A SYSTEMATIC EVALUATION FRAMEWORK FOR SMART CONTRACT SECURITY ANALYZERS: METHODS, METRICS, AND FRAMEWORK

Niosha Hejazi

A THESIS SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF ARTS

GRADUATE PROGRAM IN INFORMATION TECHNOLOGY

YORK UNIVERSITY
TORONTO, ONTARIO

August 2025

Abstract

Smart contracts automate agreements in blockchain systems but their immutable nature makes them vulnerable to permanent flaws once deployed. This thesis evaluates 256 smart contract vulnerability detection tools developed between 2018 and 2024, including approaches such as fuzzing, symbolic execution, formal verification, and artificial intelligence–based analysis. Tools were classified by detection strategy (static, dynamic, hybrid), domain (academic or industry), and scope. The evaluation involved a theoretical review of architecture, usability, and documentation, alongside an empirical assessment of accuracy, speed, and false positive rates. Findings show that while certain tools excel in specific areas, none achieve balanced performance or comprehensive coverage. To address these gaps, a modular six-layer evaluation framework is introduced, defining functional areas such as code analysis, coverage, integration, and user experience. The framework offers a benchmark for tool assessment and future development. Additionally, a graph-based detection model is proposed, demonstrating improved accuracy in both binary and multi-class settings.

Acknowledgement

I would like to sincerely thank my family and friends for their constant support and encouragement throughout my Master's journey. I am also truly grateful to my supervisor, Prof. Arash Habibi Lashkari, whose guidance and unwavering support have been essential to my academic growth and research efforts.

Contents

List of Tables

List of Figures

1 Introduction

The rapid expansion and widespread use of blockchain technology have made Smart Contracts (SCs) a key element in secure and automated digital transactions. SCs, often referred to as self-executing agreements, operate directly on blockchain networks, eliminating intermediaries and improving transparency. Their ability to simplify processes, build trust, and lower costs has made them vital to modern blockchain-based applications. As a result, these contracts are widely utilized in various domains, including finance, supply chain management, and healthcare.

However, the immutability and autonomy that make SCs appealing also introduce significant security challenges. Once deployed, SCs cannot be altered, meaning any vulnerabilities in their code could lead to irreversible losses. As a result, ensuring the security and reliability of SCs has become a critical focus for researchers and practitioners in the field. This research underscores the pressing need for robust vulnerability detection tools for smart contracts (SCs) to enhance the security and reliability of blockchain-based applications. Our study offers a thorough evaluation of current tools based on both theoretical insights and practical applications.

SCs vulnerabilities are complex and continually evolving, making them difficult to analyze and address effectively. This thesis responds to that challenge by conducting a thorough review and categorization of 256 tools, using a balanced approach that combines theoretical research with practical evaluation. This work required a significant investment of time and effort, with over 780 hours dedicated to theoretical analysis and more than 1300 hours spent on hands-on testing and validation. To ensure accuracy and relevance, the analysis focused on peer-reviewed publications, emphasizing developments up to the year 2024. The goal of this thesis is to provide a comprehensive overview of the current landscape of SCs vulnerability detection and to offer a reliable foundation for future research and improvements in this critical area.

This thesis begins by presenting the search methodology used to collect and screen literature on SCs vulnerability detection tools. Two distinct strategies are applied—one for survey papers and another for technical studies—ensuring a comprehensive and unbiased selection of high-quality research. Clear inclusion and exclusion criteria are defined to focus on relevant work. In the following section, eighteen survey papers are analyzed and grouped into experimental, theoretical, and comparative categories. This review identifies several gaps in existing research, such as limited evaluation depth and insufficient attention to specific aspects of vulnerability detection. These observations help clarify areas where further development is needed and provide direction for this thesis.

The subsequent analysis examines 256 available tools, categorized by methodology, including fuzzing, machine learning, symbolic execution, and formal verification. These tools are classified as either detection or identification tools to clearly distinguish their roles in SCs security. Each methodology is examined in terms of the types of vulnerabilities it addresses and the techniques it employs. The Evaluation Criteria, Analysis,

and Discussion section further assesses these tools by dividing them into academic and industry categories and classifying them as static, dynamic, or hybrid. This structured evaluation offers a detailed understanding of tool capabilities and limitations, supporting both academic research and real-world applications.

A key part of this thesis is the Practical Analysis of SCs Vulnerability Tools section, where selected tools from both academia and industry are evaluated in detail using the most up-to-date dataset. This analysis emphasizes the practical relevance of the study by focusing on seven tools chosen for their demonstrated effectiveness in detecting vulnerabilities. The substantial time and effort invested in this practical component reflect a strong commitment to advancing the understanding and evaluation of SCs security tools.

Building on insights from both theoretical and hands-on analysis, this thesis introduces a structured six-layer framework for evaluating SCs vulnerability detection tools. The framework is developed by analyzing the strengths of high-performing tools and organizing their capabilities into functional layers—such as code analysis, detection, monitoring, integration, and usability. Each layer is composed of specific components that reflect practical performance expectations. These components are assessed using a binary scoring method, combined with weighted contributions from each layer. This structure enables consistent and transparent evaluation of tools, while also revealing commonly missing or underdeveloped features in current solutions.

In summary, this thesis provides a comprehensive theoretical and practical analysis of SCs vulnerability detection tools, offering detailed insights into the current state of the field and establishing a strong foundation for future advancements. The main contributions of this work include: (1) a structured review and classification of existing survey and technical papers on SCs vulnerabilities, (2) an in-depth categorization and evaluation of 256 tools based on core methodologies such as fuzzing, machine learning, symbolic execution, and formal verification, (3) a systematic evaluation approach that addresses the distinct needs of academic and industry contexts by organizing tools into static, dynamic, and hybrid categories, (4) an extensive hands-on evaluation of selected tools to bridge theoretical understanding with real-world applicability, and (5) the introduction of a modular, six-layer evaluation framework that defines key components and scoring methods for assessing tool completeness and capability. (6) The development and evaluation of a custom graph-based detection model that outperforms existing tools in both binary and multi-class vulnerability classification tasks, using function-level call graphs and semantic embeddings.

The chapters that follow are organized to support these contributions. Chapter 2 details the literature search methodology and provides a critical review of existing surveys, identifying research gaps that motivate this study. Chapter 3 presents a detailed analysis of SCs vulnerability detection tools, organized by methodology and tool type, along with a comparative evaluation based on defined criteria. Chapter 4 introduces the six-layer evaluation framework, highlighting its structure, components, and application. Finally, Chapter 5 concludes the thesis with a summary of key findings, a discussion of current limitations in detection tools, and suggestions for future research directions.

2 Literature Review

2.1 Search Methodology

A structured search methodology was adopted in this thesis to ensure a comprehensive and unbiased review of the existing literature. Two distinct approaches were used to support this process: one targeting survey literature and the other focusing on technical papers. In both cases, peer-reviewed articles and academically published documents relevant to smart contracts (SCs) vulnerability analysis and detection were gathered from reputable academic databases. Each method used screening and exclusion criteria to ensure that only the most relevant and high-quality studies were included. The subsections below describe the specific steps taken to collect and refine the literature related to SCs detection tools.

2.1.1 Survey Literature Search Methodology

To collect survey papers focusing on SCs detection tools, a comprehensive database search was performed as the first phase of the review. This search strategy is built using a carefully selected set of keywords, specifically using "smart contracts" and "tools". The keywords were applied across several major academic databases, including Web of Science, IEEE Xplore, ACM Digital Library, Elsevier ScienceDirect, and Springer, covering publications dated from 2018 to 2024. This initial step identified a total of 199 survey papers.
In the second phase, the identified papers were filtered using a set of predefined exclusion criteria to ensure both relevance and quality. The exclusion criteria included the following:

- Studies not written in English.
- Studies not related to the field of computer science.
- Studies not related to SCs security.
- Studies without a full text available.
- Studies have not yet been published but documented on arXiv.
- Duplicate studies.

After applying these filters, 155 papers were excluded from the review. The third phase involved a closer inspection of the remaining papers by reviewing their titles and abstracts to determine alignment with the research objectives. For those potentially relevant, a full-text review was conducted to confirm their suitability. The complete process is illustrated in Figure 1, which outlines the main stages and outcomes of this methodical selection approach.

2.1.2 Technical Literature Search Methodology

This section defines a strategy to find technical papers on tools for detecting vulnerabilities in SCs. It consists of two main steps: a thorough search and a detailed screening. In the first step, specific keywords

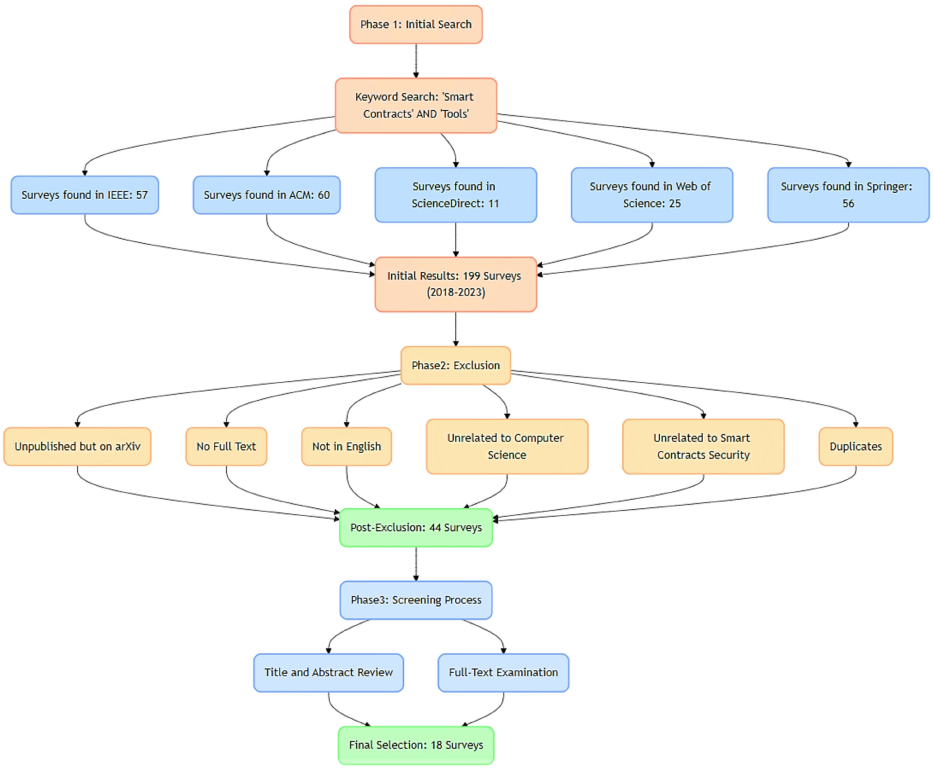

Figure 1: Survey Selection Process

such as ''smart contracts" and ''tools", ''security", ''detection" and ''vulnerabilities", are used to perform searches across several major academic databases. These included Elsevier ScienceDirect, IEEE Xplore, ACM Digital Library, and SpringerLink. To ensure coverage of recent developments, the search focused on publications from 2016 onwards. The goal was to capture a broad range of technical contributions related to SCs vulnerability detection. Table 1 summarizes the number of relevant papers retrieved for each keyword across these databases.

After gathering a large collection of papers, a careful screening process is employed in the second step to select the most relevant ones. For this screening, papers are excluded based on the following criteria:

- Studies not written in English.

Table 1: Distribution of Retrieved Papers by Source and Keyword

Keyword	**ACM**	**IEEE**	**Sceince Direct**	**Springer**
Smart Contracts and Tools	147	357	60	2126
Smart Contracts and Security	257	368	293	1094
Smart Contracts and Detection	164	245	73	724
Smart Contracts and Vulnerabilities	141	341	64	720
Smart Contracts and Detection Tools	81	138	15	69

- Studies unrelated to the field of computer science.
- Studies unrelated to SCs tools.
- Studies without a full text available.
- Duplicate studies.

The remaining papers were then reviewed in greater detail, with close attention to their abstracts, methodologies, results, and conclusions, to ensure they were directly related to the development or evaluation of tools for detecting SCs vulnerabilities. In total, 256 papers are selected as directly relevant to this study. The overall methodology for identifying and selecting these technical papers is illustrated in Figure 2.

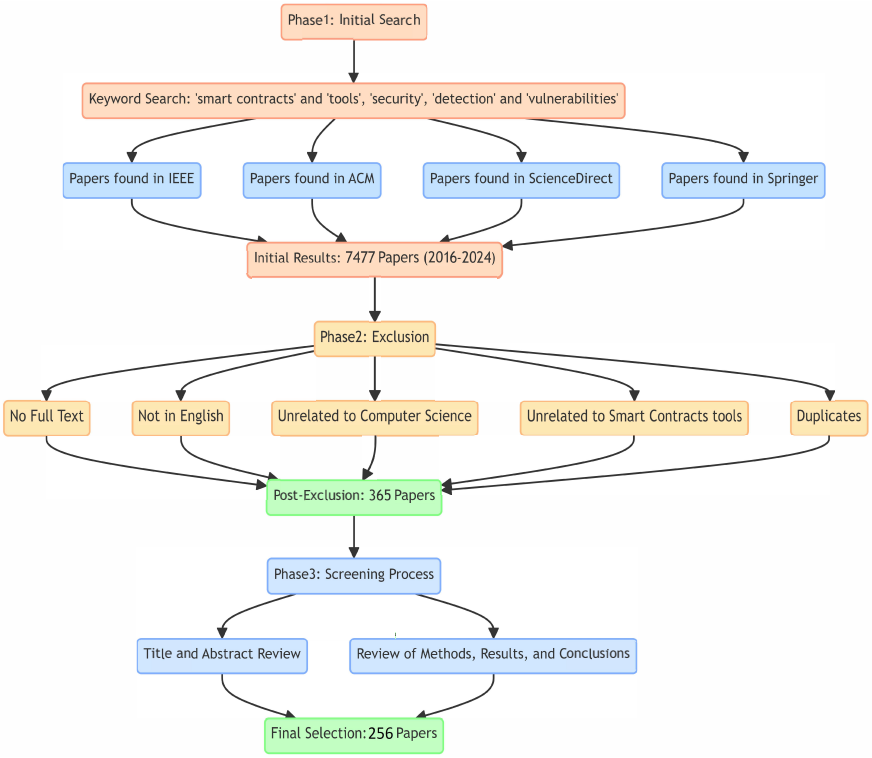

Figure 2: Technical Paper Selection Process

2.2 Previous Survey Analysis and Motivation

According to the surveys identified in Subsection 2.1.1, this section presents an analysis of 18 papers that specifically focus on SCs vulnerability detection tools. These surveys examine various aspects of existing tools, including their capabilities, limitations, and methodological foundations. To enable a clearer and more structured analysis, the reviewed surveys are categorized into three distinct groups: experimental analysis, theoretical analysis, and comparative analysis. The experimental category includes surveys that evaluate tools by testing them against specific datasets to measure their performance and accuracy. The theoretical

category comprises surveys that describe tools in terms of their underlying methods, techniques, and intended applications, often providing a categorization based on these approaches. The comparative category involves surveys that assess tools using multiple criteria such as detection accuracy, vulnerability coverage, usability, and performance. To support this classification and enhance clarity, a summary table is included that outlines the key features of the selected surveys. This structured approach not only facilitates a comprehensive understanding of the current research landscape but also helps identify critical gaps and areas for improvement in the field of SCs vulnerability detection.

2.2.1 Survey Papers with Experimental Evaluations

Some surveys have focused on evaluating SCs detection tools through experimental testing, with the aim of measuring their performance metrics and execution times. For example, Durieux et al. [1] performed a detailed study where they tested nine automated analysis tools for finding vulnerabilities in Solidity-based SCs. To make the tests fair and easy to repeat, they created a new tool called SmartBugs. This framework was built to provide a consistent environment where different tools could be tested and compared properly.
The researchers used two datasets for their tests. The first dataset, called sbcurated, included 69 SCs that were already known to have 115 vulnerabilities. These vulnerabilities were divided into ten categories based on a commonly used list called DASP10. The second dataset, named sbwild, contained 47,518 SCs taken from the Ethereum blockchain. This large dataset was used to test how well the tools worked on real-world contracts.
Out of 35 tools they found during their research, only 9 tools met all the requirements to be included in the tests. These requirements included being available to the public, supporting command-line usage, working with Solidity code, only needing the source code, and being designed to find vulnerabilities or bad practices. The tools that were included are: HoneyBadger, Maian, Manticore, Mythril, Osiris, Oyente, Securify, Slither, and SmartCheck. Table 2 shows a breakdown of which tools were included and which were excluded based on these requirements.

Table 2: Tools Included and Excluded Based on Criteria

	Inclusion criteria	Tools that violate criteria
Excluded (26)	Available and CLI (C1)	Ether, Gasper, ReGuard, Remix, SASC, sCompile, teEther, Zeus
	Compatible Input (C2)	MadMax, Vandal
	Only Source (C3)	Echidna, VeriSol
	Vulnerability Finding (C4)	contractLarva, E-EVM, Erays, Etherplay, EtherTrust, EthIR, FSolidM, KEVM, Octopus, Porosity, rattle, Solgraph, SolMet, Solhint
Included (9)	HoneyBadger, Maian, Manticore, Mythril, Osiris, Oyente, Securify, Slither, Smartcheck	

The testing process was huge, involving 428,337 tests, which took about 564 days and 3 hours to complete. This large-scale effort was made possible by using powerful servers from Scaleway and Google Cloud.

The results of the study were split into three parts: how accurate the tools were, how they performed on real-world contracts, and how fast they worked.

- The tools only found 42% of the vulnerabilities in the sbcurated dataset. Out of all the tools, Mythril performed the best, finding about 27% of the vulnerabilities by itself. The authors also tried combining tools to see if that would help. They found that using Mythril and Slither together gave the best result, finding 42 out of 115 vulnerabilities (37%). This is shown in Table 3, which lists how many vulnerabilities each tool found for each category.
- The tools could not find vulnerabilities in two categories: Bad Randomness and Short Addresses. This means there are still gaps in the tools' abilities. Table 4 shows the total number of vulnerabilities found by each tool, including those not tagged in the sbcurated dataset.

Table 3: Vulnerabilities Detected Per Tool (sbcurated Dataset)

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/19 0%	0/19 0%	4/19 21%	4/19 21%	0/19 0%	0/19 0%	0/19 0%	4/19 21% (1)	2/19 11%	5/19 26%
Arithmetic	0/22 0%	0/22 0%	4/22 18%	15/22 68%	11/22 50% (2)	12/22 55% (2)	0/22 0%	0/22 0%	1/22 5%	19/22 86%
Denial Service	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/ 7 0%
Front Running	0/7 0%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	0/8 0%	0/8 0%	2/8 25%	5/8 62%	5/8 62%	5/8 62%	5/8 62%	7/8 88% (2)	5/8 62%	7/ 8 88%
Time Manipulation	0/5 0%	0/5 0%	1/5 20%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	2/5 40% (1)	1/5 20% (1)	3/ 5 60%
Unchecked Low Calls	0/12 0%	0/12 0%	2/12 17%	5/12 42% (1)	0/12 0%	0/12 0%	3/12 25%	4/12 33% (3)	4/12 33% (1)	9/12 75%
Other	2/3 67%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	3/3 100% (1)	0/3 0%	3/ 3 100%
Total	2/115 2%	0/115 0%	13/115 11%	31/115 27%	16/115 14%	17/115 15%	10/115 9%	20/115 17%	13/115 11%	48/115 42%

When the tools were tested on the 47,518 contracts from the sbwild dataset, they found that 93% of the contracts were labeled as vulnerable by at least one tool. This high percentage probably includes many false positives. One tool, Oyente, reported vulnerabilities in 73% of the contracts, mostly related to Arithmetic errors.

The researchers looked for patterns to see if the tools agreed with each other about certain vulnerabilities. When four or more tools agreed, they considered that a stronger result. For example, 937 contracts were marked as having Arithmetic vulnerabilities by four or more tools. This part of the study is shown in Table 5, which summarizes how many contracts were marked as vulnerable by each tool.

