Journal Pre-proof

Unveiling Smart Contracts Vulnerabilities: Toward Profiling Smart Contracts Vulnerabilities using Enhanced Genetic Algorithm and Generating Benchmark Dataset

Sepideh HajiHosseinKhani, Arash Habibi Lashkari and Ali Mizani Oskui

PII: S2096-7209(24)00066-6
DOI: https://doi.org/10.1016/j.bcra.2024.100253
Reference: BCRA 100253

To appear in: *Blockchain: Research and Applications*

Received date: 14 June 2024
Revised date: 21 October 2024
Accepted date: 24 November 2024

Please cite this article as: S. HajiHosseinKhani, A. Habibi Lashkari and A. Mizani Oskui, Unveiling Smart Contracts Vulnerabilities: Toward Profiling Smart Contracts Vulnerabilities using Enhanced Genetic Algorithm and Generating Benchmark Dataset, *Blockchain: Research and Applications*, 100253, doi: https://doi.org/10.1016/j.bcra.2024.100253.

Unveiling Smart Contracts Vulnerabilities: Toward Profiling Smart Contracts Vulnerabilities using Enhanced Genetic Algorithm and Generating Benchmark Dataset

Sepideh HajiHosseinKhani[i,a], Arash Habibi Lashkari[a,b] and Ali Mizani Oskui[c,i]

[a]Computer Science, Department of Computer Science and Engineering, York University, Toronto, Ontario, Canada
[b]Behavior-Centric Cybersecurity Center (BCCC), School of Information Technology, York University, Toronto, Ontario, Canada
[c]Financial and Crypto Advisory of Switzerland (FiCAS AG), Zug, Switzerland

ARTICLE INFO

Keywords:
Smart Contracts (SCs), Vulnerability, Vulnerable Smart Contracts, Vulnerability Profiling, Genetic Algorithm.

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: an updated analyzer named BCCC-SCsVulLyzer (V2.0) and an advanced Genetic Algorithm (GA) profiling method. The analyzer 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.

1. Introduction

Smart Contracts (SCs), as autonomous and self-executing programs, are transforming the management and security of digital transactions and agreements, making them a captivating area of interest in blockchain technology [1, 2]. Despite their advantages, such as immutability and automated execution, SCs are susceptible to various security threats and potential attacks due to coding errors, arithmetic issues, and other unique characteristics [3]. Over the past decade, researchers have developed various tools for detecting SC vulnerabilities using different techniques, including static, dynamic, and hybrid approaches.

The first group of researchers focused on static analysis, proposing various methodologies and frameworks to detect vulnerabilities in SCs, enhance security, and improve transparency [4, 5, 6, 7]. These methods analyze the source code or its transformed versions before compilation. Ndiaye *et al.* [8] introduced ADEFGuard, which uses SC behavior derived from source code as a key feature for detecting malicious contracts. In addition, Ren *et al.* [9] proposed Blass, an approach that combines the semantic structure of the source code with a self-designed neural network to identify vulnerabilities. Leveraging the benefits of other programming languages, Tikhomirov *et al.* [10] developed SmartCheck, which transforms Solidity code into an XML-based intermediate format and uses XPath

[i]Sepideh Hajihosseinkhani
ORCID(s):

patterns to detect vulnerabilities. Furthermore, Liu *et al.* [11] proposed a new framework that uses a graph-based source code, combined with expert-defined patterns within a deep learning framework, to identify vulnerabilities.

The next group of researchers proposed dynamic approaches, utilizing the compiled version of the source code as the foundation to detect vulnerabilities in SCs [12, 13, 14, 15]. Most of these techniques build upon the bytecode and opcode of the SCs [16, 17, 18, 19]. For instance, Chen *et al.* [20] introduced DefectChecker, which processes opcodes into basic blocks and symbolically executes the instructions within each block. Moreover, Zhou *et al.* [21] developed a security assurance technique that employs syntax rules in the context of SCs to help developers understand the structure of SC invocation relationships. Additionally, Feist *et al.* [22] created Slither, which converts Solidity SCs into an intermediate representation called SlithIR, enhancing the understanding of SCs and aiding in vulnerability detection.

Researchers have combined dynamic and static approaches for the final category of studies to create more comprehensive tools [23, 24, 25]. Specifically, Hajihosseinkhani *et al.* [25] developed an advanced genetic profiling algorithm that integrates features from various categories such as source code, AST and ABI to achieve greater precision and introduces two profiles that characterize secure and vulnerable SCs. However, detecting smart contract vulnerabilities presents numerous challenges, as past research highlights. These challenges include coverage issues and unexplored vulnerabilities [21, 13, 17], accuracy challenges [12], scaling constraints [26], and data diversity limitations [27].

This study introduces a novel profiling approach that builds upon our previous work [25], addressing the issues of unexplored vulnerabilities, low accuracy, scaling constraints, and diversity limitations in detecting vulnerable SCs. Our proposed Enhanced Genetic Algorithm(EGA) profiling method stands out by drawing inspiration from natural selection and genetics, enabling developers not only to simulate program behavior but also to quickly identify performance bottlenecks and optimize code [28, 29, 30]. This approach provides precise data on execution time at every function, method, and line of code level while being cross-platform compatible and user-friendly. Additionally, the feature selection method identifies relevant data attributes within SCs, discarding superfluous or irrelevant ones, thereby enhancing the genetic algorithm's efficiency and improving vulnerability detection accuracy. This integrated approach promises to significantly improve the speed and precision of vulnerability detection in SCs, strengthening the security of SCs by profiling vulnerabilities.

Furthermore, We created a new dataset, BCCC-SCsVul-2024, for our experiments to evaluate the innovative profiling approach. This dataset merges Solidity source codes from various sources, including the Smart Bugs dataset, the Ethereum Smart Contracts (ESCs) dataset, the Slither-audited smart contracts dataset, and the SmartScan dataset. Our dataset includes a substantial sample size of 111,897 instances, covering a broad range of 11 potential vulnerabilities such as External Bugs, Gas Exceptions, Mishandled Exceptions, Timestamps, and Transaction Order Dependence. Through the experiment and evaluation process, our findings highlight the effectiveness of the proposed profiling technique, with reported metrics such as accuracy, precision, recall, F1-score, and time and space complexity.

Our contributions in this study are multi-faceted, providing significant advancements in the realm of vulnerability detection and profiling in SCs:

- To design and implement an SCs analyzer, BCCC-SCsVulLyzer(V2.0.0), for experiments demonstrating effective feature extraction and optimization techniques, improving SC analysis performance.
- To design an enhanced GA tailored for SCs vulnerabilities profiling, using three new components: penalty fitness function, retention of elites, and adaptive mutation rate.
- To create a new SC vulnerability dataset, BCCC-SCsVul-2024, by combining Smart Bugs, Ethereum SCs (ESCs), slither-audited smart contracts, and the SmartScan dataset, with the primary goal of testing and evaluating the proposed model on a substantial dataset comprising 111,897 samples.

The structure of the remainder of this paper is as follows. Section 2 delves into the existing literature within this research field and identifies the limitations inherent in current solutions. Section **??** explores the background of feature extraction and provides a detailed account of profiling techniques. Section 3 is devoted to an in-depth discussion of our proposed model. Section 4 outlines the extensive experiment and results. Sections 5 and 6 thoroughly analyze and discuss the research questions, proposing the common profiles. Finally, Section 7 presents our conclusions and suggests directions for future work.

2. Literature Review

This section offers a thorough review of research on SCs vulnerability detection and identification (profiling) techniques from 2016 to 2024. Our review has three primary goals: to classify earlier models as either learning-based or non-learning-based, to create a chronological taxonomy of the tools developed over these years, and to pinpoint the limitations of prior research. The first subsection examines earlier studies, the second subsection delves into the background of feature extraction, and the final subsection synthesizes our findings, emphasizing the gaps and shortcomings identified in the existing literature.

2.1. Available Profiling Models

This subsection categorizes available profiling techniques into two main categories based on their main fundamental methodologies: learning-based and non-learning-based approaches. Learning-based methods employ two main techniques: deep learning and machine learning. On the other hand, non-learning-based methods utilize rule-based, fuzzy, Genetic Algorithm (GA), symbolic execution, and Finite State Machine (FSM) techniques.

2.1.1. Non Learning-based

Non-learning approaches to SC vulnerability detection encompass several methodologies, each tailored to specific security aspects. These methods include rule-based, fuzzy logic, genetic algorithms (GAs), symbolic execution, and finite state machine (FSM) techniques.

- **Rule-based**: Rule-based techniques involve methods that use specified patterns and rules to detect vulnerabilities in SCs [27, 15, 22, 26]. For instance, Tikhomirov *et al.* [10] developed SmartCheck, which transforms Solidity code into an XML-based intermediate format and uses XPath patterns to identify vulnerabilities. Additionally, Grishchenko *et al.* [12] introduced EtherTrust, a tool for the automated analysis of EVM bytecode that provided a scalable solution to the security challenges on the Ethereum platform. Brent *et al.* [27] developed Vandal, a dynamic security analysis framework that uniquely utilizes EVM bytecode and extracts semantic logic relations to detect anomalies and vulnerabilities.

  Furthermore, Wang *et al.* [17] proposed NPChecker, which goes beyond traditional patterns by modeling contract bytecode components and exposing nondeterministic factors through information flow tracking. Rodler *et al.* [18] developed Secure Ethereum (Sereum), which utilizes a taint engine and an attack detector through dynamic taint-tracking specifically for SCs. Grech *et al.* [15] introduced MadMax, employing logic-based specifications to systematically uncover the complex characteristics of SCs. Likewise, Luu *et al.* [31] proposed Oyente, which analyzes the CFG, enabling precise alignment of instructions with their constraints and supporting an in-depth analysis. Moreover, Albert *et al.* [19] introduced ETHIR, a rule-based tool for SC bytecode analysis, enabling high-level analyses. Additionally, Feist *et al.* [22] introduced Slither, which employs a Static Single Assignment (SSA) form and a simplified instruction set. Zhou *et al.* [21] developed SASC, which uses syntax rules in the context of SCs to help developers grasp the structure of SC invocation relationships.

  In addition, Kalra *et al.* [26] presented ZEUS, which employs techniques like abstract interpretation, symbolic model checking, and constrained horn clauses on AST to ensure comprehensive analysis. Norvill *et al.* [32] introduced E-EVM, enabling users to examine the CFG of the code in each block, trace control flow paths, identify loops, and suggest optimizations, functioning as both an analytical tool and a learning aid. Moreover, Zhou *et al.* [33] introduced Erays, which examines contract complexity through cyclomatic complexity, a metric for measuring the number of linearly independent paths in a CFG. Praitheeshan *et al.* [34] developed SolGuard, utilizing the CFG to identify potential vulnerabilities during the contract's compilation phase. Krupp *et al.* [35] developed teEther, analyzing the CFG to identify critical and state-changing instructions. Lastly, Tsankov *et al.* [36] introduced Securify, which utilizes Control Dependency graphs to assess patterns of compliance and violation, defined in their specialized language.

- **Fuzzy**: Fuzzy techniques leverage the principles of fuzzy logic to address uncertainty and imprecision in data, enabling more flexible decision-making processes [3, 37]. Liu *et al.* [3] created ReGuard, a tool based on fuzzing techniques to detect re-entrancy bugs in Ethereum SCs, utilizing the C++ version of the source code. Ji *et al.* [38] proposed Effuzz, which analyzes the dependencies of branch constraints and identifies a subset of the input parameters that effectively satisfy the branch constraints. Additionally, Nguyen *et al.* [39] introduced sFuzz, examining the logs from test cases to identify potential vulnerabilities. Finally, Jiang *et al.* [37] presented

ContractFuzzer, which employs fuzzing to uncover security vulnerabilities in the ABI version of Ethereum SCs. It creates fuzzing inputs based on the SCs' ABI specifications and sets up test oracles to pinpoint vulnerabilities.

- **Genetic Algorithm**: GA techniques, inspired by natural selection and genetics principles, solve optimization and search problems through evolving solutions in each iteration. Utilizing the natural characteristics of GAs, Hajihosseinkhani *et al.* [**?**] proposed an advanced GA-based method for analyzing SCs. The primary objective of their study was the effective profiling of vulnerable SCs. Furthermore, they developed an analyzer named SCsVulLyzer, designed explicitly for profiling SCs using GA techniques.

- **Symbolic Execution**: Symbolic execution techniques analyze SCs with symbolic inputs instead of concrete data values, allowing them to explore multiple execution paths and identify potential vulnerabilities. For instance, Mossberg *et al.* [5] introduced Manticore, which maximizes code coverage testing by exploring a program's state space using constraint solving. Additionally, Ndiaye *et al.* [8] proposed ADEFGuard, which analyzes SC vulnerabilities by comparing the system's current state to a baseline to identify any deviations flagged as potential vulnerabilities. Chen *et al.* [16] developed GASPER, which explores all possible paths in the contract's code to detect inefficient coding by assessing gas efficiency through bytecode analysis of SCs. Moreover, Nikolic *et al.* [13] developed MAIAN, which analyzes a contract's bytecode within the context of the current blockchain state. By employing interprocedural symbolic analysis, MAIAN scrutinizes transaction sequences and their interaction patterns with contracts over multiple blockchain blocks.

  Furthermore, Torres *et al.* [40] proposed HONEYBADGER, a tool that detects honeypot issues in SCs by executing the code and analyzing EVM bytecode as input, producing a detailed report on the various honeypot issues it identifies. Wu *et al.* [41] developed TaintGuard, which monitors code within the program to track differences in the AST during execution, safeguarding against any unauthorized changes to contract privileges by external entities. Pasqua *et al.* [42] introduced EtherSolve, which extracts the precise CFG of the source code and identifies vulnerability paths during the symbolic execution process. Driessen *et al.* [43] presented SolAR, which automates the generation of test suites for SCs. SolAR ensures comprehensive coverage of all logical paths to effectively identify vulnerabilities by designing a test suite that traverses each edge at least once. Likewise, Chen *et al.* [20] proposed DefectChecker, which detects issues by symbolically executing the instructions within each block. Torres *et al.* [23] developed Osiris, structured around three primary components: symbolic analysis, taint analysis, and integer error detection, offering comprehensive and reliable results across all analyzed cases.

- **Finite State Machine**: A finite state machine (FSM) is a computational model used to analyze issues in SCs by consisting of a set of states and defining transitions between these states based on input conditions and actions [7, 44]. This approach focuses on converting source code into FSMs. Mohajerani *et al.* [6] developed extended finite state machines (EFSMs) to identify security vulnerabilities in SCs, specifically focusing on a casino SC scenario. Almakhour *et al.* [7] introduced a formal verification approach for composite SC security using FSM (FVACSC-FSM). It employs FSM models and model-checking methods for modeling and verifying SCs. Finally, Sharma *et al.* [44] introduced Mythril, a tool designed to assess vulnerabilities in Ethereum SCs. Mythril operates by decompiling the bytecode of an SC back into EVM opcode instructions and exploring all possible program states over a series of transactions with FSM.

2.1.2. Learning-based

In the first category of our analysis of previous works, we examine learning-based approaches divided into two main sub-categories: machine learning and deep learning techniques. The machine learning sub-category encompasses basic machine learning methods. Under deep learning techniques, we explore basic architectures and advanced architectures.

- **Machine Learning**: Machine learning techniques have significantly enhanced the security of SCs due to their improved accuracy.

  -Basic Architectures: The first group of machine learning methods utilizes basic algorithms such as XGBoost and clustering. Wang *et al.* developed ContractWard, which enhances the efficiency of vulnerability detection by extracting Bi-gram features from simplified source codes. It maintained an impressively brief average detection

time of just 4 seconds per contract. However, Ye *et al.* developed Vulpedia, which involves clustering contracts based on similarity by using tree edit distances calculated from their ASTs.

-Advanced Architectures: The second group of machine learning approaches focuses on advanced architectures, including the combination of Active Learning Queries and Semi-supervised Learning. Sun *et al.* introduced ASSBert to address the challenges of classifying SCs vulnerabilities when labeled source codes are scarce. In ASSBert, the active learning component involves selecting a subset of unlabeled samples for manual annotation, specifically targeting those with the highest uncertainty. Additionally, semi-supervised learning identifies samples most likely to belong to a specific category, assigning them predictive pseudo-tags based on the lowest uncertainty.

- **Deep Learning**: Deep learning solutions for SCs vulnerability detection can be categorized into two groups: basic and advanced architectures. Basic architectures encompass methods such as Graph Neural Networks (GNNs), Deep Neural Networks, and Long Short-Term Memory (LSTM) networks. On the other hand, advanced architectures involve more complex combinations and variations, including adversarial multi-task learning, the integration of Convolutional Neural Networks (CNNs) with attention mechanisms, and multi-layer transformer encoders.

  -Basic Architectures: The first category of basic architectures in deep learning techniques focuses on graph neural networks (GNNs). Liu *et al.* [11] proposed a new SC Vulnerability Detection Mechanism Based on Deep Learning and Expert Rules (SCVDM-DLER). First, it transforms the source code into a graph format. Next, in a subsequent phase, it can block risky transactions directly at the EVM level while producing error reports. Li *et al.* [45] introduced ExpenGas, a graph neural network method utilizing evolutionary computation-based machine learning, to detect expensive operation patterns in Ethereum SCs. It takes an AST as input and employs the neural transformer architecture, incorporating linear layers to produce vulnerability detection results. Moreover, Chen *et al.* [46] proposed EA-RGCN which captures the SCs code's local code content and global semantic features by constructing an AST and a novel Semantic Graph (SG) for each function and learning the SGs using Graph Convolutional Networks (GCNs) with residual blocks and edge attention.

