha, and beta parameters. These parameters are integral to achieving an optimal balance between the exploration of new solutions and the exploitation of known ones, which is crucial for the algorithm's effective navigation through the solution space.

The process begins with adjusting the number of generations, which dictates the depth of the algorithm's search for optimal solutions. Increasing the number of generations allows the algorithm to undergo additional cycles of selection, crossover, and mutation. This leads to a more comprehensive exploration of the solution space, potentially resulting in improved outcomes. However, this benefit comes at the cost of increased

Ref.	Name	# of samples	Format	Cat.	Details	Advantages	Disadvantages
Samreen *et al.* [44]	DeFi, Globalcryptox, FairDare, Moneybox, AIRToken QuizBLZ	6	Solidity	1	Re-entrancy	Simplicity	Labeled with other tools.
Liu *et al.* [2]	TokenSender, UGToken, E4Token, DAO RoT	5	Solidity	1	Re-entrancy	Simplicity	Labeled with other tools.
Vivar *et al.* [98]	the100th Ethereum contract(since the Ethereum blockchain was launched)	1	Byte-code	1	Not mentioned	Simplicity	Labeled with other tools.
Aquilina *et al.* [167]	Bank, DelayUnderflow, ProductVote, simulationERCToken, simulationKotET TargetUnderflow	6	Solidity	1	Over/underflow DoS Re-entrancy	Simplicity	Insufficient samples.
Hu *et al.* [91]	110 contracts from etherscan	110	Solidity	0	Not mentioned	Simplicity	Only used for checking typeability.
Liu *et al.* [41]	Samples from Etherscan	1,382	Solidity	2	Safe Not safe	Sufficient samples.	No details about categories.
Gupta *et al.* [97]	Google BigQuery	7,000	Solidity	2	Safe Not safe	Sufficient samples.	No details about categories.
Zhang *et al.* [105]	SmartBugs Dataset-Wild	47,587	Solidity	2	Safe Not safe	Sufficient samples.	No details about categories.
Qian *et al.* [42]	Dataset processing for vulnerabilities	40,700	Solidity Byte-code Binary-code	4	Delegatecall Integer Overflow Reentrancy Timestamp	Sufficient samples.	Covers only 4 attacks.
Liu *et al.* [40]	ESC	40,932	Solidity	3	Reentrancy Timestamp Infinite-Loop	Sufficient samples.	Covers only 3 attacks.
Liu *et al.* [40]	VSC	4,170	Solidity	3	Reentrancy Timestamp Infinite-Loop	Sufficient samples.	Covers only 3 attacks.
Ding *et al.* [43]	SCs come from Fabric, Caliper, and IBM	N/A	Including HF version 0.6 and HF version 1.1	2	Logic loopholes Integer overflow	Fit for HF task.	Not general.
Hajihosseinkhani *et al.* [36]	Solidity SCs	36,670	Solidity	2	Binary dataset	Sufficient samples and fit for binary classification.	Do not have vulnerabilities.

Table 7: Comparative Analysis of Previous Datasets

BCCC-SCsVul-2024	
Label	# of Samples
Total	111897
ExternalBug	3604
GasException	6879
MishandledException	5154
Timestamp	2674
TransactionOrderDependence	3562
UnusedReturn	3229
WeakAccessMod	1918
CallToUnknown	11131
DenialOfService	12394
IntegerUO	16740
Re-entrancy	17698
Secure	26914

Table 8: BCCC-SCsVul-2024 Dataset Overview

Number of Generations vs. Profiling Features	
# of Generations	**# of Profiling Features Selected**
50	59
100	65
150	75
200	77
500	80

Impact of Alpha and Beta on Profiling Features		
Alpha	**Beta**	**# of Profiling Features Selected**
0.5	0.7	66
0.5	1.0	67
0.5	1.5	68
0.5	2.0	69
0.1	1.0	63
0.1	1.5	63
0.1	2.0	64
0.3	1.0	65
0.3	1.5	68
0.3	2.0	69

Table 9: Hyper-parameter Tuning Results

computational resources. Concurrently, the alpha and beta parameters fine-tune the algorithm's handling of semantic similarity. Specifically, alpha determines the threshold for considering solutions as distinct, fostering the identification of novel solutions, while beta manages the equilibrium between exploring new areas of the solution space and refining existing solutions.

An important observation from this process is that modifications to hyper-parameters minimally impact the algorithm's output, suggesting its robustness and a high degree of generalization. This indicates that the model's performance relies less on the fine-tuning of hyper-parameters and more on its inherent stability and consistency across various conditions. Table 9 presents the number of profiling features selected by the EGA for the profiling technique during the hyper-parameter tuning process.

In conclusion, the minimal impact of hyper-parameter variations on the profiling approach's output underscores the model's robustness. Its quality is affirmed by its stability, as demonstrated by consistent performance across various settings. These characteristics confirm the model's reliability and efficacy, making it a highly valuable approach for addressing complex vulnerabilities in SCs.

4.3 Experimental Results of SC Vulnerability Detection

This sub-chapter details the experimental outcomes of SC vulnerability detection methodology. It includes classification reports for a variety of vulnerabilities in Table 10. This Table systematically lists metrics including Precision, Recall, F1-Score, Accuracy, Execution Time, Macro Average, Weighted Average, and Best Fitness for each vulnerability type. Additionally, data on the performance of secure instances is included for comparison. The metrics, detailed in Table 6, offer a comprehensive view of how the proposed method performs against each vulnerability.

The classification report in Table 10 provides a structured overview of the proposed method's performance across various vulnerabilities. Using evaluation metrics defined in Table 6, it details metrics such as Precision and Recall, highlighting the approach's coverage for each vulnerability. The F1-Score merges Precision and Recall to measure overall test accuracy, while the Accuracy metric reflects the total efficiency of the detection process. Additionally, Execution Time and Best Fitness illustrate the operational efficiency and optimization success of each method. Macro and weighted averages offer insights into a model's overall performance across different classes; macro averaging assigns equal importance to each class, while weighted averaging adjusts for class imbalances by weighting each class's contribution based on its prevalence. Together, these reports offer a panoramic assessment of the tested methods' abilities to identify and manage different types of SC vulnerabilities, furnishing a holistic view of their performance and reliability in various scenarios.

4.4 Concluding Remarks

This chapter evaluates the proposed model, addressing contribution CONT6. It begin by analyzing existing datasets and conclude that a new dataset is required. As a result, a new dataset for SC vulnerabilities, titled BCCC-SCsVul-2024, has been created. This dataset is compiled by amalgamating sources such as Smart Bugs, ESCs, slither-audited SCs, and the SmartScan-Database, with the primary goal of applying the proposed model to a large sample set of over 100,000 samples. Subsequently, experiments are conducted to tune the parameters of the proposed profiling EGA. This chapter concludes with the classification reports for various vulnerabilities. In the following chapter, the results obtained in this chapter are analyzed and discussed.

Classification Report						
	Precision	**Recall**	**F1-Score**	**Accuracy**	**Execution Time**	**Best Fitness**
ExternalBug	0.85	0.48	0.61	0.93	48.63	65
benign	0.94	0.99	0.96			
macro avg	0.89	0.73	0.79			
weighted avg	0.93	0.93	0.92			
GasException	0.82	0.53	0.64	0.88	49.71	67
benign	0.89	0.97	0.93			
macro avg	0.86	0.75	0.79			
weighted avg	0.88	0.88	0.87			
MishandledException	0.83	0.46	0.60	0.89	49.39	65
benign	0.90	0.98	0.94			
macro avg	0.87	0.72	0.77			
weighted avg	0.89	0.89	0.88			
Timestamp	0.84	0.44	0.58	0.94	49.448	68
benign	0.95	0.99	0.97			
macro avg	0.89	0.71	0.77			
weighted avg	0.94	0.94	0.93			
TransactionOrderDependence	0.88	0.47	0.61	0.93	48.82	70
benign	0.93	0.99	0.96			
macro avg	0.90	0.73	0.78			
weighted avg	0.92	0.93	0.92			
UnusedReturn	0.87	0.41	0.56	0.93	48.94	69
benign	0.93	0.99	0.96			
macro avg	0.90	0.70	0.76			
weighted avg	0.92	0.93	0.92			
WeakAccessMod	0.88	0.66	0.75	0.97	47.29	68
benign	0.98	0.99	0.99			
macro avg	0.93	0.83	0.87			
weighted avg	0.97	0.97	0.97			
CallToUnknown	0.81	0.55	0.65	0.84	51.29	66
benign	0.84	0.95	0.89			
macro avg	0.83	0.75	0.77			
weighted avg	0.83	0.84	0.83			
DenialOfService	0.79	0.51	0.62	0.83	52.74	67
benign	0.80	0.94	0.86			
macro avg	0.80	0.72	0.74			
weighted avg	0.80	0.80	0.79			
IntegerUO	0.83	0.69	0.75	0.83	52.74	67
benign	0.83	0.91	0.87			
macro avg	0.83	0.80	0.81			
weighted avg	0.83	0.83	0.82			
Reentrancy	0.79	0.59	0.67	0.78	53.22	68
benign	0.78	0.90	0.83			
macro avg	0.78	0.75	0.75			
weighted avg	0.78	0.78	0.77			

Table 10: Classification Report

5 Analysis and Discussion

This chapter is focused on thoroughly analyzing the results obtained from our experiments chapter 4, as detailed in Table 10. Initially, our analysis concentrates on evaluating performance in sub-chapter 5.1. Next, sub-chapter 5.2 provides insights gained from the proposed profiling technique. We specifically examine how the extracted features, as detailed in sub-chapter 3.3, contribute to the understanding and identification of various vulnerabilities. Following this, the discussion and evaluation of the proposed EGA model are split into three subsections. Sub-chapter 5.3 examines the model's computational efficiency through its time and space complexity. Next, sub-chapter 5.4 focuses on its stability and generalizability across different testing scenarios. Subsequently, sub-chapter 5.5 compares our model with other recently proposed models. The final sub-chapter 5.6 introduces the concept of designing specific profiling genes for each vulnerability and proposes two common profiles addressing both vulnerable and secure SCs.

5.1 Performance Analysis

This sub-chapter assesses the performance of the proposed EGA model using the evaluation criteria outlined in chapter 3.7. Table 10 presents the comprehensive results of the EGA profiling model, including various parameters. Beginning with accuracy, which measures the closeness of the model's predictions to the true values, the predictions from the EGA profiling model are generally accurate. This suggests that the model consistently performs well across various vulnerabilities.

Turning to the Best Fitness value details, achieving a fitness score between 65 and 70 out of 100 in our EGA model represents a significant accomplishment. This score demonstrates the model's proficiency in detecting vulnerabilities, indicating that the proposed model is highly effective and substantially reduces the likelihood of overlooking any issues. Maintaining a score in this range also helps prevent the model from becoming overly complex, ensuring it remains practical and manageable without sacrificing accuracy.

Further evaluating the performance, the accuracy of the model ranges from 0.78 to 0.97. This range underscores the model's robustness in correctly identifying vulnerabilities and secure instances. The wide span reflects the inherent variability in the complexity of different vulnerabilities and the model's ability to adapt to these challenges. Achieving such high accuracy, demonstrates the model's effectiveness in providing reliable predictions, even when dealing with diverse and intricate security issues.

Moving forward, there are three parameters: precision, recall, and F1-score. Precision measures the accuracy of the optimistic predictions made by the model, representing the fraction of true positive results among all positive predictions. Considering the definition of precision, the proposed model demonstrates commendable performance in identifying various vulnerabilities, ranging from 0.79 to 0.88 across different vulnerability types. Results in Table 3.6 indicate that the precision of our proposed model, when predicting a specific type of vulnerability, ranges between 0.79 and 0.88. This reflects the model's ability in correctly identifying true instances of each vulnerability type. Moreover, the model recognizes secure instances, achieving an impressive precision of 0.91. This means that when it classifies a SC as secure, there is a 91% chance that it is indeed free from vulnerabilities.

Recall, also known as sensitivity, assesses the model's ability to identify all positive cases. Even though the

model is good at precisely identifying positive instances, the range of recall, which shows the proportion of actual positives correctly identified, goes from 0.41 to 0.69. Despite the lower end of this range, it does not imply the overall inadequacy of the model. Rather, it suggests a potential for missing some positive instances while maintaining satisfactory accuracy across the dataset.

The F1-score is the harmonic mean of precision and recall, combining these two metrics to provide a balanced measure of a model's accuracy. Based on this definition and upon scrutinizing the F1-score values presented in Table 10, the F1-score spans a range from 0.56 to 0.75. This range of scores reflects the inherent complexity and subtlety associated with detecting vulnerabilities. These vulnerabilities are known for their complicated nature, often requiring nuanced detection strategies that may affect overall F1-score metrics. However, achieving F1-scores in this range against such challenging vulnerabilities underscores the model's ability to provide valuable insights and detections, even when faced with the complex behaviors these vulnerabilities exhibit.

Continuing to the next measures, Macro Avg and Weighted Avg: Macro Avg calculates the average score for each metric without considering the class distribution within the dataset. This makes it particularly beneficial for analyzing imbalanced datasets like ours. Conversely, Weighted Avg calculates the average of the metrics, adjusting for the prevalence of each class in the dataset. This provides a view that is more aligned with real-world performance, which is particularly valuable in the context of imbalanced datasets.

The Macro averages—approximately 0.81 for precision, 0.73 for recall, and 0.79 for F1-scores—provide a balanced overview of the model's performance across different classes. These values reveal a moderate level of effectiveness, without favoring any particular class size. However, this assessment also points to improvement areas, especially recall. In contrast, the Weighted averages, defined to reflect class distribution within the dataset, exhibit precisions around 0.93, recalls about 0.91, and F1-scores close to 0.89. These values indicate that the model performs particularly well in identifying the more frequently occurring secure class, indicating a variation in performance across different classes that suggests opportunities for refinement and balance.

In conclusion, the evaluation of the proposed EGA model demonstrates a strong alignment with actual outcomes, particularly highlighted by its overall accuracy and best fitness scores. Despite certain challenges, especially with complex vulnerabilities leading to lower recall rates, the model exhibits commendable precision, thereby minimizing false positives and enhancing security. The consistency in performance, demonstrated by the F1-scores and the Macro and Weighted averages, highlights the model's reliability and strong overall performance. This evaluation underscores the effectiveness of the EGA model in SCs vulnerability detection, while suggesting avenues for refinement that could enhance its capabilities even further.

5.2 Profiling Analysis

This chapter explores the profiling of the EGA through the use of violin plots to visualize the results. We examine each category of features, as discussed in sub-chapter 3.3 and illustrated in Fig. 50, individually. Twelve plots are presented to showcase the vulnerabilities and corresponding features within each category, alongside fluctuations observed in both secure and vulnerable samples. In all figures, vulnerable contracts are denoted in red, while secure samples are presented in green.

In Fig. 53, the first feature category, AST , is explored. AST provides a structural view of the code, thereby enhancing the understanding of its syntax and semantics. The proposed analyzer selected five unique features from this category including ID, the number of exported symbols, the number of nodes, node type, and source ID.

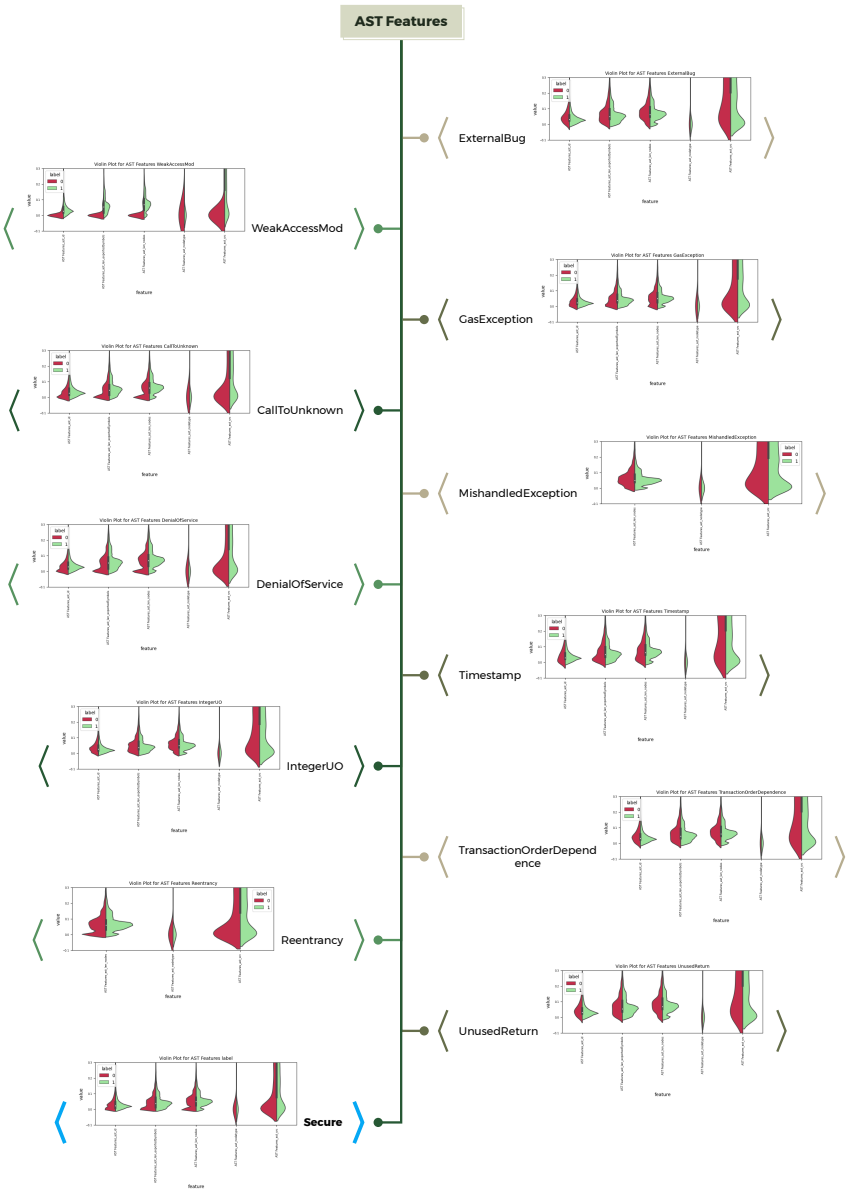


Figure 53: Profiling details: AST Features

Among these, three features — the number of exported Symbols, node type, and source ID — are almost consistently represented in the majority of plots for the identified vulnerabilities. The reason for this repetition is multifaceted:

- The number of nodes reflects the complexity and structural richness of its code, offering insights into its logic, nesting depth, size, and abstraction level.