Table 4: Total Number of Detected Vulnerabilities

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	10	28	24	0	0	6	20	3	91
Arithmetic	0	0	11	92	62	69	0	0	23	257
Denial of Service	0	0	0	0	27	11	0	2	19	59
Front Running	0	0	0	21	0	0	55	0	0	76
Reentrancy	0	0	4	16	5	5	32	15	7	84
Time Manipulation	0	0	4	0	4	5	0	5	2	20
Unchecked Low Level Calls	0	0	4	30	0	0	21	13	14	82
Unknown Unknowns	5	2	25	32	0	0	0	28	8	100
Total	5	12	76	215	98	90	114	83	76	769

The researchers also made a graph to show how often tools agreed with each other. Figure 3 shows how many vulnerabilities were found by only one tool, two tools, three tools, and four or more tools. Most vulnerabilities were only found by one tool, which suggests that many of the findings are not reliable.

Table 5: Contracts Marked as Vulnerable (sbwild Dataset)

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0 0%	44 0%	47 0%	1,076 2%	0 0%	2 0%	614 1%	2,356 4%	384 0%	3,801 8%
Arithmetic	1 0%	0 0%	102 0%	18,515 39%	13,922 29%	34,306 72%	0 0%	0 0%	7,430 15%	37,597 79%
Denial of Service	0 0%	0 0%	0 0%	0 0%	485 1%	880 1%	0 0%	2,555 5%	11,621 24%	12,419 26%
Front Running	0 0%	0 0%	0 0%	2,015 4%	0 0%	0 0%	7,217 15%	0 0%	0 0%	8,161 17%
Reentrancy	19 0%	0 0%	2 0%	8,454 17%	496 1%	308 0%	2,033 4%	8,764 18%	847 1%	14,747 31%
Time Manipulation	0 0%	0 0%	90 0%	0 0%	1,470 3%	1,452 3%	0 0%	1,988 4%	68 0%	4,069 8%
Unchecked Low Calls	0 0%	0 0%	4 0%	443 0%	0 0%	0 0%	592 1%	12,199 25%	2,867 6%	14,656 30%
Unknown Unknows	26 0%	135 0%	1,032 2%	11,126 23%	0 0%	0 0%	561 1%	9,133 19%	14,113 29%	28,355 59%
Total	46 0%	179 0%	1,203 2%	22,994 48%	14,665 30%	34,764 73%	8,781 18%	22,269 46%	24,906 52%	44,589 93%

The tools took an average of 4 minutes and 31 seconds to check each contract. Some tools were very fast, like Slither and SmartCheck, which only took 5 seconds and 10 seconds per contract, while others, like Manticore, took much longer at around 24 minutes and 28 seconds per contract. The researchers noted that tools that use static analysis (like Slither) are usually faster than those that use symbolic execution or concolic analysis (like Manticore).

Overall, the study by Durieux et al. [1] showed that many tools still have trouble finding certain types of vulnerabilities. They suggested that combining tools, especially Mythril and Slither, works better than using any single tool alone. They also pointed out that the high number of false positives is a big problem that needs to be fixed. The SmartBugs framework they built is a useful tool that can help other researchers test and compare new tools in the future.

Another study conducted by Kushwaha et al. [2] focused on evaluating 16 tools using the SolidiFI benchmark. They categorized the tools based on their underlying analysis approach and their input type, which could be either Solidity source code or EVM bytecode. The tools were grouped into three categories: static analysis tools, dynamic analysis tools, and hybrid analysis tools. Static analysis tools, which include Slither, SmartCheck, and Solhint, primarily focus on analyzing source code without executing the contracts. Dynamic analysis tools, such as Manticore and Maian, analyze contracts by executing them and observing their runtime behavior. Hybrid tools, like Mythril and Oyente, combine both static and dynamic analysis to enhance detection capabilities. Table 6 provides an overview of these tools, detailing their functionalities and methodologies.

The authors conducted their evaluation using the SolidiFI benchmark, a dataset comprising SCs annotated with known vulnerabilities. They assessed each tool based on its ability to detect specific vulnerabilities, execution time, and precision-recall metrics. The tools were evaluated against a total of 30 SCs, which included common vulnerabilities such as re-entrancy, arithmetic overflow/underflow, gas-related issues, timestamp dependency, and transaction-ordering dependency.

Their results are detailed across several tables and figures. Table 6 provides a comprehensive overview of the tools considered in their study, detailing their functionalities and detection methods. Table 7 summarizes the experimental setup, including the datasets, model architectures, and training hyperparameters used in the analysis. Table 8 compares the tools based on their vulnerability detection performance, specifically their precision, recall, and F1 scores. The authors further break down their findings by categorizing tools according to their input type and analysis approach. Table 9 presents the detection performance of static analysis tools, while Table 10 focuses on dynamic analysis tools.

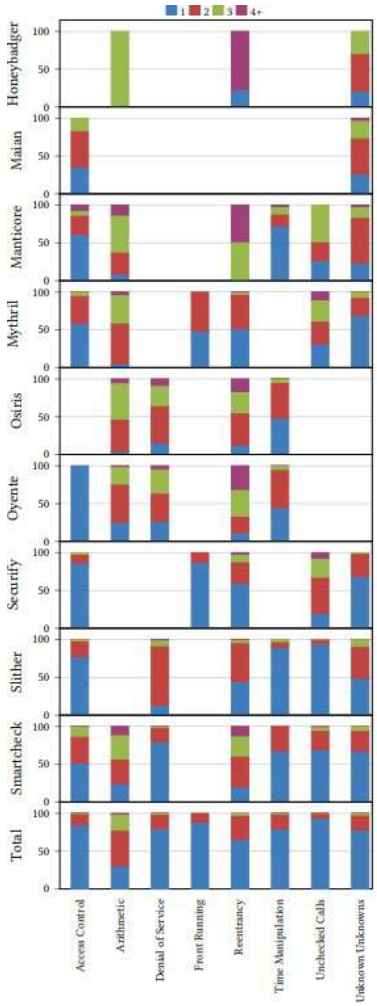

Figure 3: Proportion of Vulnerabilities Identified by Number of Tools

They also conducted a comparative evaluation of hybrid tools in Table 11. The performance of each tool was assessed based on the categories of vulnerabilities they could detect. Table 12 compares dynamic analysis tools that accept Solidity code as input, while Table 13 does the same for tools that accept EVM bytecode. Additionally, Table 14 summarizes the vulnerabilities detected by each tool and their corresponding analysis approaches.

The authors provide graphical representations of their findings to illustrate the performance and detection capabilities of the tools. Figure 4 shows the architecture of their experimental setup, providing an overview of the process used for evaluating the tools. Figure 5 illustrates the proportion of the top five vulnerabilities detected by static and dynamic analysis tools. This comparison highlights which vulnerabilities are most commonly detected and which ones are often missed.

Additionally, Figure 6 displays the share of various analysis approaches (such as symbolic execution, fuzz testing, and code instrumentation) utilized by the tools. This figure provides insight into the most commonly used methodologies and their effectiveness. Figure 7 presents the average execution time of each tool when tested against 30 contracts, showing that Slither, Solhint, and SmartCheck performed best in terms of speed,

Table 6: Overview of Tools

Tool	Source Code	Organization	Academic	Command Line Interface	Web Interface	Year	Platform
ContractWard	✗	✗	✓	✓	✗	2019	-
Echidna	✓	✗	✗	✓	✗	2020	Haskel
Eth2Vec	✗	✗	✓	✓	✗	2021	-
Ethainter	✗	✗	✓	✓	✗	2020	-
EthVer	✗	✗	✓	✓	✗	2021	-
FEther	✓	✗	✗	✓	✗	2019	Coq
FSPVM	✗	✗	✓	✓	✗	2020	Coq
Gas Gauge	✗	✗	✓	✓	✗	2021	-
GasTap	✓	✗	✓	✗	✓	2018	Python
MuSC	✓	✗	✗	✓	✗	2019	JAVA
Neucheck	✗	✗	✓	✓	✗	2019	JAVA
Pakala	✓	✗	✗	✓	✗	2018	Python
Remix-IDE	✓	✗	✗	✗	✓	2016	Java Script
SASC	✗	✗	✓	✓	✗	2018	C (83%), Python (15.7%), Markfile(0.5%)
sCompile	✗	✗	✓	✓	✗	2018	Python
SESCon	✗	✗	✓	✓	✗	2021	-
SIF	✓	✗	✗	✓	✗	2019	C++
Slither	✓	✗	✗	✓	✗	2018	Python
SmartAnvil	✓	✗	✗	✓	✗	2018	-
SmartBugs	✓	✗	✗	✓	✗	2020	Python
SmartCheck	✓	Smart Check	✗	✓	✗	2017	JAVA
SmartEmbed	✓	✗	✗	✗	✓	2019	Java Script
Smart-Graph	✗	✗	✓	✗	✓	2021	-
SmartInspect	✗	✗	✓	✓	✗	2018	Pharo
SmartScan	✗	✗	✓	✗	✓	2021	-
Sol Analyzer	✓	✗	✓	✓	✗	2019	GO
Solc-Verify	✓	✗	✗	✓	✗	2019	C++, Solidity
Solgraph	✓	✗	✗	✓	✗	2016	Java Script
SolGuard	✗	✗	✓	✓	✗	2021	-
Solhint	✓	Protofire	✗	✓	✗	2017	JAVA
Solidifier	✓	✗	✓	✓	✗	2020	-
Solidity check	✓	✗	✓	✓	✗	2019	C++
SolMet	✓	✗	✗	✓	✗	2018	JAVA
VeriSmart	✓	Software Analysis Laboratory, Korea University	✓	✓	✗	2020	Ocaml
VeriSol	✓	✗	✗	✓	✗	2019	C#
VeriSolid	✓	✗	✓	✗	✓	2019	Java Script
Zeus	✗	✗	✓	✓	✗	2018	-

while Manticore had the highest execution time.

Figure 8 compares the detection performance of each tool on the top five vulnerabilities. Slither, Mythril, and Oyente demonstrated better detection performance for re-entrancy and arithmetic overflow/underflow vulnerabilities. Figure 9 further illustrates the false positive rates of each tool when detecting these vulnerabilities. Finally, Figure 10 summarizes the overall detection accuracy of the tools.

The authors emphasize that although certain tools like Slither, Mythril, and Oyente performed well in detecting specific vulnerabilities, no single tool was able to detect all vulnerabilities present in the benchmark. They suggest that combining tools or developing hybrid approaches may be a more effective strategy. Moreover, high false positive rates remain a significant challenge, especially for tools like Mythril and Securify.

The study concludes that while current tools are effective to some extent, there is considerable room for improvement, particularly in terms of reducing false positives and increasing detection coverage. They recommend future work to focus on developing more comprehensive benchmarks and improving the underlying detection methodologies.

Another study that follows an experimental evaluation approach is by Lashkari et al. [3] who conducted a domain-specific evaluation of various vulnerability detection tools applied to SCs, particularly focusing on

Table 7: Experimental Setup

Tool	Source Code	Organization	Academic	Command Line Interface	Web Interface	Year	Platform
DEFECTCHECKER	✗	✗	✓	✓	✗	2021	JAVA
E-EVM	✓	✗	✗	✓	✗	2018	Python
Erays	✓	✗	✗	✓	✗	2018	Python
ESCORT	✗	✗	✓	✓	✗	2021	-
Ether (S-GRAM)	✗	✗	✓	✓	✗	2018	Python (Based on S-gram artifact)
EtherTrust	✓	✗	✗	✓	✗	2018	Java, Javascript
EthIR	✓	✗	✗	✓	✗	2018	Python
eThor	✗	✗	✓	✓	✗	2020	Java Script
EthPloit	✗	✗	✓	✓	✗	2020	-
GasChecker	✗	✗	✓	✓	✗	2020	-
Gasper	✗	✗	✓	✓	✗	2017	Python
HoneyBadger	✓	✗	✗	✓	✗	2019	Python
KEVM	✓	✗	✗	✓	✗	2016	Mixture of markdown syntax and K specification language.
MadMax	✓	✗	✗	✓	✗	2018	GogaHose IR
Mythril	✓	ConsenSys	✗	✓	✗	2017	Python
Octopus	✓	quoscient	✗	✓	✗	2018	Python
Osiris	✓	✗	✗	✓	✗	2018	Python
Oyente	✓	✗	✗	✓	✗	2016	Python
Porosity	✓	Camae	✗	✓	✗	2017	C++
RA	✓	✗	✗	✓	✗	2020	Python
Rattle	✓	Trail of Bits	✗	✓	✗	2018	Python
Securify	✓	Chain Security (Closed Version)	✗	✓	✗	2018	JAVA
SMARTSHEILD	✗	✗	✓	✓	✗	2020	-
TEEther	✗	✗	✓	✓	✗	2018	Python
Vandal	✓	✗	✗	✓	✗	2018	Python
VerX	✗	✗	✓	✗	✓	2020	-

transactive energy systems. The study assessed the performance of popular tools such as Mythril, Slither, Smartcheck, Honeybadger, Osiris, Solhint, Oyente, Conkas, and Confuzzius. The evaluation was carried out using the curated SB dataset, which contains known vulnerabilities such as reentrancy, access control, time manipulation, bad randomness, and arithmetic issues.

Table 15 provides a comprehensive comparison of all tools against different vulnerabilities, noting their performance based on true positives, false negatives, average runtime, and accuracy metrics. Slither demonstrated the highest accuracy and shortest runtime, while Honeybadger and Solhint were noted for poor detection performance. Mythril, although commonly used, showed mediocre accuracy with high processing times.

Figures 11 and 12 provide a graphical summary of detected and undetected vulnerabilities across both energy and non-energy contracts. The study noted that energy contracts are more prone to vulnerabilities and have higher processing times than non-energy contracts. This highlights the importance of developing domain-specific tools or enhancing existing tools to improve detection accuracy in energy contracts.

The authors emphasized that while tools like Slither perform well in general contexts, their effectiveness varies significantly when applied to specific domains such as energy systems. Furthermore, the study suggests that combining tools or developing new approaches that cater to domain-specific requirements could enhance detection performance.

Table 8: Vulnerability Detection Performance

Tool	Source Code	Organization	Academic	Command Line Interface	Web Interface	Year	Platform
Conkas	✓	✗	✗	✓	✗	2019	Python
GASOL	✗	✗	✓	✓	✗	2020	-
SAFEVM	✓	✗	✗	✗	✓	2019	Python
FSolidM	✓	✗	✓	✗	✓	2017	Java Script

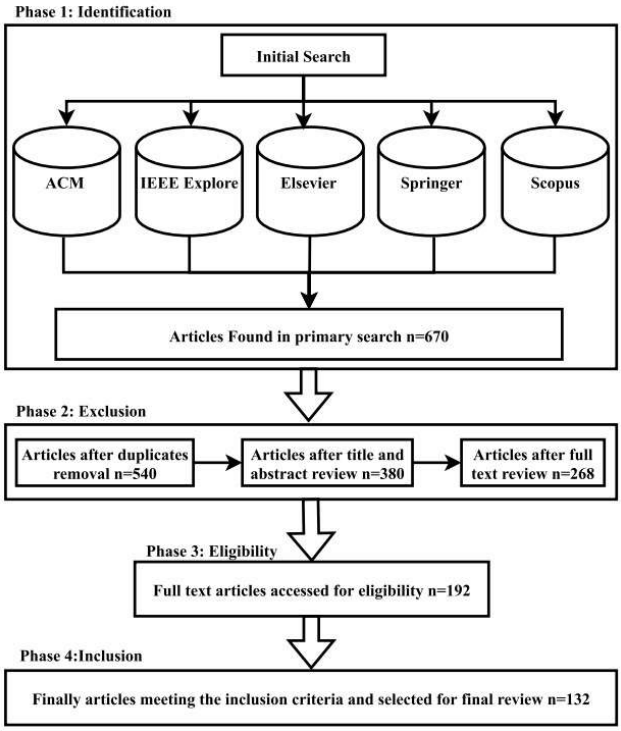

Figure 4: Experimental Setup Architecture

These papers suggest that future work should focus on improving the overall coverage of vulnerabilities detected by existing tools. Current tools often miss certain types of vulnerabilities or produce high numbers of false positives, which limits their practical use. Researchers recommend enhancing tool quality through better algorithms and techniques to reduce false positives and improve accuracy. Another important direction is to integrate analysis methods more smoothly into the development process. This would allow developers to detect vulnerabilities earlier and address them before deployment. Making tools more user-friendly and adaptable to different development environments is also highlighted as a priority.

2.2.2 Survey Papers with Methodological Descriptions

Unlike the experimental evaluations, some surveys have focused more on describing tools and their functionalities rather than testing them. These papers often provide theoretical categorizations based on the tools' underlying techniques, intended applications, or operational mechanisms. For example, Almakhour et al. [4] conducted a comprehensive survey of 25 tools for verifying Ethereum SCs, with a focus on both formal

Table 9: Static Analysis Tools

Tool	Checked Vulnerability													Analysis Approach											
	Timestamp Dependancy	Reentrency	*TOD	tx.origin	Blockhash/Block Number	Gas Related Issues	Delegate Call	**Underflow/Overflow	Freezing Ether	Unchecked Call	Self Destruct	Access Control	Denial of Service	Symbolic Execution	Dis-assembler	Graphic Visualizer	Fuzz Testing	Constraint Solving	Machine Learning	Code Instrumentation	Mutation Testing	Code-Transformation	Formal Verification	Abstract Interpretation
ContractWard	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Echidna	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Eth2Vec	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Ethainter	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
EthVer	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
FEther	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
FSPVM	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗
Gas Gauge	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
GasTap	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
MuSC	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Neucheck	✓	✓	✗	✓	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
Pakala	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Remix-IDE	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
SASC	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
sCompile	✗	✓	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
SESCon	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗
SIF	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Slither	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗
SmartAnvil	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
SmartBugs	✗	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
SmartCheck	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
SmartEmbed	✓	✓	✗	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗
Smart-Graph	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
SmartInspect	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗	✗
SmartScan	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
Sol Analyzer	✓	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Solc-Verify	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Solgraph	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗
Solguard	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
Solhint	✗	✓	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Solidifier	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Solidity check	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
SolMet	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗
VeriSmart	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
VeriSol	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
VeriSolid	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Zeus	✗	✓	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗

*TOD-Transaction Ordering Dependency, **Related to Arithmetic

verification methods and vulnerability detection methods. Their work aimed to improve the security and correctness of SCs by providing a detailed description of each tool and method, including their strengths and limitations.

The survey identifies two major aspects of SCs verification: correctness verification and vulnerabilities detection. For correctness verification, the study explores methods such as theorem proving, model checking, and runtime verification. Vulnerabilities detection focuses on techniques like symbolic execution, abstract interpretation, and fuzzing.

The paper provides an extensive overview of vulnerabilities detection tools, which are categorized based on their verification methods: symbolic execution, abstract interpretation, and fuzzing. Table 16 presents a comparison of these tools in terms of their type, level, and verification methods, while Table 17 lists the specific vulnerabilities they detect along with the related attacks.

The authors emphasize that the existing tools are primarily effective for simple SCs and highlight the need for further research to address more complex contracts. They suggest that combining formal verification methods with vulnerability detection approaches could enhance the robustness and comprehensiveness of the verification process.

Table 10: Dynamic Analysis Tools

	Checked Vulnerability														Analysis Approach										
Tool	Timestamp Dependancy	Reentrency	*TOD	tx.origin	Blockhash/Block Number	Gas Related Issues	Delegate Call	**Underflow/Overflow	Freezing Ether	Unchecked Call	Self Destruct	Access Control	Denial of Service	Symbolic Execution	Dis-assembler	Graphic Visualizer	Fuzz Testing	Constraint Solving	Machine Learning	Code Instrumentation	Mutation Testing	Code Transformation	Formal Verification	Abstract Interpretation
Conkas	✗	✓	✓	✗	✓	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
DEFECTCHECKER	✓	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
E-EVM	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Erays	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
ESCORT	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Ether (S-GRAM)	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
EtherTrust	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓
EthIR	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
eThor	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
EthPloit	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
FSolidM	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
GasChecker	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
GASOL	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Gasper	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
HoneyBadger	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
KEVM	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
MadMax	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Mythril	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Octopus	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Osiris	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Oyente	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Porosity	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
RA	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Rattle	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
SAFEVM	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Securify	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
SMARTSHEILD	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
TEEther	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Vandal	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
VerX	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

*TOD-Transaction Ordering Dependency, **Related to Arithmetic

Similarly, Vacca et al. [5] reviewed 26 tools that detect vulnerabilities in SCs. They focused on how these tools work, their methods of analysis, and how well they perform across different blockchain platforms. The authors grouped the tools based on their approach: static analysis, dynamic analysis, formal verification, or a combination of these methods. They also checked which platforms the tools support, such as Ethereum and Hyperledger.