  The second category of basic architectures in deep learning techniques focuses on deep neural networks. Ali *et al.* [14] proposed SRP, which minimizes the on-chain burden of runtime verification by integrating an off-chain mechanism with on-chain contract execution. It utilizes a deep neural network on the bytecode version of the source code. Narayana *et al.* [47] proposed ASMD-SCVB-DL, which allows for structured navigation of the AST to effectively extract features. It uses the extracted features to train a deep learning model for vulnerability detection in SCs. Similarly, Tian *et al.* [48] introduced TrVD, which utilizes tree decomposition to extract semantic features from code fragments. This method employs a deep learning-based model, serving as the core framework for analyzing the vulnerabilities in SCs. Xie *et al.* [49] introduced Block-gram, which extracts eight-dimensional opcode block features from the sequence of opcodes. Based on these pre-defined eight-dimensional attribute features, they detected vulnerabilities using a deep learning model. Finally, Ashizawa *et al.* [50] presented Eth2Vec, a learning-based tool for detecting vulnerabilities in Ethereum SCs. It uses a neural network designed for natural language processing to learn the features of vulnerable EVM bytecodes, opcodes, and AST.

  The final category of basic architectures in deep learning techniques focuses on Long Short-Term Memory (LSTM). Wang *et al.* [51] proposed TTPS, a Long Short-Term Memory based method for detecting Ponzi schemes, considering the time series transaction information of SCs. TTPS analyzed both transactional features and bytecode features of SCs. They employed Adaptive Synthetic Sampling (ADASYN) to effectively enhance the feature data of minority-class Ponzi scheme samples.

  -Advanced Architectures: Researchers focusing on advanced architectures within the deep learning domain often integrate multiple techniques. To begin with, Zhou *et al.* [52] introduced SCVDM-AMTL, an adversarial multi-task learning approach that analyzes Solidity source code to detect vulnerabilities in smart contracts (SCs). Moreover, Yuan *et al.* [24] proposed OSCV-MCEE, which utilizes transfer learning techniques. This approach consists of a multi-layer transformer encoder, augmented by the Bug Injection framework and a Softmax classifier to identify new SC vulnerabilities. Liu *et al.* [53] focused on addressing the problem of locating vulnerable functions within SCs by constructing a Multi-Relational Nested Contract Graph (MRNG),

which effectively captures both the syntactic and semantic information in SC code. Cai *et al.* [54] proposed CSJG-GNN-SCVD, which employs a bidirectional gated GNN model to identify vulnerable patterns.

Furthermore, Ren *et al.* [9] proposed Blass based on a semantic code structure and a self-designed neural network. They trained the Bi-LSTM-Att model on semantic structure for vulnerability detection. Zhang *et al.* [55] developed SVScanner, which processes structural sequences from ASTs to extract global and structural-semantic insights using a sequence model paired with an attention mechanism. Lastly, Zhang *et al.* [56] introduced the Multiple-Objective Detection Neural Network (MODNN). They employed a Convolutional Neural Network (CNN) enhanced with an attention mechanism and a backpropagation process. This setup enables it to identify various types of vulnerabilities.

In conclusion, the analysis of available profiling approaches reveals a predominant focus on detection rather than comprehensive profiling that includes both detection and identification. The landscape of non-learning-based methods for SC vulnerability detection incorporates various techniques such as rule-based, fuzzy logic, GAs, symbolic execution, and FSMs. However, these methods often lack precision, fail to address a broad spectrum of vulnerabilities, and can be time-consuming. Moreover, exploring learning-based techniques in SC security has significantly advanced vulnerability detection. These methods range from basic machine learning models to sophisticated deep learning architectures, enhancing security assessments' accuracy and scope. Nonetheless, these methods' complexity and extensive processing time, combined with the opaque nature of deep learning models, pose challenges regarding transparency and manageability.

2.2. Profiling Feature Extraction

In the context of profiling in SCs, feature extraction is crucial for analyzing characteristics (identification) [57, 58]. Effective profiling necessitates selecting the optimal set of features specifically tailored to identify the common behavior of an object [59, 60]. This subsection explores features derived from the AST, ABI specification, bytecode, opcodes, and source code of SCs. It reviews existing research on utilizing these features for profiling vulnerabilities in SCs. Table 1 provides a summary of the features that have been extracted for our analysis.

2.2.1. AST

The AST is an essential data structure in computer science that captures the syntactic structure of source code in a formal language [57]. Each construct in the source code corresponds to a node within the tree structure of the AST, with edges linking the nodes, representing the relationships among the components [58]. ASTs have several applications, ranging from compilers and interpreters to static code analysis tools [61].

Hu *et al.* [62] proposed a novel application of the AST. They developed a code generator that primarily uses the AST of an SC to assign security classes based on predefined rules. This unique application of the AST allows the code generator to recognize security vulnerabilities and allocate suitable security types to different sections of the code, improving the security of SCs. Moreover, Gorchakov *et al.* [63] proposed a comparative review of recent approaches to source code translation into vector-based representation. They used first- and second-order Markov chains based on ASTs and histograms of AST node types.

2.2.2. ABI specifications

The Ethereum ecosystem relies heavily on the contract ABI, which serves as a common interface for contracts to connect with entities outside the blockchain and with each other [64]. In the Ethereum environment, the ABI ensures that data is encoded according to its type, following the specifications encoded in the ABI [65]. This method is crucial for ensuring interoperability among diverse SCs and applications within the ecosystem [66].

Samreen *et al.* [67] proposed a novel approach for testing SCs that involves generating the ABI and bytecode of the contract under scrutiny, post-compilation. This process aims to find vulnerabilities in SCs, focusing on detecting Re-entrancy attacks. Furthermore, Sun *et al.* [68] offered an automatic method that applies a common text classification method based on ABI granularity. Their tests used fundamental machine learning techniques (SVM, XGBoost, Stacking, and Bagging) to classify SCs using TF-IDF vectors. The findings showed that this strategy is effective, and their proposed solution outperformed the source code-based categorization method by more than 0.1 on average.

2.2.3. Bytecode

bytecode is a distinctive form of computer object code that can be interpreted and translated into binary machine code by a virtual machine (VM), making it executable by a computer's hardware processor [69]. This unique

Table 1
Summary of features

Ref.	Feature Category	Description
[57, 58, 61, 62, 63]	AST	Generating security type assignments based on proposed rules for SCs and providing a tree representation of the abstract syntactic structure of source code.
[64, 65, 66, 67, 68]	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.
[69, 70, 71, 72, 73]	bytecode	Form of computer object code that an interpreter converts into binary machine code.
[71, 74, 75, 76, 77]	opcode	Represents a specific operation in the environment, with 256 possible opcodes.
[78, 79, 80, 2]	Source code	Involving the conversion of raw text into numerical representations that machine learning algorithms can more easily process.

characteristic facilitates superior code portability across various platforms since the VM can interpret and adapt the bytecode for the specific target platform [70].

In recent academic research, studies by Gupta *et al.* [71] and Vivar *et al.* [72] effectively utilized bytecode within the spheres of deep learning and security analysis of systems. These works developed novel system analysis and security approaches, taking advantage of bytecode's interpretability and portable characteristics. Moreover, Sendner *et al.* [73] proposed ESCORT, a deep learning-based vulnerability detection technique. ESCORT learns the generic bytecode semantics of SCs using a common feature extractor and identifies the features of each vulnerability class with distinct branches.

2.2.4. Opcode

Within the field of computers, an opcode holds a unique position within a machine language instruction, indicating the operation to be executed [74]. Each opcode signifies a distinct operation within the computing environment, typically with a size of 1-byte [75]. This structure enables 256 different opcodes, each representing a unique operation within the computational paradigm [76].

Recently, Gupta *et al.* [71] presented a malware detection methodology, highlighting the opcodes' significant role. This approach positions opcodes as the primary feature in their deep learning algorithm. Likewise, Huang *et al.* [77] 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.

2.2.5. Source code

Source code analysis is the process of examining and evaluating the source code of a program to identify potential vulnerabilities or improvements for enhancing overall quality and security [81]. Additionally, Natural Language Processing (NLP), which intersects computer science and human languages, can be leveraged for sophisticated analyses such as detecting vulnerabilities [78]. In this context, techniques like Word2Vec and FastText are essential as they provide vector representations of words, also known as word embeddings [79].

Qian *et al.* [80] and Zhang *et al.* [2] utilized Word2Vec, a neural network model, to learn these embeddings. In contrast, FastText is an open-source package that enables users to acquire text representations and classifiers, offering a productive method for examining the textual elements of SCs [2]. By merging Word2Vec and FastText, Zhang *et al.* [2] created a novel hybrid model that improves detecting vulnerabilities in SCs by demonstrating high accuracy and efficiency. It demonstrates the effectiveness of using NLP techniques to accurately interpret the semantics of SCs codes and the relationships between different parts of the SCs, which is essential for detecting vulnerabilities.

In conclusion, previous studies [57, 63, 64, 71, 79, 2] have highlighted the importance of feature categories in providing valuable insights into the complex structures and behaviors of SCs. For thorough analysis and precise vulnerability detection in SCs, it is crucial to extract and analyze features from these categories [72, 82, 63]. Each

feature type within these categories offers unique and essential insights that significantly enhance the vulnerability detection process [80, 73].

2.3. Synthesis

In synthesizing the available literature on SC vulnerability detection, we propose two taxonomies in this section. The first taxonomy, shown in Fig. 1, categorizes available techniques based on the fundamental models utilized to detect vulnerabilities in SCs. It further includes the key features that these models work with to detect vulnerabilities, illustrating how each model approaches the detection process. The second taxonomy, shown in Fig. 2, provides a chronological overview, tracking the fluctuations in the number of works in this area since 2016. This analysis highlights two notable peaks in publications in this field: one in 2018 and another ongoing from 2023 to 2024.

While profiling and identification are often paired with detection, the first taxonomy reveals that most current models primarily focus on the detection aspect. Most available tools utilize non-learning techniques due to their simplicity and transparency, such as rule-based approaches. However, the lack of accuracy in these detection methods has spurred the development of learning-based solutions, including simple machine learning models and more complex neural network models. Despite these advancements, there are several shortcomings in the current research on SC security and vulnerability detection:

Lack of Complete Profiling Coverage: Research often focuses on profiling specific vulnerability types in SCs, potentially overlooking the full range of vulnerabilities. This limited scope might leave certain threats unaddressed.

Lack of Comprehensive Profiling Features: Different research approaches adopt varying feature categories for profiling SCs. However, a unified, comprehensive set of profiling features is still lacking. Additionally, some categories, such as NLP techniques, should be further customized to align with the unique nature of the Solidity programming language.

False Positives and Negatives: 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 expands.

Limited Dataset Diversity: The success of specific 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.

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.

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.

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.

To address these shortcomings, ongoing research and development efforts are necessary to enhance coverage, reduce false positives and negatives, improve scalability, increase dataset diversity, and lessen dependency on accessible source code. Continuously evaluating and refining existing approaches while exploring new methodologies are crucial for improving the security of SCs, as discussed in the next section. This study proposes a new profiling technique to effectively identify different types of vulnerabilities in SCs. This method represents the second version of the enhanced genetic profiling algorithm initially proposed in our previous study [25].

3. Proposed Model

This section presents a comprehensive methodology for profiling SC vulnerabilities into seven stages. We begin with 'Data Pre-Processing' (Section 3.1), which lays the foundation for our framework by pre-processing the SCs and assigning corresponding hash IDs to each. Next, we proceed to the 'Best Compatible Compiler' (Section 3.2), where we systematically compile SCs by identifying the most suitable compiler version for each contract.

The analysis progresses to stage 'Feature Engineering' (Section 3.3), where we first employ methods to identify and extract critical features from the SCs. This stage helps uncover fundamental elements essential for analyzing

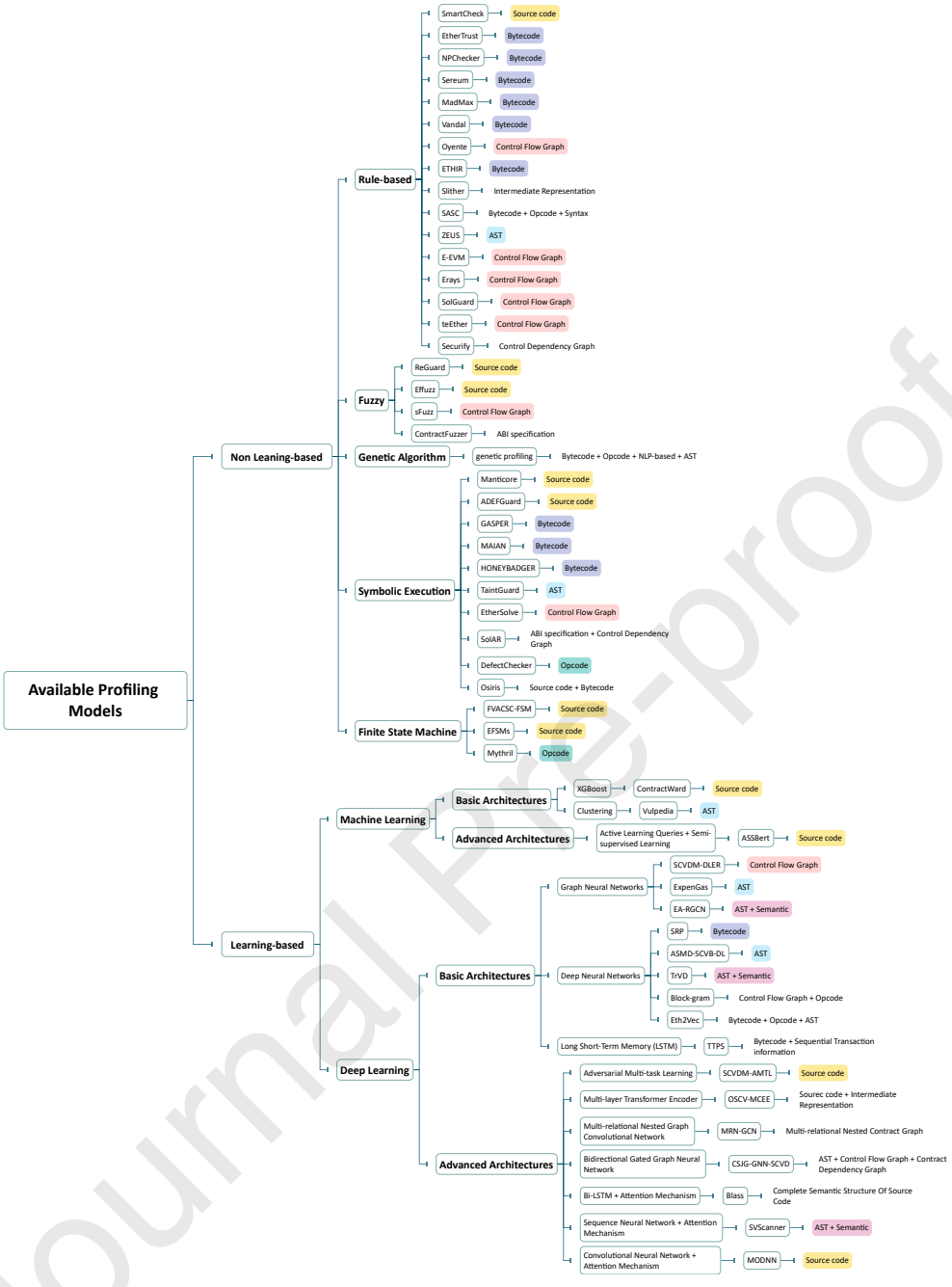


Figure 1: Taxonomy of Available Techniques

the characteristics of SCs in profiling vulnerabilities. Second, we design an ensemble feature selection model to strategically filter our features, focusing solely on those with significant impact.

The next stage, 'Proposed Profiling Model' (Section 3.4.2), involves designing and implementing an enhanced version of GA to profile vulnerabilities. These enhancements include three main components: Penalty Fitness Function, Retention of Elites, and Adaptive Mutation Rate. These build on the previous improvements we proposed in our earlier study [25].

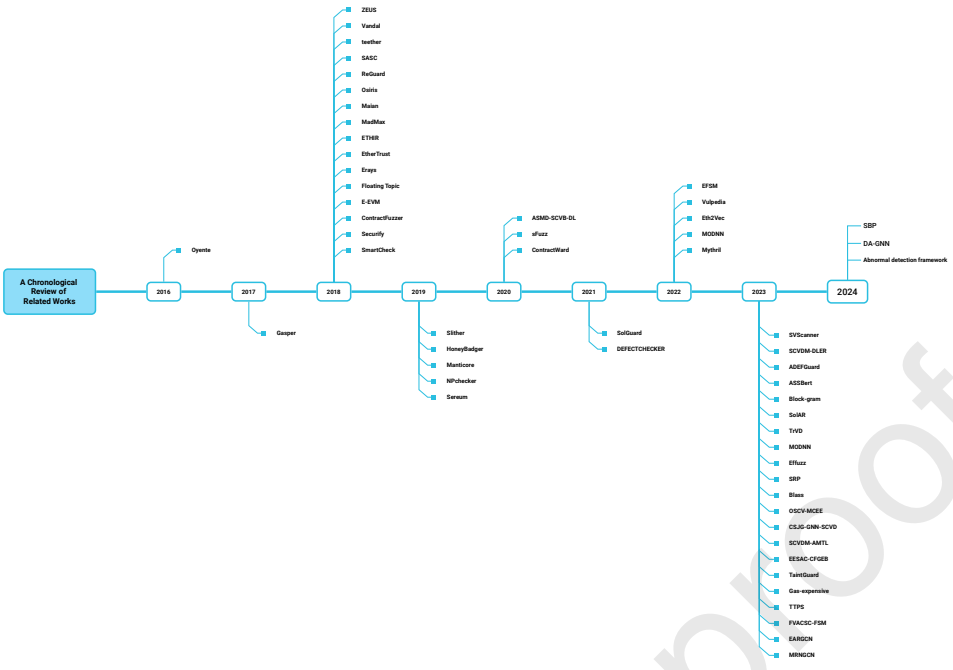


Figure 2: Chronological Overview of Work Development: A Year-by-Year Evolution Analysis

The final two stages are 'Classification' and 'Evaluation Criteria' (Section 3.5), where we implement a simple classifier to evaluate the performance of the EGA profiling model. Additionally, we outline the standards for assessing our model. The objective is to provide an advanced analytical model that utilizes existing insights and introduces novel concepts and methodologies, thereby advancing the state of SC analysis and enhancing security measures. The main architecture of the proposed model is displayed in Fig. 3, followed by a detailed explanation of each stage in the next subsections.