- Node types shed light on the risks associated with specific code constructs, highlighting those that may pose greater security risks.
- The source ID aids in pinpointing areas within the codebase that are more susceptible to introducing vulnerabilities, thereby facilitating targeted code reviews and improvements.

Collectively, these features offer insights into how different aspects of code structure and origin correlate with security vulnerabilities, as demonstrated in Fig. 53 for Re-entrancy and MishandledException.
However, the essential aspect of profiling is to characterize each vulnerability uniquely. While these three critical features — number of nodes, node type, and source ID — significantly differentiate between vulnerabilities like DenialOfService and CallToUnknown, they exhibit similar fluctuations for other vulnerabilities. Therefore, additional features are introduced to aid in creating distinct profiles. By expanding the feature set to include more characteristics, such as ”number of exportedSymbols” and ”ID,” unique profiles for specific vulnerabilities can be developed. For instance, EGA profiling creates a distinctive profile exclusively for the TransactionOrderDependence vulnerability.
The same situation arises when we add the final two features to the AST feature set for profiling. For vulnerabilities such as Timestamp, UnsentReturns, TransactionOrderDependence, and WeakAccessControl, we utilized five distinct features for profiling. However, it became evident that some vulnerabilities, such as Timestamp, UnsentReturns, and TransactionOrderDependence, exhibit similar patterns in their violin plots. This similarity likely stems from the inherent nature of these specific vulnerabilities within this particular feature set. The similarities are attributed to shared underlying coding patterns that these vulnerabilities often exhibit. These patterns include similar control structures, error-handling mechanisms, or arithmetic operations, which are reflected in their AST representations. Consequently, the AST features, such as node types and structures become comparable, leading to analogous profiles in the violin plots for these vulnerabilities.
The second category is ABI, which is represented in Fig. 54. ABI features define the interaction between different program components at the binary level, such as calling conventions, data types, and memory layouts. The proposed analyzer identifies eleven features, encompassing the number of constants, the total number of inputs (along with separate counts for zero and non-zero values), the total number of outputs (with counts for zero and non-zero values), the length of the contract name, the count of payable functions, the number of stateMutabilities, and the variety of types used. Among these features, two in particular — the number of non-zero input values and the number of zero input values — consistently appear across eleven plots associated with the identified vulnerabilities. This suggests that contracts containing a more significant number of non-zero and zero input values could be more susceptible to attacks. This highlights the vital necessity of proper examination and control of these input values to enhance security.
Although the number of non-zero input values and the number of zero input values are crucial in identifying vulnerabilities, it’s complemented by additional features that aid in crafting unique profiles. These encompass the count of constants, total inputs, zero-value outputs, contract name length, and type diversity. Collectively, these features incorporate a broad range of potential vulnerability characteristics. However, for specific vulnerabilities like Timestamp and TransactionOrderDependence, the profiling model includes a number of constants to refine and distinguish these profiles from others, such as Re-entrancy.

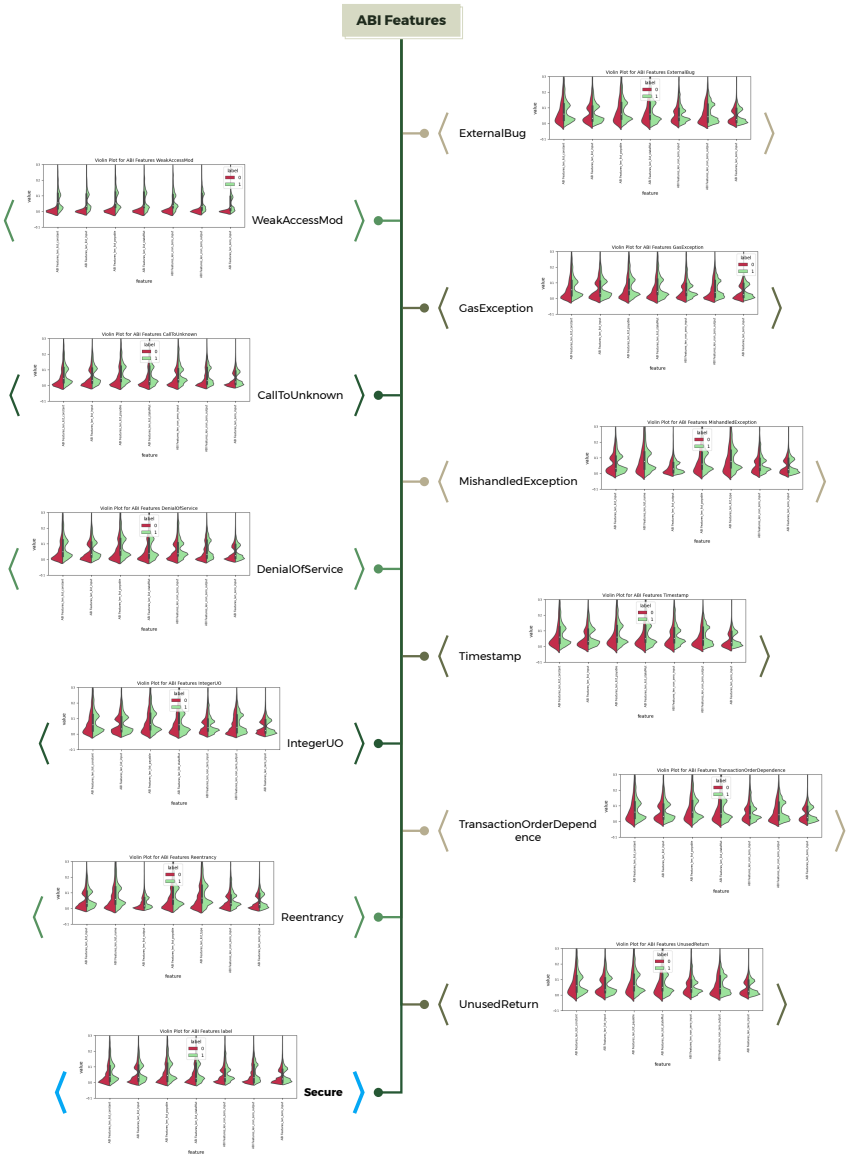


Figure 54: Profiling details: ABI Features

Expanding on this category highlights the unique characteristics of the ABI feature for WeakAccessMod profile. ABI features are directly connected to contract accessibility, function call patterns, and resource permission and management, making them crucial. Specifically, WeakAccessControl's association with incorrect permission configurations gives this vulnerability a distinct signature within the ABI profile.

Conversely, we identify a distinct group of vulnerabilities that have analogous ABI profiles due to their similar characteristics, including CallToUnknown and DenialOfService. Both CallToUnknown and DenialOfService are closely related in ABI features as they both involve scenarios where certain functions or resources are improperly utilized or abused, potentially leading to security breaches or service disruptions. As a summary, ABI features distinguish between different SC vulnerabilities primarily by focusing on evaluating input values. While certain vulnerabilities like WeakAccessMod have distinct ABI profiles, others such as

CallToUnknown and DenialOfService share analogous profiles due to similar characteristics like improper resource utilization.

Transitioning to the Bytecode category (see Fig. 55), the analyzer extracts three types of features. First is the count of each character in the bytecode, second is the entropy of the bytecode, calculated using the formula provided in sub-chapter 3.3.2, and third is the total length of the bytecode. Among these profiles in Fig. 55, two situations occur: firstly is the difference in vulnerabilities reflected in the frequency of characters in their profiles. It is evident that from 10 to 15 features are repeated in the plots; however, the difference between features that represent and also their fluctuations makes the difference in this category of features. Secondly, the ”entropy” feature is almost repeated among all profiles; regardless, the slight differences in the fluctuations make it a distinguishing feature.

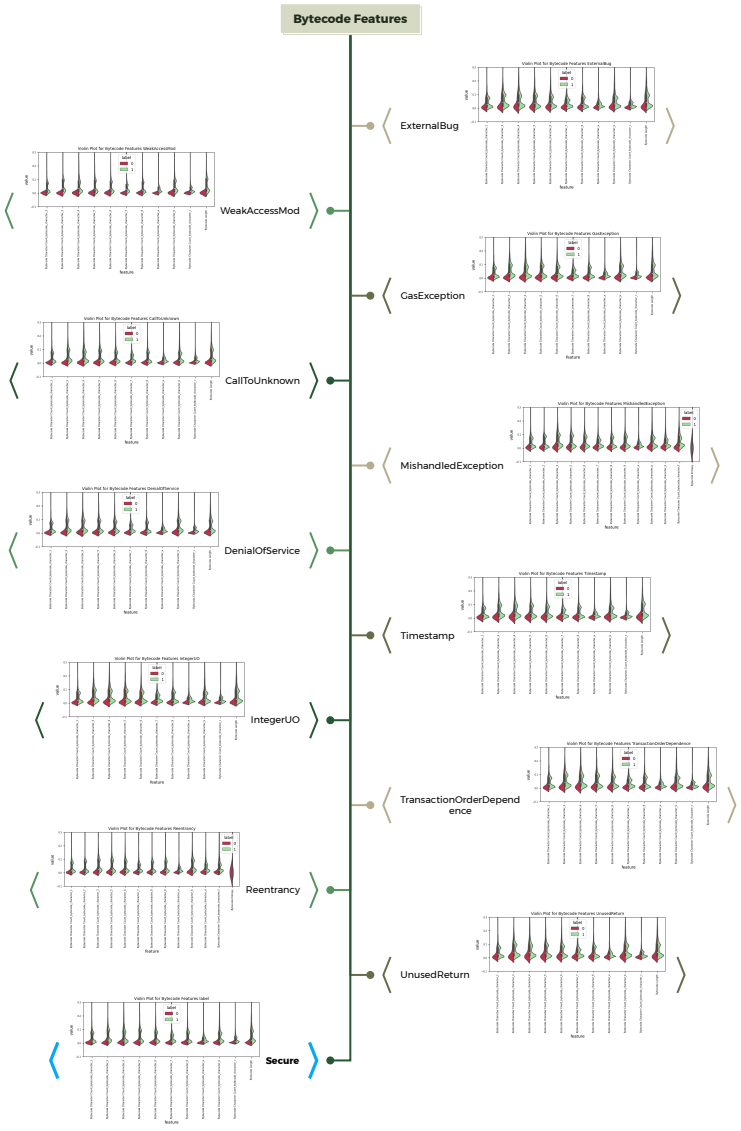


Figure 55: Profiling details: Bytecode Features

While profiles like ExternalBug and Re-entrancy stand out significantly, others do not exhibit as much dis-

tinction. The bytecode profiles of ExternalBug and Re-entrancy stand out significantly due to their unique bytecode patterns and operational characteristics, which are distinct from those found in other vulnerabilities. These distinct patterns are the result of specific code structures, making their profiles more pronounced and identifiable. On the other hand, within the context of similar profiles in 55 for IntegerUO and Timestamp, the differences in "length of bytecode" fluctuations and some character counts aid in differentiation. In summary, although some profiles in Bytecode feature profiling may appear similar, the inclusion of even a single distinct feature can significantly differentiate and individualize a profile.

Advancing to the Functional Features category presented in Fig. 56, the analyzer identifies five key features. These encompass the count of specific instructions pertinent to function details, including the number of loops, public functions, conditional statements, external calls, and total functions. These elements indicate the complexity and access control of functions, directly affecting contract behavior and security.

The profiling model uniquely employs 'public' as the sole feature. However, fluctuations differ significantly among profiling violin plots, notably with WeakAccessMod exhibiting higher values. It is attributed to WeakAccessMod's inherent reliance on access control mechanisms within SCs. Since WeakAccessMod vulnerability involves inadequate access control settings, which are associated with the visibility of functions (i.e., whether they are declared as public), the correlation with the number of public functions becomes evident. In other words, a larger number of public functions increases the likelihood of weak access control settings, thereby leading to a higher probability of the occurrence of WeakAccessMod vulnerability.

Continuing in the same category, we discover that none of the profiles make use of the number of loops, conditional statements, or external calls. This is an intentionally omission because the total number of public functions effectively define the profiles apart.

Transitioning to the opcode category reveals notable variations in profiling, as illustrated in Fig. 57. Despite varied feature sets in AST, ABI, Bytecode, and Functional Features potentially leading to similar vulnerability profiles, the opcode analysis stands apart. We examine 36 unique instruction features derived from opcode instruction counts in this category. These features reveal distinct patterns, underscoring the diversity in opcode usage. The uniqueness of these Opcode features stems from their direct correlation with low-level operations in the source code, which differ markedly depending on the contract's specific logic and functionality. As each opcode signifies a distinct action within the EVM, different vulnerabilities manifest through unique sequences or frequencies of these opcodes.

Even when opcode patterns seem similar across various vulnerabilities, there are subtle yet significant differences that can be observed. For instance, a vulnerability impacting transaction order utilizes opcodes associated with state changes more than those for arithmetic operations. Taking a specific look, the count of the 'STOP' instruction aligns with vulnerabilities like MishandledException, TransactionOrderDependence, and GasException but significantly differs in cases like WeakAccessMod and DenialOfService. This variation in opcode features separates vulnerabilities into distinct categories, creating eleven unique profiles. Hence, while there may be general similarities in opcode usage across various vulnerabilities, it's the crucial distinctions that are vital for distinguishing them. These distinctions provide a detailed insight into the security landscape of the contract.

In the Solidity category depicted in Fig. 58, which focuses on attributes unique to Solidity, the analyzer

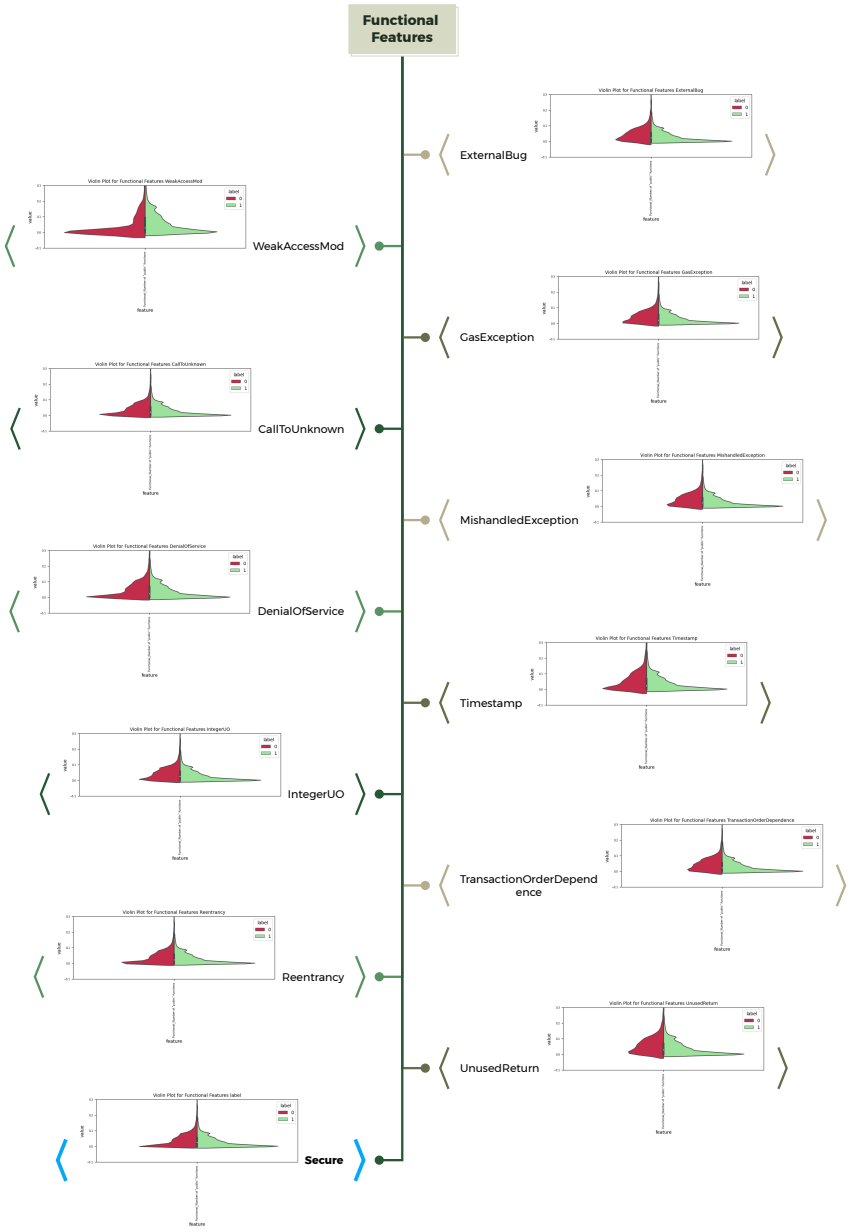


Figure 56: Profiling details: Functional Features

extracts eight distinct features. However, only one is pivotal in the profiling process: the number of ”Call” functions. This feature is crucial in the Solidity language because it directly influences a contract’s interactions with external contracts and controls its execution context. The ”Call” function allows a contract to invoke another contract’s function within the same execution context, potentially leading to security vulnerabilities if not managed properly.