The paper compares how well these tools detect common vulnerabilities like reentrancy, integer overflow, and denial-of-service attacks. They use tables and figures to show the strengths and weaknesses of each tool. The authors also explain the main techniques these tools use, such as symbolic execution, formal verification, taint analysis, and machine learning.

Table 18 shows a summary of the 26 tools. It includes their analysis methods, supported platforms, detected vulnerabilities, and other important features. This table helps to quickly compare the tools and understand what each tool focuses on.

Table 11: Hybrid Analysis Tools

Tool	Source Code	Organization	Academic	Command Line Interface	Web Interface	Year	Platform
ContractLarva	✓	✗	✗	✓	✗	2017	Haskell(70%) TeX(30%)
Ethlint	✓	✗	✗	✓	✗	2016	JavaScript
Harvey	✗	✗	✓	✓	✗	2020	-
ModCon	✗	✗	✓	✗	✓	2020	JavaScript
Solitor	✓	✗	✓	✓	✗	2018	Java
VULTRON	✓	✗	✗	✓	✗	2019	Java Script

Table 12: Solidity Code Analysis Tools

Tool	Source Code	Organization	Academic	Command Line Interface	Web Interface	Year	Platform
ContractFuzzer	✓	✗	✗	✓	✗	2018	Go
ContractGuard	✗	✗	✓	✓	✗	2019	Java Script
EthBMC	✓	✗	✗	✓	✗	2020	Rust
Etherolic	✗	✗	✓	✓	✗	2020	Rust
EVMFuzz	✓	✗	✓	✓	✗	2019	Python
MAIAN	✓	✗	✗	✓	✓	2018	Python
Manticore	✓	Trail of Bits	✗	CLI and Python API	✗	2017	Python
Sereum	✗	✗	✓	✓	✗	2019	-
sFuzz	✓	✗	✗	✓	✗	2020	C++
SODA	✓	✗	✗	✓	✗	2020	Go

The paper points out problems with these tools, such as difficulty handling large-scale contracts, user-friendliness issues, and trouble adapting to new types of vulnerabilities. They suggest improving automation, making tools work better across different platforms, and combining different methods to increase accuracy.

Another survey by Wu et al. [6] reviewed 32 SCs vulnerability detection tools, categorizing them based on their primary methods, including auditing, formal verification, and anomaly detection. The paper provides a general overview of the tools' methodologies, discussing their strengths and limitations without delving into specific performance metrics or detailed evaluations.

The authors describe auditing methods as involving manual or automated reviews of SCs to identify vulnerabilities through feature code matching, formal verification, and symbolic execution. They note that feature code matching is fast and responsive but has limitations in detecting unknown vulnerabilities.

Formal verification is highlighted as a robust technique for ensuring SCs security, though it is limited by the need for precise specifications and may require manual interventions. The paper also mentions that formal verification is widely used to eliminate logical gaps and security vulnerabilities but does not always guarantee complete coverage.

Anomaly detection is discussed mainly in terms of its application to detecting suspicious transactions and abnormal contract behavior. The authors describe several methods, including graph-based approaches and machine learning algorithms, which aim to identify irregular patterns and potential attacks.

Overall, this survey provides a high-level categorization of tools but lacks detailed quantitative comparisons. It does not include relevant tables or figures to summarize the tools or their performance, making it difficult to assess how the tools compare in terms of accuracy, efficiency, or coverage.

Table 13: EVM Bytecode Analysis Tools

Tool	Source Code	Organization	Academic	Command Line Interface	Web Interface	Year	Platform
EASYFLOW	✓	✗	✗	✓	✓	2019	Go
ReGuard	✗	✓	✗	✓	✗	2018	Python
SoliAudit	✗	✗	✓	✓	✗	2019	-

Table 14: Vulnerabilities Detected and Analysis Approaches

	Checked Vulnerability													Analysis Approach									
Tool	Timestamp Dependancy	Reentrancy	*TOD	tx.origin	Blockhash/Block Number	Gas Related Issues	Delegate Call	**Underflow/Overflow	Freezing Ether	Unchecked Call	Self Destruct	Access Control	Denial of Service	Symbolic Execution	Dis-assembler	Visualizer	Fuzz Testing	Taint analysis	Machine Learning	Code Instrumentation	Mutation Testing	Model Based Testing	Formal Verification
ContractFuzzer	✓	✓	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
ContractGuard	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
ContractLarva	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
EASYFLOW	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
EthBMC	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Etherolic	✗	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗
Ethlint	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
EVMFuzz	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Harvey	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
MAIAN	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Manticore	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
ModCon	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
ReGuard	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Sereum	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
sFuzz	✓	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
SODA	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
SoliAudit	✓	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
Solitor	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
VULTRON	✗	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗

Hewa et al. [7] also reviewed 17 SCs vulnerability detection tools, focusing mainly on their methodologies and a general overview of SCs. The survey highlights the principles of SCs, their various applications across industries, and potential challenges. Additionally, the authors discuss future prospects for improving SCs security, emphasizing the importance of developing better detection tools and standardizing approaches to enhance reliability and effectiveness.

Unlike other surveys that provide detailed comparisons or performance evaluations, this paper mainly presents a broad overview of the tools without quantitative analysis or systematic categorization. The tools are discussed alongside their general functionalities, but the survey lacks specific tables or figures summarizing their methodologies, performance, or comparative results. Therefore, while this survey offers useful context on the application of SCs vulnerability detection tools, it does not provide in-depth analysis or comprehensive evaluation of their effectiveness.

Another relevant survey is provided by Chu et al. [8] who categorized 20 tools based on their detection methods, including static analysis, dynamic analysis, formal verification, and hybrid methods. They provide

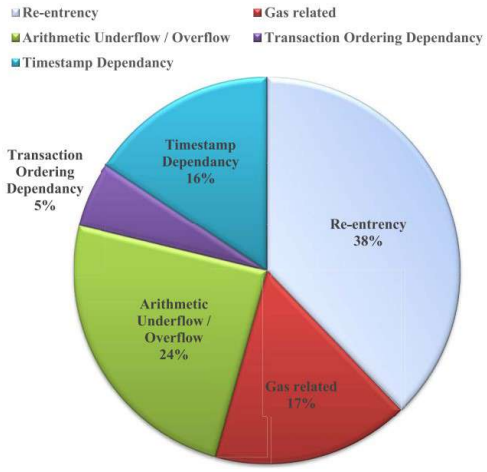

Figure 5: Top Five Vulnerabilities Detected

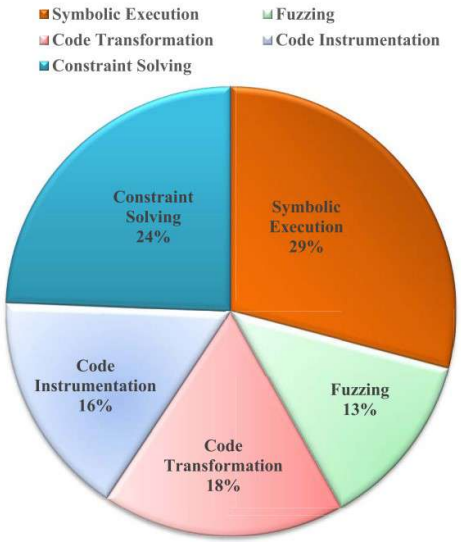

Figure 6: Analysis Approaches Used

descriptions of each tool's operational framework, input requirements, and the specific vulnerabilities each tool targets. This categorization is organized into several tables to help readers understand the scope and capabilities of these tools.

The survey explains that most tools are designed for Ethereum, but some are also applicable to other platforms. The authors discuss the tools' strengths and weaknesses, with emphasis on aspects such as scalability, precision, and adaptability. However, the paper mainly focuses on categorization rather than a detailed experimental evaluation of tools.

Table 19 provides an overview of the tools categorized by their detection techniques. The authors classify the tools based on their methodologies, such as symbolic execution, formal verification, static analysis, dynamic analysis, and machine learning. This classification highlights the variety of techniques used across the tools and their applicability to different blockchain platforms. The paper also mentions that the effectiveness of these tools varies significantly depending on the types of vulnerabilities they are designed to detect.

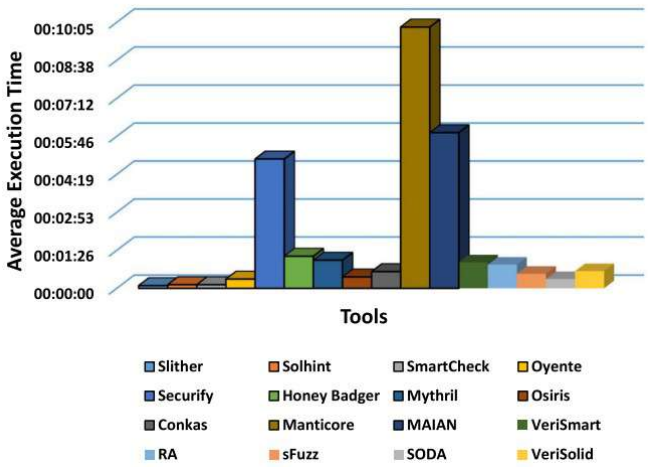

Figure 7: Average Execution Time of Each Tool

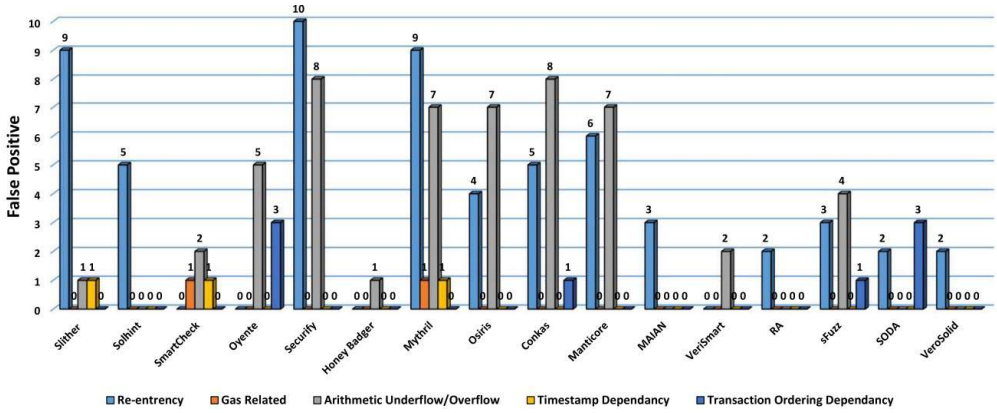

Figure 8: Detection Performance of Top Five Vulnerabilities

Despite this categorization, the survey lacks experimental evaluations and comparisons of tools. The authors do not provide quantitative assessments, leaving gaps in understanding how the tools perform in real-world scenarios. Additionally, the survey does not offer a comprehensive benchmark to test the tools' capabilities across multiple vulnerability types.

Liu et al. [9] conducted a broad survey on the security verification of SCs, focusing primarily on two aspects: security assurance and correctness verification. The paper categorizes related research into various approaches, including programming correctness, formal verification, and vulnerability scanning. The survey places strong emphasis on formal verification methods, identifying them as a promising approach for enhancing SCs reliability.

Regarding tools, the paper briefly mentions several tools used for vulnerability detection and formal verification, particularly in the context of static and dynamic analysis. Most of these tools are described as part of broader discussions on formal verification techniques and programming standards. However, the survey does not provide detailed evaluations of individual tools or their effectiveness in detecting specific vulnerabilities. Instead, the tools are mainly referenced to illustrate different verification approaches, such as program-based verification and behavior-based verification.

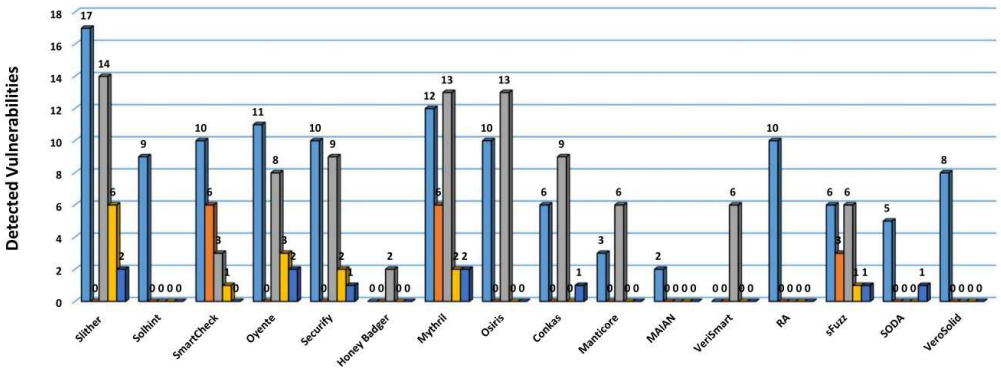

Figure 9: False Positive Rate of Each Tool

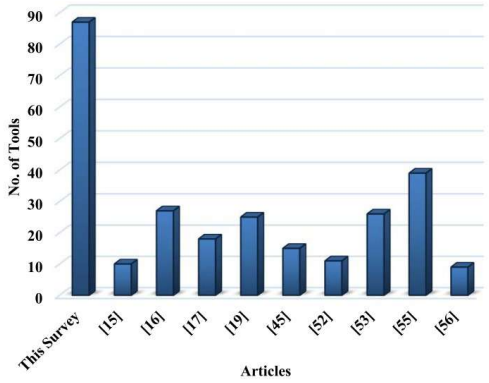

Figure 10: Overall Detection Accuracy

Overall, the survey provides a high-level overview of SCs security verification, with limited focus on individual tools. It emphasizes formal verification as a crucial area for future research but lacks systematic comparisons or experimental assessments of existing tools.

Tolmach et al. [10] provided an overview of 34 tools that utilize formal verification techniques, categorizing them by their verification methods, including model checking, theorem proving, program verification, symbolic execution, and runtime verification. The survey details the use of formal verification tools to rigorously prove or disprove SCs properties using abstract specifications.

The authors list several tools such as NuSMV, SPIN, CPN, PRISM, UPPAAL, Maude, Coq, Isabelle/HOL, and Agda, classifying them by their verification technique and detailing their application to SCs. For instance, Coq, Isabelle/HOL, and Agda are used primarily for theorem proving, while tools like Oyente and Mythril are applied for symbolic execution. This categorization helps identify the strengths and limitations of various tools and highlights areas where improvements are needed.

Table 20 from the paper provides a partial summary of formal verification tools, categorizing them based on their analysis techniques and the programming logic they apply. The table is helpful for understanding the range of tools used in the field and their respective functionalities. While the survey provides valuable insights into how formal verification methods are applied to SCs, it also highlights several challenges. The authors emphasize that although formal verification methods are powerful, they are not fully automated and

Table 15: Comparison of Tools Against Vulnerabilities

		Analysis Tools								
Vulnerabilities	**Metrics**	**Confuzzius**	**Conkas**	**Honeybadger**	**Mythril**	**Osiris**	**Oyente**	**Slither**	**Smartcheck**	**Solhint**
Bad randomness # 1	Duration	8.737881184	43.17423201	42.6293509	117.7532399	35.8527348	12.72187996	2.460914135	7.229845047	2.348021984
	Detection	✓	×	×	✓	✓	✓	✓	✓	✓
Bad randomness #2	Duration	5.140118837	4.026548862	77.663939	35.02412128	4.400139093	9.356572151	2.505437136	8.295716047	2.406748772
	Detection	✓	✓	×	✓	✓	✓	✓	✓	✓
Bad randomness #3	Duration	5.883338928	5.733039141	4.730108023	23.37867403	5.190639257	4.574568987	4.68672204	12.09117603	4.312572956
	Detection	×	×	✓	✓	×	×	✓	✓	×
Bad randomness #4	Duration	13.40132713	25.54713297	270.5662198	665.0775077	52.94841003	31.93285203	2.299463034	7.967276096	2.468337059
	Detection	✓	✓	✓	×	✓	✓	✓	✓	✓
Access Control #1	Duration	5.200572014	4.012099981	4.676064968	29.09435225	3.299962044	3.316371918	1.745553017	9.402060032	2.072764874
	Detection	×	✓	✓	✓	✓	✓	✓	✓	✓
Access Control #2	Duration	8.697600126	32.12349224	4.971230984	287.8219483	4.574355125	4.499389887	2.58026576	9.426314116	2.655647039
	Detection	✓	✓	✓	✓	✓	✓	✓	✓	✓
Access Control #3	Duration	12.31246901	7.738770008	4.431187153	75.809196	4.460098982	3.817485809	3.226504803	12.97784209	5.038857222
	Detection	✓	✓	✓	✓	✓	✓	✓	✓	✓
Access Control #4	Duration	9.177275181	12.17278075	6.860949993	187.0252879	3.602584839	5.654606819	4.357161045	8.529126167	2.500974655
	Detection	✓	✓	✓	✓	✓	✓	✓	✓	✓
Reentrancy #1	Duration	26.17377186	556.3664987	22.80289483	64.55575109	106.8687789	56.45254993	3.139178038	103.8020959	7.275887251
	Detection	✓	✓	×	×	✓	✓	✓	×	×
Reentrancy #2	Duration	8.124588966	3.576477051	2.981256962	66.90424418	3.465710878	4.1645298	2.385936975	8.54885602	2.344093084
	Detection	✓	✓	✓	✓	✓	✓	✓	✓	✓
Reentrancy #3	Duration	10.48157692	3.779005289	2.671264648	18.282516	4.615365267	4.575246096	2.574982882	11.23986912	2.040110826
	Detection	✓	✓	×	×	✓	✓	✓	×	×
Reentrancy #4	Duration	3.70638895	3.037296772	2.180539131	26.53004694	4.16880393	3.100974798	1.809810877	5.179394245	2.098110199
	Detection	✓	✓	×	×	✓	✓	✓	×	×
Time Manipulation #1	Duration	4.84670186	3.274656057	5.653620005	21.299891	3.301398754	2.669829845	1.798388243	11.70897198	1.994317055
	Detection	✓	✓	×	✓	✓	×	✓	×	×
Time Manipulation #2	Duration	20.45597315	3.69738698	4.367240906	271.6076109	3.699479103	5.648472071	2.266718149	14.23034596	4.238348961
	Detection	×	✓	×	✓	×	×	✓	×	×
Time Manipulation #3	Duration	5.510486841	4.209807873	3.619537115	10.61389804	4.293602228	3.043504953	2.796922207	30.98754811	3.026561022
	Detection	✓	✓	×	×	✓	✓	×	×	×
Time Manipulation #4	Duration	4.280456066	5.245132685	6.81735301	47.40231466	3.229922056	9.511505842	2.836236	115.5346749	10.58052206
	Detection	✓	✓	×	×	×	×	×	×	×

often require expert intervention. Additionally, challenges like state explosion in model checking and the complexity of theorem proving are highlighted as areas requiring further improvement. They recommend developing more efficient algorithms and creating user-friendly tools to make formal verification more accessible and effective for practical use cases.

Zheng et al. [11] briefly discussed 12 tools for SCs vulnerability detection, with a focus on bytecode analysis, machine learning methods, and various vulnerability types such as re-entrancy and Ponzi schemes. They primarily examined techniques for analyzing SCs through different methods, including control-flow analysis, decompilation, and symbolic analysis. Tools like Gigahorse, which decompiles bytecode to a higher-level representation, were mentioned alongside symbolic analysis techniques that map dependencies and check compliance with security patterns.