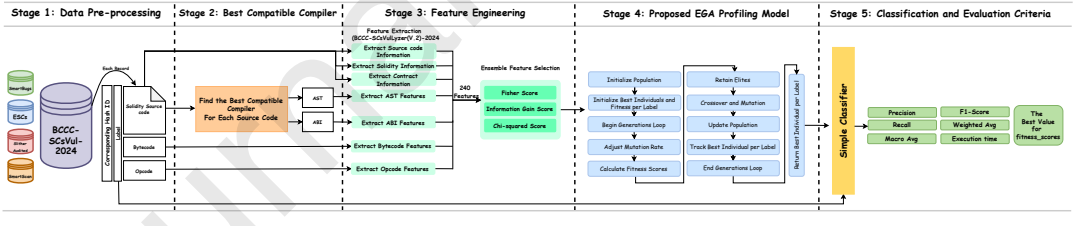


Figure 3: Proposed Model Architecture

3.1. Data Pre-processing

This section introduces the first stage of the proposed model, Data Pre-processing, as shown in Fig. 3. This stage involves collecting and processing data, which is crucial for setting up the vulnerability analysis. In this stage, we introduce a new dataset, BCCC-SCsVol-2024, with details and information in Section 4.1.2.

3.2. Best Compatible Compiler

This section discusses the second stage of the proposed model, Best Compatible Compiler, as shown in Fig. 3. According to [25], an SC compiler plays a crucial role in developing and deploying SCs on blockchain platforms like Ethereum. The compiler transforms high-level, human-readable code into low-level code or bytecode that can run on the EVM [25, 83]. Ensuring compatibility between the compiler version and the SC code is essential for maintaining

the contract's integrity and security [84]. If a contract is compiled with a different version than specified, unforeseen consequences such as security vulnerabilities or execution errors can occur due to differences in compiler behaviors.

In Solidity, the programming language used for writing SCs on Ethereum, the intended compiler version is typically specified at the beginning of the code. For instance, in the example provided in 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.

```
/**
 * @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

In brief, stage two ensures compatibility between the compiler version and the SC by specifying the most stable compiler version. This is crucial to avoid risks associated with different compiler behaviors and optimizations.

3.3. Feature Engineering

This section introduces the third stage of the proposed model, Feature Engineering, as shown in Fig. 3. It analyzes the feature extraction and selection processes in the following two subsections. The first subsection details the updated version of our previously proposed analyzer, named BCCC-SCsVulLyzer(V2.0.0), which can extract 240 features across 9 distinct categories. The second subsection explores the ensemble model used for feature selection, integrating three different approaches to leverage the strengths of each method and ensure a comprehensive and effective selection of optimal features.

3.3.1. Feature Extraction (BCCC-SCsVulLyzer(V2.0.0)

As the first part of the Feature Engineering stage, this section explains the proposed analyzer for extracting features from SCs. Syntheses from related works and feature extraction background reveal that a critical issue is extracting and examining features across diverse categories. To address this issue, our research introduces the new version of BCCC-SCsVulLyzer, which categorizes features into two broad types: compiler-based and non-compiler-based. This classification maximizes the breadth of feature extraction.

Furthermore, our research introduces innovative features under two newly defined categories: Source Code Information and Contract 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 features is intended to enrich our analysis, offering a more detailed perspective.

The architecture of BCCC-SCsVulLyzer(V2.0.0) is depicted in Fig. 4. Moreover, Fig. 5 illustrates our proposed taxonomy of the features extracted by BCCC-SCsVulLyzer(V2.0.0). Table 2 and Table 3 provide details on each extracted feature's category, definition, and measurement.

3.3.2. Feature Selection

This subsection discusses the feature selection process for preparing a set of features for the profiling step. In our analysis, BCCC-SCsVulLyzer(V2.0.0) extracted a substantial set of 240 features, which is more than sufficient for vulnerability profiling. Therefore, we implemented an ensemble feature selection method that identifies the most relevant data attributes within the SCs while eliminating redundant or irrelevant ones. This not only optimizes the

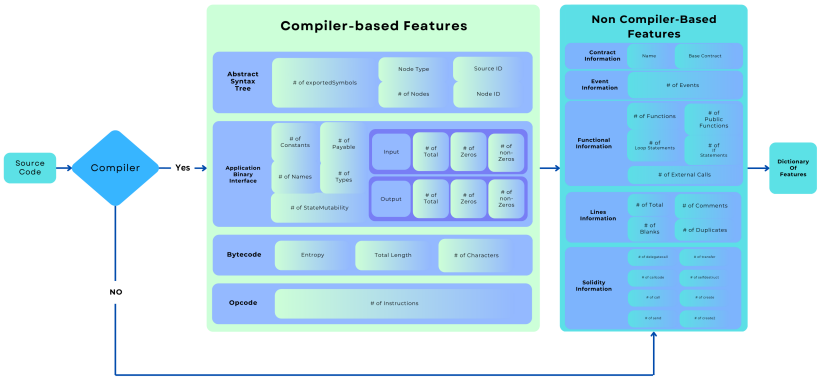


Figure 4: Architecture: BCCC-SCsVulLyzer(V2.0.0)

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 node class, 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.

efficiency of the profiling process but also improves vulnerability detection accuracy. In the following, we explain and discuss each feature selection method, highlighting its advantages and proposing the ensemble model.

The first feature selection method is the Fisher Score algorithm, detailed in 1. It is used to identify the features that are most discriminative between different classes [85]. This 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 sample and the class mean [86]. 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 [87, 88].

The second feature selection method is the Information Gain algorithm, detailed in 2. It begins by evaluating the dataset's overall entropy, which measures the initial state of disorder or uncertainty [89]. As it cycles through each dataset attribute, the algorithm calculates the reduction in entropy, known as information gain, resulting from segmenting the data according to the distinct values of each attribute [90]. This systematic approach enables the identification of the attribute that most significantly decreases entropy [91, 92]. 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 and this conditional entropy yields the information gain associated with each attribute.

The last feature selection method is the Chi-squared (χ^2), detailed in 3. It is a statistical method to evaluate the independence between each feature and the target variable in a dataset [93]. It calculates the Chi-squared statistic between each feature and the target, which measures the lack of independence between categorical variables. In

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.

practice, for each feature, the algorithm computes the expected frequencies of each category under the assumption that the feature and target are independent [94, 95]. It then compares these expected frequencies with the actual observed frequencies, accumulating the discrepancies across all categories to form the Chi-squared score [96]. Features with higher scores are considered more important as they indicate a stronger association with the target variable, suggesting that changes in the feature significantly impact the outcome.

Algorithm 1 Feature Selection: Fisher Score

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

Algorithm 2 Feature Selection: Information Gain

Require: D: Dataset with classes $C_1, C_2, , C_n$, F: Set of features
Ensure: A^i: The feature with the highest information gain
  $maxGain \leftarrow 0$ ▷ Maximum information gain
  $A^i \leftarrow NULL$ ▷ Best feature
  $H.D/ \leftarrow *_{i=1}^{n} p.C_i/\log_2 p.C_i/$ ▷ 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 $A = v$
      $H.D_v/ \leftarrow *_{i=1}^{n} p.C_iD_v/\log_2 p.C_iD_v/$
      $conditionalEntropy \leftarrow conditionalEntropy + \frac{D_v}{D} H.D_v/$
    end for
    $infoGain \leftarrow H.D/ * conditionalEntropy$
    if $infoGain > maxGain$ **then**
      $maxGain \leftarrow infoGain$
      $A^i \leftarrow A$
    end if
  end for
  return A^i

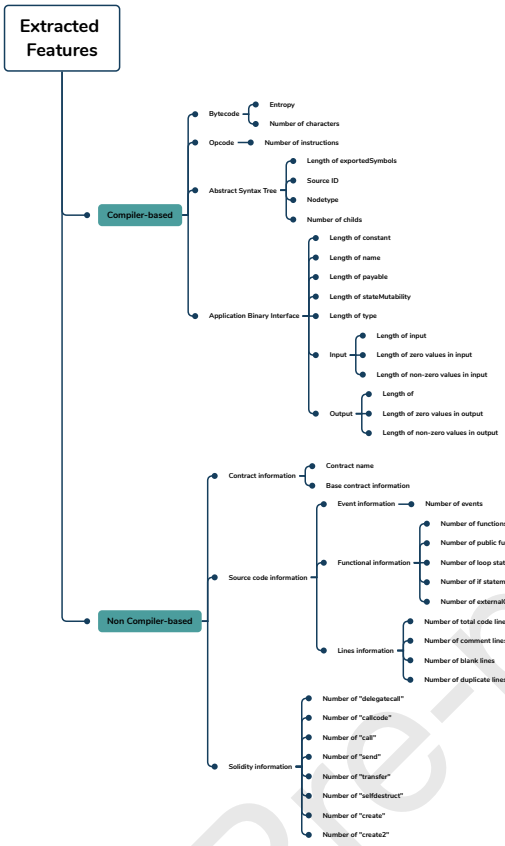


Figure 5: Taxonomy: Extracted Features Categories

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 $chi2\_scores$

Our analysis strategically combined three feature selection algorithms: Fisher Score, Information Gain, and Chi-squared (χ^2). This selection was driven by our goal to harness diverse methodologies illuminating different facets of feature relevance and their interactions with the target variable, ensuring a thorough and multifaceted analytical approach.

Starting with the Fisher Score algorithm, which is renowned for its ability to distinguish features based on their class separability, it is 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 crucial for

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.

class differentiation. 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.

To address the limitations of the Fisher Score, the Information Gain algorithm focuses on reducing entropy. It provides a deeper understanding of how each feature reduces uncertainty within the dataset, making it an asset in crafting decision trees. This approach shines in mapping out non-linear relationships and complex feature interactions, areas that the Fisher Score's linear focus may overlook. Moreover, the ensemble incorporates the Chi-squared algorithm to enhance our strategy further by targeting categorical data and assessing feature independence from the target variable. This inclusion broadens our feature selection scope, addressing biases inherent in the other two algorithms, particularly their focus on linear separability and continuous variables. Table 4 summarizes the advantages and disadvantages of these feature selection algorithms.

The ensemble approach addresses the multifaceted nature of our data, ensuring that we capture the most informative and discriminative features. By doing so, it streamlines our analysis and enhances overall model performance. AS presented in Table 5, Eighty features were selected based on their top scores, as the scores dropped significantly and remained low after the first 80.

3.4. Profiling Model

Profiling is a crucial technique for evaluating the performance of systems, algorithms, or applications, to optimize effectiveness and identify areas for improvement [97]. This process involves assessing various operational aspects, such as execution time, resource usage, and the accuracy and reliability of machine learning models. It also examines the interaction between input features and the predictions derived from them [98, 99]. The primary goal of profiling is to enhance the system's effectiveness, reliability, and overall quality [99]. This subsection analyzes the background of profiling and then explains the proposed enhanced GA profiling model in this study.

3.4.1. Profiling Algorithms Background

This segment provides a detailed exploration of various profiling algorithms. Moreover, a comprehensive taxonomy of existing profiling methods is presented in Fig. 6, providing a clear overview of the field.

- ***Statistical*** Statistical profiling collects and analyzes state data about a system, algorithm, or application to assess its performance [97]. Statistical profiling is used to gather CPU, memory, and I/O operations data to identify inefficient or low-performing areas in the system. The statistical techniques utilized in the profiling process include the Anderson-Darling test, Mahalanobis distance, Hotelling's t-squared, Hypothesis testing, Girvan-Newman algorithm, and Louvain method. The Kolmogorov-Smirnov and Kullback-Leibler (KL) divergence methodologies are also considered prominent and widely used in statistical analysis. Specifically, in this context, our attention is directed toward Kolmogorov-Smirnov and KL to maximize effectiveness and locate established and extensively studied statistical methods.

  Rassokhin *et al.* [59] pioneered the Kolmogorov-Smirnov profiling technique 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. Moreover, Zhang *et al.* [100] devised the KL divergence profiling method to effectively combine medium and low spatial resolution images. This tool examined the similarity between unknown pixels and pure winter wheat samples.

- ***Fuzzy*** Fuzzy profiling uses fuzzy logic to evaluate system performance [101]. It analyzes performance data considering several variables and their interactions using fuzzy sets, which are collections of values with different membership levels. In [102], authors provided a technique for creating fuzzy information retrieval

Table 5
Result of Feature Selection Algorithm

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

(IR) systems. They combined three relevancy profiles—a task profile, a user profile, and a document profile—to create a single index by integrating relevance feedback and fuzzy rule-based summarization algorithms.

Furthermore, Xu *et al.* [103] introduced a novel student profiling system that leverages the combined efforts of multiple agents. Fuzzy techniques enhance the system's adaptive nature, ensuring a customized and optimized learning experience for each student. Mencar *et al.* [104] proposed a unique fuzzy set-based method for characterizing preference profiles. The authors' simulations yielded strong proof of fuzzy profiling's effectiveness in creating customized recommendations according to the user's preferences. Lastly, Dickerson *et al.* [105] presented a new method of fuzzy network profiling in intrusion detection. Their approach involved meticulously examining network traffic data to create a fuzzy profile of network behavior.

- ***Genetic Algorithm*** GA-based profiling techniques employ the principles of GAs or fuzzy logic to analyze and enhance system performance [106]. These methods use iterative procedures that entail the evolution of a population of practical options using mechanisms derived from genetics and natural selection [107].

  -GA logic. In their research on machining parameter optimization, Asokan *et al.* [108] focused on achieving continuous finished profiles from cylindrical stock. They used a GA to successfully handle this machining optimization challenge because of the problem's high level of complexity. Gupta *et al.* [82] presented and evaluated a GA-based profiling technique. Their method enabled enterprises to select the lowest-cost security profile with the broadest vulnerability coverage. Finally, Resende *et al.* [109] introduced a GA-based adaptive approach for profiling features and determining parameters in anomaly-based intrusion detection. The reported tests conducted on the CICIDS2017 dataset showcased excellent results.

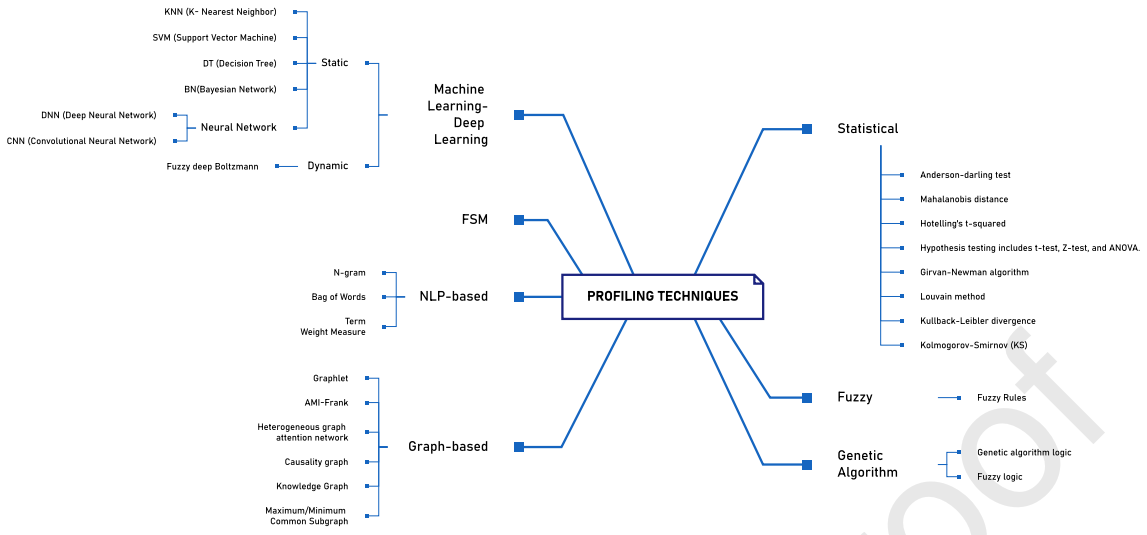


Figure 6: Taxonomy: Profiling

-Fuzzy logic. Hamamoto *et al.* [83] proposed using a GA to discover network anomalies. They suggested using flow analysis and the GA to create a digital signature for a network segment. This entailed predicting the network's traffic patterns for a given time frame using data from network flows. The authors also used GA with a fuzzy logic technique to assess if a given case is an anomaly.

- ***Machine Learning-Deep Learning*** The main goal of deep learning and machine learning profiling techniques is to thoroughly analyze and enhance prediction models' performance [110]. These innovative techniques aim to measure models' performance, reliability, and precision, meticulously identifying areas that warrant improvements to augment their effectiveness and efficiency [111]. The goal is not only to improve performance but also to improve the overall quality of the models [112]. Moving forward, we identify two primary categories of profiling methods for deep learning and machine learning: static and dynamic. In the following, we explain each method related to these categories in detail.

  -Static Static machine learning and deep learning profiling is a type of learning strategy that makes predictions or choices based on a predetermined data set, without considering new data over time [112]. In the subsequent, we explain several static learning profiling techniques, such as K-nearest neighbor, support vector machine, decision tree, deep neural network, Bayesian network, and Convolutional Neural Network.

  K Nearest Neighbor (KNN). Haque *et al.* [113] proposed a solution to the indoor localization problem by determining the Cartesian coordinates of individuals or objects within a building. They introduced a profiling-based localization method utilizing the KNN strategy. Tsalera *et al.* [114] applied KNN profiling to accurately detect noise in the environment, considering one to three nearest neighbors, which resulted in nine alternative models. Finally, Nagaraj *et al.* [115] combined KNN profiling with Feature Weighted algorithms to extract information from previous student profiles that were successful in gaining admission.

  Support vector machine (SVM). In their study on profiling, Bayot *et al.* [116] studied profiling across multiple languages. They employed SVMs, a popular machine learning algorithm for profiling, and word embedding averages, which encode words as vectors. By combining word embeddings with SVMs, the authors aimed to obtain a more accurate and adaptable method for author profiling, particularly when working with various languages. This methodology can potentially enhance the accuracy and adaptability of author profiling techniques in many linguistic contexts [116].

  Decision tree (DT). Batterham *et al.* [117] proposed a DT profiling to assess the evolution of suicide risk in a population cohort of 6656 Australian individuals. Additionally, Duchessi *et al.* [118] used DT profiling and survey data to analyze the online and mobile technologies and services of ski resorts.

Convolutional Neural Network (CNN) and Deep Neural Network (DNN). Rana *et al.* [119] investigate the usage of CNNs profiling to detect and classify different types of malware.Cura *et al.* [120] conducted a study comparing LSTM and CNN architectures for driver profiling, where the CNN architecture outperformed LSTM in detecting aggressive driving behavior [120]. In addition, Hawley *et al.* [121] suggested a deep auto-encoder model as a profiling technique, where the model is conditioned using the target audio effect's control settings samples.

Bayesian Network (BN). Baumgartner *et al.* [122], the application of the BN profiling technique in criminal profiling has the potential to offer valuable insights to law enforcement. Similarly, Xiang *et al.* [123] suggested using BNs in their study to assess behavioral patterns to utilize noisy and sparse data from indoor and outdoor surveillance scenarios.

-Dynamic In dynamic machine learning and deep learning profiling, the term refers to utilizing learning techniques capable of adjusting to evolving data or environments in real-time [124]. Various techniques are included in a dynamic category, such as online learning, transfer learning, active learning, meta-learning, reinforcement learning, and fuzzy logic [125]. Specifically, in this discussion, we concentrate on fuzzy deep logic, which is the final approach mentioned [124].

Fuzzy deep Boltzmann. Zheng *et al.* [124] developed a deep-learning strategy 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.

- ***Finite State Machine (FSM).*** FSM profiling is an effective method for understanding complicated behaviors in data analysis, particularly when combined with fuzzy logic [29]. Langensiepen *et al.* [28] explored the use of fuzzy finite state machines (FFSM) for activity recognition and worker profiling in an intelligent office environment. The authors' major goal was to improve worker efficiency and production by monitoring and evaluating their actions and behaviors with FFSM.

- ***NLP-based: N-Gram, Bag Of Words, Term Weight Measure.*** NLP technologies offer transformative potential in diverse areas, one of which is the creation of semantically rich document profiles for efficient document identification and retrieval. Guillén *et al.* [30] and Anrig *et al.* [126] highlighted the power of NLP in creating document profiles by harnessing semantic metadata extracted from text, ultimately catering to specific user requirements. Moreover, Smmarwar *et al.* [127] designed a CBG-based malware detection framework for IIoT. The framework pulls API N-gram features from the Cuckoo Sandbox, normalizes them, and then uses a method to identify sequences of API calls of varied lengths.

  While previous works focused on semantic and N-Gram profiling, Fkih *et al.* [128] introduced an approach to author profiling using the Bag of Words technique. Lastly, Kavuri *et al.* [129] proposed a novel NLP-based profiling technique called Term Weight Measure (TWM), which employs the Feature Selection algorithm to identify key features, calculating each feature value in the vector representation a crucial research task.

- ***Graph-based*** graph-based profiling is a technique for evaluating the performance of a system, algorithm, or application using graph data structures [130].The following provides details on the six categories of graph-based techniques.

  -Graphlet. Karagiannis *et al.* [130] developed comprehensive end-host profiles to capture and visualize diverse user activities and behavior. They introduced a unique graph-based profiling approach allowing flow-level data extraction and representation at the transport layer.

  -Heterogeneous Graph Attention Network (HGAT). Chen *et al.* [131] proposed a semi-supervised technique to enhance user profiles by leveraging heterogeneous graph learning. Xue *et al.* [132] introduced AppDNA, which generated concise representations of an app's behavior using a combination of function-call and heterogeneous graphs. Moreover, Labadie *et al.* [133] developed a graph-based model that encodes tweets as nodes within a graph structure. This graph representation was subsequently input into a neural network for profile classification.

  AMI-Frank. Han *et al.* [134] proposed an advanced graph-based model called AMI-Frank, designed exclusively for personalized search among folksonomies. The model used a community clustering strategy and an ant algorithm-based evaporation method to update the network.

Causality graph. Asai *et al.* [135] extracted distinctive patterns for identification using a graph-mining algorithm. The authors evaluated their framework on packet traces from seven P2P applications, which served as ground truth, and achieved an impressive overall accuracy.

Knowledge Graph. Munir *et al.* [136] recommended utilizing a knowledge graph, a graph-based data structure that captures entities and their interactions, as a profiling model. Additionally, semantic modeling was employed to define real-world concepts for computer analysis [136].

Maximum/Minimum Common Subgraph. Daoud *et al.* [137] proposed a method for personalized document ranking based on a semantic user profile represented as a graph. The suggested model used graph representation learning approaches to consider the user's preferences and interests across several topics.

In conclusion, the integration of various profiling methodologies—including graph-based, NLP-based, FSM, dynamic learning, machine learning, and statistical approaches—demonstrates their effectiveness in addressing a wide range of challenges, such as software performance optimization, document profiling, and behavioral analysis. Each method offers specific strengths, such as graph-based techniques enhancing system operations through semantic modeling and optimization and NLP-based profiling improving document identification through semantic metadata extraction and machine learning. However, these methods exhibit limitations in the context of SCs vulnerability detection, primarily due to the evolving and complex nature of security threats in decentralized systems.

In contrast, GA-based profiling techniques present distinct advantages for SCs vulnerability detection. Genetic algorithms' adaptability and iterative optimization process enable them to handle dynamic and multifaceted security challenges more effectively than static or traditional learning-based methods. GAs can evolve solutions over successive generations, allowing them to respond to shifting attack patterns and optimize multiple security parameters simultaneously. Furthermore, integrating fuzzy logic within GA frameworks enhances their capacity to detect anomalies in uncertain environments. These characteristics make GA-based approaches particularly well-suited for addressing SCs vulnerabilities, positioning them as a more robust solution than other profiling methods in this domain.

Given the diverse nature and objectives of profiling tasks, combining multiple algorithms, each with its unique strengths and approaches, may seem advantageous at first glance. However, when these algorithms are integrated, their differing methodologies and optimization strategies can often conflict, leading to contradictory outcomes. Furthermore, the increased computational complexity resulting from such combinations can impose significant demands on system resources, ultimately detracting from the desired efficiency and transparency in profiling.

Specifically, combining algorithms like graph-based, machine learning, and NLP-based techniques with GA can introduce challenges such as conflicting optimization paths and excessively high resource consumption. Therefore, instead of pursuing an overly complex multi-algorithm solution, we refined and improved the traditional genetic algorithm. The GA's iterative, population-based approach to optimization provides a more streamlined solution for vulnerability detection in SCs. By focusing on evolving and enhancing this single algorithm, we aim to retain computational efficiency while improving its adaptability to evolving vulnerability patterns by proposing an enhanced version of the GA-based profiling in the section 3.4.2.

3.4.2. Proposed Enhanced GA-based Profiling Model

Inspired by natural selection and genetics principles, GA-based profiling is a powerful computational optimization technique that adeptly handles complex logic and relationships. It leverages the evolutionary process to explore vast, multidimensional search spaces where traditional methods may struggle [138]. This technique intelligently examines various potential solutions or profiles, iterating over generations to refine and evolve these profiles. Genetic algorithms combine and modify existing solutions through crossover and mutation, which helps uncover hidden patterns or weaknesses that might not be immediately apparent [139]. This iterative improvement process is particularly valuable in detecting issues arising from complex interactions or intricate logical structures, such as vulnerabilities in smart contracts or other code-heavy systems.

GA-based algorithms extensively explore the search space and balance exploration with exploitation. This means they efficiently identify promising areas in the solution space (exploitation) while continuing to introduce novel variations to avoid local optima (exploration) [140]. By using a fitness function tailored to the specific profiling task, such as identifying security vulnerabilities, GAs can continuously evolve solutions toward higher accuracy, ensuring the discovered profiles are robust and reflective of the underlying problem [141]. Using genetic operators like selection, crossover, and mutation allows the algorithm to maintain diversity in the population, ensuring that it doesn't miss potential solutions and can adapt to changing or newly discovered patterns [142].

However, traditional GA encounters specific challenges. The first challenge is premature convergence, which occurs when the algorithm settles on a sub-optimal solution too early due to insufficient diversity or exploration in its early stages. Next is a potential loss of genetic diversity within the population over generations, leading to local optima and limiting the algorithm's ability to explore a broader range of possible solutions. Following these challenges, an imbalance in feature selection can occur, resulting in underfitting or overfitting, which leads to poor algorithm performance. Moreover, valuable information guiding the algorithm toward optimal solutions may be lost without a mechanism to preserve high-quality solutions from previous generations.

Challenges in GA include the issue of a static mutation rate, which can cause problems as different phases of the algorithm require varying levels of exploration and exploitation for efficient convergence. This static rate might not be suitable throughout the entire run of the algorithm. Additionally, a lack of solution diversity can lead to premature convergence, so the crossover process must introduce sufficient diversity to prevent the population from becoming too homogeneous. Finally, without a mechanism to ensure that at least some features remain active, the algorithm may converge on trivial solutions where all features are turned off.

This subsection explains the proposed profiling model, representing an enhanced iteration of the traditional GA. The proposed profiling model incorporates three additional enhancements beyond the improvements previously introduced in [25]. Below, we describe the enhancements at each step of the EGA profiling technique. The baseline is represented by "1," while "2" to "5" indicate improvements over the previous version. The latest advancements are numbered "6," "7," and "8."

1. **Regular Genetic Algorithm**: GA is a search and optimization technique inspired by the process of natural selection and genetics [108]. It simulates the evolution of a population of candidate solutions to a problem. This population evolves through processes mirroring natural selection, including crossover, mutation, and survival of the fittest [109]. 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 [25]. GAs are widely used for solving complex problems where traditional optimization methods are ineffective.
2. **Increased Population Size**: The EGA employs a significantly enlarged population size of 10,000 individuals, 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 [25]. By incorporating a vast array of genetic variations, the EGA improves its ability to thoroughly explore the multidimensional solution space. This extensive search capability is crucial for mitigating the risk of premature convergence on suboptimal solutions, thereby augmenting the algorithm's proficiency in identifying and evolving toward the global optimum.
3. **Semantic Similarity Distance (SSD) Checks**: Incorporating SSD 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 EGA maintains genetic diversity within the population. Simultaneously, it avoids the combination of drastically different individuals to prevent the generation of implausible or inefficient offspring [25].
4. **Initialization with Bias Towards Zeros**: Initiating the population with a bias towards zeros, 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 [25].
5. **Handling Zero Features**: The EGA incorporates a safeguard mechanism to avoid generating invalid solutions—specifically, those devoid of any selected features. 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 [25].
6. **Penalty Fitness Function**: Integrating a penalty fitness function introduces a methodical approach to discourage selecting solutions exhibiting certain undesirable traits. In feature selection, solutions that incorporate too few features might be penalized. This penalization mechanism reduces the fitness score of less desirable solutions, effectively guiding the evolutionary process toward more favorable and viable solutions. This function improves the EGA's efficiency by focusing the search on solution regions more likely to yield optimal outcomes.
7. **Retention of Elites**: By preserving a subset of elite individuals—the top performers within each generation—EGA ensures the continuity of superior genetic material across generations. This elite retention strategy

accelerates the population's convergence toward optimal solutions. By safeguarding the genetic integrity of the optimal solutions, the EGA can build upon these high-quality foundations, thereby enhancing the overall rate of evolutionary progress and increasing the likelihood of achieving superior outcomes.

8. **Adaptive Mutation Rate**: The EGA'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 evolution progresses and promising solutions 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.

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.

Introducing a penalty fitness function, preserving 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 "6" to "8," builds on its predecessor, demonstrating a deliberate strategy to overcome the limitations of traditional GAs and expand the capabilities of evolutionary computation. The EGA profiling, detailed in 4, systematically outlines and presents the sequence and integration of each innovative component in detail.

Algorithm 4 EGA 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
```