In the final category, referred to as ”Lines Features,” the analysis examines five different attributes related to the source code lines: the number of total lines, code lines, comment lines, blank lines, and duplicated lines. According to Fig. 59, only the number of total lines, code lines, and duplicated lines are utilized in the

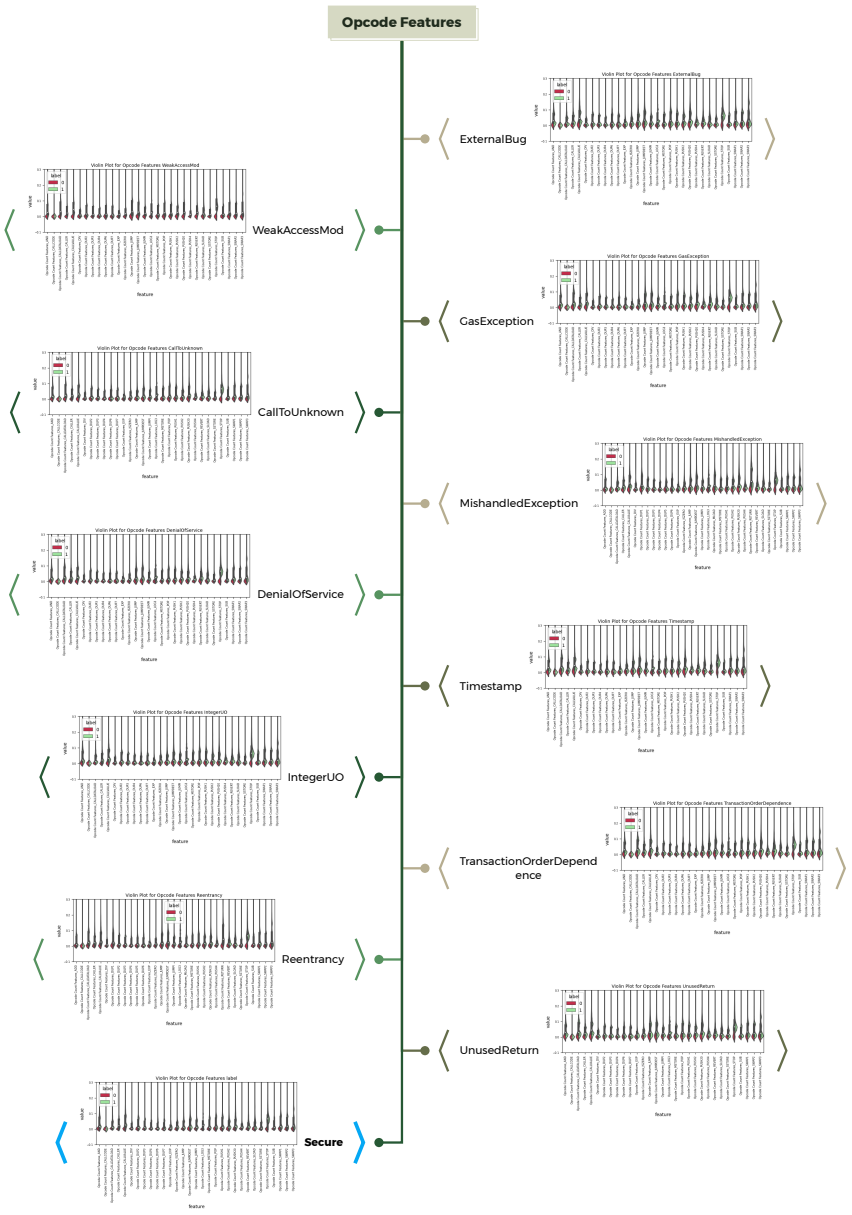


Figure 57: Profiling details: Opcode Features

profiling process. This selective usage suggests that these particular metrics are significant for distinguishing and understanding the characteristics of the source code within the profiling framework. They are defined as follows:

- Number of Total Lines: This metric provides a basic overview of the size and scale of the source code. It is a fundamental measure that offers an immediate sense of the contract's complexity.
- Number of Code Lines: By focusing on the actual lines of code (excluding comments and blank lines), this metric offers a clearer view of the functional content of the code. It is critical for assessing the amount of actual programming involved and understanding the density and potential complexity of the

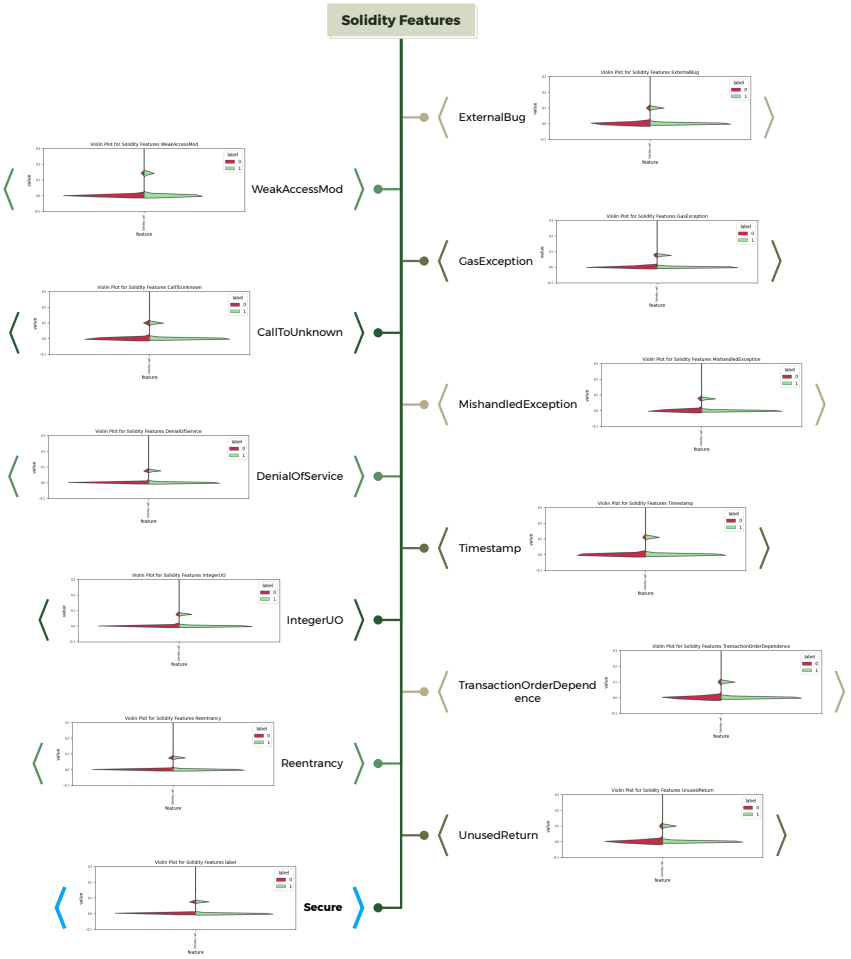


Figure 58: Profiling details: Solidity Features

logic embedded in the source code.

- Number of Duplicated Lines: This feature is crucial for identifying redundancy within the code. High levels of duplication can indicate poor coding practices and potential maintenance issues. It also hints at possible logical flaws or vulnerabilities that may arise from repeated code blocks, which might not have been thoroughly reviewed or tested.

Continuing our analysis of Fig. 59, the Re-entrancy profile incorporates the ”Number of Duplicated Lines” and ”Number of Total Lines” to address specific logical issues in its fallback mechanism. Meanwhile, MishandledException and GasException combine these features with the ”Number of Code Lines” to gain a deeper understanding of the contract’s complexity. Same, the rest of the vulnerabilities use all three features to create distinctive profiles. However, the fluctuations in the violin plot for ”Number of Total Lines” are more noticeable than the other two features, particularly in cases like Timestamp and CallToUnknown. This underscores the effectiveness of ”Number of Total Lines” as a valuable feature for comparison purposes.
In conclusion, the analysis demonstrates the model’s effectiveness in identifying diverse SCs vulnerabilities, revealing distinct patterns across feature categories. This highlights the complexity of SCs vulnerabilities,

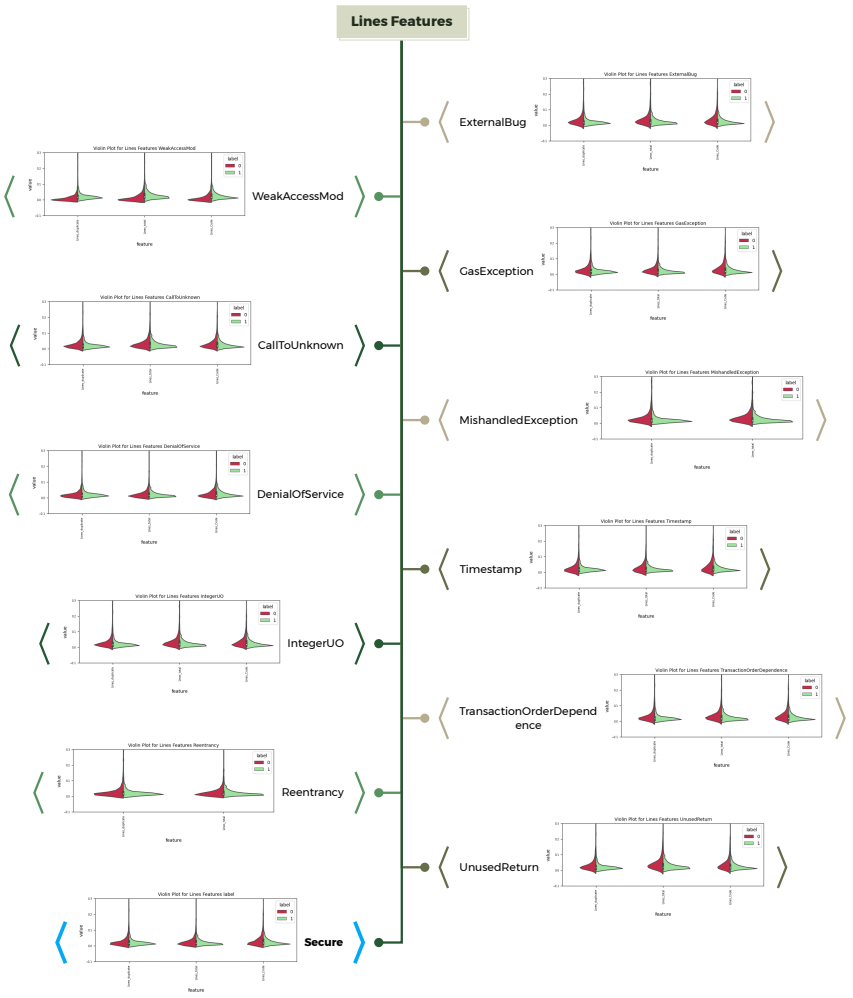


Figure 59: Profiling details: Lines Features

showing that while they share characteristics within certain categories, they also exhibit unique traits distinguishing them from one another. This delicate balance between similarity and distinctiveness highlights the importance of a sophisticated approach. Comprehensive visualization is essential in vulnerability profiling to ensure effective detection and classification.

5.3 Space and Time Complexities

This sub-chapter delves into the space and time complexity of the proposed method, beginning with the execution times detailed in Table 10. A simplified and effective procedure is indicated by the execution time of about 49 seconds. The minimum execution time of 47.29 seconds underscores the system's notable efficiency. It suggests commendable speed and minimal delays, with the majority of tasks completed in a time-effective manner. With a maximum duration of 53.22 seconds, the range supports the consistency and dependability of the system with regard to time management.

The subsequent analysis explores the distinct space and time complexities pertinent to the components of

the EGA, along with the feature selection and training phase. This refined iteration of the genetic algorithm is enriched with four sophisticated components: the Penalty Fitness Function, Retention Strategy, Adaptive Learning Rate, and Semantic Similarity Distance Check.

- **Penalty Fitness Function**
  Space Complexity: This function's space requirement primarily depends on the size of the individual array. Given that no additional arrays or data structures are created in proportion to the input's size, the space complexity remains constant, $O(1)$.
  Time Complexity: The internal operations, including `np.power`, `np.sum`, and conditional checks, execute in linear time relative to the individual's size. Consequently, the time complexity is $O(n)$, where n represents the length of the individual.

- **Retention Strategy (retain_elites)**
  Space Complexity: The primary space consumption is attributable to the elites list, which holds no more than `elite_count` individuals. As the size of this list does not expand with the population size, the space complexity is $O(1)$.
  Time Complexity: This procedure involves sorting the fitness scores, generally taking $O(m \log m)$ time, where m is the population size. The subsequent steps, including selecting indices and appending elites, are linear in terms of the number of elites. Thus, the predominant factor is the sorting operation.

- **Adaptive Learning Rate**
  Space Complexity: This component does not utilize additional space proportional to the input size, maintaining a space complexity of $O(1)$.
  Time Complexity: Given that the operations are straightforward mathematical computations and conditional checks, the time complexity remains $O(1)$.

- **Semantic Similarity Distance (SSD) Check**
  Space Complexity: Both functions operate on arrays corresponding in size to the input arrays `st1` (e.g., [0.2, ..., 0.5]) and `st2` (e.g., [0.1, ..., 0.4])without generating new data structures that scale with input size, resulting in a space complexity of $O(1)$.
  Time Complexity: These functions execute operations linearly related to the input arrays' sizes; hence, the time complexity is $O(k)$, where k denotes the length of `st1` or `st2`.

- **Main Genetic Algorithm Loop**
  Space Complexity: Significant contributors to space complexity include the arrays for population, offspring, and elites. The size of these arrays correlates with the population size and individual length. Denoting n as the individual length and m as the population size, the space complexity is $O(mn)$.
  Time Complexity: The time complexity involves multiple components:
  - Fitness calculation for each individual is $O(n)$, culminating in $O(mn)$ for all m individuals.
  - Sorting for elite retention takes $O(m \log m)$.

– The crossover and mutation processes are estimated as $O(mn)$, presuming these operations are linear with respect to individual length.

Thus, the total time complexity for one generation is $O(mn + m\log m)$, and for `num_generations`, it is $O(g \times (mn + m\log m))$, where g represents the number of generations.

- **Feature Selection and Model Training**
  Space Complexity:

  – Feature Selection algorithms such as Fisher Score, Information Gain, and Chi-squared maintain a constant space complexity relative to the input dataset's size. This complexity primarily depends on the number of features and classes, rather than the number of instances, making it O(1) in this scenario.

  – Model Training: Space complexity of the training process is $O(T \times d \times F)$ where T is the number of trees, d is the average depth of the trees, and F is the number of features (80 in our case).

  Time Complexity:

  – Feature Selection algorithms: The time complexity for these methods generally scales with the product of the number of instances, features, and classes, nominally O(NCF).

  – Model Training: Time complexity of the training phase split into Training Time and Prediction Time (for M instances) which are $O(T \times N \times F \times \log(N))$ and $O(M \times T \times d)$ where N is the number of samples (100,000), F is the number of features, T is the number of trees, d is the depth of the trees, and M is the number of instances to predict.

In conclusion, the detailed analysis of the proposed EGA, incorporating feature selection, a novel fitness function, and a semantic similarity check, reveals that the model balances efficiency and complexity. It is demonstrated that space complexities are predominantly constant ($O(1)$), ensuring minimal space requirements. In contrast, time complexities range from constant to linear and logarithmic ($O(1)$ to $O(mn + m\log m)$), depending on the specific operation. This highlights the model's capability to handle various computational tasks efficiently, making it a viable solution for complex vulnerability detection scenarios while maintaining manageable computational demands.

5.4 Stability, Generalizability, and Overall Performance Characteristics

This sub-chapter verifies the proposed model's stability and generalization ability through multiple rounds of cross-validation. This involves splitting the dataset into 'k' folds and then training and testing the model 'k' times, each time using a different fold as the test set and the remaining data as the training set. Table 11 provides the model's performance through this k-fold process for k = 5, 10, and 15.
Firstly, the model's generalizability is supported by its consistent performance across labels, such as ExternalBug, GasException, MishandledException, and others. The precision, recall, and F1-scores remain relatively stable across various labels, indicating that the model can generalize well across different labels

Label	K-fold	Precision	Recall	F1-Score
ExternalBug	5	0.82	0.46	0.58
	10	0.83	0.46	0.59
	15	0.85	0.48	0.61
GasException	5	0.76	0.45	0.58
	10	0.79	0.53	0.56
	15	0.82	0.53	0.64
MishandledException	5	0.80	0.43	0.55
	10	0.81	0.44	0.57
	15	0.83	0.46	0.60
Timestamp	5	0.82	0.38	0.51
	10	0.82	0.40	0.53
	15	0.84	0.44	0.58
TransactionOrderDependence	5	0.82	0.39	0.52
	10	0.85	0.43	0.57
	15	0.88	0.47	0.61
UnusedReturn	5	0.84	0.38	0.52
	10	0.86	0.39	0.53
	15	0.87	0.41	0.56
WeakAccessMod	5	0.80	0.60	0.68
	10	0.83	0.63	0.71
	15	0.88	0.66	0.56
CallToUnknown	5	0.80	0.48	0.60
	10	0.81	0.50	0.61
	15	0.81	0.55	0.65
DenialOfService	5	0.74	0.47	0.57
	10	0.75	0.50	0.60
	15	0.79	0.51	0.62
IntegerUO	5	0.80	0.64	0.71
	10	0.81	0.67	0.73
	15	0.83	0.69	0.75
Re-entrancy	5	0.72	0.53	0.61
	10	0.75	0.51	0.60
	15	0.79	0.59	0.67

Table 11: Evaluation metrics for different models

and vulnerabilities. Moreover, the model's stability is evidenced by the incremental metrics changes with the K-fold increase from 5 to 15. The precision, recall, and F1-scores show slight but consistent improvements or maintain stability as the K-fold increases. This suggests the model is relatively sensitive to the data splits used in the validation process.

In conclusion, the analysis confirms that the EGA stands out for its generalizability and stability. Its consistent performance across various labels and K-fold validations underscores its robustness, making it a viable solution for practical scenarios.

5.5 Comparing Our Model with Existing Methods

This sub-chapter provides a comparative analysis of the proposed EGA with two distinct categories of existing methods: traditional and NN-based vulnerability detectors. Table 12 offers a detailed evaluation, focusing on the performance of multi-class vulnerability detection methods.

To assess the effectiveness of the proposed EGA, first, we evaluate it against leading traditional contract multi-vulnerability detection approaches. Following the methodology outlined by Chen *et al.* [81], we selected two benchmark tools, Mythril and Slither, which are widely recognized in top conferences and journals. The results, as detailed in Table 12, emphasize the performance on Re-entrancy vulnerabilities. Traditional tools generally showed weaker performance on BCCC-SCsVul-2024 dataset, with Slither achieving the highest accuracy at 0.73%. EGA, however, enhanced this accuracy by 0.05%, reaching 0.78%.