The authors conclude that while existing tools provide some coverage for well-known vulnerabilities like re-entrancy, they still struggle with complex attack patterns and require improvements in precision and efficiency. They suggest that integrating multiple techniques, such as combining static and dynamic analysis with machine learning, could enhance the overall effectiveness of SCs vulnerability detection. The need for

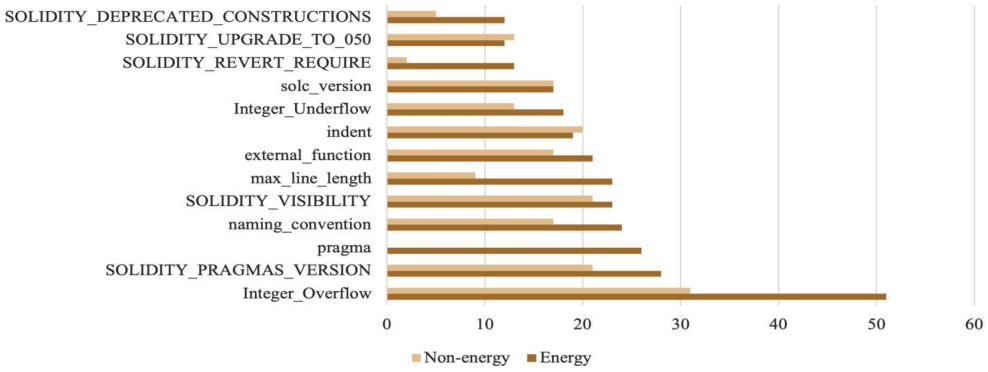

Figure 11: Detected Vulnerabilities Across Energy and Non-Energy Contracts

Table 16: Summary of Vulnerabilities Detection Tools (Type, Level, and Methods)

Tools	Verification methods	Type	Level
Oyente	Symbolic execution	Static analysis	Bytecode
OSIRIS	Symbolic execution	Static analysis	Bytecode
MAIAN	Symbolic execution	Dynamic analysis	Bytecode
SmartCheck	Symbolic execution	Static analysis	Solidity code
Slither	Symbolic execution	Static analysis	Bytecode
Mythril	Symbolic execution	Static analysis	Bytecode
Gasper	Symbolic execution	Static analysis	Bytecode
MadMax	Abstract interpretation	Static analysis	Bytecode
Vandal	Abstract interpretation	Static analysis	Bytecode
Ethir	Abstract interpretation	Static analysis	Bytecode
Securify	Abstract interpretation	Dynamic analysis	Bytecode
ContractFuzzer	Fuzzing	Dynamic analysis	Bytecode
ReGuard	Fuzzing	Dynamic analysis	Solidity or Bytecode

more comprehensive benchmarks and standardized testing frameworks is also emphasized to ensure consistent and reliable evaluations.

Hu et al. [12] provided a thorough analysis of 40 SCs vulnerability detection tools, systematically categorizing them based on their underlying techniques and the specific vulnerabilities they target. The survey divides tools into various categories, including those utilizing static analysis, dynamic analysis, formal verification, and hybrid approaches. For each category, the paper discusses the tools' operational principles, their effectiveness in detecting vulnerabilities, and their limitations.

The survey presents a detailed comparison of the tools through several tables, among which Table 21 provides a comprehensive summary of the most representative research work. This table evaluates tools based on criteria such as the auxiliary approach used, the analysis level, the number of detectable vulnerabilities, and whether the tools are open source.

The authors also discuss the strengths and weaknesses of various approaches. For example, symbolic execution is identified as one of the most widely used techniques but is limited by issues like path explosion. Formal verification offers greater semantic understanding but suffers from low automation levels. Fuzzing is noted for its efficiency but struggles when analyzing contracts without source code.

The survey highlights that deep learning-based detection methods are more extensible than traditional methods but lack interpretability. While the paper provides useful insights into the tools, its focus is more on categorizing them based on their methods rather than offering a detailed comparison of their performance or usability.

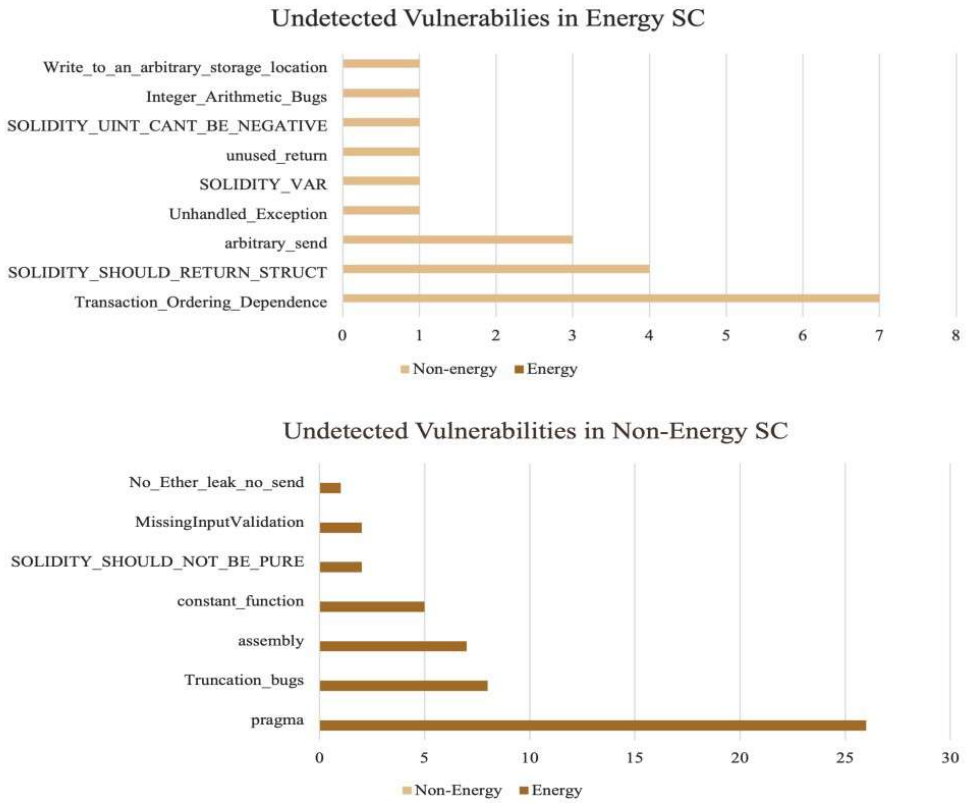

Figure 12: Undetected Vulnerabilities Across Energy and Non-Energy Contracts

These papers emphasize the importance of combining static and dynamic analysis methods to improve vulnerability detection. Using both approaches can help cover a wider range of vulnerabilities and reduce false positives by cross-verifying results. Researchers also point out the need for comprehensive benchmarks that include various types of vulnerabilities to ensure tools are evaluated consistently. Establishing standardized metrics is crucial for comparing tools fairly and identifying their strengths and weaknesses. This can also help guide the development of more reliable and versatile tools that work across multiple blockchain platforms. Creating unified frameworks for evaluation would contribute to better progress in the field.

2.2.3 Survey Papers with Comparative Analyses

Furthermore, some surveys contributed to the discussion by conducting comparative studies that evaluate the tools from various perspectives. For example, Harz et al. [13] reviewed 10 verification tools and methods for SCs analysis. Their focus was on examining various aspects such as the underlying analysis techniques, automation levels, compatibility with different programming languages, and the open-source status of the tools. The paper also highlights how the selected tools differ in their ability to detect various vulnerability types, emphasizing that tools with higher automation levels tend to be more user-friendly and applicable to real-world scenarios. Additionally, the authors point out that most tools are limited to specific languages or platforms, which restricts their broader applicability.

The paper categorizes tools based on their verification methods, which include formal verification, symbolic

Table 17: Vulnerabilities Detection Tools Based on Vulnerabilities and Related Attacks

Tools	Detecting vulnerabilities	Attacks
Oyente	Re-entrancy, Exception handling, Transaction ordering, Block timestamp dependency, Call stack depth limitation	Integer overflow/under flow The DAO attack
Vandal	Re-entrancy, Unchecked and failed send, Destroyable/suicidal contract, Unsecured balance, tx.origin	The DAO attack, Parity multisig wallet attack
Ethir	Re-entrancy, Exception handling, Transaction ordering, Block timestamp dependency	The DAO attack
Securify	Exception handling, Transaction ordering, Call stack depth limitation, Unchecked and failed send, No restricted write, No restricted transfer, Non-validated arguments	Parity multisig wallet attack
Osiris	Integer bugs	Integer overflow/under flow attack
MAIAN	Call stack depth limitation, Destroyable/suicidal contract, Unsecured balance, Greedy contracts, Prodigal contracts	Parity multisig wallet attack
Smartcheck	Re-entrancy, Transaction ordering, Block timestamp dependency, Integer over/under flow, unchecked, failed send, Destroyable/suicidal contract, Unsecured balance	The DAO attack, Integer over/under flow attack
Slither	Re-entrancy, Exception handling, Transaction ordering, Block timestamp dependency	The DAO attack
Gasper	Gas costly code patterns exist	–
Madmax	Unbounded mass operations, integer overflows	The DAO attack, Integer over/under flow attack
ContractFuzzer	Re-entrancy vulnerabilities, Unchecked call	The DAO attack
ReGuard	Re-entrancy vulnerabilities	The DAO attack
Mythril	Re-entrancy vulnerabilities, Unchecked call, tx.origin, Unchecked math, manipulated balance vulnerability	The DAO attack

execution, static analysis, and runtime verification. In particular, the authors emphasize the importance of automation in the verification process, noting that fully automated tools are generally preferred due to their usability and efficiency. Furthermore, they discuss how compatibility with various SCs languages can impact the applicability of a tool across different blockchain platforms. The survey also addresses the difficulty in achieving complete automation for formal verification, due to the inherent complexity of SCs and the diverse range of potential vulnerabilities. This challenge is identified as a critical area for future improvement.

Additionally, Harz et al. provide detailed comparisons through Table 22 and Table 23. Table 22 summarizes the primary characteristics of the tools, including their verification approach, language compatibility, and automation level. Table 23 further examines the specific techniques employed by each tool, highlighting their strengths and limitations in detecting common vulnerabilities in SCs. The tables provide useful insights into the landscape of SCs verification tools, helping to identify trends and gaps in existing approaches.

While the survey provides useful insights into the different methods employed by verification tools, it lacks comprehensive performance evaluations or quantitative comparisons. The authors conclude that future research should focus on developing more robust benchmarks and enhancing automation capabilities to improve the effectiveness of vulnerability detection tools. They also recommend that efforts should be directed towards improving compatibility across different platforms and languages to broaden the applicability of these tools.

López Vivar et al. [14] reviewed 18 open-source tools for detecting vulnerabilities in SCs, analyzing their operation, installation requirements, and performance across different blockchain platforms. They examined each tool individually, describing how it works, the types of analysis it performs, and the platforms it supports. The tools were grouped based on their methods, including static analysis, dynamic analysis, formal verification, or hybrid approaches. Common tools covered in their study include Oyente, Mythril, Slither, and Securify.

After describing each tool, the authors summarized their findings through two tables. Table 24 presents the

Table 18: Vulnerabilities Detection Tools Based on Vulnerabilities and Related Attacks

Tool/Framework	Technology	Technique	Description	Experimental Dataset	Findings
CONTRACTLA-RVA (Ellul and Pace, 2018)	Ethereum	Standard techniques for runtime verification.	A tool for runtime verification of smart contracts to ensure safety.	Parity Multisig Wallet smart contracts.	Useful to ensure that smart contracts have certain specifications.
KEVM (Hildenbrandt et al., 2018)	Ethereum	Formal analysis.	A formal specification executable EVM's bytecode stack-based language.	Official Ethereum test suite.	KEVM is complete, performing and can be used in an environment of continuous integration.
SmartCheck (Tikhomirov et al., 2018)	Ethereum	Static code analysis.	Static analysis tool to detect code issue related to security, operations, development and functionality.	4,600 verified smart contracts.	The tool identified 99.9% contracts with issues, 63.2% contracts with vulnerability.
MadMax (Grech et al., 2018)	Ethereum	Static code analysis.	Static analysis program to detect gas-vulnerabilities.	91,800 smart contracts.	81% of the samples considered present warnings.
RA (Re-entrancy Analyzer) (Chinen et al., 2020)	Ethereum	Static code analysis.	A static analysis tool to find re-entrancy attacks.	Known re-entrancy vulnerabilities.	The tool precisely identifies vulnerable smart contracts.
Samreen & Alalfi (Samreen and Alalfi, 2020)	Ethereum	Combination of static and dynamic analysis.	A framework to detect Re-entrancy vulnerabilities.	5 modified smart contracts.	The combination of the two techniques improves the performance and decreases the false positives.
SolMet (Hegedüs, 2019)	Ethereum	Code parsing, static code analysis.	A tool to measures static OO metrics in Solidity smart contracts.	10,206 Solidity source code file with nearly 45,000 smart contracts.	The tool is immature at the current state.
PASO (Pierro and Tonelli, 2020)	Ethereum	Code parsing, static code analysis.	A web-based parser for Solidity language analysis.	No validation has been provided.	Calculate metrics of smart contract and can be used with different versions of the Solidity language.
VerX (Permenev et al., 2020)	Ethereum	Abstraction and symbolic execution engine.	A symbolic execution engine to find functional properties of Ethereum smart contracts.	12 real world Ethereum projects.	Verx is practical and useful to verify functional properties of smart contracts.
Gasper (Chen et al., 2017)	Ethereum	Symbolic execution.	A tool to locate gas-costly programming pattern (Dead Code, Opaque Predicates,Expensive Operations in a Loop).	4,240 smart contracts.	93.5%, 90.1% and 80% smart contracts suffer from three different gas-costly pattern patterns.
GasChecker (Chen et al., 2020a)	Ethereum	Symbolic execution.	A tool for automatic identification of inefficient code for gas consumption in smart contracts.	1,500 representative smart contracts from 599,934 contracts.	Most real contracts contain inefficient code.
MAIAN (Nikolić et al., 2018)	Ethereum	Symbolic execution.	A tool to trace properties of smart contracts to find exploit.	970,898 smart contracts.	Most of the smart contracts identified by the tool are true positives.
SmartEmbed (Gao et al., 2020)	Ethereum	Code parsing.	A prototype to identify code clones.	More than 22,000 smart contracts.	The tool identified 90% of the clones.
SIF (Peng et al., 2019)	Ethereum	Code instrumentation and analysis.	A framework for monitoring, instrumenting, and generating code.	51 smart contracts.	The tool can analyze the AST, retrieve information and generating Solidity code.
Oyente (Luu et al., 2016)	Ethereum	Symbolic execution.	An analysis tool for smart contracts to detect security bugs.	19,366 Ethereum contracts	The tool identifies vulnerable smart contracts.
TEETHER (Krupp and Rossow, 2018)	Ethereum	Symbolic execution.	The tool creates exploits for contracts given its binary bytecode.	38,757 Ethereum contracts.	The tool finds exploits for 815 on 38,757 smart contracts.
Slither (Feist et al., 2019)	Ethereum	Static code analysis.	A tool to find vulnerabilities, code optimization, code review, bug detection.	1,000 smart contracts.	The tool is better than other in terms of speed, robustness and false positives.
Securify (Tsankov et al., 2018)	Ethereum	Symbolic analysis.	Security analyzer for Ethereum smart contracts.	18,000 smart contracts.	The tool demonstrates the correctness of smart contracts and discovers violations.
Contract-fuzzer (Jiang et al., 2018)	Ethereum	Static code analysis, fuzz testing.	A fuzzer to test Ethereum smart contracts for security vulnerabilities.	6,991 smart contracts.	The tool detects 459 vulnerabilities on 6,991 smart contracts tested.
EVMFuzz (Fu et al., 2019)	Ethereum	Static code analysis, differential fuzz testing.	Differential fuzz testing to detect vulnerabilities of EVMs.	36,295 smart contracts.	Most of the smart contracts generated have different performance.

main features of the tools, indicating whether they support Solidity or EVM bytecode, their analysis methods, and whether they perform static analysis, dynamic analysis, or formal verification. Table 25 compares the tools based on their ease of installation, usefulness, dependency requirements, and whether they are actively maintained. Tools like Slither and Mythril are considered easy to install and effective, while others like EtherTrust are noted for their complexity and outdated dependencies.

The authors highlight several problems affecting these tools, such as outdated dependencies, limited documentation, and lack of active maintenance. They recommend improving compatibility across platforms, providing better documentation, and ensuring active maintenance to enhance usability and effectiveness.

Rameder [15] provided a detailed examination of 140 SCs security analysis tools, focusing on their methods for detecting vulnerabilities. The tools are categorized based on their approach, such as static analysis, dynamic analysis, formal verification, and hybrid methods. This categorization helps highlight common weaknesses and gaps in existing tools, such as their limited coverage of certain vulnerability types and lack of support for complex contracts.

The survey describes each tool's methodology, availability, publication date, supported platforms, and detection techniques. Instead of presenting all tools in a single table, the author organizes them into smaller groups based on specific criteria. Some groups include tools classified by their detection technique, applica-

Table 19: Categorization of Tools Based on Detection Methods

Main technology	Tools	Assistive technology	Analysis level	Number	Open source	Method comparison
Symbol execution	Oyente	–	Bytecode	5	Yes	**Advantages**, the most widely used method in traditional tools, **Disadvantages**, in the face of multi-layer deep calling sequences, there will be a path explosion problem.
	Osiris	Taint analysis	Bytecode	1	Yes	
	Mythril	Taint analysis	Bytecode	14	Yes	
	DEPOSafe	Behavior modeling	Bytecode	1	No	
	Artemis	–	Bytecode	4	No	
	MAIAN	–	Source code	3	Yes	
Formal verification	ZEUS	–	Source code	7	No	**Advantages**, security specifications are used to detect contracts. **Disadvantages**, there is a problem with unreachable execution paths.
	Securify	–	Source code	4	Yes	
Fuzzing	ConFuzzius	–	Bytecode	7	Yes	**Advantages**, different inputs and coverage can be customized. **Disadvantages**, it cannot be tested for smart contracts without source code.
	Harvey	Program analysis	Source code	4	Yes	
	sFuzz	–	Bytecode	9	Yes	
	EVMFuzzer	–	Bytecode	5	Yes	
Other technology	Slither	Program analysis	Source code	26	Yes	**Advantages**, the daily overhead is relatively small. **Disadvantages**, the detection rate of false positives is high.
	SmartCheck	Program analysis	Source code	20	Yes	
Deep learning	TMP	–	Source code	3	Yes	**Advantages**, no need to manually formulate detection rules. **Disadvantages**, the method has the problem of poor interpretability.
	CodeNet	–	Bytecode	4	No	
	AME	Expert knowledge	Source code	3	Yes	
	CGE	Expert knowledge	Source code	3	Yes	
	ContractWard	–	Bytecode	6	No	
	DeeSCVHunter	–	Source code	2	No	

Table 20: A Summary of Formal Verification Tools

Verification Techniques	Tools	Selected References	Total
Model Checking	NuSMV, nuXmv	[15, 100, 118, 121, 128, 129]	18
	SPIN	[29, 123]	
	CPN	[61, 115]	
	BIP-SMC	[10, 120]	
	PRISM	[40]	
	UPPAAL	[18]	
	MCK	[169]	
	Maude	[25]	
	FDR	[141]	
	LDLf	[147]	
Theorem Proving	Coq	[19, 22, 37, 130, 132, 152, 161, 184]	20
	Isabelle/HOL	[16, 72, 88, 89, 105, 109]	
	Agda	[46]	
Program Verification	Datalog & Soufflé	[43, 44, 77, 136, 168]	42
	Boogie Verifier & Corral	[20, 83, 176]	
	LLVM & SMACK	[97, 175]	
	SeaHorn	[14, 97]	
	F*	[39, 78]	
	KeY	[11, 34, 35]	
	Why3	[58, 127, 187]	
	$\mathbb{K}$ framework	[96, 98, 135]	
	Custom static analyses	[64, 69, 105, 149, 153, 154, 157]	
Symbolic & Concolic Execution	`Oyente`	[67, 116, 167, 189]	22
	`Mythril`	[125, 140, 177]	
	`teEther`	[102]	
	`Maian`	[131]	
	`Manticore`	[124]	
Runtime Verification & Testing	`ContractLarva`	[26, 63]	26
	`EVM*`	[117]	
	`ReGuard`	[111]	
	`SODA`	[50]	
	`Solythesis`	[107]	
	`ECFChecker`	[80]	

tion scope, availability, and supported platforms. The survey provides brief descriptions of each tool to give a clear overview of their functionalities and limitations.