3.5. Classification and Evaluation Criteria

The final stage of the proposed model architecture elaborates on the classification and evaluation of the proposed profiling model. To evaluate, we utilize a simple classifier during the initial training phase, which allows us to assess

Table 6
Details of Evaluation Metrics

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.	$F1 = 2 \odot \frac{Precision \odot Recall}{Precision+Recall}$
Accuracy	Fraction of all correct predictions.	$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$
Macro Average	Average metric computed independently for each class.	$MacroAvg = \frac{1}{N}\sum_{i=1}^{N} Metric_i$
Weighted Average	Average metric weighted by class size.	$WeightedAvg = \sum_{i=1}^{N} w_i \odot Metric_i$
Execution Time	Total duration required for a process or algorithm to complete its task.	$T_{exec} = T_{end} * T_{start}$
Best Fitness	The highest value of a fitness function achieved by an individual in a population.	$F_{best} = \max.F\,x_i//$

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, ensuring that the results primarily reflect the quality of the pre-processed and structured data rather than the complexity of the classification algorithm. Additionally, this subsection outlines the metrics employed for evaluation and explains their significance. 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 section 4.

4. Experiments and Results

This section explores the experimental setup and methodologies used to efficiently validate the proposed profiling. A key aspect of the validation process is the creation of a comprehensive, specifically designed dataset that provides a robust testing ground and diverse vulnerabilities to assess the model's performance. The validation process was followed to measure the model's proficiency, starting with Hyper-parameter tuning and culminating in experiments on various vulnerabilities, detailed in the following subsections.

4.1. A New SCs Dataset

This subsection reviews the publicly available datasets listed in Table 7. After that, it introduces the newly created dataset for this study, called BCCC-SCsVul-2024.

4.1.1. Available Datasets

This section critically analyze the datasets utilized in the field of SCs vulnerability detection research, categorizing them by their respective sizes.

- **Small-scale (Less than 100 samples):** The first group of studies introduced datasets with fewer than one hundred samples. Samreen *et al.* [67] utilized a dataset comprising six samples from Etherscan, illustrating the challenges of small datasets, such as limited scope and potential reliance on external tools for labeling. Liu *et al.* [3] analyzed five SCs exclusively containing Re-entrancy bugs and Vivar *et al.* [72] focused on a single bytecode sample. Additionally, In [143], Aquilina *et al.* used a dataset of six SCs, each characterized by specific attack types.

- **Medium-scale (Between 100 and 10,000 samples):** The second group of studies provided datasets with more samples, though still fewer than 10,000. In [62], researchers evaluated their type verifier system using 110 samples from Etherscan, but this dataset is limited in its ability to detect security issues because the samples lack labels. In contrast, some datasets are binary and provide information about secure or vulnerable SCs. Liu *et al.* [84] provided a general dataset with 1382 samples. Similarly, Gupta *et al.* [71] and Zhang *et al.* [2] utilized two different binary datasets with less than 10,000 samples which still are not enough for training the models.

- **Large-scale (More than 10,000 samples):** The final group of studies utilized larger datasets, providing a broader range of information on vulnerable SCs. Qian *et al.* [80] provided a dataset comprising 40,700 SCs in various formats (Solidity, bytecode, and Binary code), but it is limited to four types of vulnerabilities. Liu

Table 7
Comparative Analysis of Previous Datasets

Cat.	Ref.	Name	# of samples	Format	Label	Details	Advantages	Disadvantages
Small-scale	Samreen *et al.* [67]	DeFi, Globalcryptox, FairDare, Moneybox, AIRToken QuizBLZ	6	Solidity	1	Re-entrancy	Simplicity	Labeled with other tools.
	Liu *et al.* [3]	TokenSender, UGToken, E4Token, DAO RoT	5	Solidity	1	Re-entrancy	Simplicity	Labeled with other tools.
	Vivar *et al.* [72]	the100th Ethereum contract(since the Ethereum blockchain was launched)	1	Byte-code	1	Not mentioned	Simplicity	Labeled with other tools.
	Aquilina *et al.* [143]	Bank, DelayUnderflow, ProductVote, simulationERCToken, simulationKotET TargetUnderflow	6	Solidity	1	Over/underflow DoS Re-entrancy	Simplicity	Insufficient samples.
Medium-scale	Hu *et al.* [62]	110 contracts from etherscan	110	Solidity	0	Not mentioned	Simplicity	Only used for checking typeability.
	Liu *et al.* [84]	Samples from Etherscan	1,382	Solidity	2	Safe Not safe	Sufficient samples.	No details about categories.
	Gupta *et al.* [71]	Google BigQuery	7,000	Solidity	2	Safe Not safe	Sufficient samples.	No details about categories.
	Zhang *et al.* [2]	SmartBugs Dataset-Wild	47,587	Solidity	2	Safe Not safe	Sufficient samples.	No details about categories.
Large-scale	Qian *et al.* [80]	Dataset processing for vulnerabilities	40,700	Solidity Byte-code Binary-code	4	Delegatecall Integer Overflow Re-entrancy Timestamp	Sufficient samples.	Covers only 4 attacks.
	Liu *et al.* [144]	ESC	40,932	Solidity	3	Re-entrancy Timestamp Infinite-Loop	Sufficient samples.	Covers only 3 attacks.
	Liu *et al.* [144]	VSC	4,170	Solidity	3	Re-entrancy Timestamp Infinite-Loop	Sufficient samples.	Covers only 3 attacks.
	Ding *et al.* [145]	SCs come from Fabric, Caliper, and IBM	-	Including HF version 0.6 and HF version 1.1	2	Logic loopholes Integer overflow	Fit for HF task.	Not general.
	Hajihosseinkhani *et al.* [25]	Solidity SCs	36,670	Solidity	2	Binary dataset	Sufficient samples and fit for binary classification.	Do not have vulnerabilities.

et al. [144], and Ding *et al.* [145] offered datasets with a focus on specific vulnerability types, the latter in HF versions 0.6 and 1.1, but confined to a narrow range of attacks. Hajihosseinkhani *et al.* [25] introduced a dataset comprising 36,670 samples for binary classification of vulnerable and secure SCs.