The performance gap can largely be attributed to the traditional tools' reliance solely on simplistic patterns for detecting vulnerabilities. For example, Mythril detects Re-entrancy by checking for a '.value' invocation followed by any internal function call. Furthermore, Mythril's inability to differentiate between 'send()', 'transfer()', and 'call()' functions, leads to frequent misidentification of vulnerabilities. In contrast, EGA's sophisticated analytical capabilities offer a significant advantage in accurately identifying vulnerabilities.

Apart from comparing with traditional methods, we also assess EGA against NN-based vulnerability detection tools. We selected seven open-source methods employing various NN models for a comprehensive evaluation: VanillaRNN, LSTM, GRU, DR-GCN, TMP, AME, and SCVHunter. The first three models process the textual sequence of SC code, while the latter four utilize SC graph data. These methods are representative of common approaches in vulnerability detection and are frequently used as benchmarks in recent studies on SC vulnerability detection.

EGA demonstrates significant enhancements across various evaluation metrics, particularly in accuracy. For Re-entrancy and timestamp dependency vulnerabilities, which were detected by all previous methods, EGA showed improvements of 2.0% and 15.0% over SCVHunter and DR-GCN, the top-performing models in sequence processing, respectively. In comparisons with GNN-based tools, EGA surpassed the best-performing approach, AME, by 8.0% and 22.0% for Re-entrancy and timestamp vulnerabilities. This superior performance is attributed to the use of multiple feature categories and the EGA profiling technique, which capture more valuable information than previous methods.

Moreover, a significant drawback of NN-based approaches is their black-box nature, which means their internal workings are not easily interpretable. This lack of explainability renders them suboptimal for assessing the security of SCs. Conversely, the EGA model provides a more transparent and understandable solution

Table 12: Performance Comparison

Methods	Re-entrancy				Timestamp				TransactionOrderDependence			
	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1
Performance comparison with traditional approaches												
Mythril[168]	0.64	0.54	0.42	0.47	-	-	-	-	0.75	0.43	0.57	0.49
Slither[76]	0.73	0.53	0.68	0.59	-	-	-	-	0.46	0.17	0.69	0.28
Performance comparison with NN-based approaches												
Vanilla-RNN [169]	0.51	0.56	0.50	0.52	0.48	0.44	0.53	0.48	0.47	0.52	0.44	0.47
LSTM [170]	0.57	0.65	0.53	0.58	0.53	0.59	0.52	0.55	0.53	0.60	0.50	0.54
GRU [170]	0.61	0.67	0.55	0.60	0.57	0.60	0.52	0.55	0.51	0.63	0.53	0.57
DR-GCN [171]	0.65	0.62	0.59	0.60	0.79	0.63	0.72	0.67	-	-	-	-
TMP [171]	0.69	0.60	0.56	0.57	0.73	0.58	0.74	0.61	-	-	-	-
AME [40]	0.70	0.65	0.61	0.62	0.72	0.62	0.71	0.62	-	-	-	-
SCVHunter [170]	0.76	0.56	0.73	0.63	0.68	0.63	0.73	0.63	0.79	0.51	0.77	0.60
EGA (Proposed method)	0.78	0.59	0.79	0.67	0.94	0.44	0.84	0.58	0.93	0.47	0.88	0.61

by detecting SC vulnerabilities based on precisely defined attributes connected to the different aspects of the source code. This unique benefit highlights the robustness and reliability of the proposed approach, making it a preferable option for determining vulnerabilities in SCs.

In summary, the proposed EGA distinguishes itself through its exceptional detection capabilities and the delicate balance it maintains between precision and recall. Its transparent process further underscores its viability as a promising alternative for comprehensive vulnerability detection, significantly outperforming traditional and NN-based methods.

5.6 Visualizing Generated Profiles Using Genes

This sub-chapter introduces a profiling formula and details the visualization of profiling genes for each type of vulnerability in SCs, as well as for secure SCs. It begins by defining four key parameters: Global Weight, EDM, Peaks, and Local Weight (Average). Following this, the main formula is established. The section concludes by proposing two common profiles: one tailored for vulnerable SCs and another for secure SCs.

5.6.1 Visualization Parameters

This segment details the four key parameters involved in formulating the profiling of SCs. It begins by defining each parameter and presenting the associated formulas. Additionally, it provides explanations for the selection of each parameter, illustrating their relevance and importance in accurately formulating the profiles.

- **Parameter 1: Global weight**

This segment introduces the first parameter of the formula, Global Weight. It is determined by three distinct feature selection algorithms identified in Section 3.4. Table 5 displays a unique score for each feature, based on the feature selection process. We use averaging to consolidate the feature selection values from various

features within the same category into a single comprehensive score. This approach accurately represents each category, yielding seven distinct global features. Due to the variation in these values, ranging from 3681.07 to 81.81, we adopt the z-score normalization method to convert them into standardized weights.
The z-score is calculated by subtracting the mean of the dataset from each value and then dividing the result by the standard deviation of the dataset. The calculated mean and standard deviations were 2740.61 and 2617.72, respectively. The z-score formula applied is:

$$z = (X - \mu)/\sigma$$

where X is a value from the dataset, μ is the mean, and σ is the standard deviation. This technique effectively standardizes data points from disparate scales to uniform ones, thereby preserving their range and behavioral distinctions without distortion.
However, since the main focus is on the magnitude of deviation rather than its direction, we apply the absolute value to the z-scores, thus obtaining positive values. This step ensures that all values are positive and maintains the relative differences between each original data point. The normalized values (absolute z-scores) for each feature category are as follows:

- Opcode: 0.533
- Bytecode: 0.595
- Functional: 0.514
- Solidity: 0.875
- AST: 2.214
- ABI: 0.310
- Lines: 0.577

In essence, these values represent the number of standard deviations an element is from the mean. By converting them to positive values, we emphasize the magnitude of their deviations. This approach facilitates the comparison of different metrics on the same scale, while preserving the significance of their variations.

- **Parameter 2: Earth Mover's Distance (EDM)**

The second parameter of the formula, referred to as EDM, is introduced in this segment. It is the discrepancy between two probability distributions which quantified by the Wasserstein distance, also commonly known as the Earth Mover's Distance (EMD). This metric has gained significant attention for its intuitive interpretation and robust mathematical foundation. The Wasserstein distance originates from the field of optimal transport, where it conceptualizes the minimal "cost" required to transform one distribution into another. Imagine two distributions as two different ways of piling up a certain amount of earth. The Wasserstein distance then represents the least amount of work needed to reshape one pile into the other, where "work" is quantified as the amount of earth moved times the distance it's moved.
Mathematically, for one-dimensional distributions, the Wasserstein distance between cumulative distribution functions (CDFs) $F(x)$ and $G(x)$ is defined as:

$$W(F,G) = \int_{-\infty}^{\infty} |F(x) - G(x)|\, dx$$

This integral calculates the total area between the two CDFs across all values. Intuitively, this measures how much one distribution needs to be "shifted" to resemble the other, offering a clear geometric interpretation.
For empirical distributions, those derived from data samples, the calculation simplifies into a discrete form. Given two ordered samples $X = \{x_1, x_2, ..., x_n\}$ and $Y = \{y_1, y_2, ..., y_n\}$, the empirical Wasserstein distance can be expressed as:

$$W(X,Y) = \frac{1}{n} \sum_{i=1}^{n} |x_i - y_i|$$

In this context, n represents the total number of observations, with x_i and y_i denoting the i-th smallest values in each respective sample.
In summary, the EDM parameter serves to highlight and quantify the cumulative absolute discrepancies between corresponding elements of two ordered samples. This allows it to capture the differences in their distributions effectively. Consequently, EDM is an essential parameter for comparing two different categories of samples to each other.

- **Parameter 3: Peaks Difference**

This segment discusses the third parameter, named Peaks, and focuses on comparing peak patterns between secure and vulnerable contracts. It specifically examines the presence and quantity of peaks identified within each plot, which is crucial for understanding the distinct characteristics of each contract type.
The concept of peaks in a one-dimensional array ‘x[n]‘, where ‘n‘ denotes the index, is fundamental. A point ‘x[p]‘ is recognized as a peak if it fulfills the condition:

$$x[p] \geq x[p-h] \quad \text{and} \quad x[p] \geq x[p+h]$$

for all ‘0 ¡ h ¡= prominence‘. Here, ‘prominence‘ plays a crucial role as it dictates how much higher a point needs to be relative to its surroundings to qualify as a peak, reflecting the point's prominence within the data. The importance of a peak's prominence is highlighted by its ability to measure the peak's relative height above the nearest low point that does not include any higher peak. This assessment helps determine how distinct a peak is in terms of both height and spatial position compared to others.
Additionally, a ‘threshold‘ is employed to exclude insignificant peaks that do not sufficiently surpass the height of their surroundings. A point ‘x[p]‘ is considered a genuine peak if:

$$x[p] - x[p-h] \geq \text{threshold} \quad \text{and} \quad x[p] - x[p+h] \geq \text{threshold}$$

for at least one ‘h‘ within the defined range. Meanwhile, the ‘distance‘ parameter determines the minimal horizontal spacing required between adjacent peaks, ensuring that each detected peak is distinctly separated from others.

In summary, this segment introduces a parameter that analyzes the differences between secure and vulnerable contracts by comparing their peak patterns. It delves into the technical definition of peaks within a data array, emphasizing conditions such as prominence, threshold, and distance to identify significant peaks. Due to their clear distinctiveness, peaks are essential for differentiating between vulnerable and secure SCs.

- **Parameter 4: Local weight(Average)**

This segment introduces the fourth parameter, named Local Weight, which uses a combined approach involving two distinct metrics: EMD and Peaks. The EMD metric measures discrepancies between distributions, while Peaks analysis focuses on structural anomalies within the data. Together, these metrics provide a comprehensive evaluation of the contract's features.
We employ an averaging approach to consolidate these insights into a comprehensive evaluation metric for each feature. It involves calculating the "EDM weight" from the distributional discrepancies and the "Peaks weight" from the structural anomalies. Then, for each feature, we compute the average of these two weights:

$$\text{Local weight} = \frac{\text{EDM Weight} + \text{Peaks Weight}}{2}$$

Local weight value serves as a unified metric, encapsulating the distributional shifts and the prominence of peaks in the data. It provides a balanced perspective on each feature's impact or association with vulnerabilities in SCs. This methodology ensures a holistic view, combining the mathematical precision of EMD with the empirical insights of peak analysis.
In summary, the local weight is a consolidated metric that aggregates the EDM and Peak values with equal weighting proportions. It offers an equitable viewpoint regarding the influence or correlation of each feature with vulnerabilities in SCs. As an illustrative example, the four parameters associated with the ExternalBug vulnerability are systematically delineated beneath corresponding column headings across various feature categories. This detailed information is provided in the associated tables: 13, 14, 15, 16, 17, 18, and 19.

5.6.2 Visualization Genes

This section introduces the aggregation of the four proposed parameters to mathematically profile each vulnerability. It includes a comprehensive table that details the values of each feature under separate labels for vulnerable and secure scenarios. Additionally, the section outlines individual profiles for each vulnerability and presents two common profiles for vulnerable and secure SCs.
To methodically profile each vulnerability, we adopt a two-pronged weighting approach that combines category weights derived from the feature selection process with local weights obtained from integrating EMD and Peaks analysis. Firstly, the global weight for each feature category is determined through the feature selection process. This process evaluates the significance of each feature category in distinguishing between various classes of vulnerabilities. It assigns a weight that reflects the category's overall importance or relevance within the dataset.
Secondly, the local weight is calculated for each feature within the context of specific vulnerabilities. This calculation involves averaging the weights derived from EMD and peak analysis. The EMD weight assesses

how well the feature distinguishes between different distributional shapes, while the peak weight evaluates the feature based on the presence and characteristics of peaks within the data.

For each vulnerability, we multiply the category weight by the local weight to determine a final value for each feature. This approach is applied to all features j from 1 to M within each category i from 1 to N.

$$\sum_{i=1}^{N} G(i) \cdot \sum_{j=1}^{M} L(j)$$

This method provides a thorough and accurate assessment of each feature's relevance to the identified vulnerabilities. By this method, we assign a distinct valuation to each feature for each type of vulnerability, capturing both the overall importance of the feature and its specific relevance to particular vulnerabilities. Utilizing dual weights in this approach allows for a detailed analysis, enabling the identification of features that are particularly indicative of specific types of vulnerabilities.

For an illustrative example, this process has been comprehensively detailed for the ExternalBug vulnerability, as presented in tables 13, 14, 15, 16, 17, 18, and 19. For other vulnerabilities, the presentation is streamlined by summarizing the final total weights of each feature in Table 20, focusing on the aggregate impact of each feature across different vulnerabilities.

Table 13: ExternalBug Opcode features

Group	Global weight	Feature name	EMD weight	Peaks	Local Weight (Average)	Total weight
Opcode	0.533	Opcode Count Features_CALLCODE	0.000000	0.000000	0.000000	0.000000
		Opcode Count Features_EXP	0.008190	0.947959	0.478074	0.254814
		Opcode Count Features_DUP7	0.096911	0.877673	0.487292	0.259727
		Opcode Count Features_SSTORE	0.045180	0.937939	0.491560	0.262001
		Opcode Count Features_DIV	0.041457	0.954987	0.498222	0.265552
		Opcode Count Features_PUSH4	0.044443	0.959773	0.502108	0.267624
		Opcode Count Features_CALLDATALOAD	0.058461	0.949903	0.504182	0.268729
		Opcode Count Features_LOG3	0.173608	0.835801	0.504704	0.269007
		Opcode Count Features_DUP6	0.101349	0.930612	0.515980	0.275018
		Opcode Count Features_DUP4	0.073090	0.967250	0.520170	0.277251
		Opcode Count Features_CALLER	0.102651	0.949454	0.526053	0.280386
		Opcode Count Features_ISZERO	0.074786	0.979662	0.527224	0.281011
		Opcode Count Features_SLOAD	0.098097	0.960221	0.529159	0.282042
		Opcode Count Features_REVERT	0.106569	0.960371	0.533470	0.284340
		Opcode Count Features_DUP3	0.105159	0.973381	0.539270	0.287431
		Opcode Count Features_SUB	0.119726	0.970988	0.545357	0.290675
		Opcode Count Features_JUMPI	0.146752	0.975624	0.561188	0.299113
		Opcode Count Features_MSTORE	0.148133	0.977868	0.563000	0.300079
		Opcode Count Features_JUMP	0.168118	0.968596	0.568357	0.302934
		Opcode Count Features_DUP2	0.167806	0.984148	0.575977	0.306996
		Opcode Count Features_AND	0.188713	0.972035	0.580374	0.309339
		Opcode Count Features_PUSH2	0.179585	0.984447	0.582016	0.310215
		Opcode Count Features_SWAP2	0.207045	0.976073	0.591559	0.315301
		Opcode Count Features_PUSH1	0.204834	0.986092	0.595463	0.317382
		Opcode Count Features_SWAP3	0.239905	0.965605	0.602755	0.321268
		Opcode Count Features_SWAP1	0.240056	0.984148	0.612102	0.326251
		Opcode Count Features_PUSH20	0.256618	0.970390	0.613504	0.326998
		Opcode Count Features_POP	0.250264	0.984148	0.617206	0.328971
		Opcode Count Features_JUMPDEST	0.261531	0.981606	0.621569	0.331296
		Opcode Count Features_CALLVALUE	0.319322	0.952894	0.636108	0.339046
		Opcode Count Features_STOP	0.510519	0.870794	0.690656	0.368120

Continuing with this analysis, we propose profiling images for each vulnerability based on Table 20. This

Table 14: ExternalBug Bytecode Features

Group	Global weight	Feature name	EMD weight	Peaks	Local Weight (Average)	Total weight
Bytecode	0.595	Bytecode Character Count_bytecode_character_c	0.049448	0.992523	0.520985	0.309986
		Bytecode Character Count_bytecode_character_a	0.073613	0.987588	0.530600	0.315707
		Bytecode Character Count_bytecode_character_7	0.199179	0.995962	0.597571	0.355555
		Bytecode Character Count_bytecode_character_1	0.205110	1.000000	0.602555	0.358520
		Bytecode Character Count_bytecode_character_8	0.221888	0.998804	0.610346	0.363156
		Bytecode Character Count_bytecode_character_b	0.233483	0.994317	0.613900	0.365271
		Bytecode Character Count_bytecode_character_6	0.274494	0.996112	0.635303	0.378005
		Bytecode Character Count_bytecode_character_5	0.287914	0.996112	0.642013	0.381998
		Bytecode Length	0.317259	0.990130	0.653694	0.388948
		Bytecode Character Count_bytecode_character_3	0.322957	0.995065	0.659011	0.392111
		Bytecode Character Count_bytecode_character_4	0.357230	0.992074	0.674652	0.401418

Table 15: ExternalBug Functional Features

Group	Global weight	Feature name	EMD weight	Peaks	Local Weight (Average)	Total weight
Functional	0.514	Functional_Number of ”public” functions	0.308032	0.944220	0.626126	0.321829

Table 16: ExternalBug ABI Features

Group	Global weight	Feature name	EMD weight	Peaks	Local Weight (Average)	Total weight
ABI	0.310	ABI Features_len_non_zero_input	0.018212	0.926723	0.472468	0.146465
		ABI Features_len_zero_input	0.027332	0.926125	0.476729	0.147786
		ABI Features_len_list_input	0.038155	0.938837	0.488496	0.151434
		ABI Features_len_non_zero_output	0.105717	0.942575	0.524146	0.162485
		ABI Features_len_list_stateMut	0.142183	0.908778	0.525481	0.162899
		ABI Features_len_list_constant	0.119574	0.947361	0.533467	0.165375
		ABI Features_len_list_payable	0.205937	0.921190	0.563563	0.174705

Table 17: ExternalBug AST Features

Group	Global weight	Feature name	EMD weight	Peaks	Local Weight (Average)	Total weight
AST	2.214	AST Features_ast_nodetype	0.067997	0.154778	0.111388	0.246612
		AST Features_ast_len_exportedSymbols	0.233452	0.906386	0.569919	1.261800
		AST Features_ast_len_nodes	0.238983	0.908629	0.573806	1.270407
		AST Features_ast_id	0.412125	0.974278	0.693202	1.534749
		AST Features_ast_src	1.000000	0.994467	0.997233	2.207875

Table 18: ExternalBug Solidity Features

Group	Global weight	Feature name	EMD weight	Peaks	Local Weight (Average)	Total weight
Solidity	0.875	Solidity call	0.096416	0.267534	0.181975	0.159228

Table 19: ExternalBug Lines Features

Group	Global weight	Feature name	EMD weight	Peaks	Local Weight (Average)	Total weight
Lines	0.577	Lines_duplicate	0.243175	0.952894	0.598034	0.345066
		Lines_Code	0.241212	0.996261	0.618737	0.357011
		Lines_total	0.273696	0.999551	0.636624	0.367332

is complemented by secure profiling using a series of heatmap plots, ranging from Fig. 60 to Fig. 71. As observable from the results, there is a noticeable similarity among the different vulnerabilities. However, the secure profile differs in both features and values. Furthermore, figure 72 illustrates all vulnerabilities alongside secure features, with each category of features represented in a different color. From this analysis, it can be inferred that the behavior of the vulnerabilities is similar to one another.