The author discusses general patterns and challenges among the tools. Common problems include difficulty detecting certain vulnerabilities, limited compatibility with blockchain platforms beyond Ethereum, and challenges in handling complex SCs. Despite the existence of many tools, most share similar weaknesses and do not provide a complete solution.

The survey concludes that future research should focus on creating tools that can detect a wider range of vulnerabilities and improve accuracy by combining different analysis methods. Enhancing compatibility across various blockchain platforms is also considered a priority.

Di Angelo et al. [16] conducted a comprehensive evaluation of 27 tools designed for analyzing Ethereum SCs. Their study aimed to fill a gap in the existing literature by reviewing tools originating from academic, community, and corporate environments. The tools were evaluated based on their purpose, input types, analysis techniques, and their ability to detect various security issues.

The authors categorized the tools based on their purpose, code level, type, preprocessing requirements, and

Table 21: Summary of Representative Vulnerability Detection Tools.

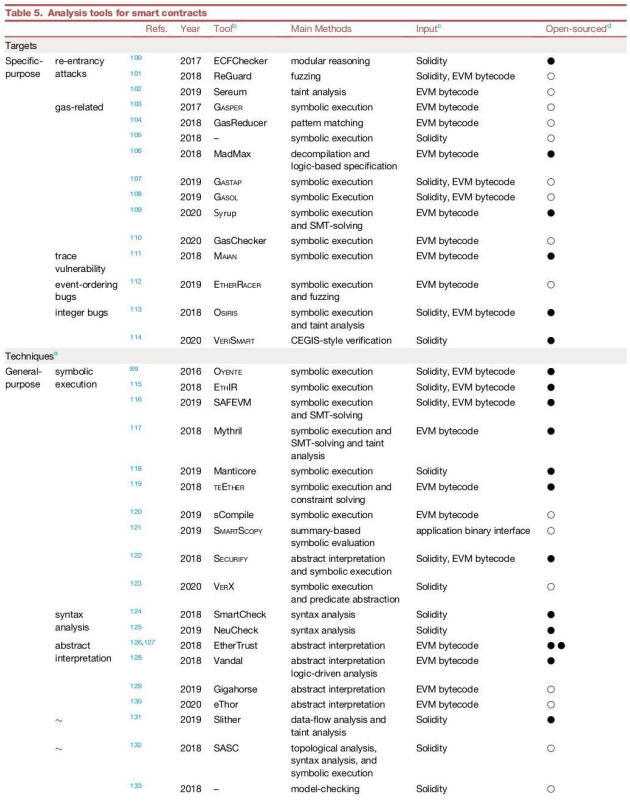

Table 5. Analysis tools for smart contracts

		Refs.	Year	Tool[b]	Main Methods	Input[c]	Open-sourced[d]
Targets							
Specific-purpose	re-entrancy attacks	[100]	2017	ECFChecker	modular reasoning	Solidity	●
		[101]	2018	ReGuard	fuzzing	Solidity, EVM bytecode	○
		[102]	2019	Sereum	taint analysis	EVM bytecode	○
	gas-related	[103]	2017	Gasper	symbolic execution	EVM bytecode	○
		[104]	2018	GasReducer	pattern matching	EVM bytecode	○
		[105]	2018	–	symbolic execution	Solidity	○
		[106]	2018	MadMax	decompilation and logic-based specification	EVM bytecode	●
		[107]	2019	Gastap	symbolic execution	Solidity, EVM bytecode	○
		[108]	2019	Gasol	symbolic Execution	Solidity, EVM bytecode	○
		[109]	2020	Syrup	symbolic execution and SMT-solving	EVM bytecode	●
		[110]	2020	GasChecker	symbolic execution	EVM bytecode	○
	trace vulnerability	[111]	2018	Maian	symbolic execution	EVM bytecode	●
	event-ordering bugs	[112]	2019	EtherRacer	symbolic execution and fuzzing	EVM bytecode	○
	integer bugs	[113]	2018	Osiris	symbolic execution and taint analysis	Solidity, EVM bytecode	●
		[114]	2020	VeriSmart	CEGIS-style verification	Solidity	●
Techniques[a]							
General-purpose	symbolic execution	[89]	2016	Oyente	symbolic execution	Solidity, EVM bytecode	●
		[115]	2018	EthIR	symbolic execution	Solidity, EVM bytecode	●
		[116]	2019	SAFEVM	symbolic execution and SMT-solving	Solidity, EVM bytecode	●
		[117]	2018	Mythril	symbolic execution and SMT-solving and taint analysis	EVM bytecode	●
		[118]	2019	Manticore	symbolic execution	Solidity	●
		[119]	2018	teEther	symbolic execution and constraint solving	EVM bytecode	●
		[120]	2019	sCompile	symbolic execution	EVM bytecode	○
		[121]	2019	SmartScopy	summary-based symbolic evaluation	application binary interface	○
		[122]	2018	Securify	abstract interpretation and symbolic execution	Solidity, EVM bytecode	●
		[123]	2020	VerX	symbolic execution and predicate abstraction	Solidity	○
	syntax analysis	[124]	2018	SmartCheck	syntax analysis	Solidity	●
		[125]	2019	NeuCheck	syntax analysis	Solidity	●
	abstract interpretation	[126,127]	2018	EtherTrust	abstract interpretation	EVM bytecode	●●
		[128]	2018	Vandal	abstract interpretation logic-driven analysis	EVM bytecode	●
		[129]	2019	Gigahorse	abstract interpretation	EVM bytecode	○
		[130]	2020	eThor	abstract interpretation	EVM bytecode	○
	~	[131]	2019	Slither	data-flow analysis and taint analysis	Solidity	●
	~	[132]	2018	SASC	topological analysis, syntax analysis, and symbolic execution	Solidity	○
		[133]	2018	–	model-checking	Solidity	○

methods of analysis. Table 26 provides an overview of all tools considered in their study. The table highlights which tools are designed for static analysis, dynamic analysis, or hybrid approaches, along with their specific functionalities and methodologies.

The study also investigated the implementation details of these tools, such as the number of contributors, active months, programming languages used, and repository commits. This information is summarized in Table 27. Although this table primarily addresses development aspects, it provides useful context for understanding the maturity and maintenance status of each tool.

Independent tool comparisons were conducted to assess their accuracy and effectiveness in detecting known vulnerabilities. Table 28 presents the results of these comparisons, highlighting which tools are considered most effective across various criteria. Additionally, the authors compared tools against each other from their own perspective, which is summarized in Table 29.

A critical aspect of this study is the comparison of the security issues addressed by each tool. Table 30 provides a detailed list of vulnerabilities that each tool is capable of detecting. This table is particularly relevant for understanding the strengths and weaknesses of individual tools and for identifying gaps in vulnerability coverage.

The authors emphasized that many tools are specialized, targeting specific categories of vulnerabilities, while others attempt broader coverage with varying degrees of success. They recommend combining tools or adopting hybrid approaches to achieve more comprehensive detection capabilities. Furthermore, the study highlights the need for consistent benchmarks and standardized evaluation methodologies to improve the

Table 22: Verification Tools and Methods Overview

Language	Paradigm	Instructions	Semantics	Metering	Ref.
Solidity	contract-oriented	complete	informal	gas, limit	[30]
Vyper	procedural	restricted	informal	gas, limit	[15]
Bamboo	procedural	complete	semi†	gas, limit	[14]
Flint	procedural	complete	informal	gas, limit	[16]
Pyramid Scheme	functional	complete	informal	gas, limit	[41]
Obsidian	object-oriented	–	informal	–	[42]
Rholang	concurrent	complete	informal	phlogiston	[43]
Liquidity	functional	restricted	semi†	gas, limit	[31]
DAML	functional	restricted	–	–	[44]–[52]
Pact	functional	restricted	semi†	gas, limit	[19]
Simplicity	pure functional	restricted	formal	–	[32]
Scilla	functional	restricted	formal	gas, limit	[17]
Yul	procedural	complete	informal	gas, limit	[40]
EthIR	procedural	complete	informal	gas, limit	[24]
IELE	register-based	complete	formal	gas, limit	[20]
Bitcoin Script	stack-based	restricted	informal	script size	[33]
EVM	stack-based	complete	informal‡	gas, limit	[34]
eWASM	stack-based	complete	informal	gas, limit	[53]
Michelson	stack-based	restricted	semi†	gas, limit	[18]

Table 23: Techniques Employed by Verification Tools

Tool	Approach	Automation	Coverage	Languages	Source	Ref.
Securify	model	full	full	Solidity, EVM	open	[22]
Mythril	model	full	property	EVM	open	[66]
Oyente	model	full	property	Solidity, EVM	open	[21], [24]
ZEUS	model	full	property	Solidity, Go, Java	closed†	[23]
ECF	model	full	property	EVM	open	[67]
Maian	model	full	property	EVM	open	[68]
$\mathbb{K}$	proof	partial	full	EVM, IELE	open	[26], [71]
Lem	proof	partial	full	EVM	open	[27], [72]
Coq	proof	partial	partial‡	Scilla, Michelson	open	[17], [18]
*F**	proof	partial	partial‡	EVM	open	[25], [38]

reliability of future tool assessments.

Li et al. [17] performed a comparative analysis of 16 SCs vulnerability detection tools. The authors focus on the tools' detection approaches, vulnerability coverage, open-source status, and detection accuracy. By examining these aspects, the paper aims to highlight the strengths and weaknesses of each tool, providing a clear understanding of their effectiveness and limitations. The survey also considers the impact of different detection techniques on accuracy and coverage, emphasizing the need for diverse approaches to handle various types of vulnerabilities.

The comparative analysis is organized in a table that contains the detection approach, supported vulnerability types, availability of open-source code, and detection accuracy for each tool. This structured presentation allows readers to easily compare the tools based on their characteristics and functionalities. The authors provide explanations of how each tool works, discussing their methodologies and the specific vulnerabilities

Table 24: Summary of security tools specifications.

		Oyente	Remix-IDE	Solgraph	MadMax	Manticore	SmartCheck	Mythril	ContractLarva	SolMet	Vandal	EthIR	MAIAN	Erays	Rattle	Osiris	Securify	Slither	EtherTrust
Nivel	Bytecode	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Solidity	✗	✓	✓	✗	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
Analysis	Dynamic analysis	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
	Static analysis	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	Formal verification	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓	✗	✓

Table 25: Summary of the installation of the security tools.

Tool	Ease of Installation	Usefulness	Stays Up to Date	Dependencies
Oyente	1/5	3/5	No	Python 2 (pip2), Z3 Prover, web3 (pip3), solc, Go-Ethereum
Remix-IDE	5/5	5/5	Yes	nodejs, npm
Solgraph	4/5	5/5	No	nodejs, npm, graphviz
MadMax	4/5	5/5	No	Python 3
Manticore	5/5	5/5	Yes	Python 3 (+3.6v), pip3, solc
SmartCheck	5/5	5/5	Yes	npm
Mythril	5/5	5/5	Yes	npm, Python 3, pip3
ContractLarva	3/3	3/3	Yes	Haskell (ghc)
SolMet	5/5	5/5	No	Java, maven, CSV reader
Vandal	4/4	4/4	No	Python 3
EthIR	1/5	3/5	Yes	Python 2 (pip2), Z3 Prover, web3 (pip3), solc, Go-Ethereum
MAIAN	1/5	1/5	No	solc, Z3, Python 3, web3
Erays	5/5	3/5	No	graphviz, Python
Rattle	5/5	5/5	No	graphviz, solc, Python 3
Osiris	2/5	5/5	No	Python 2 (pip2), Z3 Prover, web3 (pip3), solc, Go-Ethereum
Securify	5/5	5/5	No	soufflé, Java 8, solc
Slither	5/5	5/5	Yes	solc, Python 3
EtherTrust	1/5	2/5	No	Z3 Prover, Python, maven

they target. Additionally, they describe the pros and cons of each tool, which helps to identify areas where improvements are necessary.

The survey also discusses the limitations of the reviewed tools, including issues such as incomplete vulnerability coverage, high false-positive rates, and restricted applicability across multiple blockchain platforms. These limitations indicate the need for future research to enhance the robustness and efficiency of vulnerability detection tools. Furthermore, the authors emphasize that detection accuracy varies significantly among the tools, which poses challenges for consistent performance evaluation. They suggest that integrating different analysis techniques and developing unified benchmarks could improve accuracy and consistency across various tools.

The comparative analysis provided by the survey highlights gaps in existing tools, particularly in terms of consistency in vulnerability detection and the lack of a unified benchmark for evaluating tool effectiveness. This suggests that future research should focus on developing standardized evaluation metrics and enhancing cross-platform applicability. The authors recommend that future work should aim to improve detection methods while ensuring that tools remain accessible and easy to use for developers and auditors alike.

Kushwaha et al. [2] conducted a comprehensive study involving 86 SCs vulnerability detection tools, evaluating them based on several important criteria. These criteria include the origin of the tools, distinguishing between academic and company-developed tools, the availability of source code, whether the tools are open-

Table 26: Overview of Tools

	Purpose				Level		Type		Code transformation						Analysis method					
Tool	Security issues	Exploits	Formal guarantees	Bulk analysis	Bytecode	Solidity code	Static analysis	Dynamic analysis	Contextualization	Disassembly	Control flow graph	Call graph	AST analysis	Decompilation	Code instrumentation	Symbolic execution	Constraint solving	Abstract interpretation	Horn logic	Model checking
contractLarva	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
E-EVM	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Erays	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
EthIR	✗	✗	✗	✗	✓+	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✗	✗	✗
EtherTrust	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗
FSolidM	✗	✗	✗	✗	form.spec		✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
KEVM	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
MAIAN	✓	✓+	✗	✓	✓+	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Manticore	✓	✓	✗	✗	✓+	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Mythril	✓	✓	✗	✗	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Osiris	✓	✗	✗	✓	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Oyente	✓	✗	✗	✓	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Porosity	✓	✗	✗	✗	✓+	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Rattle	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Remix-IDE	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Securify*	✓	✗	✓	✓	✓+	✗	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	✗
SmartCheck*	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Solgraph	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
SolMet	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Vandal	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✓	✗	✓	✓	✗
Ether*	✓	✗	✗	✓	✓+	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Gasper	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
ReGuard	✓	✗	✗	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
SASC	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗
sCompile	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
teEther	✓	✓+	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
Zeus	✓	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗

source or proprietary, interface type such as command-line or web-based, launch year, and programming language used for implementation. The survey aims to provide a structured overview of the current landscape of vulnerability detection tools by organizing them according to these attributes.

The authors categorize the tools by their detection methods, including static analysis, dynamic analysis, formal verification, and hybrid techniques. They provide detailed descriptions of each category, explaining how different methods work and what types of vulnerabilities they are most effective at detecting. The paper emphasizes that while some tools focus on specific vulnerabilities, others attempt to cover a broader range, though often with reduced accuracy or efficiency.

Additionally, the survey provides comparisons of tools based on their vulnerability coverage, ease of use, performance, and maintenance status. For instance, some tools excel in detecting specific vulnerabilities such as reentrancy or integer overflow but struggle with more complex or less common issues. The authors also note that tools developed by companies tend to prioritize usability and performance, whereas academic tools often focus on novel detection techniques or enhancing precision.

The paper includes various tables that summarize the tools' characteristics, methodologies, and effectiveness. These comparisons highlight significant differences in approach, performance, and applicability. The authors use these tables to illustrate trends in the development of SCs vulnerability detection tools, noting that many tools focus on Ethereum while others attempt to generalize to different blockchain platforms.

Table 27: Implementation Details of Tools

Tool	Commits	First commit mm/yy	Active months	Publication tool	Contributors	Affiliation	k loc	Language
contractL.	14	12/17	7	✓	1	ac	2.8	hs
E-EVM	2	1/18	1	✓	1	ac	2.0	py
Erays	6	8/18	3	*	3	ac	5.6	py
EthIR	394	2/18	9		1	ac	7.9	py
EtherTr.	1	5/18	na	✓	na	ac	13.0	java
FSolidM	183	9/17	13		4	ac	35.0	js
KEVM	1581	10/16	25		21	ac	23.0	$\mathbb{K}$,py
MAIAN	14	3/18	2	✓	2	ac	3.1	py
Manticore	597	2/17	21		56	co	37.0	py
Mythril	1988	9/17	14		42	co	9.0	py
Osiris	4	9/18	1	*	1	ac	1.2	py
Oyente	799	1/16	31		22	cm	6.6	py
Porosity	94	2/17	12		10	co	4.2	cpp
Rattle	23	8/18	2		2	co	2.6	py
Remix-I	4972	11/14	48		63	cm	17.0	js
Securify*	21	9/18	2	✓	4	ac	13.0	java
SmartCh.*	161	5/17	10	✓	4	ac	2.1	java
Solgraph	68	7/16	26		3	co	0.1	js
SolMet	9	2/18	7	*	1	ac	0.6	java
Vandal	811	8/16	22		6	ac	7.3	py

Despite the comprehensive analysis, the authors point out several limitations in existing tools. Common issues include limited vulnerability coverage, a high rate of false positives, and insufficient support for complex or non-standard SCs. They suggest that future work should focus on developing more robust benchmarks, improving detection accuracy, and creating tools that are both comprehensive and easy to use. Additionally, enhancing compatibility across various blockchain platforms remains a critical challenge that needs further attention.

2.3 Conclusion Remarks

While the analyzed surveys have provided valuable insights into SCs vulnerability detection tools, several unresolved challenges persist. One major limitation is the absence of standardized benchmarks, which complicates objective comparison across tools. Without consistent evaluation criteria or datasets, it remains difficult to determine which tools perform best under specific conditions. Additionally, the usability of these tools for developers and auditors is often underexplored, and there is limited focus on providing actionable programming guidelines to improve SCs languages and reduce the likelihood of vulnerabilities.

Beyond the 18 surveys examined in detail, some other studies briefly mention detection tools but do not conduct in-depth evaluations. For example, Bartoletti et al. [18] reference tools like Mythril, Oyente, and Securify, though their primary focus is on the design and semantics of SCs languages. Similarly, Ressi et al. [19] highlight the potential of AI-based approaches, such as anomaly detection, for enhancing blockchain security. However, these studies do not provide technical assessments or comparative analyses of specific

Table 28: Independent Tool Comparisons

INDEPENDENT TOOL COMPARISONS

	Remix	Porosity	SmartCheck	Securify	Mythril
Oyente	[56]	[55]	[54], [56]	[54], [56]	[54], [55]
Mythril		[55]	[54]	[54]	
Securify	[56]		[54], [56]		
Smartcheck	[56]				

Table 29: Tool Comparisons from Authors' Perspective

COMPARISON OF OWN TOOLS WITH OTHERS

Tool to tool	Oyente	Mythril	Securify	Remix	Zeus	MAIAN	EthIR	Rattle
Vandal	[44]	[44]					[44]	[44]
SmartCh.	[38]		[38]	[38]				
Securify	[35]	[35]						
sCompile	[50]					[50]		
teEther	[51]				[51]			
SASC	[49]							
ReGuard	[48]							
Zeus	[52]							
Osiris					[26]			

detection tools. As such, they offer useful peripheral insights but fall outside the focused scope of this section, which centers on comprehensive surveys.

Table 33 summarizes the characteristics of the reviewed surveys, outlining their methodologies—whether experimental, theoretical, or comparative—and cataloging the number of tools each study examined. The table also differentiates between relevant vulnerabilities, which are associated with specific tools, and general vulnerability references, where no explicit tool mapping is provided. In some cases, surveys describe detection approaches without specifying the tools that implement them. This inconsistency further complicates the comparison of studies and the identification of clear patterns.