These datasets from prior research exhibit significant limitations. Small-scale datasets often face issues of overfitting and lack broad applicability. Medium-scale datasets, while offering more samples, are still insufficient for comprehensive model training and typically provide limited binary classifications. Moreover, large-scale datasets 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 SCs.

4.1.2. New Vulnerable SCs datasets (BCCC-SCsVul-2024)

This research introduces a new dataset by collecting Solidity source code of 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 provides additional details about the new dataset.

Table 8
BCCC-SCsVul-2024 Dataset Overview: An (*) indicates the secure examples, meaning they do not contain any known vulnerabilities.

BCCC-SCsVul-2024	
Label	# of Samples
Total	111897
External Bug	3604
Gas Exception	6879
Mishandled Exception	5154
Timestamp	2674
Transaction Order Dependence	3562
UnusedReturn	3229
Weak Access Mod	1918
Call to Unknown	11131
Denial of Service	12394
Integer Underflow/Overflow	16740
Re-entrancy	17698
Secure*	26914

Table 9
Hyper-parameter Tuning Results: The (*) indicates the final values used in the EGA.

Impact of Number of Generations on the Number of Features in EGA Profiling	
# of Generations	**# of Features**
50	59
100 *	65
150	75
200	77
500	80

Impact of Alpha and Beta on the Number of Features in EGA Profiling		
Alpha	**Beta**	**# of Features**
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

4.1.3. Hyper-parameter Tuning

This section discusses the tuning of hyper-parameters that are pivotal for the performance optimization of the proposed EGA profiling. The main focus is primarily on the number of generations (the number of iterations the EGA performs), alpha (the lower bound of SSD range), and beta (the upper bound of SSD range) 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 EGA'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 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 EGA's output, suggesting its robustness and a high degree of generalization. Table 9 provides detailed information on how the number of features selected by the EGA varies with different hyper-parameter settings. 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. The final setup of hyper-parameters is indicated in the table by an (*) sign next to the value.

In conclusion, tuning hyper-parameters such as the number of generations, alpha, and beta is crucial for optimizing the performance of the proposed EGA. These parameters help balance the exploration of new solutions with the

exploitation of known ones, which is essential for the EGA's effective navigation through the solution space. Notably, changes to these hyper-parameters have minimal impact on the EGA's output, demonstrating its exceptional robustness and strong generalization capabilities.

4.2. Vulnerability Classification Results

In this study, we conduct a classification process to validate the proposed profiling model's ability to detect various vulnerabilities. Table 10 provides a comprehensive view of how the proposed method performs in identifying secure and different vulnerabilities, based on the evaluation metrics defined in Table 6. Furthermore, the evaluation details on the complexity, performance analysis, and comparing with available models are provided in Section 5.

5. Analysis and Discussion

This section provides a comprehensive analysis of the proposed model by addressing four critical questions. First, it evaluates the model's performance by examining the results presented in Section 4 and Table 10. Next, it examines the model's time and space complexity and analyzes its stability and generalizability through a K-fold process. Finally, it compares the proposed model with traditional and NN-based methods regarding their effectiveness in detecting vulnerabilities.

(1) RQ1: How efficient is the proposed model in terms of the defined evaluation parameters outlined in Table 6? The classification report in Table 10 provides a structured overview of the proposed method performance across various vulnerabilities. Focusing on the precision parameter, representing the fraction of true positive results among all positive predictions, the proposed model demonstrates strong precision in identifying vulnerabilities, with scores ranging from 0.79 to 0.88 across different types. It also achieves a precision of 0.91 for recognizing secure instances, indicating that when it classifies an SC as secure, there is a 91% chance it is indeed free from vulnerabilities. Furthermore, recall, which measures the model's ability to identify all positive cases, ranges from 0.41 to 0.69. This reflects the proportion of actual positives correctly identified. While the lower recall indicates some missed positives, the model's high precision ensures that the identified instances are mostly accurate, maintaining satisfactory performance.

Continuing the performance analysis, the F1-score, representing the harmonic mean of precision and recall, ranges from 0.56 to 0.75. This range reflects the inherent complexity of detecting vulnerabilities, highlighting the model's ability to manage the nuanced detection strategies required for such complicated issues. Achieving F1-scores within this range against challenging vulnerabilities demonstrates the model's capability to provide valuable insights and detections, even when faced with complex behaviors.

Further evaluating the performance, the model's accuracy 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 on to Macro Avg and Weighted AVG, Macro AVG calculates the average score for each metric without considering class distribution. At the same time, Weighted AVG adjusts for the prevalence of each class. Macro averages—approximately 0.81 for precision, 0.73 for recall, and 0.79 for F1-scores—offer a balanced overview of the model's performance across different classes, highlighting moderate effectiveness and areas for improvement, especially recall. Weighted averages, which reflect class distribution, show 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, highlighting variations in performance across different classes and suggesting opportunities for refinement.

Finally, achieving the Best Fitness value 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 high effectiveness and substantially reducing the likelihood of overlooking any issues. Maintaining a score in this range also ensures the model remains practical and manageable without becoming overly complex, thus preserving accuracy.

Table 10
Classification Report

Classification Report						
	Precision	**Recall**	**F1-score**	**Accuracy**	**Execution Time**	**Best Fitness**
External Bug	0.85	0.48	0.61	0.93	48.63	65
secure	0.94	0.99	0.96			
macro avg	0.89	0.73	0.79			
weighted avg	0.93	0.93	0.92			
Gas Exception	0.82	0.53	0.64	0.88	49.71	67
secure	0.89	0.97	0.93			
macro avg	0.86	0.75	0.79			
weighted avg	0.88	0.88	0.87			
Mishandled Exception	0.83	0.46	0.60	0.89	49.39	65
secure	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	49.44	68
secure	0.95	0.99	0.97			
macro avg	0.89	0.71	0.77			
weighted avg	0.94	0.94	0.93			
Transaction Order Dependence	0.88	0.47	0.61	0.93	48.82	70
secure	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
secure	0.93	0.99	0.96			
macro avg	0.90	0.70	0.76			
weighted avg	0.92	0.93	0.92			
Weak Access Mod	0.88	0.66	0.75	0.97	47.29	68
secure	0.98	0.99	0.99			
macro avg	0.93	0.83	0.87			
weighted avg	0.97	0.97	0.97			
Call to Unknown	0.81	0.55	0.65	0.84	51.29	66
secure	0.84	0.95	0.89			
macro avg	0.83	0.75	0.77			
weighted avg	0.83	0.84	0.83			
Denial of Service	0.79	0.51	0.62	0.83	52.74	67
secure	0.80	0.94	0.86			
macro avg	0.80	0.72	0.74			
weighted avg	0.80	0.80	0.79			
Integer Underflow/Overflow	0.83	0.69	0.75	0.83	52.74	67
secure	0.83	0.91	0.87			
macro avg	0.83	0.80	0.81			
weighted avg	0.83	0.83	0.82			
Re-entrancy	0.79	0.59	0.67	0.78	53.22	68
secure	0.78	0.90	0.83			
macro avg	0.78	0.75	0.75			
weighted avg	0.78	0.78	0.77			

(2) RQ2: How efficient is each component of the proposed profiling model in terms of time and space complexity? The efficiency and reliability of the proposed EGA profiling model are highlighted by its execution time metrics detailed in Table 10. The model demonstrates a streamlined and effective process with an average execution time of approximately 49 seconds. The minimum execution time of 47.29 seconds underscores the system's high efficiency, indicating swift performance and minimal delays. Additionally, the maximum execution time of 53.22 seconds supports the system's consistency and dependability in time management, ensuring that most tasks are completed promptly and efficiently.

Moreover, the following analysis examines the specific space and time complexities of each component in the proposed EGA and the feature selection and training phases. This improved version of the genetic algorithm

incorporates four advanced elements: the Penalty Fitness Function, Retention Strategy, Adaptive Learning Rate, and SSD Check.

- **Penalty Fitness Function**
  Space Complexity: The space requirement is mainly determined by the size of the individual array. Since no extra arrays or data structures are introduced that scale with the input size, the space complexity is constant, O(1).
  Time Complexity: The internal operations, including `np.power`, `np.sum`, and conditional checks, are executed in linear time relative to the size of the individual. As a result, the time complexity is O(n), where n denotes the length of the individual.

- **Retention Strategy (retain_elites)**
  Space Complexity: The primary space consumption comes from the elites list, which contains no more than `elite_count` individuals. Since the size of this list remains fixed and does not grow with the population size, the space complexity is O(1).
  Time Complexity: This procedure involves sorting the fitness scores, which generally takes O(m log m) time, where m is the population size. The subsequent steps, such as selecting indices and appending elites, run linearly concerning the number of elites. Therefore, the dominant factor in the time complexity is the sorting operation.
  Space Complexity: This component does not use additional space proportional to the input size, maintaining a space complexity of O(1).
  Time Complexity: Since the operations involve simple mathematical calculations and conditional checks, the time complexity remains constant at O(1).

- **SSD Checks**
  Space Complexity: The SSD Checks 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 creating additional data structures that scale with the input size, leading to a space complexity of O(1).
  Time Complexity: The SSD Checks functions perform operations that scale linearly with the size of the input arrays; therefore, the time complexity is O(k), where k represents the length of `st1` or `st2`.

- **Main GA Loop**
  Space Complexity: Letting n represent the individual length and m the population size, the space complexity is O(mn).
  Time Complexity: The time complexity involves multiple components:

  – Fitness calculation for each individual has a time complexity of O(n), resulting in O(mn) for all m individuals.
  – Sorting for elite retention requires O(m log m).
  – The crossover and mutation processes are estimated at O(mn), assuming these operations are linear concerning individual length.

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

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

  – Feature Selection algorithms: The ensemble feature selection algorithm maintains a constant space complexity relative to the size of the input dataset. This complexity primarily depends on the number of features and classes, rather than the number of instances, resulting in a space complexity of O(1) in this scenario.
  – Model Training: The space complexity of the training process is O(T * d * 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 this case).

  Time Complexity:

  – Feature Selection algorithms: The time complexity scales with the product of the number of instances (N), features (F), and classes (C), resulting in a time complexity of O(NCF).

Table 11
Space and Time Complexity of Each Component

Component	Space Complexity	Time Complexity
Penalty Fitness Function	O(1)	O(n)
Retention Strategy (retain_elites)	O(1)	O(m log m) m: population size
Adaptive Learning Rate	O(1)	O(1)
SSD Checks	O(1)	O(k) k: length of arrays in SSD
Main GA Loop	O(mn) n: individual length m: population size	O(g * mn + m log m) m: population size n: individual length g: number of generations
Feature Selection and Model Training	O(T * d * F) T: number of trees d: depth of the tree F: number of features	O(T * N * F * log(N)) N: number of samples F: number of features

– Model Training: Time complexity of the training phase split into Training Time and Prediction Time (for M instances) which are O(T * N * F * log(N)) and O(M * T * 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.

The detailed analysis of the proposed EGA shows that space complexities remain predominantly constant at O(1), ensuring minimal space usage. In contrast, time complexities vary from constant to linear and logarithmic, ranging from O(1) to O(mn + m log m), depending on the specific operation. This demonstrates the model's ability to efficiently manage various computational tasks, making it a suitable solution for complex vulnerability detection scenarios while keeping computational demands manageable. Table 11 summarizes the ablation studies, which assess the impact of each component of the model.

(3) RQ3: How does the proposed model perform in terms of stability and generalizability?
The stability and generalizability of the proposed EGA profiing model is verified using multiple rounds of cross-validation. This process 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. In table 12, we provide the model's performance results through the k-fold cross-validation process for k = 5, 10, and 15.

First, the results indicate that the model has demonstrated excellent generalizability, evidenced by its consistent performance across various labels, including External Bug, Gas Exception, Mishandled Exception, and others. The precision, recall, and F1-scores remain relatively stable across various labels, indicating that the model can generalize well across different labels and vulnerabilities. Second, 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.

(4) RQ4: How does the effectiveness of the proposed model in detecting vulnerabilities compare with traditional and NN-based vulnerability detection methods?

To evaluate the effectiveness of the proposed EGA, we compare it against two categories of approaches: traditional and NN-based. The key insight, as highlighted in Table 13, is that most previous methods primarily focused on detecting re-entrancy vulnerabilities, neglecting others and failing to identify more than three types of vulnerabilities. As a result, the table contains numerous (-) symbols, indicating that these tools are not applicable for certain vulnerabilities.

Secondly, we assess EGA against leading traditional vulnerability detection approaches. Following the methodology of Chen et al. [20], we selected two benchmark tools, Mythril and Slither, known for their effectiveness in detecting vulnerabilities. The traditional tools performed weaker on the BCCC-SCsVul-2024 dataset, with Slither achieving the highest accuracy at 0.73. In contrast, EGA outperformed these tools, improving accuracy by 0.05 and reaching 0.78.

Table 12
Evaluation metrics for different models

Label	K-fold	Precision	Recall	F1-score
External Bug	5	0.82	0.46	0.58
	10	0.83	0.46	0.59
	15	0.85	0.48	0.61
Gas Exception	5	0.76	0.45	0.58
	10	0.79	0.53	0.56
	15	0.82	0.53	0.64
Mishandled Exception	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
Transaction Order Dependence	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
Weak Access Mod	5	0.80	0.60	0.68
	10	0.83	0.63	0.71
	15	0.88	0.66	0.56
Call to Unknown	5	0.80	0.48	0.60
	10	0.81	0.50	0.61
	15	0.81	0.55	0.65
Denial of Service	5	0.74	0.47	0.57
	10	0.75	0.50	0.60
	15	0.79	0.51	0.62
Integer Underflow/Overflow	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

The performance gap is largely due to traditional tools relying on simplistic patterns to detect vulnerabilities. For instance, Mythril identifies re-entrancy by merely checking for a '.value' invocation followed by an internal function call. Additionally, Mythril's inability to distinguish between 'send()', 'transfer()', and 'call()' functions often results in frequent misidentification of vulnerabilities. In contrast, EGA leverages EGA profiling and a comprehensive feature set, offering a more holistic view of smart contracts. This allows it to accurately identify a broader range of vulnerabilities, outperforming traditional solutions.

Apart from comparing with traditional methods, we also assess EGA against NN-based vulnerability detection approaches. We selected seven open-source methods employing various neural network 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. In comparison with NN-based tools, EGA surpassed the best-performing approach, AME, by 0.08 and 0.22 accuracy for Re-entrancy and timestamp vulnerabilities. Furthermore, EGA showed 0.02 and 0.15 accuracy improvements over SCVHunter and DR-GCN, the top-performing models in sequence processing, respectively.

The performance difference between EGA profiling and existing NN-based models stems primarily from a key limitation of NN-based approaches: their black-box nature, which makes their internal processes difficult to interpret. This lack of explainability renders them less suitable for assessing SCs security. In contrast, the EGA model offers greater transparency and interpretability by detecting SC vulnerabilities through well-defined attributes tied to various

Table 13
Performance Comparison: A (-) signifies that the approach cannot detect the specific vulnerability.

Vulnerability	Metric	Traditional Approaches		Neural Network-based Approaches							Proposed Model
		Mythril	Slither	Vanilla-RNN	LSTM	GRU	DR-GCN	TMP	AME	SCVHunter	EGA
External Bug	Acc	-	-	-	-	-	-	-	-	-	0.93
	F1	-	-	-	-	-	-	-	-	-	0.61
Gas Exception	Acc	-	-	-	-	-	-	-	-	-	0.88
	F1	-	-	-	-	-	-	-	-	-	0.64
Mishandled Exception	Acc	-	-	-	-	-	-	-	-	-	0.89
	F1	-	-	-	-	-	-	-	-	-	0.6
Timestamp	Acc	-	-	0.48	0.53	0.57	0.79	0.73	0.72	0.68	0.94
	F1	-	-	0.48	0.55	0.55	0.67	0.61	0.62	0.63	0.58
Transaction Order Dependence	Acc	0.75	0.46	0.47	0.53	0.51	-	-	-	0.79	0.93
	F1	0.49	0.28	0.47	0.54	0.57	-	-	-	0.6	0.61
UnusedReturn	Acc	-	-	-	-	-	-	-	-	-	0.93
	F5	-	-	-	-	-	-	-	-	-	0.56
Weak Access Mod	Acc	-	-	-	-	-	-	-	-	-	0.97
	F1	-	-	-	-	-	-	-	-	-	0.75
Call to Unknown	Acc	-	-	-	-	-	-	-	-	-	0.84
	F1	-	-	-	-	-	-	-	-	-	0.65
Denial of Service	Acc	-	-	-	-	-	-	-	-	-	0.83
	F1	-	-	-	-	-	-	-	-	-	0.62
Integer Underflow/Overflow	Acc	-	-	-	-	-	-	-	-	-	0.83
	F1	-	-	-	-	-	-	-	-	-	0.75
Re-entrancy	Acc	0.64	0.73	0.51	0.87	0.61	0.65	0.69	0.7	0.76	0.78
	F1	0.47	0.59	0.52	0.58	0.6	0.6	0.57	0.62	0.63	0.67

aspects of the source code. This unique advantage underscores the robustness and reliability of the EGA approach, making it a more favorable option for vulnerability detection in SCs. Furthermore, the analysis of all evaluation metrics demonstrates that EGA consistently outperforms NN-based solutions in accuracy across all vulnerabilities.

However, the F1-score of the proposed EGA for detecting the Timestamp vulnerability is lower than previous solutions. This discrepancy can be attributed to the limited number of samples available for the Timestamp vulnerability, as shown in Table 8. The scarcity of samples results in a lower recall value, which, in turn, reduces the overall F1-score. To address this, future work will expand the dataset to include more samples for the Timestamp vulnerability, thereby improving performance in this area.

In conclusion, the proposed EGA stands out for its exceptional ability to detect 11 types of vulnerabilities while maintaining a well-balanced performance between F1-score and accuracy. Its transparent and interpretable process further strengthens its position as a viable and promising alternative for comprehensive vulnerability detection, significantly outperforming traditional and NN-based methods.