Analysis of previous behavior profiling reveals that attitudes associated with various vulnerabilities are notably similar. Further evidence is provided in figure 73, which depicts a heatmap illustrating the correlations among all vulnerabilities and secure SCs. The heatmap clearly shows that most vulnerabilities are closely correlated, supporting the notion of their similarities. For example, the tight correlations among Timestamp, TransactionOrderDependence, and UnsendReturn indicate that distinct profiling based on the analyzed features is challenging.

Given these observed similarities, it is essential to establish a common vulnerability profile. Fig.74 and Fig. 75 propose two distinct profiles, depicted through Histogram2DContour plots. These profiles provide a visual representation of common patterns in both secure and vulnerable SC behaviors.

The secure profile in Fig. 75 exhibits pronounced density around the center, specifically between coordinates (40, 0.5) and (60, 0.5). This concentration suggests that secure behaviors typically cluster around these values, which fall under the 'Opcode' category (according to Table 20). The profile's predominantly horizontal extension indicates greater variability along the X-axis compared to the Y-axis, suggesting that one variable significantly influences secure behavior which is 'AST_nodetype' (according to Table 20). Additionally, a less dense area around (20, 2) highlights the presence of atypical secure behaviors associated with the 'Bytecode' category (according to Table 20).

Conversely, the vulnerabilities profile in Fig. 74 exhibits a significant central concentration, ranging from coordinates (20, 0.5) to (60, 0.5). This clustering correlates with categories such as Bytecode, Opcode, Functional, and Lines, as detailed in Table 20. However, this profile is more spread out, suggesting a broader range of behaviors that belong to different types of vulnerabilities. Similar to the secure profile, it extends predominantly horizontally, indicating variability in the same influencing factor. Despite this variability, the minimal presence of outliers suggests that vulnerabilities are more uniformly distributed within the observed range.

Both profiles exhibit a similar shape and horizontal spread, suggesting a common influencing factor that significantly impacts both secure and vulnerable behaviors in SCs. However, the secure profile is more tightly clustered around specific regions, signifying a more defined range of secure behaviors, while the vulnerabilities profile is broader, highlighting a wider range of potential vulnerabilities.

In a nutshell, this section outlines a systematic method to mathematically profile each vulnerability by ag-

Table 20: Total Weights

		Total weight										
Group name	Feature name	GasException	MishandledException	Timestamp	TransactionOrderDependence	UnusedReturn	WeakAccessMod	CallToUnknown	DenialOfService	IntegerUO	Re-entrancy	secure
ABI	ABI Features_len_list_constant	0.15925	0.00000	0.22036	0.17683	0.18529	0.15938	0.15485	0.16163	0.15022	0.00000	0.16258
	ABI Features_len_list_input	0.14867	0.14425	0.19390	0.15991	0.16408	0.15612	0.14852	0.15081	0.13469	0.12641	0.15351
	ABI Features_len_list_name	0.00000	0.15051	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.13718	0.00000
	ABI Features_len_list_output	0.00000	0.14515	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.12988	0.00000
	ABI Features_len_list_payable	0.15961	0.14762	0.23487	0.18747	0.19247	0.15734	0.15296	0.15998	0.15697	0.12988	0.17059
	ABI Features_len_list_stateMut	0.15076	0.00000	0.21355	0.17226	0.17782	0.15426	0.14909	0.15388	0.14071	0.00000	0.16596
	ABI Features_len_list_type	0.00000	0.14819	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.13095	0.00000
	ABI Features_len_non_zero_input	0.14565	0.14201	0.19448	0.15312	0.16047	0.15614	0.14875	0.15178	0.12986	0.12435	0.15188
	ABI Features_len_non_zero_output	0.15714	0.00000	0.21150	0.17468	0.18183	0.15667	0.15385	0.15952	0.14939	0.00000	0.15318
	ABI Features_len_zero_input	0.14208	0.14134	0.17815	0.15723	0.15878	0.15022	0.14369	0.14541	0.13149	0.12637	0.13898
AST	AST Features_ast_id	1.11830	0.00000	1.66154	1.51190	1.60363	1.11882	1.03792	1.04754	1.24105	0.00000	1.09345
	AST Features_ast_len_exportedSymbols	1.01390	0.00000	1.33169	1.30233	1.36859	1.07663	0.98779	1.00350	1.04163	0.00000	1.07177
	AST Features_ast_len_nodes	1.02808	1.02578	1.38156	1.30857	1.37269	1.09331	1.00263	1.01947	1.04941	0.81742	1.08755
	AST Features_ast_nodetype	1.14978	0.26290	0.16747	0.16335	0.16562	1.27354	1.10700	1.10700	0.10018	0.55329	2.21400
	AST Features_ast_src	1.86304	1.33157	2.21400	2.21400	2.21351	1.67214	1.67830	1.80655	2.08272	1.27887	1.71695
Bytecode	Bytecode Character Count_bytecode_character_1	0.32360	0.30776	0.41982	0.37864	0.38900	0.30771	0.31399	0.32275	0.35702	0.28731	0.25962
	Bytecode Character Count_bytecode_character_2	0.00000	0.30663	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.28428	0.00000
	Bytecode Character Count_bytecode_character_3	0.33652	0.31325	0.46710	0.41687	0.41922	0.31217	0.32012	0.32923	0.39226	0.29001	0.27675
	Bytecode Character Count_bytecode_character_4	0.33379	0.00000	0.47486	0.42484	0.42459	0.31103	0.31916	0.33184	0.39750	0.00000	0.27277
	Bytecode Character Count_bytecode_character_5	0.33356	0.31144	0.45537	0.40369	0.41197	0.31165	0.31965	0.32934	0.38228	0.29143	0.27123
	Bytecode Character Count_bytecode_character_6	0.33034	0.31039	0.44494	0.40120	0.40624	0.30987	0.31543	0.32569	0.37025	0.28784	0.26600
	Bytecode Character Count_bytecode_character_7	0.31969	0.30559	0.40691	0.37638	0.38254	0.30511	0.30999	0.31746	0.35041	0.28343	0.25782
	Bytecode Character Count_bytecode_character_8	0.32613	0.30897	0.42464	0.38391	0.39436	0.30762	0.31368	0.32209	0.36174	0.28721	0.26258
	Bytecode Character Count_bytecode_character_9	0.00000	0.30881	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.28887	0.00000
	Bytecode Character Count_bytecode_character_a	0.30244	0.29595	0.35369	0.33568	0.34558	0.29874	0.30135	0.30678	0.30148	0.27365	0.24905
	Bytecode Character Count_bytecode_character_b	0.33408	0.30730	0.43047	0.38453	0.41305	0.31265	0.31535	0.33231	0.35572	0.29138	0.26431
	Bytecode Character Count_bytecode_character_c	0.30064	0.00000	0.34477	0.32975	0.34054	0.29826	0.30160	0.30669	0.30014	0.00000	0.24371
	Bytecode Character Count_bytecode_character_d	0.00000	0.30386	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.28096	0.00000
	Bytecode Character Count_bytecode_character_f	0.00000	0.31132	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.28858	0.00000
	Bytecode Entropy	0.00000	0.59500	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.57997	0.00000
	Bytecode Length	0.33539	0.00000	0.49081	0.41385	0.41787	0.31317	0.32086	0.32661	0.38039	0.00000	0.27205
Functional	Functional_Number of "public" functions	0.25997	0.25422	0.00000	0.33731	0.36564	0.23921	0.00000	0.22342	0.28866	0.19733	0.25593
	Functional_Number of functions	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
Lines	Lines_duplicate	0.28800	0.28504	0.34261	0.34568	0.35844	0.26552	0.26156	0.24255	0.30088	0.21971	0.29199
	Lines_total	0.29555	0.29989	0.35937	0.36716	0.40030	0.27920	0.27538	0.25644	0.33519	0.23661	0.28343
	Lines_Code	0.30100	0.00000	0.36919	0.36377	0.37864	0.27746	0.27230	0.25805	0.32482	0.00000	0.28669
Opcode	Opcode Count Features_ADD	0.00000	0.26753	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.25789	0.22075
	Opcode Count Features_AND	0.28318	0.00000	0.37235	0.33123	0.33495	0.26852	0.27713	0.28365	0.31195	0.00000	0.00000
	Opcode Count Features_CALLCODE	0.00382	0.00086	0.01079	0.02846	0.03949	0.00097	0.04278	0.02220	0.00898	0.01674	0.00252
	Opcode Count Features_CALLDATALOAD	0.28368	0.25659	0.38522	0.33953	0.34095	0.26521	0.27548	0.28333	0.26763	0.25398	0.20772
	Opcode Count Features_CALLER	0.27196	0.25865	0.34203	0.30978	0.31030	0.25980	0.26794	0.27274	0.28688	0.24960	0.20872
	Opcode Count Features_CALLVALUE	0.29587	0.26993	0.40192	0.36570	0.36039	0.26913	0.27975	0.28951	0.34159	0.25302	0.24370
	Opcode Count Features_DIV	0.26234	0.25621	0.30032	0.28623	0.29339	0.25604	0.26074	0.26578	0.26415	0.24271	0.20509
	Opcode Count Features_DUP1	0.00000	0.27590	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.25330	0.00000
	Opcode Count Features_DUP2	0.28094	0.26968	0.35848	0.32418	0.33493	0.26888	0.27299	0.27959	0.30117	0.24994	0.22812
	Opcode Count Features_DUP3	0.27207	0.26447	0.33158	0.30611	0.31583	0.26369	0.26791	0.27427	0.28314	0.24674	0.22208
	Opcode Count Features_DUP4	0.26835	0.26100	0.31370	0.29523	0.30705	0.25961	0.26554	0.27074	0.27933	0.24864	0.20241
	Opcode Count Features_DUP5	0.00000	0.25634	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.26099	0.00000
	Opcode Count Features_DUP6	0.26386	0.25361	0.34229	0.30142	0.30229	0.25333	0.26316	0.26852	0.25918	0.25101	0.17541
	Opcode Count Features_DUP7	0.24479	0.00000	0.30810	0.27491	0.28880	0.23647	0.25294	0.25629	0.26274	0.00000	0.14926
	Opcode Count Features_EXP	0.26204	0.25407	0.28551	0.27353	0.28345	0.25224	0.27165	0.27469	0.27402	0.27049	0.14050
	Opcode Count Features_ISZERO	0.27127	0.26378	0.31991	0.29782	0.30993	0.26567	0.27132	0.27679	0.27690	0.25144	0.21034
	Opcode Count Features_JUMP	0.28720	0.26564	0.35515	0.31980	0.34604	0.27539	0.26753	0.28311	0.27486	0.24914	0.22377
	Opcode Count Features_JUMPDEST	0.29329	0.27382	0.39862	0.35033	0.35813	0.27567	0.27634	0.29115	0.30260	0.25437	0.23557
	Opcode Count Features_JUMPI	0.27611	0.26602	0.34191	0.31560	0.32773	0.26739	0.27058	0.27635	0.28843	0.24329	0.23330
	Opcode Count Features_LOG3	0.24439	0.23013	0.34254	0.29276	0.29625	0.22998	0.24633	0.24836	0.27475	0.23501	0.17376
	Opcode Count Features_MLOAD	0.00000	0.27216	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.23959	0.00000
	Opcode Count Features_MSTORE	0.27741	0.26662	0.35246	0.31710	0.32749	0.26665	0.27009	0.27651	0.29206	0.24607	0.22942
	Opcode Count Features_POP	0.28978	0.00000	0.38208	0.34725	0.35622	0.27300	0.27849	0.28496	0.31815	0.00000	0.23487
	Opcode Count Features_PUSH1	0.28593	0.27175	0.37095	0.33428	0.34514	0.27223	0.27668	0.28359	0.31033	0.25023	0.23560
	Opcode Count Features_PUSH2	0.28465	0.27022	0.36307	0.32538	0.33476	0.27159	0.27679	0.28759	0.29964	0.26100	0.21183
	Opcode Count Features_PUSH20	0.28842	0.27070	0.39955	0.34870	0.34905	0.27202	0.27910	0.27757	0.29713	0.24571	0.22857
	Opcode Count Features_PUSH4	0.26009	0.25618	0.29283	0.29095	0.29858	0.25697	0.26114	0.26535	0.26099	0.24331	0.20225
	Opcode Count Features_RETURN	0.00000	0.27494	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.25242	0.00000
	Opcode Count Features_REVERT	0.27922	0.25973	0.31240	0.33676	0.30990	0.26797	0.26239	0.27407	0.26480	0.23305	0.22723
	Opcode Count Features_SLOAD	0.27187	0.26020	0.34871	0.30539	0.31339	0.26156	0.26916	0.27356	0.28303	0.24866	0.21043
	Opcode Count Features_SSTORE	0.26111	0.25200	0.30667	0.28509	0.29288	0.25365	0.25955	0.26350	0.26430	0.23968	0.20676
	Opcode Count Features_STOP	0.29673	0.25889	0.41345	0.41022	0.38389	0.25814	0.26819	0.27921	0.36264	0.22400	0.28854
	Opcode Count Features_SUB	0.27529	0.26524	0.33462	0.31178	0.31894	0.26297	0.26996	0.27637	0.29490	0.25366	0.20961
	Opcode Count Features_SWAP1	0.29354	0.27405	0.38986	0.34713	0.35570	0.27321	0.28268	0.29128	0.33374	0.26385	0.22347
	Opcode Count Features_SWAP2	0.28686	0.27053	0.37169	0.33611	0.34058	0.26820	0.27708	0.28537	0.32183	0.25996	0.21692
	Opcode Count Features_SWAP3	0.29174	0.26794	0.39869	0.34670	0.34235	0.26998	0.28069	0.29103	0.32075	0.26361	0.21373
Solidity	Solidity call	0.12486	0.12083	0.21697	0.19409	0.19723	0.10788	0.16848	0.15628	0.23733	0.18739	0.04136

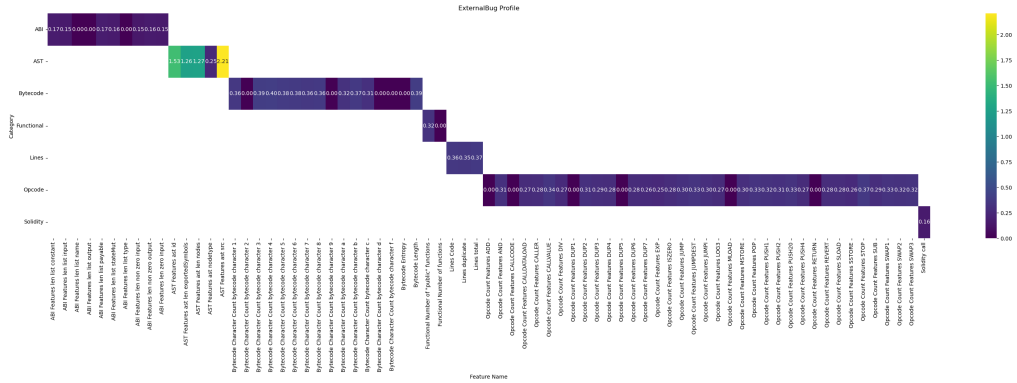


Figure 60: Individual Heatmap Behavior Profile: ExternalBug

gregating four proposed parameters. The method effectively combines category and local weights through a two-pronged approach, yielding a precise valuation for each feature's relevance to different vulnerabilities. Moreover, the section explores the behavioral patterns of secure and vulnerable SCs through detailed heatmap plots and comparative analyses. It demonstrates that the similarities in code structure, arithmetic issues, and exploitation methods across different vulnerabilities lead to highly similar profiling for each vulnerability.

Continuing this analysis, the data in the comprehensive Table 20 indicate differences in feature values across various vulnerabilities. For example, while the entropy of the bytecode shows positive and slightly different values for both Re-entrancy and Mishandled Exceptions, it is zero for the secure label. This pattern is consistent across other features, where the values for vulnerabilities are similar to each other but distinct from those of the secure label. Furthermore, ranging from Fig. 60 to Fig. 71, we observe clear similarities across these plots. This similarity leads us to create two common profiles for vulnerable and secure SCs. Common profiles aggregate characteristics from multiple vulnerabilities into unified profiles for vulnerable and secure SCs. They help clarify the comprehensive anomalies, providing a clearer comparative perspective for identifying key areas of risk and resilience in SC security.

As a summery, this section presents a methodical approach to mathematically profile vulnerabilities in SCs by consolidating four proposed parameters. By amalgamating category and local weights, a precise assessment of each feature's relevance to different vulnerabilities is achieved. The analysis explores the behavioral patterns of both secure and vulnerable SCs through detailed heatmap plots and comparative examinations, revealing striking similarities in code structure, arithmetic issues, and exploitation methods across different vulnerabilities.