A notable limitation among prior surveys is their tendency to either evaluate a limited set of tools or neglect to articulate the connection between tools, vulnerabilities, and detection methodologies. For instance, works such as [14] and [15] mention general categories of vulnerabilities but do not specify which tools are capable of detecting them. Others, like [13] and [22], offer only high-level overviews of detection approaches without detailing their tool-specific implementations. These gaps indicate a need for a more unified and detailed evaluation framework—one that explicitly maps tools to vulnerabilities and methods.

Table 30: Security Issues Detected by Tools

	Blockchain				EVM		Solidity											
Tool	TOD	Random number	Timestamp	Unpredictable state	Callstack depth	Lost Ether	Reentrancy	Unchecked call	tx.origin	Blockhash	send	selfdestruct	Visibility	Unchecked math	Costly pattern	Bad coding pattern	Deprecated	Other
MAIAN	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Manticore	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	✓
Mythril	✓	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓
Osiris	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Oyente	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Porosity	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Remix-IDE	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Securify*	✓	✗	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
SmartCheck*	✗	✗	✓	✓	✗	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
Solgraph	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓
Vandal	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓

Table 31: Comparison of 16 v Vulnerability Detection Tools

Detection Tool	Detection Approach	Supported Type of Vulnerabilities									Open-Source Language	Detection Accuracy
		Solidity Layer				EVM Layer			Block Layer			
		Reentrancy	Integer Error	Exception Handling	Logical Error	Short Address Attack	Tx.origin	Call-Stack Overflow	Timestamp Dependency	Transaction Order Dependency		
F* framework	Formal Verification	√	√	√	-[1]	-	-	-	√	√	-	Medium
EthIR		√	√	√	-	-	-	-	√	√	Python	Medium
EthBMC		√	-	-	-	-	-	-	√	√	Python	High
Oyente	Symbolic Execution	√	√	√	-	√	-	√	√	√	Python	Low
teEther		√	√	√	√	√	-	-	-	-	Python	Medium
Slither		√	-	√	-	-	√	√	√	-	Python	High
Manticore		√	√	-	-	√	-	-	-	-	Python	Medium
ContractFuzzer	Fuzzing	√	√	-	-	√	-	-	√	-	Go	Medium
sFuzz		√	√	-	√	√	-	-	√	-	C++	Medium
Harvey		√	√	-	√	√	-	-	√	-	Solidity	Medium
Sereum	Taint Analysis	√	√	√	-	-	-	-	√	√	Java	High
Ethainter		√	√	-	-	-	-	-	√	-	Solidity	High
Vaas	Integrated	√	√	√	-	-	√	-	√	-	Solidity	High
Mythril		√	√	-	-	-	√	-	√	-	Python	High
Securify		√	-	√	√	√	√	√	√	√	Solidity	Medium
Mythx		√	√	-	-	-	√	-	√	-	python	High

Table 32: Comparison of SCs Analysis Tools Based on Various Characteristics

Tool	Byte Code	Solidity Code	Static Analysis	Dynamic Analysis	Type of Tool	Implementation Language	Publicly Available
ContractLarva [103]	✗	✗	✗	✗	Academic	Haskell	✓
E-EVM [25]	✗	✗	✗	✗	Academic	Python	✓
EtherTrust [104]	✓	✗	✓	✗	Academic	java	✓
EthIR [105]	✓	✗	✓	✗	Academic	Python	✓
FSolidM [106]	✗	✗	✓	✗	Academic	Java Script	✓
Gasper [42]	✓	✗	✓	✗	*SDSLabs	Go	✓
KEVM [107]	✓	✗	✓	✗	Academic	Python	✓
MAIAN [59]	✓	✗	✓	✗	Academic	Python	✓
Manticore [108]	✓	✗	✓	✓	*Trail of Bits	Python	✓
Mythril [72]	✓	✗	✓	✗	*ConsenSys	Python	✓
Osiris [61]	✓	✗	✓	✗	Academic	Python	✓
Oyente [10]	✓	✗	✓	✗	Community	Python	✓
Porosity [109]	✓	✗	✓	✗	*Comae Technologies	C++	✓
Rattle [110]	✓	✗	✓	✗	*Trail of Bits	Python	✓
ReGaurd [76]	✓	✓	✗	✗	*Chieftin Lab	C++	✗
Remix-IDE [111]	✗	✓	✓	✓	Community	Java Script	✓
SASC [112]	✗	✓	✓	✗	*Fujitsu	Python	✗
sCompile [113]	✗	✓	✓	✗	Academic	C++	✗
Securify [71]	✓	✗	✓	✗	Academic	Java	✓
SmartCheck [69]	✗	✓	✓	✗	Academic	Java	✓
Solgraph [114]	✗	✓	✓	✗	*Raine Revere	Java Script	✓
SolMet [93]	✗	✓	✓	✗	Academic	Java	✓
teEther [26]	✓	✗	✓	✗	Academic	Python	✗
Vandal [115]	✓	✗	✓	✓	Academic	Python	✓
Zeus [116]	✗	✓	✓	✗	*IBM Research India	C++	✗

Table 33: Summary of Previous Surveys

Survey	Advantage	Disadvantage	Experimental Analysis	Theoretical Analysis	Comparative Analysis	Number of tools	Number of Relevant Approaches	Number of Irrelevant Approaches	Number of Relevant Vulnerabilities	Number of Irrelevant Vulnerabilities	Dataset
Harz et al. [13]	Various criteria for the comparison of tools	Focused on different factors related to SCs rather than tools	✗	✗	✓	10	2	✗	✗	✗	✗
Di Angelo et al. [16]	Comparison of the tools in various areas	Only mentioning 27 tools	✗	✓	✓	27	6	✗	17	✗	✗
Liu et al. [9]	Discussing verification methods thoroughly	Focusing on the methods rather than the tools Mentioning only 11 tools	✗	✓	✗	11	✗	2	6	✗	✗
Durieux et al. [1]	Comprehensive experimental evaluation on two datasets	The experiment is done only on 9 tools	✓	✗	✗	9	✗	✗	✗	✗	SmartBugs Dataset [20]
Almakhour M. et al. [4]	Describing the approaches used in tools	Focusing on the methods rather than the tools Mentioning only 13 tools	✗	✓	✓	13	6	✗	8	15	✗
Vivar et al. [14]	Comparison of tools in new criteria such as Ease of Installation, Usefulness, Stays Up to Date	Not covering every tool up to the publication time	✓	✗	✓	18	✗	3	✗	12	Unspecified datasets
Tolmach et al. [10]	Evaluating the effectiveness of tools in analysing SCs	Focusing on the approaches rather than the tools	✗	✓	✗	34	6	✗	✗	✗	✗
Zheng et al. [11]	Summarizes platforms and challenges in smart contract security	Brief descriptions without mapping tools to specific vulnerabilities	✗	✓	✗	12	✗	5	✗	5	✗
Vacca et al. [5]	A comprehensive comparison table with innovative assessments of various tools	Focusing on different aspects in coding of SCs rather than tools	✗	✓	✓	26	16	✗	✗	✗	✗
Hu et al. [12]	Comparison of 40 analysis tools, type and number of vulnerabilities they can detect	Not focused on the tools	✗	✓	✓	40	5	✗	5	✗	✗
Rameder et al. [15]	Discussing 140 analysis tools	A brief theoretical analysis for each tool	✗	✓	✓	140	12	✗	✗	54	✗
Wu et al. [6]	Provides a clear focus on understanding different types of vulnerabilities in SCs	Limited cross-platform focus and practical details	✗	✓	✗	32	9	✗	8	9	✗
Hewa et al. [7]	Thorough discussion of challenges and future potentials in blockchain-based SCs	Brief mention of some vulnerabilities and tools	✗	✓	✗	17	6	✗	6	4	✗
Kushwaha et al. [2]	A comprehensive comparison of tools Experimental performance evaluation of 30 tools	Not cover the high-level description related to the full flesh working of the tools Not mentioning the tools without a name	✓	✗	✓	86	2	✗	8	13	SolidiFI Benchmark (30 contracts) [21]
Kushwaha et al. [22]	Providing tables that detail vulnerabilities alongside their associated attacks, causes, and detection tools	Focusing on the vulnerabilities rather than tools	✗	✗	✓	27	2	✗	23	✗	✗
Li et al. [17]	Presenting a statistical analysis of the existing tools	Covering only 16 tools Focusing on comparisons without providing further explanation	✗	✗	✓	16	5	✗	9	✗	✗
Chu et al. [8]	Comparison of detection tool in various criteria such as Main technology, Assistive technology, Analysis level,adv & dis adv	A brief theoretical analysis on only 20 tools	✗	✓	✓	20	5	✗	✗	12	✗
Lashkari et al. [3]	Domain-specific evaluation of tools	Limited non-energy domain focus	✓	✓	✗	9	✗	7	✗	5	Etherscan (40 contracts) [23] SmartBugs (20 contracts) [20]
This work	**Comprehensive evaluation with a new scoring framework and both theoretical and practical analysis**	**Limited automation in evaluation process**	✓	✓	✓	**256**	**13**	✗	**239**	✗	**HajiHosseinkhani et al. [24]**

3 Evaluating the Available Smart Contracts Analyzers

Building on the papers identified in Subsection 2.1.2, this thesis analyzes 256 tools proposed for SCs vulnerability detection and identification. Detection tools are designed to uncover potential weaknesses and ensure the security and correctness of SCs code. These tools apply a variety of techniques, including fuzzing, machine learning, symbolic execution, and formal verification, to carry out their analyses. Identification tools extend this functionality by not only detecting vulnerabilities but also classifying and specifying their exact types. Using advanced techniques such as symbolic execution, static analysis, and machine learning, these tools provide in-depth insights into the specific nature of vulnerabilities, thereby helping developers understand and address the root causes of security issues in SCs code.
In this chapter, we categorize and describe these tools based on their methodologies and core capabilities. While some related studies focus on enhancing SCs testing efficiency through the development of test suites—such as the work by Górski [25]—their primary objective is optimization rather than vulnerability detection or classification. Therefore, such studies are excluded from this analysis, which centers exclusively on tools explicitly developed for vulnerability detection and identification.

3.1 Overview of SCs Analysis Tools by Methodology

In this thesis, the reviewed tools are organized based on the methodologies employed in each paper to detect or identify SCs vulnerabilities. To achieve this, we examined the underlying techniques used across all tools and grouped them according to methodological similarities, resulting in 14 distinct categories. These categories include abstract interpretation, AI-based methods, code instrumentation, control flow analysis, disassembly analysis, formal verification, fuzzing, model-based testing, mutation testing, pattern matching and syntactical analysis, runtime verification, symbolic execution, taint analysis, and visualization analysis. Within each category, tools are further divided based on their primary function—either detection or identification. For instance, tools using formal verification are classified as either formal verification–detection or formal verification–identification tools, depending on their capabilities.

3.1.1 "Abstract Interpretation" Tools

This category involves the use of abstract models to represent program behavior and analyze its properties with the aim of identifying potential vulnerabilities. Through mathematical abstraction, these tools can systematically explore a wide range of execution paths, enabling the detection of issues that might be missed by traditional testing methods. Among the tools reviewed under abstract interpretation, the first two are focused on detection, while the remainder support vulnerability identification in SCs.

1. **Asparagus–detector** [26]: It synthesizes parametric gas upper bounds for SCs using polyhedral and real algebraic geometry. It outperforms existing methods, such as the Gas-Aware SCs Analysis Platform (GASTAP), in terms of applicability and bound tightness. Asparagus, published in 2023, detects vulnerabilities including "gas limit underestimation," "out-of-gas exceptions," and "excessive gas consumption in loops."

2. **MichelsonLiSA–detector** [27]: Introduced in 2024, it is a static analysis tool based on abstract interpretation designed for Tezos SCs. It uses the Library for Static Analysis (LiSA) framework to analyze the Michelson language, translating its low-level stack-based code into a high-level Intermediate Representation (IR). This tool supports various analyses, such as taint and data flow analyses, to identify "untrusted cross-contract invocations" in SCs by examining Control Flow Graphs (CFGs) and semantic checkers.

3. **Gastap–identifier** [28]: Released in 2019, Gastap is a gas-aware analysis platform that prevents "out-of-gas" vulnerabilities by automatically inferring gas upper bounds for public functions in Ethereum SCs. It inputs Solidity source code or Ethereum Virtual Machine (EVM) bytecode and uses CFG construction, size analysis, and gas equation generation to provide precise gas consumption bounds. Gastap is effective for debugging, verifying, and certifying gas usage and is available via a web interface.

This category provides an effective approach to analyzing SCs by using mathematical models to approximate and explore a wide range of possible execution paths. Abstract interpretation facilitates the detection and identification of different vulnerability types, including issues related to gas consumption and unhandled exceptions. The tools discussed in this category demonstrate how this technique can uncover subtle and complex risks, contributing to the development of more secure SCs.

3.1.2 "AI-based" Tools

The second category consists of tools that leverage AI-based techniques for SCs vulnerability analysis. This methodology applies advanced artificial intelligence algorithms—particularly machine learning and deep learning models—to detect and identify vulnerabilities with high accuracy and efficiency. By recognizing patterns and anomalies in SCs behavior, these tools can uncover potential security risks that might be difficult to detect through traditional approaches. In this category, the first twenty-four tools are designed for vulnerability detection, while the remaining thirty-three are used for vulnerability identification in SCs.

1. **Block-Gram–detector** [29]: This tool extracts low-dimensional, knowledgeable features from Ethereum bytecode to improve vulnerability detection efficiency. It converts bytecode to opcodes, segments them, and mines block and attribute features. This method significantly reduces detection latency and enhances feature interpretability using Shapley additive explanations (SHAP) values. Blockgram, released in 2023, can detect various vulnerabilities, including "integer overflow and underflow," "call stack depth attack," "Transaction-Ordering Dependence (TOD)," "timestamp dependency," and "reentrancy vulnerabilities."

2. **Cross-Modality Mutual Learning Vulnerability Detector–detector** [30]: This framework was developed in 2023 to improve SCs vulnerability detection by using a teacher network trained on both source code and bytecode and a student network trained on bytecode alone. This mutual learning approach combines multiple modalities for more accurate identification of vulnerabilities, including "reentrancy," "timestamp dependence," "integer overflow/underflow," and "delegatecall."

3. **CodeNet–detector** [31]: Published in 2022, it is a convolutional neural network (CNN)-based architecture for SCs vulnerability detection. It transforms SCs into images while preserving their semantics and context. CodeNet employs a unique data pre-processing method and specialized convolutions to detect vulnerabilities, including "reentrancy," "unchecked low-level calls," "timestamp dependency," and "tx.origin."

4. **ContractArmor–detector** [32]: It was published in 2024 as a tool for generating attack surfaces in Solidity SCs, combining a rule-based engine and ChatGPT API for security analysis. ContractArmor identifies vulnerabilities through numerical values, key variables, and complex queries, which are evaluated on real-world contracts.

5. **Deep Learning-Based Malicious SCs Detection Scheme–detector** [33]: Launched in 2022, it uses Long Short-Term Memory (LSTM), Artificial Neural Network (ANN), and Gated Recurrent Unit (GRU) models to classify SCs as safe or malicious. It converts bytecode to opcodes, applies one-hot encoding, and generates feature vectors for classification.

6. **Deep Learning-Based Vulnerability Detection Framework–detector** [34]: This framework uses ANN, autoencoders, and multi-label classification models to identify vulnerabilities in SCs, specifically "reentrancy," "Denial of Service (DoS)," and "transaction origin" vulnerabilities. Developed in 2023, it extracts features from Solidity code's abstract syntax trees (ASTs) and applies deep learning techniques.

7. **Deep Learning and Expert Rules-Based Vulnerability Detection Mechanism–detector** [35]: This tool combines deep learning with expert rules to improve the detection of vulnerabilities in Ethereum SCs. Published in 2023, it uses Graph Neural Networks (GNNs) for initial detection and expert rules to verify and block risky transactions at the EVM level. The framework detects vulnerabilities such as "reentrancy," "call with hardcoded gas amounts," "timestamp dependency," and "code injection."

8. **DeepInfer–detector** [36]: It is a deep learning-based framework for inferring function signatures and returns from EVM bytecode. Published in 2023, It lifts the bytecode into an IR to preserve semantics and uses GNNs to extract type-related knowledge.

   DistilBERT-MLP/LSTM Reentrancy Detector–detector [37]: This tool, developed in 2023, uses a custom tokenizer and DistilBERT-based models for detecting "reentrancy" vulnerabilities in SCs. The model undergoes three stages: tokenization using a custom vocabulary, pre-training on a masked language model, and fine-tuning with Multilayer Perceptron (MLP) and Long Short-Term Memory (LSTM) for binary classification.

9. **Dynamit–detector** [38]: It is a machine-learning framework designed in 2021 for detecting "reentrancy" vulnerabilities in Ethereum SCs. It monitors transactions and extracts features, such as gas usage and balance differences, to classify them as benign or harmful. Dynamit employs a random forest classifier to analyze transaction metadata without requiring source code or instrumentation.

10. **EA-RGCN–detector** [39]: It is a novel graph convolutional network model for SCs vulnerability detection published in 2023 that constructs a semantic graph for each function. It extracts content and semantic features using residual graph convolutional networks and edge attention modules to identify vulnerabilities, including "arithmetic," "reentrancy," "timestamp dependency," and "unchecked low calls."

11. **Ensemble Models-Based Digital Forensic Framework–detector** [40]: It is a novel methodology introduced in 2023 using natural language processing and machine learning. It leverages a hybrid system combining feature extraction with ensemble modeling to classify vulnerabilities, including "DoS", "access control", "arithmetic integer overflow", "bad randomness", "reentrancy", "unchecked low-level calls", and "timestamp dependence". The framework enhances accuracy and minimizes false positives through techniques such as Synthetic Minority Oversampling Technique (SMOTE) sampling and Term Frequency-Inverse Document Frequency (TF-IDF), as well as Continuous Bag of Words (CBOW) and Skip-N Gram models for feature extraction.

12. **Extended Multimodal AI Framework–detector** [41]: Developed in 2023, this tool uses a multimodal AI approach for vulnerability detection in Ethereum SCs. It combines static analysis and various AI models to create a comprehensive framework. The tool leverages source code, build-based data, and Ethereum Virtual Machine (EVM) bytecode. It uses models such as bidirectional Long Short-Term Memory (bi-LSTM) with self-attention, textCNN, and random forest for both training and inference. The framework supports contracts with and without source code, excelling in feature fusion techniques.

13. **HGAT–detector** [42]: Hierarchical Graph Attention Network (HGAT) leverages a hierarchical graph attention network to detect SCs vulnerabilities by abstracting functions into code graphs using AST and CFG. Published in 2023, it extracts node features using graph attention mechanisms and splices vectors to detect vulnerabilities such as "reentrancy," "timestamp dependency," "integer overflow and underflow."

14. **Integrated DL-Based Vulnerability Detector–detector** [43]: Introduced in 2023, it employs a two-step hierarchical approach to enhance feature extraction for vulnerability detection. The first step uses a transformer for opcode relationship extraction and a Bidirectional Gated Recurrent Unit (Bi-GRU) for aggregating sequential information. The second step utilizes Text-CNN and spatial attention to capture local features, emphasizing significant semantics for improved vulnerability detection. It can detect 13 types of vulnerabilities in SCs. These vulnerabilities include "reentrancy", "access control," "arithmetic integer overflow/underflow," "unchecked return values for low-level calls," "denial of service," "bad randomness," "front running (TOCTOU)," "time manipulation," "short addresses," "call stack," "mishandled exceptions, ""transaction-ordering dependence," and "unprotected usage of self-destruct."

15. **ML-Based SCs Vulnerability Detector–detector** [44]: Launched in 2023, this model uses machine learning to identify valid and invalid SCs. By employing models like k-Nearest Neighbors (KNN),

Naive Bayes, Support Vector Machine (SVM), and Random Forest, it detects transaction-related vulnerabilities, including "transaction origin", "check-effects-interaction," "potential reentrancy bugs," "inline assembly," "block timestamp," "low level calls," "block hash," and "selfdestruct dealers".