In a nutshell, the EGA profiling approach exhibits high precision and accuracy in identifying vulnerabilities, with efficient execution times averaging 49 seconds and minimal space and time complexities. K-fold cross-validation demonstrates the model's consistent performance and excellent generalizability. Furthermore, the EGA model surpasses traditional tools and NN-based methods, achieving superior accuracy and precision while providing a transparent and reliable solution for detecting vulnerabilities in SCs.

6. Discussion

This section provides information on the detailed analysis and discussion of profiling results using the EGA profiling method. It then proceeds to visualize the generated profiles using Genes, introducing two common profiles for vulnerable and secure SCs.

6.1. Profiling Results

This section discusses the profiling results obtained from the EGA method, which was presented using violin plots. Through this process, each category of features from Section 2.2 is examined individually. Additionally, twelve plots are generated for each category to illustrate the vulnerabilities and features, highlighting the differences between secure and vulnerable samples. In each violin plot, vulnerable SCs are represented in red, while secure SCs are shown in green.

The first category of features, AST, provides a structural view of the code, enhancing the understanding of its syntax and semantics. BCCC-SCsVulLyzer(V2.0.0) extracted five unique features from this category: ID, number of

exported symbols, number of nodes, node type, and source ID. However, EGA profiling identifies three features—number of exported symbols, node type, and source ID—that are consistently represented in most profiling plots due to their significance:

- **Number of nodes:** Reflects code complexity, logic, nesting depth, size, and abstraction level.
- **Node types:** Highlight risks associated with specific code constructs that may pose security threats.
- **Source ID:** Pinpoints vulnerable areas in the codebase for targeted reviews and improvements.

As shown in Fig. 7a, the plots for Re-entrancy and Mishandled Exceptions highlight only three features, emphasizing their ability to profile these two vulnerabilities and uniquely characterizing them. However, these features exhibit similar patterns for other vulnerabilities. To address this issue, “number of exportedSymbols" and “ID," have been incorporated to create distinct profiles, such as the profile for Transaction Order Dependence.

Despite creating differentiated profiles with AST, profiling some vulnerabilities, such as Timestamp, Unsend Returns, Transaction Order Dependence, and Weak Access Control, remains similar. The similarities likely stem from the inherent nature of specific vulnerabilities within this particular feature set, which are attributed to shared underlying coding patterns exhibited by them. These patterns include similar control structures, error-handling mechanisms, or arithmetic operations reflected in their AST representations. Consequently, the AST features, such as node types and structures, become comparable, leading to analogous profiles.

The second category of features is ABI, as shown in Fig. 7b, which defines interactions at the binary level. BCCC-SCsVulLyzer(V2.0.0) identifies eleven features from ABI according to Table 2. However, EGA reveals that only the number of non-zero and zero input values consistently appear across the eleven profiling plots. This consistency indicates that SCs with significant numbers of these input values are more susceptible to attacks, underscoring the need for proper examination and control to enhance security.

Although the number of non-zero and zero input values is crucial in identifying vulnerabilities, additional features such as the count of constants, total inputs, zero-value outputs, contract name length, and type diversity enhance the profiling. The number of constants for specific vulnerabilities like Timestamp and Transaction Order Dependence helps refine and distinguish these profiles from others, such as Re-entrancy.

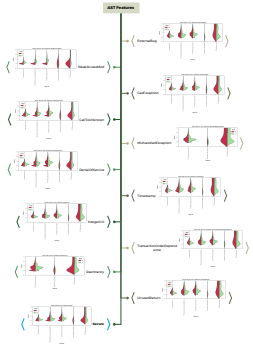
Examining Fig. 7b in more detail, it is evident that ABI features uniquely characterize some vulnerabilities while failing to do so for others. For example, the Weak Access Mod vulnerability has a unique profile. This uniqueness is due to the ability of the ABI features to relate to SC accessibility, function call patterns, and resource permissions. The distinct ABI signature of Weak Access Mod arises from its connection to incorrect permission configurations. Conversely, vulnerabilities like Call to Unknown and Denial of Service share similar ABI profiles due to their common characteristics, such as improper resource utilization.

The third category of features is bytecode, detailed in Fig. 7c. BCCC-SCsVulLyzer(V2.0.0) extracts three types of features from this category: the count of each character in the bytecode, bytecode entropy, and total bytecode length from SCs. While EGA identifies almost all features as important for profiling vulnerabilities, two notable patterns emerge. 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 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.

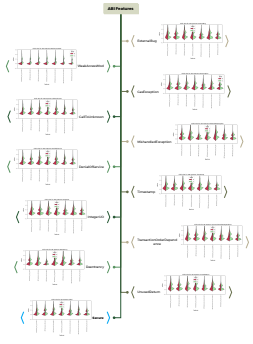
While profiles like External Bug and Re-entrancy are highly distinctive, others are less so. The bytecode profiles of External Bugs and Re-entrancy are unique due to their specific bytecode patterns and operational characteristics, which result from particular code structures, making them more pronounced and identifiable. In contrast, similar profiles for Integer Underflow/Overflow and Timestamp can be differentiated by fluctuations in bytecode length and specific character counts. Notably, a single distinct feature can significantly differentiate and individualize a profile within the bytecode feature category.

The fourth category of features is functional, as shown in Fig. 7d. BCCC-SCsVulLyzer(V2.0.0) extracts five key features from this category: the count of specific instructions related to function details, including the number of loops, public functions, conditional statements, external calls, and total functions. These elements reflect the complexity and access control of functions, directly affecting contract behavior and security.

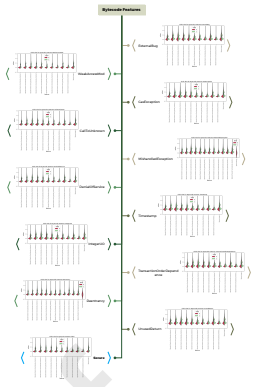
However, the EGA model uniquely employs ’public’ as the sole feature, showing significant fluctuations among profiling violin plots. Remarkably, Weak Access Mod exhibits higher values due to its reliance on access control mechanisms in SCs. This vulnerability involves inadequate access control settings related to the visibility of functions



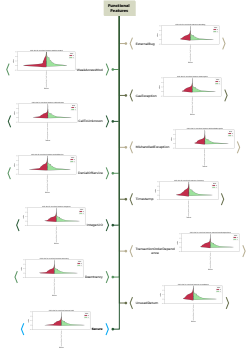
(a) Profiling details: AST



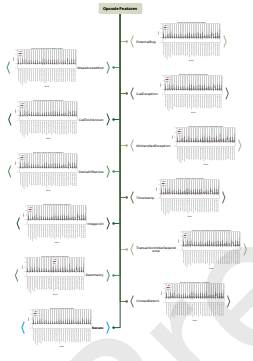
(b) Profiling details: ABI



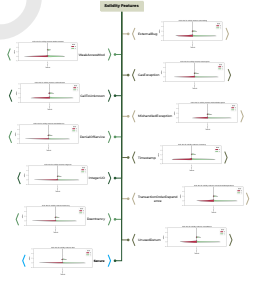
(c) Profiling details: bytecode Features



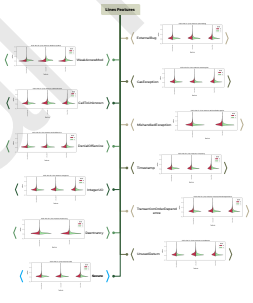
(d) Profiling details: Functional Features



(e) Profiling details: opcode Features



(f) Profiling details: Solidity Features



(g) Profiling details: Lines Features

Figure 7: Profiling details

(i.e., whether they are declared as public). Consequently, a larger number of public functions correlates with weak access control settings, increasing the likelihood of Weak Access Mod vulnerability.

The fifth category of features is opcode, which reveals notable profiling variations, as shown in Fig. 7e. EGA identifies 36 unique opcode instructions that reveal distinct patterns, highlighting the diversity in vulnerability profiles. The uniqueness of opcode features stems from their direct correlation with low-level operations in the source code, varying significantly with the SC's logic and functionality. Each opcode represents a specific action within the EVM, so different vulnerabilities manifest through unique sequences or frequencies of these opcodes. For instance, vulnerabilities affecting transaction order use opcodes related to state changes more than arithmetic operations.

Specifically, the 'STOP' instruction count aligns with vulnerabilities like Mishandled Exception, Transaction Order Dependence, and Gas Exception but differs significantly in cases like Weak Access Mod and Denial of Service.

The sixth category of features is Solidity, as shown in Fig. 7f, which focuses on attributes specific to Solidity. BCCC-SCsVulLyzer(V2.0.0) extracts eight features from this category. However, EGA identifies only one feature as pivotal: the number of "Call" functions. This feature is crucial because it directly influences a contract's interactions with external contracts and controls its execution context. It allows a contract to invoke another contract's function within the same execution context, potentially leading to security vulnerabilities if not managed properly.

Among the vulnerabilities, there are slight fluctuations in the plots for the "Call" function feature, particularly for vulnerabilities such as Transaction Order Dependence, Unsend Return, and Call to Unknown. The "Call" function introduces Transaction Order Dependence by allowing external contracts to be called within the same execution context, which can be manipulated to affect transaction order and lead to unexpected behaviors. Moreover, if the failure of the "Call" function is not properly handled, it can lead to Unsend Return vulnerabilities, where the contract proceeds under false assumptions. Additionally, because the "Call" function is low-level and does not check the target address, a malicious or incorrect address can result in unintended code execution or execution in an unknown context, increasing the risk of Call to Unknown vulnerabilities. These three vulnerability profiles, fundamentally based on the concept of the "Call" function, show a more concentrated and specific range for the number of "Call" function feature values compared to other vulnerabilities.

The final category is line features, as in Fig. 7g. BCCC-SCsVulLyzer(V2.0.0) extracts five attributes related to source code lines: total lines, code lines, comment lines, blank lines, and duplicated lines. However, EGA identifies that only the total, code, and duplicated lines are used in the profiling process, indicating their significance in distinguishing and understanding source code characteristics. These features are defined as follows:

- **Number of Total Lines:** Provides an overview of the size and scale of the source code, offering an immediate sense of the contract's complexity.
- **Number of Code Lines:** Focuses on actual lines of code (excluding comments and blank lines), giving a clearer view of the functional content and the density and complexity of the logic.
- **Number of Duplicated Lines:** Identifies redundancy within the code, which can indicate poor coding practices and potential maintenance issues, and hint at logical flaws or vulnerabilities from repeated code blocks.

For instance, the Re-entrancy profile uses several "duplicated lines" and "total lines" to address specific logical issues in its fallback mechanism. Mishandled Exception and Gas Exception combine these features with several "code lines" to better understand the contract's complexity. The rest of the vulnerabilities use all three features to create distinctive profiles. Notably, fluctuations in the violin plot for several "total lines" are more pronounced, especially in cases like Timestamp and Call to Unknown, underscoring its effectiveness as a valuable comparison feature.

In conclusion, the EGA effectively identifies diverse SCs vulnerabilities, revealing distinct patterns across each feature category. However, some categories exhibit similar profiling patterns for different vulnerabilities. This highlights the complexity of SCs vulnerabilities, showing that while they share characteristics within certain categories, they also exhibit unique traits that distinguish them.

6.2. Visualizing Generated Profiles Using Genes

This section introduces a profiling formula that can define each vulnerability, secure SCs, and visualize their profiling genes.

6.2.1. Visualization Parameters

This subsection 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 explains the selection of each parameter, illustrating their relevance and importance in accurately formulating the profiles.

- **Parameter 1: Global Weight**

The first parameter, global weight, is determined by three distinct feature selection algorithms identified in Section 3.3.2. As shown in Table 5, each feature has a unique score derived from the ensemble feature selection method. We use averaging to consolidate these values into a comprehensive score, resulting in seven distinct global features. Due to the variation in these values, we adopt z-score normalization to convert them into standardized weights. This standardization preserves the range and behavioral distinctions of the data points. Moreover, we apply the absolute

value to the z-scores to focus on the magnitude of deviation. This ensures all values are positive, maintaining the relative differences between each original data point. The z-score is calculated as:

$$z = \frac{.X * \mu/}{\sigma}$$

where X is a value from the dataset, μ is the mean, and σ is the standard deviation. 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 quantifies the discrepancy between two probability distributions using the Wasserstein distance, also known as the Earth Mover's Distance (EMD). This metric, rooted in optimal transport theory, measures the minimal "cost" to transform one distribution into another. This study defines the distribution as all samples associated with a specific vulnerability. Calculating the EDM between this vulnerability distribution and secure samples reveals the extent of the differences between them. Mathematically, for one-dimensional distributions, the EDM between cumulative distribution functions (CDFs) $F.x/$ and $G.x/$ is defined as:

$$W.F,G/ = \int_{*} F.x/ * G.x/\,dx$$

The calculation simplifies for empirical distributions derived from data samples into a discrete form. Given two ordered samples $X = \hat{}x_1, x_2, ..., x_n\text{‘}$ and $Y = \hat{}y_1, y_2, ..., y_n\text{‘}$, the empirical EDM is expressed as:

$$W.X,Y/ = \frac{1}{n} \sum_{i=1}^{n} x_i * y_i$$

Here, n represents the total number of observations, with x_i and y_i being the i-th smallest values in each sample.

In summary, the EDM parameter highlights and quantifies the cumulative absolute discrepancies between the corresponding elements of two ordered samples. This allows it to effectively capture the differences in their distributions. Consequently, EDM is an essential parameter for comparing two different categories of samples.

- **Parameter 3: Peaks Difference**

The third parameter of the formula is the peak difference, which represents the distinct characteristics of each distribution. This difference arises from the fluctuations and the number of highest and lowest points in their patterns. Similar to the second parameter, the peak difference reveals details about specific vulnerabilities, providing information on the differences between secure and vulnerable instances. A peak in a one-dimensional array $x[n]$, where n is the index, is defined as a point $x[p]$ that meets the condition:

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

for all $0 < h \leq$ prominence. The prominence indicates how much higher a point needs to be relative to its surroundings to qualify as a peak. This reflects the peak's prominence within the data and helps measure its distinctiveness in height and spatial position compared to others. Additionally, a threshold excludes insignificant peaks that do not sufficiently surpass their surroundings' height. A point $x[p]$ is 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. The distance parameter ensures minimal horizontal spacing between adjacent peaks, separating each detected peak distinctly.

In summary, the peak difference parameter analyzes the differences between secure and vulnerable contracts by comparing 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)**

The fourth parameter of the formula is local weight, which combines two metrics: EMD and Peaks Difference. This unified metric captures distributional shifts and the prominence of peaks in the data. By averaging these insights, we calculate the 'EMD weight' from distributional discrepancies and the 'Peaks Difference' from structural anomalies. The local weight provides a balanced perspective on each feature's impact on SC vulnerabilities, combining the precision of EMD with the insights of peak analysis. The average of these two weights is given by:

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

6.2.2. Visualization Genes

This section concludes the findings on formulating vulnerabilities and secure SCs, as presented in Table 14 and Fig. 8. As shown in Fig. 8, each feature related to a vulnerability has a specific weight. Further details are illustrated in Fig. 9, where it is evident that different vulnerabilities share similar feature values but vary in domain range. This observation suggests a potential relationship between different vulnerabilities.

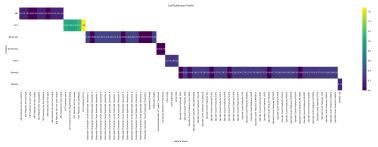
Additionally, Fig. 10 presents a heatmap that illustrates the correlations among all vulnerabilities and secure SCs. The heatmap clearly shows that most vulnerabilities are closely correlated, supporting the notion of their similarities. For example, the tight correlations among Timestamp, Transaction Order Dependence, and Unsent Return indicate that distinct profiling based on the analyzed features is challenging. Given these observed similarities, it is essential to establish a common vulnerability profile. Figures 11 and 12 present two distinct profiles through Histogram2DContour plots. These profiles visually represent common patterns in both secure and vulnerable SC behaviors.

The secure profile in Figure 12 shows a 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, corresponding to the 'opcode' category (as shown in Table 14). The profile's predominantly horizontal spread indicates greater variability along the X-axis than the Y-axis, suggesting that 'AST_nodetype' (from Table 14) significantly influences secure behavior. Additionally, a less dense area around (20, 2) highlights the presence of atypical secure behaviors associated with the 'bytecode' category.

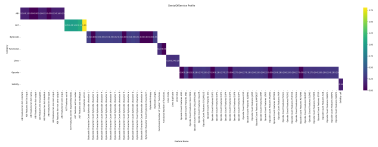
Conversely, the vulnerabilities profile in Figure 11 exhibits a significant central concentration, ranging from coordinates (20, 0.5) to (60, 0.5). This clustering correlates with categories such as bytecode, opcode, functional, and lines, as detailed in Table 14. However, this profile is more spread out, suggesting a broader range of behaviors linked to different vulnerabilities. Like the secure profile, it extends 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, indicating a more defined range of secure behaviors. In contrast, the vulnerabilities profile is broader, highlighting a wider range of potential vulnerabilities.

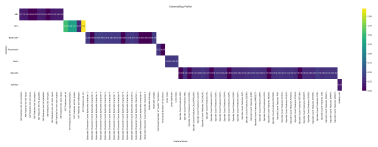
This section presents a systematic method for mathematically profiling each vulnerability by aggregating four proposed parameters. The method integrates category and local weights through a two-pronged approach, precisely valuing each feature's relevance to different vulnerabilities. Additionally, the section examines the behavioral patterns of secure and vulnerable SCs using detailed heatmap plots and comparative analyses. It demonstrates that similarities in code structure, arithmetic issues, and exploitation methods across various vulnerabilities result in similar profiling for each vulnerability. Therefore, creating a common profile for vulnerable and secure SCs provides a comprehensive approach to behavioral profiling.