Moreover, the data in Table 20 highlight variations in feature values across various vulnerabilities, with distinct patterns emerging between vulnerabilities and the secure label. The examination of heatmap plots from Fig. 60 to Fig. 71 underscores these similarities, prompting the creation of common profiles for vulnerable and secure SCs. These profiles amalgamate characteristics from multiple vulnerabilities, offering a comprehensive perspective for identifying critical areas of risk and resilience in SC security.

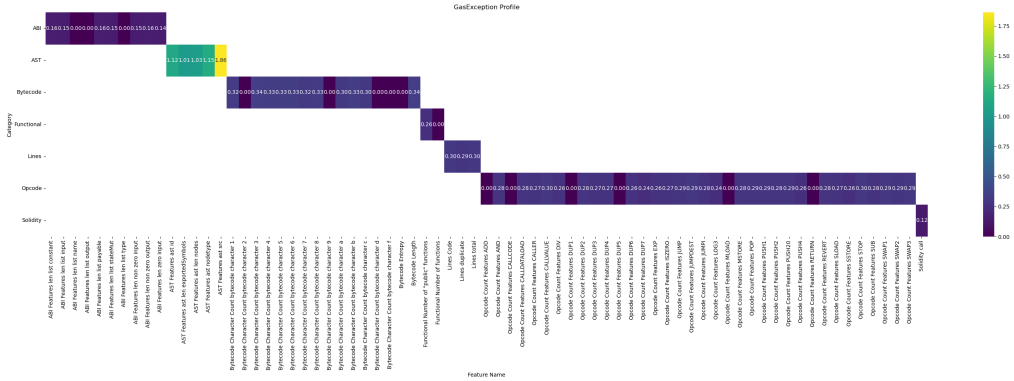


Figure 61: Individual Heatmap Behavior Profile: GasException

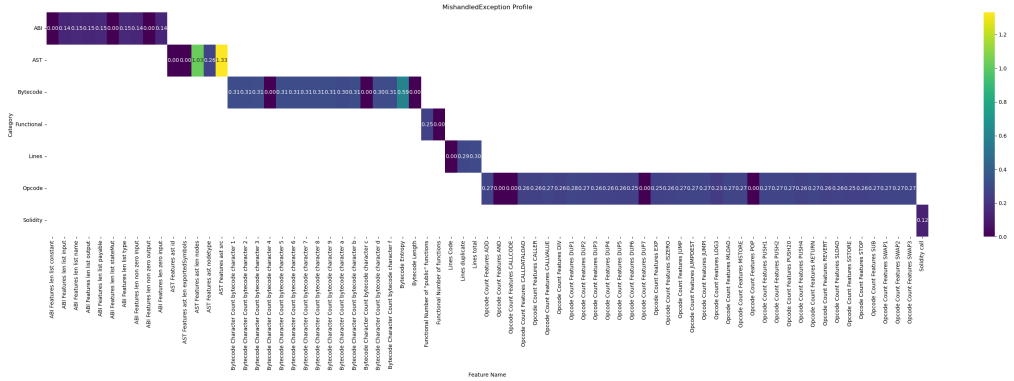


Figure 62: Individual Heatmap Behavior Profile: MishandledException

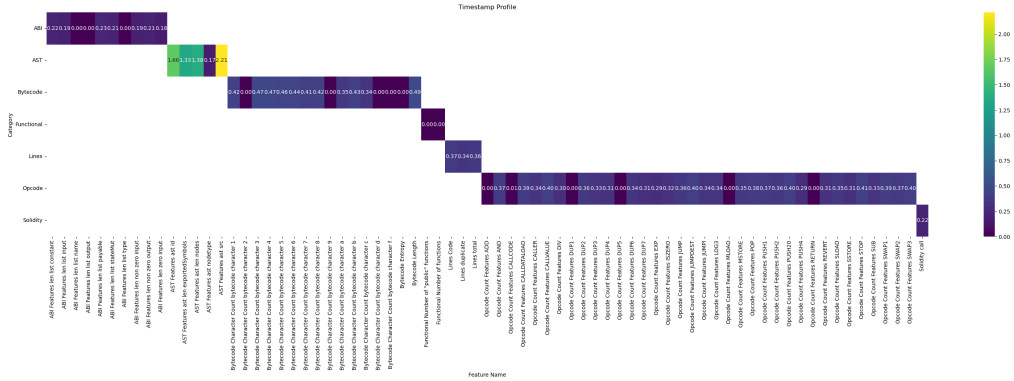


Figure 63: Individual Heatmap Behavior Profile: Timestamp

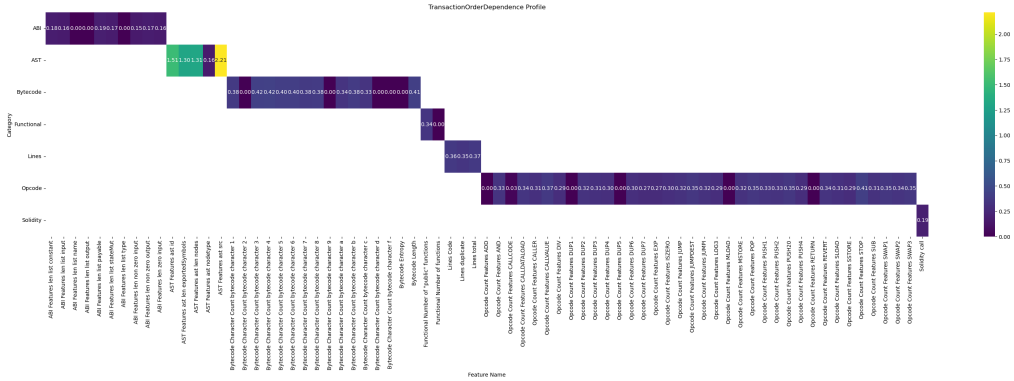


Figure 64: Individual Heatmap Behavior Profile: TransactionOrderDependence

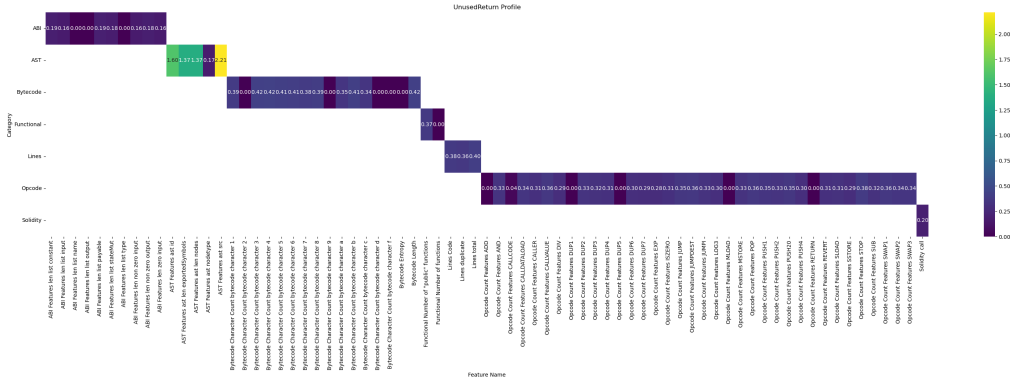


Figure 65: Individual Heatmap Behavior Profile: UnusedReturn

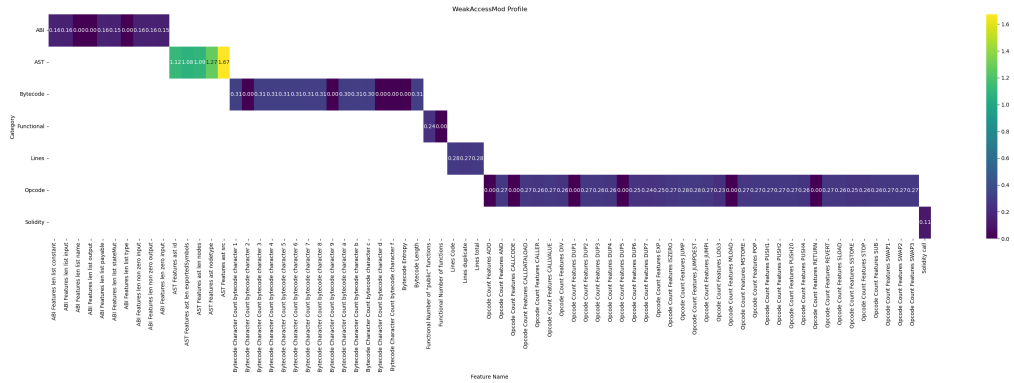


Figure 66: Individual Heatmap Behavior Profile: WeakAccessMod

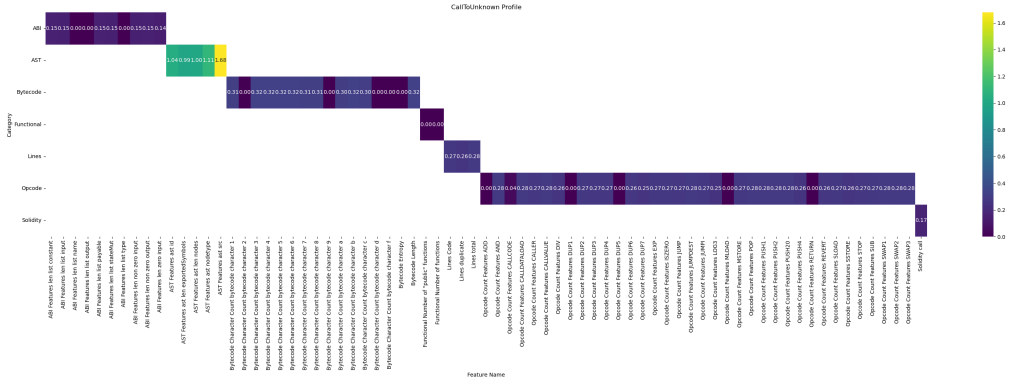


Figure 67: Individual Heatmap Behavior Profile: CallToUnknown

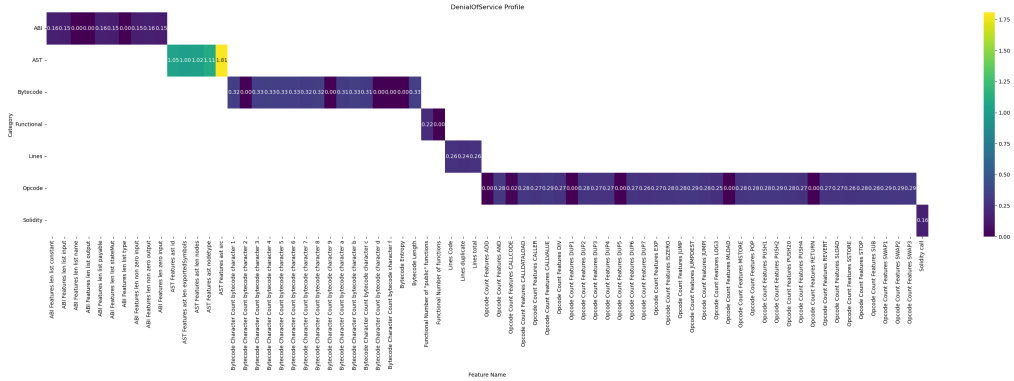


Figure 68: Individual Heatmap Behavior Profile: DenialOfService

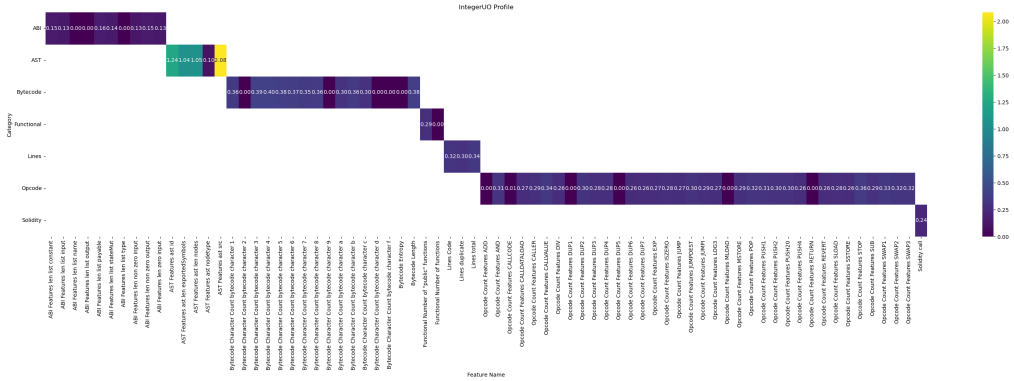


Figure 69: Individual Heatmap Behavior Profile: IntegerUO

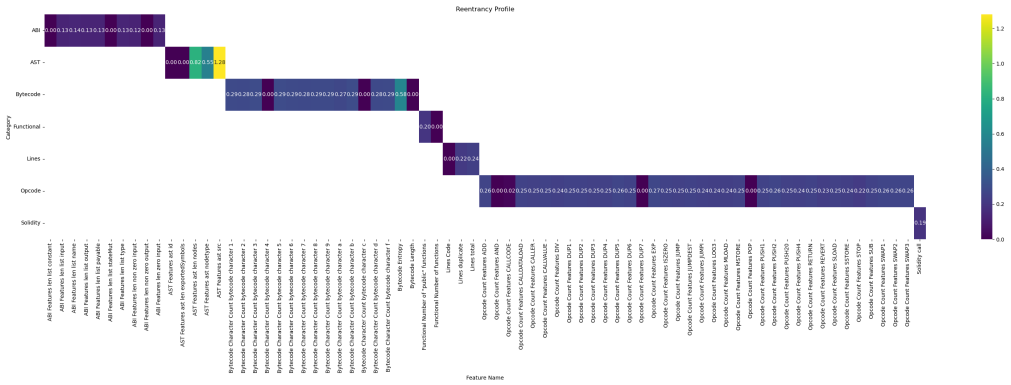


Figure 70: Individual Heatmap Behavior Profile: Re-entrancy

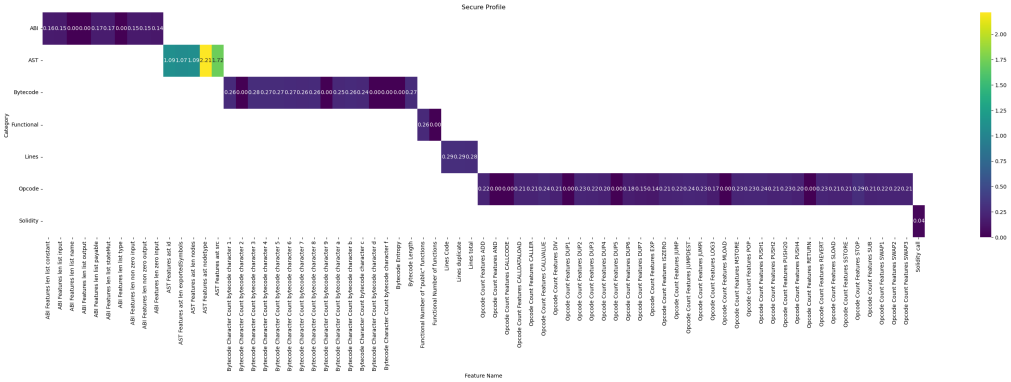


Figure 71: Individual Heatmap Behavior Profile: Secure

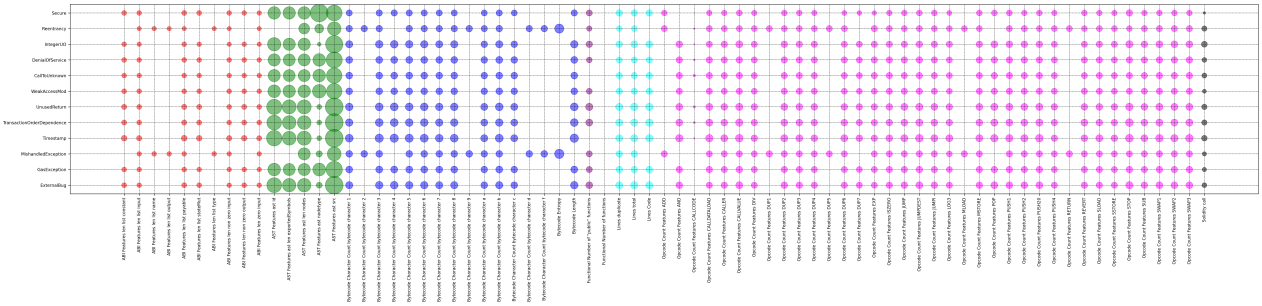


Figure 72: Comparative Profiling Visualization: The colors assigned to different categories: Red (Represents the ABI category), Green (Designated for the AST category), Blue (Used for Bytecode-related features), Purple (Highlights features in the Functional category), Cyan (Indicates features associated with Lines of code or specific line-based metrics), Magenta (Assigned to features derived from OpCodes), Black (Represents features specific to the Solidity programming language)