16. **MSgram–detector** [45]: Multi-Semantic gram (MSgram) enhances SCs vulnerability auditing by generating sequences in three tokenization standards: text, structure, and combined sequences. MSgram, developed in 2020, utilizes an N-gram language model to capture multiple semantic contexts and employs intersection and Union strategies to integrate auditing results from different semantic perspectives. It can detect vulnerabilities including "reentrancy", "timestamp dependency", "integer overflow/underflow", "unchecked low-level calls", "self-destruct usage", "call stack depth", and "tx.origin usage".

17. **Multimodal Decision Fusion Model–detector** [46]: This method employs deep learning and multimodal decision fusion to detect vulnerabilities in SCs. It extracts features from the source code, operation code, and control-flow graph, integrating them through multimodal decision fusion. Developed in 2023, The tool detects"arithmetic vulnerabilities", "reentrancy", "TOD", and "locked ether vulnerabilities"

18. **SC-Defender–detector** [47]: Published in 2023, SC-Defender is designed for detecting vulnerabilities in Internet of Things (IoT)-enabled SCs. It utilizes the Abstract Syntax Tree (AST) of Solidity code and a Tree-Based Convolutional Neural Network (TBCNN) for vulnerability detection, with a particular focus on "reentrancy" flaws. The tool incorporates an AST pruning technique to eliminate redundant nodes and improve detection efficiency.

19. **S-HGTNs–detector** [48]: Spatial Heterogeneous Graph Transformer Networks (S-HGTNs), developed in 2022, is an anomaly detection model designed for SCs fraud on Ethereum. It constructs a Heterogeneous Information Network (HIN) and employs transformer networks to automatically generate meta-paths, enabling the detection of financial fraud, illegal financing, and money laundering.

20. **Semantic ML-Based Reentrancy Detector–detector** [49]: Published in 2022, It utilizes machine learning to detect "reentrancy" vulnerabilities in SCs by analyzing their semantic structure. The tool identifies vulnerabilities and provides correction feedback by combining AST data with machine learning models.

21. **SG-EA-RGCN Vulnerability Detector–detector** [50]: Signed Graph Entity Alignment Relational Graph Convolutional Network (SG-EA-RGCN) Vulnerability Detector is a graph-based tool designed for SCs vulnerability detection. It constructs semantic graphs and uses residual GCNs with edge attention to analyze and identify vulnerabilities, including "reentrancy", "timestamp dependence", "tx.origin usage", "unchecked send", "unhandled exception", "arithmetic vulnerabilities (overflow/underflow)", and "TOD". Developed in 2023, this approach allows for precise detection by focusing on the relationships and interactions within the SCs' code.

22. **SVScanner–detector** [51]: Smart Vulnerability Scanner (SVScanner), published in 2023, uses deep learning techniques to detect vulnerabilities in Ethereum SCs. It combines code token sequences and AST features using a TextCNN for effective vulnerability detection. It detects "reentrancy" vulnerabilities, "integer bugs," and "timestamp dependence" vulnerabilities.

23. **Transaction-based analysis with LSTM network–detector** [52]: This approach, published in 2021, focuses on classifying and detecting malicious Ethereum SCs based on transaction behavior. It uses an LSTM network to train a model on features extracted from contract transactions, including balance changes and ether flow correlations. The tool detects Ponzi schemes, gambling, and high-risk contracts based on transaction behavior.

24. **ABCNN–identifier** [53]: ABCNN, introduced in 2021, is an attention-based CNN model designed for detecting SCs vulnerabilities. It combines CNN with self-attention mechanisms to improve detection accuracy and speed. The model preprocesses opcode sequences and uses feature extraction and classification layers to identify vulnerabilities like "reentrancy," "arithmetic issues," and "time manipulation."

25. **ASSBert–identifier** [54]: ASSBert, introduced in 2023, employs Active and Semi-Supervised (ASS) learning combined with the Bidirectional Encoder Representations from the Transformers (BERT) model. It efficiently labels data using active learning and improves model performance with semi-supervised learning. It addresses vulnerabilities like "timestamp dependence," "call depth issues," "reentrancy," "transaction-ordering dependence," "arithmetic errors," and "tx.origin usage."

26. **AWD-LSTM–identifier** [55]: Average Stochastic Gradient Descent Weight-Dropped LSTM (AWD-LSTM), introduced in 2020, uses a variant of the LSTM model to classify SCs such as Suicidal, Prodigal, Greedy, and Normal. The model employs a pre-trained encoder inspired by natural language processing techniques to improve classification efficiency. Analyzing opcode sequences addresses vulnerabilities like "invocation depth" issues.

27. **AMEVulDetector–identifier** [56]: Attentive Multi-Encoder Vulnerability Detector (AMEVulDetector), proposed in 2021, combines deep learning with expert patterns for detecting SCs vulnerabilities including "reentrancy," "block timestamp dependence," and "infinite loops." It employs automatic tools to extract expert patterns, constructs a semantic graph for the code, and uses an attentive multi-encoder network to fuse graph features and patterns, providing interpretable weights for feature importance.

28. **BiGAS Detection Model–identifier** [57]: BiGAS, proposed in 2024, combines a bidirectional gated recurrent unit (BiGRU) with an attention mechanism and support vector machine (SVM) to detect "reentrancy" vulnerabilities in SCs. It processes SCs into token sequences and uses a functional model to extract and classify features.

29. **Bi-GRU with Attention Vulnerability Detector–identifier** [58]: Proposed in 2023, this tool employs a BiGRU with an attention mechanism to detect vulnerabilities in Ethereum SCs. It uses opcode

sequence extraction and vector representation to focus on 'reentrancy" and "timestamp dependency" vulnerabilities. The model captures contextual information and highlights the most relevant parts of the input sequence.

30. **Blass–identifier** [59]: Blass, introduced in 2023, uses a Bi-LSTM with an attention mechanism to classify SCs vulnerabilities. It targets issues such as "reentrancy", "integer overflow", "timestamp dependence", and "dangerous delegatecall." By constructing program slices with complete semantic code structure features, Blass significantly improves vulnerability detection compared to traditional methods.

31. **CBGRU–identifier** [60]: Convolutional Bidirectional Gated Recurrent Unit (CBGRU), proposed in 2022, combines CNN and BiGRU with Word2Vec and FastText word embeddings to detect multiple SCs vulnerabilities. It focuses on vulnerabilities including "infinite loop", "reentrancy", "timestamp dependency", "call stack depth attack", "integer overflow", and "integer underflow". The hybrid model enhances feature extraction and classification by integrating the capabilities of convolutional and recurrent neural networks.

32. **CGE–identifier** [61]: Contract Graph Embedding (CGE), proposed in 2023, combines GNN with expert knowledge for SCs vulnerability detection. It constructs contract graphs to represent control and data flow semantics, highlighting key nodes through a normalization process. The tool detects "reentrancy," "timestamp dependence," and "infinite loop" vulnerabilities, by integrating temporal message propagation and security pattern features.

33. **Cider–identifier** [62]: Cider, proposed in 2022, employs reinforcement learning to infer contract invariants specifically for proving arithmetic safety in SCs. It formulates the invariant generation problem as a Markov Decision Process (MDP) and uses a neural policy to predict valuable invariants. The tool enhances the verification process by reducing runtime assertions and improving the quality of inferred invariants, specifically targeting 'arithmetic overflow' vulnerabilities.

34. **ContractWard–identifier** [63]: Proposed in 2020, ContractWard detects six types of vulnerabilities: "integer overflow," "integer underflow," "TOD," "callstack depth attack," "timestamp dependency," and "reentrancy." It employs bigram features extracted from simplified operation codes and utilizes five machine learning algorithms, including XGBoost, to detect vulnerabilities. ContractWard is also designed to detect rapid batches.

35. **Deep Learning-Based Vulnerability Detector–identifier** [64]: Proposed in 2022, this tool employs deep learning algorithms such as CNN, LSTM, CNN-BiLSTM, and residual networks for detecting SCs vulnerabilities. It focuses on vulnerabilities including "integer overflow", "integer underflow", "callstack depth attack", "TOD", "timestamp dependency", and "reentrancy". The tool enhances detection performance by using unigram and bigram feature extraction and optimizing with modified term frequency-inverse document frequency.

36. **DeeSCVHunter–identifier** [65]: Developed in 2021, Deep SCs Vulnerability Hunter (DeeSCVHunter) is a deep learning-based framework for detecting "reentrancy" and "time dependence" vulnerabilities in SCs. It uses Vulnerability Candidate Slice (VCS) to enhance detection by leveraging data and control dependencies. The framework includes models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) to learn patterns of vulnerabilities. The tool's code and datasets are publicly available.

37. **DL4SC–identifier** [66]: DL4SC, published in 2024, is a deep learning-based framework for detecting vulnerabilities in SCs. It combines transformer encoders and CNNs to analyze opcode sequences, detecting "reentrancy," "arithmetic," and "timestamp dependence" vulnerabilities. The tool optimizes hyperparameters using the Sparrow Search Algorithm (SSA). The dataset and tool are publicly available.

38. **DR-GCN and TMP–identifier** [67]: Developed in 2020, these tools utilize GNNs for SCs vulnerability detection. They create contract graphs that capture both syntactic and semantic aspects of the contract. DR-GCN employs degree-free convolutions on these graphs, while TMP uses temporal message propagation to maintain temporal data relationships. Their methods identify vulnerabilities including "reentrancy", "timestamp dependence", and "infinite loops".

39. **ESCORT–identifier** [68]: Introduced in 2023, Efficient SCs Optimization and Risk Tool (ESCORT) is a deep learning tool for detecting vulnerabilities in Ethereum SCs. It uses a standard feature extractor and multiple branches to detect various vulnerability types, including "reentrancy", "integer overflow," "integer underflow," "callstack depth attack," "transaction-ordering dependence," "timestamp dependency," "unchecked send," "tx.origin usage", "assert violation," "accessible selfdestruct", and "arbitrary jump with function type variable" concurrently. ESCORT operates on bytecodes and supports transfer learning to add new vulnerability types with minimal data, reducing the need for multiple tools and minimizing detection time.

40. **Eth2Vec–identifier** [69]: Released in 2020, Ethereum Bytecode to Vectors (Eth2Vec) is a machine learning-based static analysis tool that analyzes code similarity to detect vulnerabilities in Ethereum SCs. It automatically learns features of EVM bytecodes using a neural network for natural language processing, making it robust against code rewrites.

41. **HAM–identifier** [70]: Proposed in 2023, the Hybrid Attention Mechanism (HAM) model utilizes a combination of single-head and multi-head attention mechanisms to detect vulnerabilities in SCs. By extracting code fragments that focus on key vulnerability points, HAM improves detection accuracy for "reentrancy," "arithmetic vulnerabilities," "unchecked return values," "timestamp dependency," and "tx.origin" issues. Public datasets were used to demonstrate its effectiveness.

42. **Machine Learning-Based Vulnerability Detection Scheme–identifier** [71]: This tool, presented in 2020, focuses on detecting three specific vulnerabilities in Ethereum SCs: "has-short-address," "has-flows," and "is-greedy.". It utilizes a novel slicing matrix method to extract features, improving

vulnerability detection accuracy. The tool utilizes various machine learning models, including neural networks and random forests, demonstrating its effectiveness on publicly available datasets from GitHub.

43. **MANDO-GURU–identifier** [72]: This open-source tool is designed for vulnerability detection in SCs, utilizing heterogeneous graph embeddings to analyze control-flow and call graphs. Released in 2022, MANDO-GURU identifies seven types of vulnerabilities, including "reentrancy," "front running," "arithmetic bugs," "control flow issues," "call graph issues," "symbolic execution bugs," and "data flow vulnerabilities" at both line-level and contract-level.

44. **MODNN–identifier** [73]: Multiple-Objective Detection Neural Network (MODNN) is a machine learning-based tool developed in 2022 for detecting vulnerabilities in SCs. It utilizes the Crucial Operation Sequence (COS) for feature extraction, enabling it to identify known and unknown vulnerabilities through explicit and implicit features. MODNN can detect 12 types of vulnerabilities, including "integer overflow and underflow", "callstack depth attack," "TOD," "timestamp dependency," "reentrancy," "unchecked send," "tx.origin", "assert failure," and "block timestamp," and supports the parallel detection of multiple vulnerabilities, improving scalability and reducing training costs.

45. **Multi-Task Learning Vulnerability Detection Model–identifier** [74]: This model employs a multi-task learning framework to detect and identify vulnerabilities in SCs. Introduced in 2022, it utilizes a shared bottom layer for learning semantic information and task-specific layers with convolutional neural networks (CNNs) for detection and classification. The model effectively handles vulnerabilities such as "arithmetic errors", "reentrancy", and "unknown addresses."

46. **RLRep–identifier** [75]: RLRep is a reinforcement learning-based tool, introduced in 2024, for providing repair recommendations for SCs vulnerabilities. It uses an encoder-decoder model and policy gradient algorithm to suggest fixes without requiring extensive labeled data. The tool effectively identifies and recommends repairs for vulnerabilities including "exception disorder", "integer overflow", "reentrancy", "TOD", and "tx.origin".

47. **S-gram–identifier** [76]: Introduced in 2020, S-gram is a semantic-aware security auditing tool for Ethereum SCs. It leverages a novel S-gram model that captures the semantic patterns in contract code to detect vulnerabilities. By analyzing token sequences and ranking potential vulnerabilities, S-gram provides a comprehensive security audit, identifying complex vulnerabilities including "reentrancy", "integer overflow/underflow", "timestamp dependency", and "unchecked low-level calls" that traditional syntax-based tools may miss.

48. **SCLMF–identifier** [77]: Introduced in 2023, the Semantic Classification and Meta-Learning Framework (SCLMF) is a meta-learning-based framework for detecting vulnerabilities in Ethereum SCs. It transforms bytecode into RGB images and employs a learner-meta-learner architecture to perform few-shot learning. The model accurately identifies vulnerabilities with limited data, utilizing convolutional neural networks (CNNs) and Model-Agnostic Meta-Learning (MAML) algorithms for effective classification.

49. **SCVDIE–identifier** [78]: SCs Vulnerability Detection with Integrated Ensembles (SCVDIE) is a machine learning-based tool introduced in 2022, leveraging an ensemble of neural networks to detect vulnerabilities in Ethereum SCs. It combines models such as CNN, RNN, and transformer to enhance detection accuracy, mainly targeting vulnerabilities including "integer underflow and overflow", "call-stack depth attack," "TOD," "timestamp dependency," and "reentrancy."

50. **SecBERT–identifier** [79]: Security-enhanced Bidirectional Encoder Representations from Transformers (SecBERT) uses a BERT-based architecture for multi-label vulnerability detection in SCs. Introduced in 2023, it uses the pre-trained SecBERT model to extract features from the bytecode and employs an MLP for classification.

51. **sGuard+ –identifier** [80]: sGuard+, introduced in 2020, is a machine learning-guided automated vulnerability repair tool. It utilizes binary classification models to identify each type of vulnerability at the function level, based on features extracted from both the source code and bytecode of SCs. The tool refines and extends the repair rules of sGuard, preserving the original business logic and reducing gas overhead.

52. **SmartConDetect–identifier** [81]: SmartConDetect is a static analysis tool for detecting security vulnerabilities in Solidity SCs. Released in 2023, it employs a pre-trained BERT model to analyze code fragments and identify susceptible code patterns. This tool can detect twenty-three vulnerabilities including "gas exhaustion", "unchecked function call", "misuse of view functions", "misuse of pure function", "this.balance equality check point", "incorrect return type", "msg.value zero", "misuse of visibility," "array length manipulation," "use of insecure math functions," "redundant fallback reject," "locked ether," "data leakage when using private," "misuse of approve function in ERC20 library", "mistake in solidity compiler version 0.5.0", "misuse of var," "misuse of multiple return values in internal and private function," "misuse of the transfer function in the loop," "misuse of inline assembly," "hardcode of the address," "deprecated constructions," "false return of ERC20" and "misuse of revert require".

53. **SmartEmbed–identifier** [82]: SmartEmbed is a tool designed for clone and bug detection in SCs through structural code embedding. Launched in 2019, it utilizes deep learning and similarity-checking techniques to efficiently and accurately identify code clones and clone-related bugs. The tool uses code embedding vectors for vulnerability detection.

54. **SmartMixModel–identifier** [83]: SmartMixModel is a machine learning-based vulnerability detection model for Solidity SCs. Released in 2022, it utilizes high-level syntactic and low-level bytecode features to enhance detection accuracy. The model identifies vulnerabilities including "unsafe array's length manipulation", "costly loop," "locked money," "using tx.origin for authorization", "checking for strict balance equality," "redundant fallback function," "hardcoded address," "extra gas consumption," "send instead of transfer," and "ETH transfer inside the loop."

55. **Sliced-JGNN–identifier** [84]: Sliced Joint Graph Neural Network (Sliced-JGNN) employs a GNN to detect vulnerabilities in SCs, introduced in 2023. It combines ASTs, control flow graphs, and program

dependency graphs. It uses program slicing to eliminate irrelevant information, effectively identifying vulnerabilities including "block information dependency", "dangerous delegatecall", "integer overflow", "re-entrancy", "suicide contract", "short address attack", "timestamp dependency", "TODy" and "unchecked call return value".

56. **TP-Detect–identifier** [85]: Trigram Pixel-Detect (TP-Detect) is a machine learning-based vulnerability detection tool for Ethereum SCs that uses trigram feature extraction and pixel value extraction to create a comprehensive dataset. Introduced in 2023, it employs models like Naive Bayes and Random Forest to identify vulnerabilities including "integer underflow and overflow", "call stack depth attack", "transaction ordering dependency", "timestamp dependency" and "reentrancy vulnerability".

57. **VulDeeSmartContract–identifier** [86]: Vulnerability Deep SCs (VulDeeSmartContract) uses BiLSTM with an attention mechanism for "reentrancy detection" in Ethereum SCs. Developed in 2020, it analyzes contract snippets to capture essential semantic information and control flow dependencies.

In conclusion, AI-based tools offer a state-of-the-art approach for accurately identifying vulnerabilities in SCs. Through the use of advanced machine learning and deep learning techniques, these tools are capable of detecting complex security issues that may not be easily identified through conventional methods. The tools discussed in this category show both the strengths and limitations of AI-based methodologies in improving the security and reliability of SCs.

3.1.3 "Code Instrumentation" Tools

The third category includes tools that employ code instrumentation techniques. This approach involves modifying the source code or bytecode of SCs to insert additional logic for monitoring, analysis, or enforcing security constraints. By instrumenting the code, these tools can detect vulnerabilities, apply runtime protections, and validate contract behavior—all while preserving the original functionality of the SCs. Here are three detection tools developed for code instrumentation in SCs:

1. **HermHD–detector** [87]: Hermes High-Density (HermHD) is an automated security enhancement tool designed in 2023 to protect Ethereum SCs through code obfuscation. It employs six obfuscation patterns, including control flow flattening and various instruction-level techniques, to rewrite the bytecode without affecting functionality. HermHD enhances security by preventing reverse static analysis tools from cracking the contract.

2. **SmartShield–detector** [88]: SCs Rectification and Shielding System (SMARTSHIELD), developed in 2020, is an automated bytecode rectification system designed to fix three typical security bugs in SCs: "state changes after external calls," "missing checks for out-of-bound arithmetic operations," and "missing checks for failing external calls." By analyzing and transforming EVM bytecode while preserving semantics and optimizing gas usage, SMARTSHIELD ensures secure SCs deployment.

3. **SolAnalyser–detector** [89]: Published in 2019, SolAnalyser combines static and dynamic analysis for automated vulnerability detection in Solidity SCs. It uses code instrumentation with assertions,

automated input generation, and execution trace analysis to detect vulnerabilities. The tool supports eight different vulnerability types and includes a fault-seeding component to assess the effectiveness of its analysis. It detects vulnerabilities including "integer overflow/underflow," "division by zero," "timestamp dependency," "transaction origin misuse," "unchecked send," "repetitive call functions," and "out-of-gas conditions."