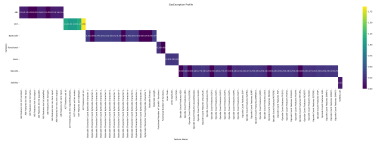
(a) Individual Heatmap Behavior Profile: Call to Unknown



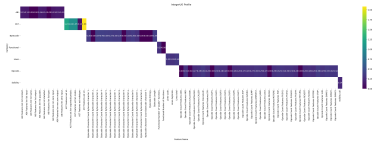
(b) Individual Heatmap Behavior Profile: Denial of Service



(c) Individual Heatmap Behavior Profile: External Bug

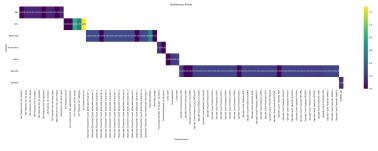


(d) Individual Heatmap Behavior Profile: Gas Exception

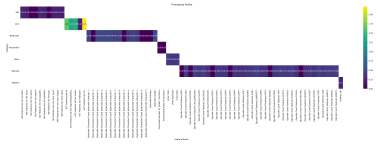


(e) Individual Heatmap Behavior Profile: Integer Underflow/Overflow

(f) Individual Heatmap Behavior Profile: Mishandled Exception



(g) Individual Heatmap Behavior Profile: Re-entrancy

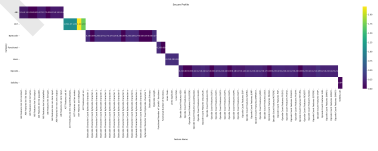


(h) Individual Heatmap Behavior Profile: Timestamp

(i) Individual Heatmap Behavior Profile: Transaction Order Dependence

(j) Individual Heatmap Behavior Profile: Unused Return

(k) Individual Heatmap Behavior Profile: Weak Access Mod



(l) Individual Heatmap Behavior Profile: Secure

Figure 8: Individual Heatmap Behavior Profile

7. Conclusion and Future Works

With the advent of blockchain networks, traditional contracts have transitioned to SCs, which are essential for maintaining and upholding trust within these systems. Regarding the security of SCs, current analysis methods can be categorized into traditional and NNs. Traditional methods like 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 created a new analyzer named BCCC-SCsVulLyzer(V2.0.0), which leverages GAs explicitly for profiling SC vulnerabilities. Additionally, we have compiled an extensive benchmark dataset of 111,897 Solidity source code

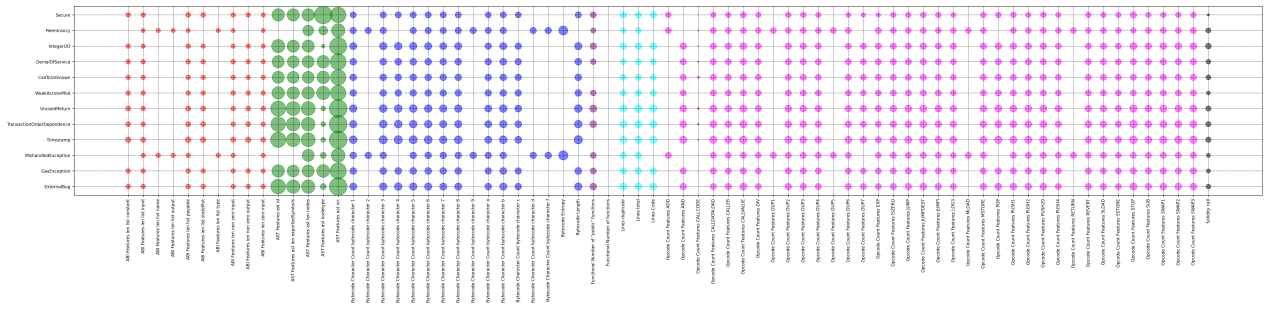


Figure 9: Comparative Profiling Visualization: The colors assigned to different categories: Red (Represents the ABI category), Green (Designated for the AST category), Blue (Used for bytecode-related features), Purple (Highlights features in the Functional category), Cyan (Indicates features associated 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)

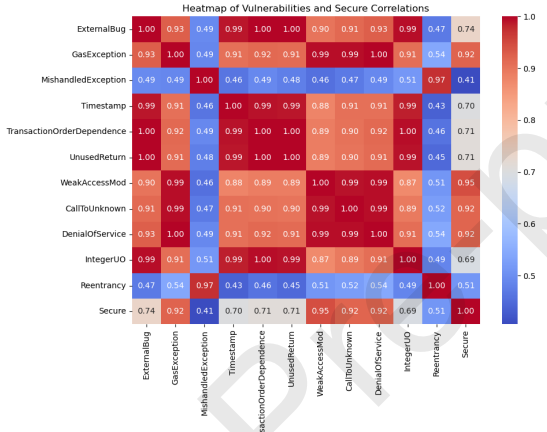


Figure 10: Heatmap of Vulnerabilities and Secure Correlations

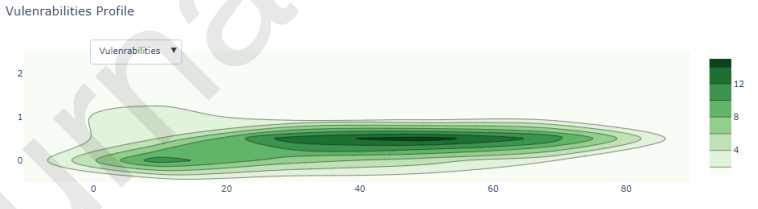


Figure 11: Common Behaviour Profile: Vulnerable

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 External Bugs, Gas Exceptions, Mishandled Exceptions, Timestamps, Transaction Order Dependence, Unused Returns, Weak Access Mod, Call to Unknown, Denial of Service, Integer Underflow/Overflow, and Re-entrancy. 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. Our model robustly identifies vulnerabilities with an accuracy ranging from 0.78 to 0.97. The precision in predicting specific types of

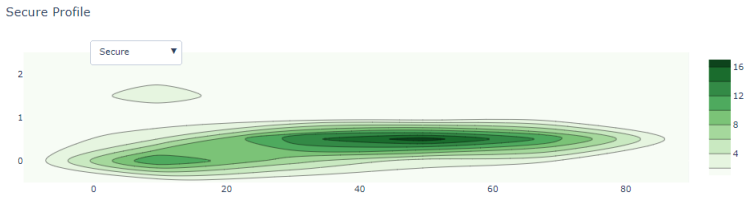


Figure 12: Common Behaviour Profile: Secure

vulnerabilities, ranging between 0.79 and 0.88, reflects its accuracy in correctly identifying true instances of each vulnerability type. To broaden the discussion, it's important to consider how the proposed approach could be adapted or extended to other blockchain platforms and smart contract languages. While this work primarily focuses on the Ethereum platform and the Solidity programming language, the steps outlined in Section 3 are adaptable to any blockchain platform and smart contract programming language.

In practical applications, our model can quickly analyze the security features of smart contracts, enabling immediate assessment and tagging of vulnerabilities. This real-time solution is critical in identifying potential threats within the blockchain ecosystem. Any platform executing Ethereum smart contracts can integrate this model into its back-end development structure to evaluate contracts before execution. For example, existing solutions like Mythril [44], Slither [22], SCVHunter [146], and AME [144] and which are currently used in the industry, could adopt this proposed solution as an alternative to their current methods for detecting and identifying E11-type vulnerabilities.

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 vulnerabilities could enhance the profiler's versatility and practicality, enabling it to address a wider range of threats. Developing a new system capable of detecting unknown vulnerabilities is a key area of future work. 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.

Acknowledgements

The authors acknowledge the Canada Research Chair - Tier II (#CRC-2021-00340) and Natural Sciences and Engineering Research Council grant from Canada—NSERC (#RGPIN-2020-04701)—to Arash Habibi Lashkari.

References

[1] V. Buterin, et al., A next-generation smart contract and decentralized application platform, white paper 3 (37) (2014) 2–1.
[2] L. Zhang, W. Chen, W. Wang, Z. Jin, C. Zhao, Z. Cai, H. Chen, Cbgru: A detection method of smart contract vulnerability based on a hybrid model, Sensors 22 (9) (2022) 3577.
[3] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, B. Roscoe, Reguard: finding reentrancy bugs in smart contracts, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2018, pp. 65–68.
[4] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, Y. Wang, Assbert: Active and semi-supervised bert for smart contract vulnerability detection, Journal of Information Security and Applications 73 (2023) 103423.
[5] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, A. Dinaburg, Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 1186–1189.
[6] S. Mohajerani, W. Ahrendt, M. Fabian, Modeling and security verification of state-based smart contracts, IFAC-PapersOnLine 55 (28) (2022) 356–362.
[7] M. Almakhour, L. Sliman, A. E. Samhat, A. Mellouk, A formal verification approach for composite smart contracts security using fsm, Journal of King Saud University-Computer and Information Sciences 35 (1) (2023) 70–86.
[8] M. Ndiaye, T. A. Diallo, K. Konate, Adefguard: Anomaly detection framework based on ethereum smart contracts behaviours, Blockchain: Research and Applications (2023) 100148.

[9] X. Ren, Y. Wu, J. Li, D. Hao, M. Alam, Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network, Computers and Electrical Engineering 109 (2023) 108766.
[10] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, Y. Alexandrov, Smartcheck: Static analysis of ethereum smart contracts, in: Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain, 2018, pp. 9–16.
[11] Z. Liu, M. Jiang, S. Zhang, J. Zhang, Y. Liu, A smart contract vulnerability detection mechanism based on deep learning and expert rules, IEEE Access (2023).
[12] I. Grishchenko, M. Maffei, C. Schneidewind, Ethertrust: Sound static analysis of ethereum bytecode, Technische Universität Wien, Tech. Rep (2018) 1–41.
[13] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, A. Hobor, Finding the greedy, prodigal, and suicidal contracts at scale, in: Proceedings of the 34th annual computer security applications conference, 2018, pp. 653–663.
[14] I. M. Ali, N. Lasla, M. M. Abdallah, A. Erbad, Srp: An efficient runtime protection framework for blockchain-based smart contracts, Journal of Network and Computer Applications 216 (2023) 103658.
[15] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, Y. Smaragdakis, Madmax: Surviving out-of-gas conditions in ethereum smart contracts, Proceedings of the ACM on Programming Languages 2 (OOPSLA) (2018) 1–27.
[16] T. Chen, X. Li, X. Luo, X. Zhang, Under-optimized smart contracts devour your money, in: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), IEEE, 2017, pp. 442–446.
[17] S. Wang, C. Zhang, Z. Su, Detecting nondeterministic payment bugs in ethereum smart contracts, Proceedings of the ACM on Programming Languages 3 (OOPSLA) (2019) 1–29.
[18] M. Rodler, W. Li, G. O. Karame, L. Davi, Sereum: Protecting existing smart contracts against re-entrancy attacks, arXiv preprint arXiv:1812.05934 (2018).
[19] E. Albert, P. Gordillo, B. Livshits, A. Rubio, I. Sergey, Ethir: A framework for high-level analysis of ethereum bytecode, in: International symposium on automated technology for verification and analysis, Springer, 2018, pp. 513–520.
[20] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, T. Chen, Defectchecker: Automated smart contract defect detection by analyzing evm bytecode, IEEE Transactions on Software Engineering 48 (7) (2021) 2189–2207.
[21] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, H. Kurihara, Security assurance for smart contract, in: 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), IEEE, 2018, pp. 1–5.
[22] J. Feist, G. Grieco, A. Groce, Slither: a static analysis framework for smart contracts, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), IEEE, 2019, pp. 8–15.
[23] C. F. Torres, J. Schütte, R. State, Osiris: Hunting for integer bugs in ethereum smart contracts, in: Proceedings of the 34th annual computer security applications conference, 2018, pp. 664–676.
[24] D. Yuan, X. Wang, Y. Li, T. Zhang, Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding, Journal of Systems and Software 202 (2023) 111699.
[25] S. HajiHosseinKhani, A. H. Lashkari, A. M. Oskui, Unveiling vulnerable smart contracts: Toward profiling vulnerable smart contracts using genetic algorithm and generating benchmark dataset, Blockchain: Research and Applications (2023) 100171.
[26] S. Kalra, S. Goel, M. Dhawan, S. Sharma, Zeus: analyzing safety of smart contracts., in: Ndss, 2018, pp. 1–12.
[27] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, B. Scholz, Vandal: A scalable security analysis framework for smart contracts, arXiv preprint arXiv:1809.03981 (2018).
[28] C. Langensiepen, A. Lotfi, 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), IEEE, 2014, pp. 873–880.
[29] A. Fernández-Isabel, P. Peixoto, I. M. de Diego, C. Conde, E. Cabello, Combining dynamic finite state machines and text-based similarities to represent human behavior, Engineering Applications of Artificial Intelligence 85 (2019) 504–516.
[30] A. Guillén, Y. Gutiérrez, R. Muñoz, Natural language processing technologies for document profiling (2017) 284–290.
[31] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 254–269.
[32] R. Norvill, B. B. F. Pontiveros, R. State, A. Cullen, Visual emulation for ethereum's virtual machine, in: NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium, IEEE, 2018, pp. 1–4.
[33] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, M. Bailey, Erays: reverse engineering ethereum's opaque smart contracts, in: 27th USENIX security symposium (USENIX Security 18), 2018, pp. 1371–1385.
[34] P. Praitheeshan, L. Pan, X. Zheng, A. Jolfaei, R. Doss, Solguard: Preventing external call issues in smart contract-based multi-agent robotic systems, Information Sciences 579 (2021) 150–166.
[35] J. Krupp, C. Rossow, ˆteEther‘: Gnawing at ethereum to automatically exploit smart contracts, in: 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 1317–1333.
[36] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, M. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, 2018, pp. 67–82.
[37] B. Jiang, Y. Liu, W. K. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 259–269.
[38] S. Ji, J. Wu, J. Qiu, J. Dong, Effuzz: Efficient fuzzing by directed search for smart contracts, Information and Software Technology 159 (2023) 107213.
[39] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, Q. T. Minh, sfuzz: An efficient adaptive fuzzer for solidity smart contracts, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 778–788.
[40] C. F. Torres, M. Steichen, et al., The art of the scam: Demystifying honeypots in ethereum smart contracts, in: 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 1591–1607.

[41] X. Wu, X. Du, Q. Yang, A. Liu, N. Wang, W. Wang, Taintguard: Preventing implicit privilege leakage in smart contract based on taint tracking at abstract syntax tree level, Journal of Systems Architecture (2023) 102925.
[42] M. Pasqua, A. Benini, F. Contro, M. Crosara, M. Dalla Preda, M. Ceccato, Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode, Journal of Systems and Software 200 (2023) 111653.
[43] S. Driessen, D. Di Nucci, D. Tamburri, W.-J. van den Heuvel, Solar: Automated test-suite generation for solidity smart contracts, Science of Computer Programming (2023) 103036.
[44] N. Sharma, A survey of mythril, a smart contract security analysis tool for evm bytecode (2022).
[45] J. Li, Z. Zhao, Z. Su, W. Meng, Gas-expensive patterns detection to optimize smart contracts, Applied Soft Computing (2023) 110542.
[46] D. Chen, L. Feng, Y. Fan, S. Shang, Z. Wei, Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention, Journal of Systems and Software 202 (2023) 111705.
[47] K. L. Narayana, K. Sathiyamurthy, Automation and smart materials in detecting smart contracts vulnerabilities in blockchain using deep learning, Materials Today: Proceedings (2021).
[48] Z. Tian, B. Tian, J. Lv, Y. Chen, L. Chen, Enhancing vulnerability detection via ast decomposition and neural sub-tree encoding, Expert Systems with Applications (2023) 121865.
[49] X. Xie, H. Wang, Z. Jian, Y. Fang, Z. Wang, T. Li, Block-gram: Mining knowledgeable features for efficiently smart contract vulnerability detection, Digital Communications and Networks (2023).
[50] N. Ashizawa, N. Yanai, J. P. Cruz, S. Okamura, Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts, in: Proceedings of the 3rd ACM international symposium on blockchain and secure critical infrastructure, 2021, pp. 47–59.
[51] L. Wang, H. Cheng, Z. Zheng, A. Yang, M. Xu, Temporal transaction information-aware ponzi scheme detection for ethereum smart contracts, Engineering Applications of Artificial Intelligence 126 (2023) 107022.
[52] K. Zhou, J. Huang, H. Han, B. Gong, A. Xiong, W. Wang, Q. Wu, Smart contracts vulnerability detection model based on adversarial multi-task learning, Journal of Information Security and Applications 77 (2023) 103555.
[53] H. Liu, Y. Fan, L. Feng, Z. Wei, Vulnerable smart contract function locating based on multi-relational nested graph convolutional network, Journal of Systems and Software (2023) 111775.
[54] J. Cai, B. Li, J. Zhang, X. Sun, B. Chen, Combine sliced joint graph with graph neural networks for smart contract vulnerability detection, Journal of Systems and Software 195 (2023) 111550.
[55] H. Zhang, W. Zhang, Y. Feng, Y. Liu, Svscanner: Detecting smart contract vulnerabilities via deep semantic extraction, Journal of Information Security and Applications 75 (2023) 103484.
[56] L. Zhang, J. Wang, W. Wang, Z. Jin, Y. Su, H. Chen, Smart contract vulnerability detection combined with multi-objective detection, Computer Networks 217 (2022) 109289.
[57] C. W. Fraser, D. R. Hanson, A Retargetable C Compiler: Design and Implementation, Addison-Wesley Professional, 1995.
[58] B. Stroustrup, The Design and Evolution of C++, Addison-Wesley Professional, 1994.
[59] D. N. Rassokhin, D. K. Agrafiotis, Kolmogorov-smirnov statistic and its application in library design, Journal of Molecular Graphics and Modelling 18 (4-5) (2000) 368–382.
[60] S. Yu, H. Zhang, T. Li, Fuzzy profiling: A framework for discovering unknown malware, in: Proceedings of the 9th International Conference on Intelligent Systems Design and Applications (ISDA), 2009, pp. 838–843.
URL https://doi.org/10.1109/ISDA.2009.214
[61] D. Grune, C. J. H. Jacobs, Parsing Techniques: A Practical Guide, Springer Science & Business Media, 2007.
[62] X. Hu, Y. Zhuang, S.-W. Lin, F. Zhang, S. Kan, Z. Cao, A security type verifier for smart contracts, Computers & Security 108 (2021) 102343.
[63] A. V. Gorchakov, L. A. Demidova, P. N. Sovietov, Analysis of program representations based on abstract syntax trees and higher-order markov chains for source code classification task, Future Internet 15 (9) (2023) 314.
[64] W. E. Boebert, The system v application binary interface, UNIX Review 10 (11) (1992) 6–18.
[65] U. S. Laboratories, System v application binary interface - intel386 architecture processor supplement, http://www.sco.com/developers/gabi/1995-07-27/contents.html (1995).
[66] H. Yin, D. X. Song, Reverse engineering of binary application program interfaces, in: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS), 2006, pp. 187–196.
URL https://doi.org/10.1145/1180405.1180434
[67] N. F. Samreen, M. H. Alalfi, Reentrancy vulnerability identification in ethereum smart contracts, in: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), IEEE, 2020, pp. 22–29.
[68] X. Sun, X. Lin, Z. Liao, An abi-based classification approach for ethereum smart contracts, in: 2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), IEEE, 2021, pp. 99–104.
[69] E. Poll, M. Steffen, A formal definition of the java bytecode language, Science of Computer Programming 47 (3) (2003) 247–267.
URL https://doi.org/10.1016/S0167-6423(02)00153-0
[70] C. Click, M. Paleczny, C. Verbrugge, Static single assignment of java bytecode, in: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2002, pp. 222–234.
URL https://doi.org/10.1145/503272.503251
[71] R. Gupta, M. M. Patel, A. Shukla, S. Tanwar, Deep learning-based malicious smart contract detection scheme for internet of things environment, Computers & Electrical Engineering 97 (2022) 107583.
[72] A. L. Vivar, A. L. S. Orozco, L. J. G. Villalba, A security framework for ethereum smart contracts, Computer Communications 172 (2021) 119–129.