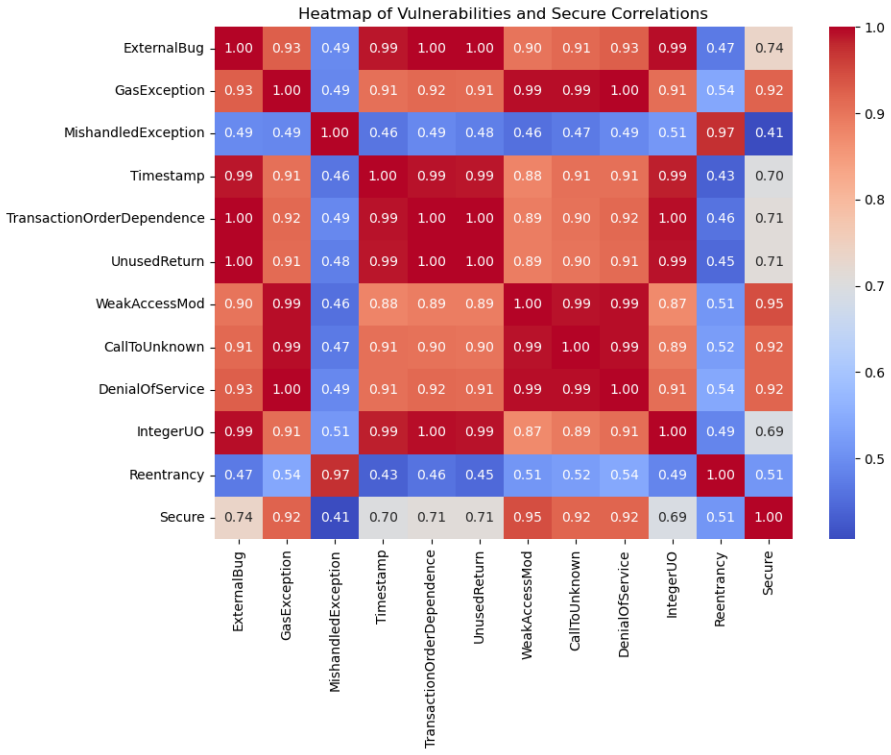


Figure 73: Heatmap of Vulnerabilities and Secure Correlations

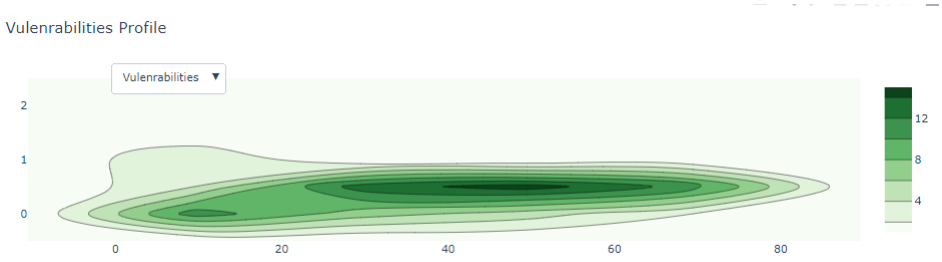


Figure 74: Common Behaviour Profile: Vulnerable

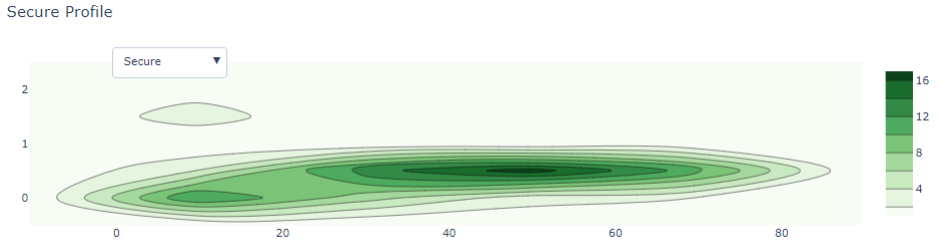


Figure 75: Common Behaviour Profile: Secure

5.7 Concluding Remarks

This chapter comprehensively analyzes the experimental results, specifically addressing the contributions referenced as CONT8 in Chapter 1. The analysis aims to identify the significance of features within each category through profiling each vulnerability. Additionally, we propose two common profiles for vulnerable and secure SCs.

6 Conclusion and Future Works

With the advent of blockchain networks, traditional contracts have transitioned to SCs, essential for maintaining and upholding trust within these systems. Regarding the security of SCs, current analysis methods can be categorized into two groups: traditional and NNs methods. Traditional methods, such as rule-based and machine learning, often lack precision and effectiveness. In contrast, advanced techniques grapple with complexity, explainability, and reliance on training datasets. This paper presents an enhanced version of the GA for detecting, identifying, and profiling SCs' vulnerabilities.

The enhancements to the regular GA encompass several strategic modifications to improve its efficiency and effectiveness in problem-solving. By increasing the population size to 10,000 individuals, the algorithm significantly broadens the diversity of potential solutions, enhancing its capability to thoroughly explore the multidimensional solution space and avoid premature convergence on suboptimal solutions. Moreover, introducing SSD checks before crossover events ensures that genetic recombination occurs between individuals that are neither too similar nor excessively dissimilar, maintaining genetic diversity while preventing the creation of implausible offspring. An adaptive mutation rate is also implemented, which modulates mutation frequencies based on the evolutionary stage. This starts with a higher mutation rate to encourage a wide exploration of the solution space, followed by a reduction in the rate as promising solutions emerge, thereby refining these solutions and ensuring a balanced approach between exploring diverse options and exploiting promising configurations.

Also, we create a new analyzer named BCCC-SCsVulLyzer, which leverages GAs explicitly for profiling SCs vulnerabilities. Additionally, we have compiled an extensive benchmark dataset of 111,897 Solidity source code samples, covering a broad spectrum of examples to ensure the effective validation of our method. We subject our methodology to thorough testing and experimentation, revealing that our model identifies 11 vulnerabilities, including ExternalBug, GasException, MishandledException, Timestamp, TransactionOrderDependence, UnusedReturn, WeakAccessMod, CallToUnknown, DenialOfService, IntegerUO, and Reentrancy. The proposed approach detects broader vulnerabilities than traditional and NN-based methods, while previous models only detect two or three vulnerabilities.

Our proposed approach, validated through comprehensive experimental analysis, demonstrates superior precision and accuracy. The profiling technique enhances the transparency and explainability of our model. With an accuracy ranging from 0.78 to 0.97, our model robustly identifies vulnerabilities. The precision in predicting specific types of vulnerabilities, ranging between 0.79 and 0.88, reflects its accuracy in correctly identifying true instances of each vulnerability type. In practical applications, our model can rapidly analyze the security features of smart contracts, allowing for immediate assessment and tagging of each smart contract's vulnerability. This real-time solution significantly contributes to identifying potential threats in the blockchain ecosystem.

Looking ahead, this work opens up several promising avenues for future research. While this paper primarily focused on identifying 11 vulnerabilities in SCs, there is potential to profile a broader range of vulnerabilities. A deeper understanding of various types of vulnerabilities could make the profiler more versatile and practical, better addressing a range of threats. Additionally, scalability is a key consideration. Enhancing the

profiler's ability to efficiently analyze a larger volume of SCs will be crucial as the number and complexity of SCs continue to grow. Lastly, the taxonomies outlined in this study should be continually revised and expanded to account for new developments. Our understanding and classification of SCs should advance in parallel with their technological and security protocols. This will ensure that our procedures remain relevant and efficient, thus enhancing the security of SCs.

Bibliography

[1] 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*, pp. 254–269, 2016.

[2] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 65–68, 2018.

[3] 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*, pp. 653–663, 2018.

[4] 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*, pp. 664–676, 2018.

[5] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pp. 422–430, Springer, 2016.

[6] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[7] J. Krupp and C. Rossow, "{teEther}: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1317–1333, 2018.

[8] 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*, pp. 259–269, 2018.

[9] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, IEEE, 2018.

[10] 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)*, pp. 1186–1189, IEEE, 2019.

[11] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[12] 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)*, pp. 1591–1607, 2019.

[13] P. Praitheeshan, L. Pan, X. Zheng, A. Jolfaei, and R. Doss, "Solguard: Preventing external call issues in smart contract-based multi-agent robotic systems," *Information Sciences*, vol. 579, pp. 150–166, 2021.

[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*, pp. 47–59, 2021.

[15] L. Zhang, J. Wang, W. Wang, Z. Jin, Y. Su, and H. Chen, "Smart contract vulnerability detection combined with multi-objective detection," *Computer Networks*, vol. 217, p. 109289, 2022.

[16] J. Ye, M. Ma, Y. Lin, L. Ma, Y. Xue, and J. Zhao, "Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures," *Journal of Systems and Software*, vol. 192, p. 111410, 2022.

[17] X. Xie, H. Wang, Z. Jian, Y. Fang, Z. Wang, and T. Li, "Block-gram: Mining knowledgeable features for efficiently smart contract vulnerability detection," *Digital Communications and Networks*, 2023.

[18] 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. 103423, 2023.

[19] H. Zhang, W. Zhang, Y. Feng, and Y. Liu, "Svscanner: Detecting smart contract vulnerabilities via deep semantic extraction," *Journal of Information Security and Applications*, vol. 75, p. 103484, 2023.

[20] Z. Liu, M. Jiang, S. Zhang, J. Zhang, and Y. Liu, "A smart contract vulnerability detection mechanism based on deep learning and expert rules," *IEEE Access*, 2023.

[21] Z. Tian, B. Tian, J. Lv, Y. Chen, and L. Chen, "Enhancing vulnerability detection via ast decomposition and neural sub-tree encoding," *Expert Systems with Applications*, p. 121865, 2023.

[22] S. Driessen, D. Di Nucci, D. Tamburri, and W.-J. van den Heuvel, "Solar: Automated test-suite generation for solidity smart contracts," *Science of Computer Programming*, p. 103036, 2023.

[23] W. Jie, Q. Chen, J. Wang, A. S. V. Koe, J. Li, P. Huang, Y. Wu, and Y. Wang, "A novel extended multimodal ai framework towards vulnerability detection in smart contracts," *Information Sciences*, vol. 636, p. 118907, 2023.

[24] S. Ji, J. Wu, J. Qiu, and J. Dong, "Effuzz: Efficient fuzzing by directed search for smart contracts," *Information and Software Technology*, vol. 159, p. 107213, 2023.

[25] I. M. Ali, N. Lasla, M. M. Abdallah, and A. Erbad, "Srp: An efficient runtime protection framework for blockchain-based smart contracts," *Journal of Network and Computer Applications*, vol. 216, p. 103658, 2023.

[26] X. Ren, Y. Wu, J. Li, D. Hao, and M. Alam, "Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network," *Computers and Electrical Engineering*, vol. 109, p. 108766, 2023.

[27] D. Yuan, X. Wang, Y. Li, and T. Zhang, "Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding," *Journal of Systems and Software*, vol. 202, p. 111699, 2023.

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

[29] K. Zhou, J. Huang, H. Han, B. Gong, A. Xiong, W. Wang, and Q. Wu, "Smart contracts vulnerability detection model based on adversarial multi-task learning," *Journal of Information Security and Applications*, vol. 77, p. 103555, 2023.

[30] X. Wu, X. Du, Q. Yang, A. Liu, N. Wang, and W. Wang, "Taintguard: Preventing implicit privilege leakage in smart contract based on taint tracking at abstract syntax tree level," *Journal of Systems Architecture*, p. 102925, 2023.

[31] J. Li, Z. Zhao, Z. Su, and W. Meng, "Gas-expensive patterns detection to optimize smart contracts," *Applied Soft Computing*, p. 110542, 2023.

[32] L. Wang, H. Cheng, Z. Zheng, A. Yang, and M. Xu, "Temporal transaction information-aware ponzi scheme detection for ethereum smart contracts," *Engineering Applications of Artificial Intelligence*, vol. 126, p. 107022, 2023.

[33] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "A formal verification approach for composite smart contracts security using fsm," *Journal of King Saud University-Computer and Information Sciences*, vol. 35, no. 1, pp. 70–86, 2023.

[34] D. Chen, L. Feng, Y. Fan, S. Shang, and Z. Wei, "Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention," *Journal of Systems and Software*, vol. 202, p. 111705, 2023.

[35] H. Liu, Y. Fan, L. Feng, and Z. Wei, "Vulnerable smart contract function locating based on multi-relational nested graph convolutional network," *Journal of Systems and Software*, p. 111775, 2023.

[36] S. HajiHosseinKhani, A. H. Lashkari, and A. M. Oskui, "Unveiling vulnerable smart contracts: Toward profiling vulnerable smart contracts using genetic algorithm and generating benchmark dataset," *Blockchain: Research and Applications*, p. 100171, 2023.

[37] P. Li, G. Wang, X. Xing, J. Zhu, W. Gu, and G. Zhai, "Detecting abnormal behaviors in smart contracts using opcode sequences," *Computer Communications*, vol. 220, pp. 12–22, 2024.

[38] J. Cai, B. Li, T. Zhang, J. Zhang, and X. Sun, "Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction," *Journal of Systems and Software*, vol. 209, p. 111919, 2024.

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

[40] 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.

[41] 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.

[42] 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. 19685–19695, 2020.

[43] M. Ding, P. Li, S. Li, and H. Zhang, "Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection," in *Evaluation and Assessment in Software Engineering*, pp. 321–328, 2021.

[44] N. F. Samreen and M. H. Alalfi, "Reentrancy vulnerability identification in ethereum smart contracts," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 22–29, IEEE, 2020.

[45] C. Langensiepen, A. Lotfi, and S. Puteh, "Activities recognition and worker profiling in the intelligent office environment using a fuzzy finite state machine," in *2014 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pp. 873–880, IEEE, 2014.

[46] A. Fernández-Isabel, P. Peixoto, I. M. de Diego, C. Conde, and E. Cabello, "Combining dynamic finite state machines and text-based similarities to represent human behavior," *Engineering Applications of Artificial Intelligence*, vol. 85, pp. 504–516, 2019.

[47] A. Guillén, Y. Gutiérrez, and R. Muñoz, "Natural language processing technologies for document profiling," pp. 284–290, 2017.

[48] X. Feng, Q. Wang, X. Zhu, and S. Wen, "Bug searching in smart contract," *arXiv preprint arXiv:1905.00799*, 2019.

[49] N. Szabo, "Formalizing and securing relationships on public networks," *First monday*, 1997.

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

[51] 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*, pp. 164–186, Springer, 2017.

[52] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn, "Flashrelate: extracting relational data from semi-structured spreadsheets using examples," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 218–228, 2015.

[53] M. A. Ferrag, M. Derdour, M. Mukherjee, A. Derhab, L. Maglaras, and H. Janicke, "Blockchain technologies for the internet of things: Research issues and challenges," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2188–2204, 2018.

[54] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen, "Exploring the attack surface of blockchain: A systematic overview," *arXiv preprint arXiv:1904.03487*, 2019.

[55] L.-H. Zhu, B.-K. Zheng, M. Shen, F. Gao, H.-Y. Li, and K.-X. Shi, "Data security and privacy in bitcoin system: a survey," *Journal of Computer Science and Technology*, vol. 35, pp. 843–862, 2020.

[56] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: a survey," *arXiv preprint arXiv:1908.08605*, 2019.

[57] B. T. Bug", ""batch overflow bug on ethereum erc20 token contracts and safemath"."

[58] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 204–217, IEEE, 2018.

[59] M. Coblenz, "Obsidian: a safer blockchain programming language," in *2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C)*, pp. 97–99, IEEE, 2017.

[60] "Solidity." `https://solidity.readthedocs.io/en/develop/`.

[61] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," Mar. 21 2019. US Patent App. 16/119,750.

[62] R. R. Vokerla, B. Shanmugam, S. Azam, A. Karim, F. D. Boer, M. Jonkman, and F. Faisal, "An overview of blockchain applications and attacks," pp. 1–6, 2019.

[63] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pp. 442–446, IEEE, 2017.

[64] 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*, pp. 9–16, 2018.

[65] P. Tsankov, A. Dan, D. Drachsler-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*, pp. 67–82, 2018.

[66] R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen, "Visual emulation for ethereum's virtual machine," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–4, IEEE, 2018.

[67] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: reverse engineering ethereum's opaque smart contracts," in *27th USENIX security symposium (USENIX Security 18)*, pp. 1371–1385, 2018.

[68] I. Grishchenko, M. Maffei, and C. Schneidewind, "Ethertrust: Sound static analysis of ethereum bytecode," *Technische Universität Wien, Tech. Rep*, pp. 1–41, 2018.

[69] 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.

[70] 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.

[71] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *International symposium on automated technology for verification and analysis*, pp. 513–520, Springer, 2018.

[72] E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-D´iez, "Object-sensitive cost analysis for concurrent objects," *Software Testing, Verification and Reliability*, vol. 25, no. 3, pp. 218–271, 2015.

[73] J. Giesl, P. Schneider-Kamp, and R. Thiemann, "Aprove 1.2: Automatic termination proofs in the dependency pair framework," in *Automated Reasoning: Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 3*, pp. 281–286, Springer, 2006.

[74] C. Borralleras, D. Larraz, A. Oliveras, J. M. Rivero, E. Rodr´iguez-Carbonell, and A. Rubio, "Verymax: tool description for termcomp 2016," in *15th International Workshop on Termination*, p. 18, 2016.

[75] A. Flores-Montoya, "Cofloco: system description," in *15th International Workshop on Termination*, vol. 20, 2016.

[76] 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)*, pp. 8–15, IEEE, 2019.

[77] 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.

[78] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervasive and Mobile Computing*, vol. 67, p. 101227, 2020.

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

[80] K. L. Narayana and K. Sathiyamurthy, "Automation and smart materials in detecting smart contracts vulnerabilities in blockchain using deep learning," *Materials Today: Proceedings*, 2021.

[81] 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.

[82] S. Mohajerani, W. Ahrendt, and M. Fabian, "Modeling and security verification of state-based smart contracts," *IFAC-PapersOnLine*, vol. 55, no. 28, pp. 356–362, 2022.

[83] M. Ndiaye, T. A. Diallo, and K. Konate, "Adefguard: Anomaly detection framework based on ethereum smart contracts behaviours," *Blockchain: Research and Applications*, p. 100148, 2023.

[84] D. He, R. Wu, X. Li, S. Chan, and M. Guizani, "Detection of vulnerabilities of blockchain smart contracts," *IEEE Internet of Things Journal*, 2023.

[85] M. Pasqua, A. Benini, F. Contro, M. Crosara, M. Dalla Preda, and M. Ceccato, "Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode," *Journal of Systems and Software*, vol. 200, p. 111653, 2023.

[86] Z. Zhen, X. Zhao, J. Zhang, Y. Wang, and H. Chen, "Da-gnn: A smart contract vulnerability detection method based on dual attention graph neural network," *Computer Networks*, p. 110238, 2024.

[87] C. Sendner, L. Petzi, J. Stang, and A. Dmitrienko, "Large-scale study of vulnerability scanners for ethereum smart contracts," in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 220–220, IEEE Computer Society, 2024.

[88] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*. Springer Science Business Media, 2007.

[89] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Professional, 1995.

[90] B. Stroustrup, *The Design and Evolution of C++*. Addison-Wesley Professional, 1994.

[91] X. Hu, Y. Zhuang, S.-W. Lin, F. Zhang, S. Kan, and Z. Cao, "A security type verifier for smart contracts," *Computers & Security*, vol. 108, p. 102343, 2021.