[73] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, F. Koushanfar, Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning., in: NDSS, 2023.
[74] J. Cocke, J. T. Schwartz, Instruction selection using microcode-like global operations, Communications of the ACM 23 (3) (1980) 162–167. URL https://doi.org/10.1145/358896.358899
[75] B. Horn, J. Sakarovitch, M. Soria, Opcode patterns and regular expressions, Theoretical Computer Science 100 (2) (1992) 273–292. URL https://doi.org/10.1016/0304-3975(92)90222-C
[76] S. Govindan, P. Ranganathan, Improving instruction cache performance by opcode morphing, in: Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA), 2001, pp. 168–179. URL https://doi.org/10.1145/383982.384003
[77] H. Huang, L. Guo, L. Zhao, H. Wang, C. Xu, S. Jiang, Effective combining source code and opcode for accurate vulnerability detection of smart contracts in edge ai systems, Applied Soft Computing (2024) 111556.
[78] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS), 2017, pp. 6000–6010. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
[79] J. Devlin, M.-W. Chang, K. Lee, 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), 2019, pp. 4171–4186. URL https://doi.org/10.18653/v1/N19-1423
[80] P. Qian, Z. Liu, Q. He, R. Zimmermann, X. Wang, Towards automated reentrancy detection for smart contracts based on sequential models, IEEE Access 8 (2020) 19685–19695.
[81] D. Binkley, Source code analysis: A road map, Future of Software Engineering (FOSE'07) (2007) 104–119.
[82] M. Gupta, J. Rees, A. Chaturvedi, J. Chi, Matching information security vulnerabilities to organizational security profiles: a genetic algorithm approach, Decision Support Systems 41 (3) (2006) 592–603.
[83] A. H. Hamamoto, L. F. Carvalho, L. D. H. Sampaio, T. Abrão, M. L. Proença Jr, Network anomaly detection system using genetic algorithm and fuzzy logic, Expert Systems with Applications 92 (2018) 390–402.
[84] L. Liu, W.-T. Tsai, M. Z. A. Bhuiyan, H. Peng, M. Liu, Blockchain-enabled fraud discovery through abnormal smart contract detection on ethereum, Future Generation Computer Systems 128 (2022) 158–166.
[85] Q. Gu, Z. Li, J. Han, Generalized fisher score for feature selection, arXiv preprint arXiv:1202.3725 (2012).
[86] L. Sun, T. Wang, W. Ding, J. Xu, Y. Lin, Feature selection using fisher score and multilabel neighborhood rough sets for multilabel classification, Information Sciences 578 (2021) 887–912.
[87] D. Aksu, S. Üstebay, M. A. Aydin, 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, Springer, 2018, pp. 141–149.
[88] B. Singh, J. S. Sankhwar, 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), IEEE, 2014, pp. 1–6.
[89] B. Azhagusundari, A. S. Thanamani, et al., Feature selection based on information gain, International Journal of Innovative Technology and Exploring Engineering (IJITEE) 2 (2) (2013) 18–21.
[90] 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, IEEE, 2012, pp. 355–358.
[91] E. O. Omuya, G. O. Okeyo, M. W. Kimwele, Feature selection for classification using principal component analysis and information gain, Expert Systems with Applications 174 (2021) 114765.
[92] C. Shang, M. Li, S. Feng, Q. Jiang, J. Fan, Feature selection via maximizing global information gain for text classification, Knowledge-Based Systems 54 (2013) 298–309.
[93] 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, IEEE, 2018, pp. 229–233.
[94] S. Ray, K. Alshouiliy, A. Roy, A. AlGhamdi, D. P. Agrawal, Chi-squared based feature selection for stroke prediction using azureml, in: 2020 Intermountain Engineering, Technology and Computing (IETC), IEEE, 2020, pp. 1–6.
[95] A.-M. Bidgoli, M. N. Parsa, A hybrid feature selection by resampling, chi squared and consistency evaluation techniques, World Academy of Science, Engineering and Technology 68 (2012) 276–285.
[96] I. S. Thaseen, 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 29 (4) (2017) 462–472.
[97] R. Gupta, N. K. Gupta, M. L. Soffa, A case for profiling-oriented software engineering, IEEE Software 13 (1) (1996) 22–31. URL https://doi.org/10.1109/52.485327
[98] B. J. Cox, The execution time measurement and profiling of a program, Communications of the ACM 15 (10) (1972) 801–805.
[99] B. Zadrozny, 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), 2002, pp. 627–632. URL https://doi.org/10.1145/775047.775141
[100] X. Zhang, F. Qiu, 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 76 (2019) 26–39.
[101] R. Zuech, A. Goodrum, Fuzzy profiling for the detection of anomalous program behavior, Computers & Security 24 (2) (2005) 123–139. URL https://doi.org/10.1016/j.cose.2004.11.001
[102] O. Alhabashneh, R. Iqbal, F. Doctor, S. Amin, Adaptive information retrieval system based on fuzzy profiling, in: 2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), IEEE, 2015, pp. 1–8.

[103] D. Xu, H. Wang, K. Su, Intelligent student profiling with fuzzy models, in: Proceedings of the 35th Annual Hawaii international conference on system sciences, IEEE, 2002, pp. 8–pp.
[104] C. Mencar, M. A. Torsello, D. Dell'Agnello, G. Castellano, C. Castiello, Modeling user preferences through adaptive fuzzy profiles, in: 2009 Ninth International Conference on Intelligent Systems Design and Applications, IEEE, 2009, pp. 1031–1036.
[105] J. E. Dickerson, 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), IEEE, 2000, pp. 301–306.
[106] E. Mezura-Montes, J. Velázquez-Reyes, C. A. Coello Coello, Profiling-based adaptive genetic algorithm, IEEE Transactions on Evolutionary Computation 13 (5) (2009) 1053–1070.
URL https://doi.org/10.1109/TEVC.2008.2011727
[107] G. Kendall, E. K. Burke, M. Gendreau, B. Ombuki-Berman, B. McCollum, E. Özcan, R. Qu, On the use of profiling techniques in genetic algorithm-based hyper-heuristics, Journal of the Operational Research Society 58 (6) (2007) 708–718.
URL https://doi.org/10.1057/palgrave.jors.2602257
[108] P. Asokan, R. Saravanan, 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 21 (1) (2003) 1–9.
[109] P. A. A. Resende, A. C. Drummond, Adaptive anomaly-based intrusion detection system using genetic algorithm and profiling, Security and Privacy 1 (4) (2018) e36.
[110] S. Lal, C. Cascaval, J. Mars, P. Dey, H. Tang, Profiling machine learning workloads, in: Proceedings of the 2019 International Symposium on Code Generation and Optimization (CGO), 2019, pp. 267–279.
URL https://doi.org/10.1109/CGO.2019.8661173
[111] J. Wang, W. Huang, H. Zhang, X. Wu, Understanding the performance of tensorflow workloads on gpus, in: Proceedings of the 2020 IEEE International Symposium on Workload Characterization (IISWC), 2020, pp. 83–92.
URL https://doi.org/10.1109/IISWC49841.2020.00015
[112] A. Samajdar, V. Sridharan, M. Zinsmaier, Z. Pan, S. Shin, R. Gao, Q. Zhou, R. K. Gupta, Auto-profiling: A framework for profiling and optimization of ml workloads, in: Proceedings of the 2020 USENIX Annual Technical Conference (ATC), 2020, pp. 631–644.
URL https://www.usenix.org/conference/atc20/presentation/samajdar
[113] I. T. Haque, C. Assi, Profiling-based indoor localization schemes, IEEE Systems Journal 9 (1) (2013) 76–85.
[114] E. Tsalera, A. Papadakis, M. Samarakou, Monitoring, profiling and classification of urban environmental noise using sound characteristics and the knn algorithm, Energy Reports 6 (2020) 223–230.
[115] P. Nagaraj, K. Saiteja, K. K. Ram, K. M. Kanta, S. K. Aditya, 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), IEEE, 2022, pp. 479–484.
[116] R. Bayot, T. Gonçalves, Multilingual author profiling using word embedding averages and svms, in: 2016 10th International Conference on Software, Knowledge, Information Management & Applications (SKIMA), IEEE, 2016, pp. 382–386.
[117] P. J. Batterham, H. Christensen, Longitudinal risk profiling for suicidal thoughts and behaviours in a community cohort using decision trees, Journal of affective disorders 142 (1-3) (2012) 306–314.
[118] P. Duchessi, E. J. Lauría, Decision tree models for profiling ski resorts' promotional and advertising strategies and the impact on sales, Expert Systems with Applications 40 (15) (2013) 5822–5829.
[119] D. S. Rana, S. C. Dimri, Machine learning enables malware detection and classification techniques, in: 2024 IEEE International Conference on Computing, Power and Communication Technologies (IC2PCT), Vol. 5, IEEE, 2024, pp. 1215–1221.
[120] A. Cura, H. Küçük, E. Ergen, İ. B. Öksüzoğlu, Driver profiling using long short term memory (lstm) and convolutional neural network (cnn) methods, IEEE Transactions on Intelligent Transportation Systems 22 (10) (2020) 6572–6582.
[121] S. H. Hawley, B. Colburn, S. I. Mimilakis, Signaltrain: Profiling audio compressors with deep neural networks, arXiv preprint arXiv:1905.11928 (2019).
[122] K. C. Baumgartner, S. Ferrari, C. G. Salfati, Bayesian network modeling of offender behavior for criminal profiling, in: Proceedings of the 44th IEEE Conference on Decision and Control, IEEE, 2005, pp. 2702–2709.
[123] T. Xiang, S. Gong, Video behavior profiling for anomaly detection, IEEE transactions on pattern analysis and machine intelligence 30 (5) (2008) 893–908.
[124] Y.-J. Zheng, W.-G. Sheng, X.-M. Sun, S.-Y. Chen, Airline passenger profiling based on fuzzy deep machine learning, IEEE transactions on neural networks and learning systems 28 (12) (2016) 2911–2923.
[125] M. Sedaghati, C. Jutten, A. Abdi, Profiling deep neural networks: Sparsity and complexity analysis, IEEE Signal Processing Letters 27 (2020) 220–224.
URL https://doi.org/10.1109/LSP.2020.2966893
[126] B. W.-G. M. Anrig, Bernhard, The role of algorithms in profiling (2008).
[127] S. K. Smmarwar, G. P. Gupta, S. Kumar, Android malware detection and identification frameworks by leveraging the machine and deep learning techniques: A comprehensive review, Telematics and Informatics Reports (2024) 100130.
[128] F. Fkih, D. Rhouma, Text mining-based author profiling: Literature review, trends and challenges, in: International Conference on Hybrid Intelligent Systems, Springer, 2022, pp. 423–431.
[129] K. Kavuri, M. Kavitha, A term weight measure based approach for author profiling, in: 2022 International Conference on Electronic Systems and Intelligent Computing (ICESIC), IEEE, 2022, pp. 275–280.
[130] T. Karagiannis, K. Papagiannaki, N. Taft, 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, Springer, 2007, pp. 186–196.

[131] W. Chen, Y. Gu, Z. Ren, X. He, H. Xie, T. Guo, D. Yin, Y. Zhang, Semi-supervised user profiling with heterogeneous graph attention networks., in: IJCAI, Vol. 19, 2019, pp. 2116–2122.
[132] S. Xue, L. Zhang, A. Li, X.-Y. Li, C. Ruan, W. Huang, Appdna: App behavior profiling via graph-based deep learning, in: IEEE INFOCOM 2018-IEEE Conference on Computer Communications, IEEE, 2018, pp. 1475–1483.
[133] R. Labadie-Tamayo, D. Castro-Castro, Graph-based profile condensation for users profiling (2022).
[134] K. Han, J. Park, 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), IEEE, 2015, pp. 225–231.
[135] H. Asai, K. Fukuda, P. Abry, P. Borgnat, H. Esaki, Network application profiling with traffic causality graphs, International Journal of Network Management 24 (4) (2014) 289–303.
[136] S. Munir, S. I. Jami, S. Wasi, Knowledge graph based semantic modeling for profiling in industry 4.0, International Journal on Information Technologies & Security 12 (1) (2020) 37–50.
[137] M. Daoud, L. Tamine, 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, Springer, 2010, pp. 171–182.
[138] M. Sedighizadeh, A. Rezazadeh, Using genetic algorithm for distributed generation allocation to reduce losses and improve voltage profile, World Academy of Science, Engineering and Technology 37 (1) (2008) 251–256.
[139] W. Zou, V. V. Tolstikov, Probing genetic algorithms for feature selection in comprehensive metabolic profiling approach, Rapid Communications in Mass Spectrometry: An International Journal Devoted to the Rapid Dissemination of Up-to-the-Minute Research in Mass Spectrometry 22 (8) (2008) 1312–1324.
[140] L. Haldurai, T. Madhubala, R. Rajalakshmi, A study on genetic algorithm and its applications, Int. J. Comput. Sci. Eng 4 (10) (2016) 139–143.
[141] N. C. Evans, D. L. Shealy, Design and optimization of an irradiance profile-shaping system with a genetic algorithm method, Applied Optics 37 (22) (1998) 5216–5221.
[142] R. Paulavičius, L. Stripinis, S. Sutavičiūtė, D. Kočegarov, E. Filatovas, A novel greedy genetic algorithm-based personalized travel recommendation system, Expert Systems with Applications 230 (2023) 120580.
[143] S. J. Aquilina, F. Casino, M. Vella, J. Ellul, C. Patsakis, Etherclue: Digital investigation of attacks on ethereum smart contracts, Blockchain: Research and Applications 2 (4) (2021) 100028.
[144] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, X. Wang, Combining graph neural networks with expert knowledge for smart contract vulnerability detection, IEEE Transactions on Knowledge and Data Engineering (2021).
[145] M. Ding, P. Li, S. Li, H. Zhang, Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection, in: Evaluation and Assessment in Software Engineering, 2021, pp. 321–328.
[146] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, 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, 2024, pp. 1–13.

Table 14
Total Weights

		Total weight										
Group name	Feature name	Gas Exception	Mishandled Exception	Timestamp	Transaction Order Dependence	UnusedReturn	Weak Access Mod	Call to Unknown	Denial of Service	Integer Underflow/Overflow	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

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The author is an Editorial Board Member/Editor-in-Chief/Associate Editor/Guest Editor for *[Blockchain: Research and Applications]* and was not involved in the editorial review or the decision to publish this article.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Sepideh HajiHosseinKhani: Conducted the primary methodology development of using a genetic algorithm for profiling malicious smart contracts. This involved designing, implementing, and troubleshooting the algorithm and its components. Ms. HajiHosseinKhani also took the lead in the analysis and interpretation of data by creating a new dataset and benchmarking the developed algorithm against previous non-machine learning-based and machine learning-based models. She also participated in the writing and editing of the manuscript, ensuring that the descriptions of the methodology and results were accurate and clearly communicated.

Arash Habibi Lashkari: Dr. Lashkari secured the funding that made the research possible. He provided oversight and supervision throughout the course of the study. His expertise in the field guided the direction of the research and the methodology used. Dr. Lashkari was integral in the process of analysis and interpretation.

Ali Mizani Oskui: Ali Mizani Oskui has made significant contributions as an industry partner.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The author is an Editorial Board Member/Editor-in-Chief/Associate Editor/Guest Editor for *[Blockchain: Research and Applications]* and was not involved in the editorial review or the decision to publish this article.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Sepideh HajiHosseinKhani: Conducted the primary methodology development of using a genetic algorithm for profiling malicious smart contracts. This involved designing, implementing, and troubleshooting the algorithm and its components. Ms. HajiHosseinKhani also took the lead in the analysis and interpretation of data by creating a new dataset and benchmarking the developed algorithm against previous non-machine learning-based and machine learning-based models. She also participated in the writing and editing of the manuscript, ensuring that the descriptions of the methodology and results were accurate and clearly communicated.

Arash Habibi Lashkari: Dr. Lashkari secured the funding that made the research possible. He provided oversight and supervision throughout the course of the study. His expertise in the field guided the direction of the research and the methodology used. Dr. Lashkari was integral in the process of analysis and interpretation.

Ali Mizani Oskui: Ali Mizani Oskui has made significant contributions as an industry partner.