[92] W. E. Boebert, "The system v application binary interface," *UNIX Review*, vol. 10, no. 11, pp. 6–18, 1992.

[93] U. S. Laboratories, "System v application binary interface - intel386 architecture processor supplement." `http://www.sco.com/developers/gabi/1995-07-27/contents.html`, 1995.

[94] H. Yin and D. X. Song, "Reverse engineering of binary application program interfaces," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pp. 187–196, 2006.

[95] E. Poll and M. Steffen, "A formal definition of the java bytecode language," *Science of Computer Programming*, vol. 47, no. 3, pp. 247–267, 2003.

[96] C. Click, M. Paleczny, and C. Verbrugge, "Static single assignment of java bytecode," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 222–234, 2002.

[97] 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.

[98] A. L. Vivar, A. L. S. Orozco, and L. J. G. Villalba, "A security framework for ethereum smart contracts," *Computer Communications*, vol. 172, pp. 119–129, 2021.

[99] J. Cocke and J. T. Schwartz, "Instruction selection using microcode-like global operations," *Communications of the ACM*, vol. 23, no. 3, pp. 162–167, 1980.

[100] B. Horn, J. Sakarovitch, and M. Soria, "Opcode patterns and regular expressions," *Theoretical Computer Science*, vol. 100, no. 2, pp. 273–292, 1992.

[101] S. Govindan and P. Ranganathan, "Improving instruction cache performance by opcode morphing," in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pp. 168–179, 2001.

[102] H. Huang, L. Guo, L. Zhao, H. Wang, C. Xu, and S. Jiang, "Effective combining source code and opcode for accurate vulnerability detection of smart contracts in edge ai systems," *Applied Soft Computing*, p. 111556, 2024.

[103] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS)*, pp. 6000–6010, 2017.

[104] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pp. 4171–4186, 2019.

[105] L. Zhang, W. Chen, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen, "Cbgru: A detection method of smart contract vulnerability based on a hybrid model," *Sensors*, vol. 22, no. 9, p. 3577, 2022.

[106] A. V. Gorchakov, L. A. Demidova, and P. N. Sovietov, "Analysis of program representations based on abstract syntax trees and higher-order markov chains for source code classification task," *Future Internet*, vol. 15, no. 9, p. 314, 2023.

[107] M. Gupta, J. Rees, A. Chaturvedi, and J. Chi, "Matching information security vulnerabilities to organizational security profiles: a genetic algorithm approach," *Decision Support Systems*, vol. 41, no. 3, pp. 592–603, 2006.

[108] Q. Gu, Z. Li, and J. Han, "Generalized fisher score for feature selection," *arXiv preprint arXiv:1202.3725*, 2012.

[109] L. Sun, T. Wang, W. Ding, J. Xu, and Y. Lin, "Feature selection using fisher score and multilabel neighborhood rough sets for multilabel classification," *Information Sciences*, vol. 578, pp. 887–912, 2021.

[110] D. Aksu, S. Üstebay, M. A. Aydin, and T. Atmaca, "Intrusion detection with comparative analysis of supervised learning techniques and fisher score feature selection algorithm," in *Computer and Information Sciences: 32nd International Symposium, ISCIS 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 20-21, 2018, Proceedings 32*, pp. 141–149, Springer, 2018.

[111] B. Singh, J. S. Sankhwar, and O. P. Vyas, "Optimization of feature selection method for high dimensional data using fisher score and minimum spanning tree," in *2014 annual IEEE India conference (INDICON)*, pp. 1–6, IEEE, 2014.

[112] B. Azhagusundari, A. S. Thanamani, *et al.*, "Feature selection based on information gain," *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, vol. 2, no. 2, pp. 18–21, 2013.

[113] S. Lei, "A feature selection method based on information gain and genetic algorithm," in *2012 international conference on computer science and electronics engineering*, vol. 2, pp. 355–358, IEEE, 2012.

[114] E. O. Omuya, G. O. Okeyo, and M. W. Kimwele, "Feature selection for classification using principal component analysis and information gain," *Expert Systems with Applications*, vol. 174, p. 114765, 2021.

[115] C. Shang, M. Li, S. Feng, Q. Jiang, and J. Fan, "Feature selection via maximizing global information gain for text classification," *Knowledge-Based Systems*, vol. 54, pp. 298–309, 2013.

[116] A. W. Haryanto, E. K. Mawardi, *et al.*, "Influence of word normalization and chi-squared feature selection on support vector machine (svm) text classification," in *2018 International seminar on application for technology of information and communication*, pp. 229–233, IEEE, 2018.

[117] S. Ray, K. Alshouiliy, A. Roy, A. AlGhamdi, and D. P. Agrawal, "Chi-squared based feature selection for stroke prediction using azureml," in *2020 Intermountain Engineering, Technology and Computing (IETC)*, pp. 1–6, IEEE, 2020.

[118] A.-M. Bidgoli and M. N. Parsa, "A hybrid feature selection by resampling, chi squared and consistency evaluation techniques," *World Academy of Science, Engineering and Technology*, vol. 68, pp. 276–285, 2012.

[119] I. S. Thaseen and C. A. Kumar, "Intrusion detection model using fusion of chi-square feature selection and multi class svm," *Journal of King Saud University-Computer and Information Sciences*, vol. 29, no. 4, pp. 462–472, 2017.

[120] E. Chitsaz, M. Taheri, S. D. Katebi, and M. Z. Jahromi, "An improved fuzzy feature clustering and selection based on chi-squared-test," in *Proceedings of the international multiconference of engineers and computer scientists*, vol. 1, pp. 18–20, 2009.

[121] R. Gupta, N. K. Gupta, and M. L. Soffa, "A case for profiling-oriented software engineering," *IEEE Software*, vol. 13, no. 1, pp. 22–31, 1996.

[122] B. J. Cox, "The execution time measurement and profiling of a program," *Communications of the ACM*, vol. 15, no. 10, pp. 801–805, 1972.

[123] B. Zadrozny and C. Elkan, "Profiling user sessions for fun and profit: Data, methods and models," in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 627–632, 2002.

[124] T. Hoffelder, "Equivalence analyses of dissolution profiles with the mahalanobis distance," *Biometrical Journal*, vol. 61, no. 5, pp. 1120–1137, 2019.

[125] F. Iorio, R. Tagliaferri, and D. d. Bernardo, "Identifying network of drug mode of action by gene expression profiling," *Journal of Computational Biology*, vol. 16, no. 2, pp. 241–251, 2009.

[126] D. N. Rassokhin and D. K. Agrafiotis, "Kolmogorov-smirnov statistic and its application in library design," *Journal of Molecular Graphics and Modelling*, vol. 18, no. 4-5, pp. 368–382, 2000.

[127] X. Zhang, F. Qiu, and F. Qin, "Identification and mapping of winter wheat by integrating temporal change information and kullback–leibler divergence," *International Journal of Applied Earth Observation and Geoinformation*, vol. 76, pp. 26–39, 2019.

[128] T. Karagiannis, K. Papagiannaki, N. Taft, and M. Faloutsos, "Profiling the end host," in *Passive and Active Network Measurement: 8th Internatinoal Conference, PAM 2007, Louvain-la-neuve, Belgium, April 5-6, 2007. Proceedings 8*, pp. 186–196, Springer, 2007.

[129] R. Labadie-Tamayo and D. Castro-Castro, "Graph-based profile condensation for users profiling," 2022.

[130] D. Xu, H. Wang, and K. Su, "Intelligent student profiling with fuzzy models," in *Proceedings of the 35th Annual Hawaii international conference on system sciences*, pp. 8–pp, IEEE, 2002.

[131] K. Han, J. Park, and M. Y. Yi, "Adaptive and multiple interest-aware user profiles for personalized search in folksonomy: A simple but effective graph-based profiling model," in *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, pp. 225–231, IEEE, 2015.

[132] W. Chen, Y. Gu, Z. Ren, X. He, H. Xie, T. Guo, D. Yin, and Y. Zhang, "Semi-supervised user profiling with heterogeneous graph attention networks.," in *IJCAI*, vol. 19, pp. 2116–2122, 2019.

[133] S. Xue, L. Zhang, A. Li, X.-Y. Li, C. Ruan, and W. Huang, "Appdna: App behavior profiling via graph-based deep learning," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 1475–1483, IEEE, 2018.

[134] H. Asai, K. Fukuda, P. Abry, P. Borgnat, and H. Esaki, "Network application profiling with traffic causality graphs," *International Journal of Network Management*, vol. 24, no. 4, pp. 289–303, 2014.

[135] S. Munir, S. I. Jami, and S. Wasi, "Knowledge graph based semantic modeling for profiling in industry 4.0," *International Journal on Information Technologies & Security*, vol. 12, no. 1, pp. 37–50, 2020.

[136] M. Daoud, L. Tamine, and M. Boughanem, "A personalized graph-based document ranking model using a semantic user profile," in *User Modeling, Adaptation, and Personalization: 18th International Conference, UMAP 2010, Big Island, HI, USA, June 20-24, 2010. Proceedings 18*, pp. 171–182, Springer, 2010.

[137] R. Zuech and A. Goodrum, "Fuzzy profiling for the detection of anomalous program behavior," *Computers & Security*, vol. 24, no. 2, pp. 123–139, 2005.

[138] C. Liu, G. Wang, and K. Peng, "Fuzzy profiling-based approach to anomaly detection in network traffic," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 2, pp. 202–211, 2007.

[139] S. Yu, H. Zhang, and T. Li, "Fuzzy profiling: A framework for discovering unknown malware," in *Proceedings of the 9th International Conference on Intelligent Systems Design and Applications (ISDA)*, pp. 838–843, 2009.

[140] O. Alhabashneh, R. Iqbal, F. Doctor, and S. Amin, "Adaptive information retrieval system based on fuzzy profiling," in *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pp. 1–8, IEEE, 2015.

[141] C. Mencar, M. A. Torsello, D. Dell'Agnello, G. Castellano, and C. Castiello, "Modeling user preferences through adaptive fuzzy profiles," in *2009 Ninth International Conference on Intelligent Systems Design and Applications*, pp. 1031–1036, IEEE, 2009.

[142] J. E. Dickerson and J. A. Dickerson, "Fuzzy network profiling for intrusion detection," in *PeachFuzz 2000. 19th International Conference of the North American Fuzzy Information Processing Society-NAFIPS (Cat. No. 00TH8500)*, pp. 301–306, IEEE, 2000.

[143] E. Mezura-Montes, J. Velázquez-Reyes, and C. A. Coello Coello, "Profiling-based adaptive genetic algorithm," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 5, pp. 1053–1070, 2009.

[144] G. Kendall, E. K. Burke, M. Gendreau, B. Ombuki-Berman, B. McCollum, E. Özcan, and R. Qu, "On the use of profiling techniques in genetic algorithm-based hyper-heuristics," *Journal of the Operational Research Society*, vol. 58, no. 6, pp. 708–718, 2007.

[145] S. Huang and B. Zhou, "Profiling genetic algorithms for improved performance on large scale problems," in *Proceedings of the 2009 IEEE Congress on Evolutionary Computation (CEC)*, pp. 3523–3530, 2009.

[146] P. Asokan, R. Saravanan, and K. Vijayakumar, "Machining parameters optimisation for turning cylindrical stock into a continuous finished profile using genetic algorithm (ga) and simulated annealing (sa)," *The International Journal of Advanced Manufacturing Technology*, vol. 21, no. 1, pp. 1–9, 2003.

[147] P. A. A. Resende and A. C. Drummond, "Adaptive anomaly-based intrusion detection system using genetic algorithm and profiling," *Security and Privacy*, vol. 1, no. 4, p. e36, 2018.

[148] A. H. Hamamoto, L. F. Carvalho, L. D. H. Sampaio, T. Abrão, and M. L. Proencca Jr, "Network anomaly detection system using genetic algorithm and fuzzy logic," *Expert Systems with Applications*, vol. 92, pp. 390–402, 2018.

[149] S. Lal, C. Cascaval, J. Mars, P. Dey, and H. Tang, "Profiling machine learning workloads," in *Proceedings of the 2019 International Symposium on Code Generation and Optimization (CGO)*, pp. 267–279, 2019.

[150] J. Wang, W. Huang, H. Zhang, and X. Wu, "Understanding the performance of tensorflow workloads on gpus," in *Proceedings of the 2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 83–92, 2020.

[151] A. Samajdar, V. Sridharan, M. Zinsmaier, Z. Pan, S. Shin, R. Gao, Q. Zhou, and R. K. Gupta, "Auto-profiling: A framework for profiling and optimization of ml workloads," in *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pp. 631–644, 2020.

[152] I. T. Haque and C. Assi, "Profiling-based indoor localization schemes," *IEEE Systems Journal*, vol. 9, no. 1, pp. 76–85, 2013.

[153] E. Tsalera, A. Papadakis, and M. Samarakou, "Monitoring, profiling and classification of urban environmental noise using sound characteristics and the knn algorithm," *Energy Reports*, vol. 6, pp. 223–230, 2020.

[154] P. Nagaraj, K. Saiteja, K. K. Ram, K. M. Kanta, S. K. Aditya, and V. Muneeswaran, "University recommender system based on student profile using feature weighted algorithm and knn," in *2022 International Conference on Sustainable Computing and Data Communication Systems (ICSCDS)*, pp. 479–484, IEEE, 2022.

[155] R. Bayot and T. Gonccalves, "Multilingual author profiling using word embedding averages and svms," in *2016 10th International Conference on Software, Knowledge, Information Management & Applications (SKIMA)*, pp. 382–386, IEEE, 2016.

[156] P. J. Batterham and H. Christensen, "Longitudinal risk profiling for suicidal thoughts and behaviours in a community cohort using decision trees," *Journal of affective disorders*, vol. 142, no. 1-3, pp. 306–314, 2012.

[157] P. Duchessi and E. J. Laur´ia, "Decision tree models for profiling ski resorts' promotional and advertising strategies and the impact on sales," *Expert Systems with Applications*, vol. 40, no. 15, pp. 5822–5829, 2013.

[158] S. H. Hawley, B. Colburn, and S. I. Mimilakis, "Signaltrain: Profiling audio compressors with deep neural networks," *arXiv preprint arXiv:1905.11928*, 2019.

[159] G. X. Yu, T. Grossman, and G. Pekhimenko, "Skyline: Interactive in-editor computational performance profiling for deep neural network training," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pp. 126–139, 2020.

[160] A. Li, S. Xue, X.-Y. Li, L. Zhang, and J. Qian, "Appdna: Profiling app behavior via deep-learning function call graphs," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 1, pp. 414–427, 2020.

[161] A. Cura, H. Küccük, E. Ergen, and İ. B. Öksüzouglu, "Driver profiling using long short term memory (lstm) and convolutional neural network (cnn) methods," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 10, pp. 6572–6582, 2020.

[162] K. C. Baumgartner, S. Ferrari, and C. G. Salfati, "Bayesian network modeling of offender behavior for criminal profiling," in *Proceedings of the 44th IEEE Conference on Decision and Control*, pp. 2702–2709, IEEE, 2005.

[163] T. Xiang and S. Gong, "Video behavior profiling for anomaly detection," *IEEE transactions on pattern analysis and machine intelligence*, vol. 30, no. 5, pp. 893–908, 2008.

[164] Y.-J. Zheng, W.-G. Sheng, X.-M. Sun, and S.-Y. Chen, "Airline passenger profiling based on fuzzy deep machine learning," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 12, pp. 2911–2923, 2016.

[165] M. Sedaghati, C. Jutten, and A. Abdi, "Profiling deep neural networks: Sparsity and complexity analysis," *IEEE Signal Processing Letters*, vol. 27, pp. 220–224, 2020.

[166] B. W.-G. M. Anrig, Bernhard, "The role of algorithms in profiling," 2008.

[167] S. J. Aquilina, F. Casino, M. Vella, J. Ellul, and C. Patsakis, "Etherclue: Digital investigation of attacks on ethereum smart contracts," *Blockchain: Research and Applications*, vol. 2, no. 4, p. 100028, 2021.

[168] N. Sharma and S. Sharma, "A survey of mythril, a smart contract security analysis tool for EVM bytecode," *Indian J Natural Sci*, vol. 13, p. 75, 2022.

[169] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Proceedings of international conference on neural networks (ICNN'96)*, vol. 1, pp. 347–352, IEEE, 1996.

[170] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, and S. Li, "Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[171] 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*, pp. 3283–3290, 2021.

Vita

Vita

Candidate's full name: Sepideh HajiHosseinKhani

University attended (with dates and degrees obtained):
Bachelor of Computer Engineering AlZahra University 2017 - 2021

Publications:
Sepideh HajiHosseinKhani, Arash Habibi Lashkari, Ali Mizani Oskui, **"Unveiling vulnerable smart contracts: Toward profiling vulnerable smart contracts using genetic algorithm and generating benchmark dataset"**, published in Blockchain: Research and Applications (ScienceDirect).

Sepideh HajiHosseinKhani, Arash Habibi Lashkari, Ali Mizani Oskui, **"Unveiling Smart Contracts Vulnerabilities: Toward Profiling Smart Contracts Vulnerabilities using EGA and Generating Benchmark Dataset"**, submitted in Blockchain: Research and Applications (ScienceDirect).

Sepideh HajiHosseinKhani, Arash Habibi Lashkari, Ali Mizani Oskui, **"Security of Smart Contracts: A Systematic Mapping Study on Vulnerabilities, Causes, and Attacks"**, submitted in ACM Computing Surveys.

Sepideh HajiHosseinKhani, Bhaskar Joshi, Arash Habibi Lashkari, **"AuthAttLyzer-V2: Unveiling Code Authorship Attribution using Enhanced Ensemble Learning Models and Generating Benchmark Dataset"**, accepted in 20th Int. Conference on Data Science (ICDATA'24).

Maëlle Gautrin, Arash Habibi Lashkari, Sepideh HajiHosseinKhani, **"A Novel Deep Learning-Based Vulnerability Detection in Smart Contracts Using Masked Attention and Control Flow Graph Analysis"**, submitted in International Journal of Information Security (Springer).