In summary, code instrumentation tools improve SCs security by injecting additional logic into the code for monitoring and enforcement purposes. The tools discussed above illustrate how varied approaches—such as obfuscation, bytecode rectification, and the combination of static and dynamic analysis—can be employed to detect and prevent critical vulnerabilities. These techniques effectively maintain the intended functionality of the contract while introducing robust layers of security.

3.1.4 "Control Flow Analysis" Tools

The fourth category focuses on control flow analysis. This methodology involves examining the execution structure of SCs by constructing control flow graphs and analyzing possible execution paths. Through this analysis, tools in this category can detect critical vulnerabilities and security risks that may compromise the intended behavior and security of the contract. Here are two detection tools developed for control flow analysis in SCs:

1. **MadMax–detector** [90]: MadMax is a static program analysis tool that detects gas-focused vulnerabilities in Ethereum SCs. It combines a control-flow analysis-based decompiler and declarative program structure queries. MadMax, developed in 2018, analyzes SCs to capture high-level concepts such as dynamic data structures and safely resumable loops, identifying vulnerabilities such as "unbounded mass operations," "wallet griefing," and "loop overflows" efficiently.

2. **WaLi–detector** [91]: WaLi (developed in 2022) is a control-flow-based analyzer for detecting security vulnerabilities in Wasm SCs. It constructs a control flow graph from the Wasm bytecode and identifies critical paths that may contain vulnerabilities. WaLi simulates a runtime environment using a Wasm virtual machine to trace these paths and detect vulnerabilities such as "access control vulnerabilities" based on predefined patterns.

Control flow analysis tools play a critical role in identifying vulnerabilities by examining execution paths within SCs. By generating and analyzing control flow graphs, these tools can detect key security issues such as gas-related inefficiencies and access control flaws. The tools discussed in this section highlight the effectiveness of control flow analysis in uncovering and addressing critical weaknesses in SCs.

3.1.5 "Disassembly Analysis" Tools

The fifth category includes tools that apply disassembly analysis. This methodology involves breaking down the bytecode of SCs into a more readable and analyzable form. By disassembling the bytecode, these tools enable a clearer understanding of the contract's internal operations and assist in the identification of potential

vulnerabilities. In addition to bytecode translation, many of these tools also generate control flow graphs and produce representations of the contract's logic, supporting comprehensive and low-level security analysis. The following tools use disassembly analysis, with the first two focusing on detection and the others on identification within software components.

1. **Octopus–detector** [92]: Octopus, developed in 2020, is an open-source security analysis framework for WebAssembly modules and blockchain SCs, including Ethereum, Bitcoin, EOS, and NEO. It offers bytecode disassembly, control flow, call flow analysis, and symbolic execution. Octopus provides a comprehensive study by generating control flow graphs and converting bytecode to a high-level Static Single Assignment (SSA) form.

2. **Porosity–detector** [93]: Porosity is an open-source decompiler for EVM bytecode developed in 2017. It translates EVM bytecode into readable Solidity syntax, aiding static and dynamic analysis. The tool enables the examination of compiled SCs, helping to identify vulnerabilities including "reentrant vulnerability/race condition", "call stack vulnerability," and "time dependence vulnerability" by generating more understandable code.

3. **EtherProv–identifier** [94]: Released in 2021, EtherProv is a provenance-aware tool that combines static and dynamic analysis to detect, analyze, and mitigate security issues, including "liquid ether", "re-entrancy" and "restricted writes" in Ethereum SCs. It uses CFG instrumentation and datalog queries for efficient security analysis. EtherProv identifies vulnerabilities across multiple contracts and transaction histories, offering real-time mitigation for deployed contracts.

Disassembly analysis tools provide valuable insights into SCs by translating complex bytecode into more understandable representations. These tools facilitate the detection and analysis of vulnerabilities by breaking down low-level code and generating control flow graphs. The tools covered in this section demonstrate how disassembly analysis enhances SCs security by enabling developers to better interpret and mitigate potential risks.

3.1.6 "Formal Verification" Tools

The sixth tool category revolves around formal verification and theorem proving. This methodology uses mathematical proofs and logical reasoning to ensure SCs adhere to their intended specifications and behavior. By translating SCs into formal models and verifying them against precise criteria, these tools help guarantee the contracts' correctness and safety. Of the tools developed for formal verification and theorem proving, the first eight are for detection and the remaining seventeen focus on vulnerability identification in SCs.

1. **Celestial–detector** [95]: It is a framework developed in 2021 for verifying Solidity SCs using $F^\star$. It translates contracts to $F^\star$ for formal verification against blockchain semantics and then erases specifications to generate deployable Solidity code. Celestial ensures functional correctness by automating the verification process and providing low entry barriers for developers.

2. **ConCert–detector** [96]: ConCert is a framework for verifying SCs using Coq, focusing on functional correctness and safety properties. It allows the extraction of verified contracts to executable blockchain code in languages like CameLIGO. ConCert, introduced in 2020, supports reasoning about contract interactions, making it suitable for complex decentralized applications. Using Coq's proof assistant capabilities helps detect vulnerabilities related to functional correctness and safety properties.

3. **FEther–detector**[97]: Formal Ether (FEther) is an extensible definitional interpreter published in 2019 for Ethereum SCs verification in Coq. It combines symbolic execution and higher-order logic theorem proving to ensure consistency between SCs and their formal models. FEther features automatic strategies for execution and verification, with verified functional correctness in Coq.

4. **HoRStify–detector** [98]: Hostile Resistant Static Analysis (HoRStify), developed in 2023, is a sound static analysis tool for Ethereum SCs, focusing on dependency analysis to verify security properties. It uses a formal proof framework for static program slicing and logical encoding with Datalog solvers. This ensures the detection of vulnerabilities like "timestamp dependency" and "single-entrancy" in SCs.

5. **NuSMV Model-Checking Framework–detector** [99]: Developed in 2018, this tool employs model-checking techniques to verify the correctness of Ethereum SCs. Translating Solidity code into the New Symbolic Model Verifier (NuSMV) input language and formalizing properties using temporal logic Computation Tree Logic (CTL) systematically inspects all possible state sequences to ensure compliance with specified behavioral properties. This framework helps detect vulnerabilities related to "state reachability," "safety properties," "liveness," and "functional correctness."

6. **PROMELA and SPIN Model-Checking Framework–detector** [100]: This framework, introduced in 2020, employs the Simple Promela Interpreter (SPIN) model checker to verify the correctness of SCs by translating Solidity code into Process Meta Language (PROMELA) models. It systematically checks the contract's logic against specified properties using assertions, deadlock detection, and linear temporal logic (LTL).

7. **SAFEVM–detector** [101]: It is a verification tool designed for Ethereum SCs, utilizing state-of-the-art verification engines for C programs. It decompiles EVM bytecode into C code with SV-COMP verification annotations, allowing verification by tools like CPAchecker, SeaHorn, and VeryMax. Introduced in 2019, SAFEVM efficiently handles "general safety annotations" and "array access verifications," providing a comprehensive analysis of SCs safety by transforming invalid bytecode operations into verifiable C program assertions.

8. **VeriMove–detector** [102]: VeriMove (2022) is a model-checking framework designed to verify Move SCs. Built on top of the VeriSolid framework, VeriMove extends its capabilities to the Move language, allowing for the verification of global properties across multiple function executions. The framework primarily targets vulnerabilities such as "reentrancy" and "integer overflow/underflow."

9. **2Vyper–identifier** [103]: 2Vyper is a 2021 Satisfiability Modulo Theories(SMT)-based automated verification tool for Ethereum SCs written in Vyper. It uses a novel specification methodology tailored to SCs, enabling sound and precise reasoning even in unverified code and arbitrary "re-entrancy." The tool supports modular reasoning about collaborating contracts. It includes domain-specific specifications for resources and resource transfers.

10. **EtherTrust–identifier** [104]: EtherTrust, proposed in 2018, is a sound static analysis tool for Ethereum bytecode. It uses a reachability analysis technique based on Horn-clause resolution to detect critical vulnerabilities like "reentrancy" and "transaction environment dependency."

11. **eThor–identifier** [105]: eThor, released in 2020, is a sound and automated static analyzer for Ethereum SCs. It employs a Horn-clause-based reachability analysis to detect security properties such as "single-entrancy." eThor provides formal security guarantees and supports the analysis of EVM bytecode, demonstrating precision in large-scale evaluations on real-world contracts.

12. **ESBMC-Solidity–identifier** [106]: Introduced in 2022, ESBMC-Solidity is an SMT-based model checker for Solidity SCs. It uses a new frontend to convert Solidity JSON AST into an IR for symbolic execution. The tool verifies various vulnerabilities, including "integer overflow", "integer underflow", "authorization through tx.origin", "static array out-of-bounds", and "dynamic array out-of-bounds", providing counterexamples for each detected issue.

13. **FSPVM-E–identifier** [107]: Formal Symbolic Process Virtual Machine for Ethereum (FSPVM-E), released in 2020, is a hybrid formal verification system for Ethereum SCs implemented in Coq. It combines symbolic execution and higher-order logic theorem proving using an FSPVM to ensure the reliability and security of SCs. FSPVM-E includes a formal memory framework, an intermediate programming language (Lolisa), and a formally verified interpreter (FEther). It can detect vulnerabilities such as "integer overflow", "stack overflow", "unchecked send bug", and "divide zero".

14. $F^\star$ **Verification–identifier** [108]: Released in 2016, $F^\star$ Verification is a formal verification tool that translates Ethereum SCs to the $F^\star$ functional programming language for comprehensive analysis. It supports Solidity source code and EVM bytecode, allowing for runtime safety and functional correctness verification through formal methods and relational reasoning.

15. **Predicate Abstraction-Based Validation Framework–identifier** [109]: This tool uses predicate abstraction to validate SCs by constructing finite labeled transition systems from contract code. Released in 2022, it supports auditors by identifying function call sequences and checking the correspondence between contract behavior and requirements. The tool can expose defects by using predicates based on required clauses and enum-type state variables.

16. **Securify–identifier** [110]: Securify is a practical and scalable security analyzer for Ethereum SCs, introduced in 2018. This static analysis tool uses compliance and violation patterns to verify security properties and detect violations efficiently. By symbolically encoding the contract's dependency graph, Securify helps identify issues such as "reentrancy", "unrestricted write to storage", "locked ether",

"TOD", "exception handling", "unchecked call return values", and "integer overflows/underflows" with minimal false positives.

17. **SmartFast–identifier** [111]: SmartFast is a formal analysis tool for Ethereum SCs. It introduces a novel intermediate representation, SmartIR, which uses preset rules and taint tracking to identify and locate vulnerabilities in contract code. Developed in 2022, SmartFast detects a wide range of vulnerabilities with high precision and recall rates. These vulnerabilities can be categorized into 15 groups, including "reentrancy and call vulnerabilities", "access control and authorization", "state management and initialization", "logic and computation issues", "mathematical errors", "contract design and standards", "security best practices", "SCs standards compliance", "gas optimization and cost management", "code quality and readability", "data handling and logic flow", "execution and functionality", "error handling and recovery", "assembly and low-level operations", and "miscellaneous issues".

18. **Solc-verify–identifier** [112]: Solc-verify is a source-level verification tool for Ethereum SCs. Released in 2020, it uses modular program analysis and SMT solvers to verify Solidity contracts by analyzing annotations directly in the source code. It detects various vulnerabilities such as "reentrancy," "overflows," and "assertion failures."

19. **Solicitous–identifier** [113]: Solicitous is a formal verification tool integrated into the official Solidity compiler. Introduced in 2020, it uses constrained Horn clauses (CHCs) to accurately model SCs behaviors, allowing fully automated verification using generic theorem provers. Solicitous excels in proving "unbounded safety properties" and generating counterexamples for property violations.

20. **Solidifier–identifier** [114]: Solidifier is a bounded model checker for Solidity SCs that uses lazy contract deployment and precise memory modeling. Introduced in 2021, it employs Boogie, an intermediate verification language, to encode Solidity and the Ethereum blockchain. This tool addresses vulnerabilities by exploring semantic properties rather than specific patterns, ensuring precise and well-formed state manipulation.

21. **Vandal–identifier** [115]: Released in 2018, Vandal is a security analysis framework for Ethereum SCs that converts low-level EVM bytecode to semantic logic relations. It uses datalog-based logic specifications to detect vulnerabilities, including "unchecked send", "reentrancy", "unsecured balance", "destroyable contract", and "use of origin". Vandal's logic-driven approach allows for easy customization and rapid prototyping of new vulnerability analyses.

22. **VERISMART–identifier** [116]: VERISMART is a safety verifier for Ethereum SCs, focusing on "arithmetic safety." Introduced in 2020, it uses a novel domain-specific algorithm to automatically discover transaction invariants, ensuring exhaustive verification without compromising precision or scalability. VERISMART effectively detects vulnerabilities like "integer overflows and underflows," reducing false positives.

23. **VERISOL–identifier** [117]: VERISOL is a formal verifier for SCs in the Azure Blockchain, introduced in 2019. It translates Solidity programs into the Boogie intermediate verification language,

leveraging the Boogie verification pipeline to perform semantic conformance checking and automatic verification. VERISOL has been used to find previously unknown bugs in contracts with Azure Blockchain Workbench.

24. **VeriSolid–identifier** [118]: VeriSolid is a framework for formally verifying Ethereum SCs, using a transition-system-based model with rigorous operational semantics. Launched in 2019, it allows developers to reason about and verify contract behavior at a high level of abstraction. VeriSolid enables the generation of Solidity code from verified models, facilitating the correct-by-design development of SCs. It can detect vulnerabilities such as "reentrancy", "TOD", "integer overflow/underflow", "access control issues", and "deadlock freedom".

25. **ZEUS–identifier** [119]: ZEUS is a formal verification framework designed to analyze the safety of SCs. Introduced in 2018, it combines abstract interpretation and symbolic model checking to verify the correctness and fairness of contracts. ZEUS is capable of checking for adherence to safe programming practices and business logic, detecting vulnerabilities including "reentrancy", "unchecked send", "integer overflow/underflow", "transaction state dependence", "block state dependence", and "TOD".

In summary, formal verification tools use mathematical methods to verify the correctness and security of SCs. By translating contracts into formal models, these tools help ensure that contracts perform as intended. The tools mentioned in this section highlight the importance of formal verification in building reliable and secure SCs.

3.1.7 "Fuzzing" Tools

The seventh category of tools emphasizes fuzz testing. Fuzz testing, or fuzzing, is a dynamic testing technique that involves providing invalid, unexpected, or random data as inputs to SCs to discover coding errors and security vulnerabilities. This method is particularly effective for identifying vulnerabilities and ensuring the robustness of SCs. Fuzzing tools often combine techniques such as symbolic execution and evolutionary fuzzing to generate meaningful transaction sequences and optimize mutation processes for thorough testing. The first twenty tools developed for fuzz testing focus on vulnerability detection, while the remaining ten are designed for vulnerability identification in SCs.

1. **ConFuzzius–detector** [120]: It is a hybrid fuzzer developed in 2021 for SCs, combining symbolic execution and evolutionary fuzzing. It uses constraint solving for complex conditions and dynamic data dependency analysis to generate meaningful transaction sequences. It detects vulnerabilities including "assertion failure", "integer overflow", "reentrancy", "TOD", "block dependency", "unhandled exception", "unsafe delegate call", "leaking ether", "locking ether", and "unprotected self-destruct".

2. **CrossFuzz–detector** [121]: It is a cross-contract fuzzing tool for detecting vulnerabilities in Ethereum SCs published in 2024. It generates constructor parameters by tracing data propagation paths and uses inter-contract data flow to optimize transaction sequence mutation. CrossFuzz can detect a range of vulnerabilities, including "reentrancy," "unhandled exceptions," "assertion failures," and "block dependency."

3. **Echidna–detector** [122]: It is a SCs fuzzer that uses random transaction generation to detect violations including “reentrancy”, “assertion violations”, “gas limit issues”, “integer overflows and underflows”, and “custom property violations” in Ethereum SCs. It integrates with Slither for initial contract analysis and then runs a fuzzing campaign, leveraging application binary interfaces and constants. Echidna, launched in 2020, excels in quickly identifying property violations, estimating gas usage, and supporting various development frameworks with minimal configuration.

4. **EFCF–detector** [123]: A high-performance fuzzer introduced in 2023 translates EVM bytecode to native C++ code for efficient fuzzing. It uses a structure-aware mutation engine for SCs transaction sequences and leverages the contract’s ABI to generate valid inputs. EFCF accurately models complex interactions, including “reentrancy”, “cross-contract interactions”, “access control bugs”, “integer overflows”, “delegatecall misuse”, and “compositional reentrancy”.

5. **EtherDiffer–detector** [124]: It performs differential testing on Remote Procedure Call (RPC) services of Ethereum nodes. It generates a non-deterministic chain using concurrent transactions and propagation delays. EtherDiffer, published in 2023, creates and executes test cases to detect deviations in error handling and return values, identifying bugs and inconsistencies including “invalid argument handling”, “gas estimation discrepancies”, “uncle block access inconsistencies”, “implementation bugs (e.g., crashes and DOS)”, “method support inconsistencies”, and “fields and formats variations”.

6. **EthPloit–detector** [125]: It is a SCs exploit generator that uses fuzzing combined with static taint analysis to generate exploit-targeted transactions. It employs a dynamic seed strategy and an instrumented EVM to handle blockchain behaviors and constraints. EtherDiffer detects vulnerabilities, including “crash bugs,” “DoS bugs,” and “incorrect gas estimation” for contract executions.

7. **EVMFuzzer–detector** [126]: It is a differential fuzz testing tool that detects vulnerabilities in EVM implementations. It generates and mutates seed contracts and runs them on multiple EVMs to identify execution discrepancies. It uses dynamic priority scheduling and benchmark EVMs as cross-referencing oracles to detect security issues efficiently. EVMFuzzer, developed in 2019, detects vulnerabilities including “illegal stack operations,” “DoS bugs,” and “segmentation faults.”

8. **FuzzDelSol–detector** [127]: Introduced in 2023, it is a binary-only coverage-guided fuzzer for Solana SCs. It models runtime specifics like SCs interactions and generates transactions to uncover vulnerabilities, including “missing signer checks”, “missing owner checks”, “arbitrary cross program invocation”, “missing key checks”, “integer bugs”, and “lamport theft”.

9. **GasFuzzer–detector** [128]: It is a fuzzing tool designed in 2020 to detect gas-oriented exception security vulnerabilities, including “exceptions disorder” and “out-of-gas” in Ethereum SCs. The tool employs a two-phase strategy: a gas-greedy strategy that prioritizes gas-heavy transactions for mutation and a gas-leveling strategy that manipulates gas allowances to expose vulnerabilities.

10. **HFContractFuzzer–detector** [129]: It is a fuzzing tool for detecting vulnerabilities in Hyperledger Fabric SCs. It employs go-fuzz and SCs written in Go, utilizing unit testing with the MockStub class

to simulate blockchain state. Developed in 2021, the tool optimizes the fuzzing process through initial corpus and mutation algorithm enhancements to effectively identify vulnerabilities, including "type conversion errors," "logic loopholes," and "integer overflow."

11. **Hydra–detector** [130]: Hydra is a framework designed in 2019 for principled, automated bug bounties on Ethereum SCs. It combines N-version programming with automated bug bounty payouts, running multiple versions of a program to detect and isolate bugs. A metaprogram coordinates actions to ensure security, incentivizing disclosure through bounty rewards. The vulnerabilities detected by Hydra include "logic errors," "reentrancy attacks," and other exploitable bugs by ensuring discrepancies between different versions of the program are identified and rewarded with bounties to encourage reporting.

12. **IcyChecker–detector** [131]: It is a fuzzing-based framework designed in 2023 to detect State Inconsistency (SI) bugs in SCs. It replays on-chain historical transactions to gather accurate contextual information, then generates and mutates transaction sequences to identify SI bugs by observing state differences. This approach effectively uncovers vulnerabilities like "state inconsistency bugs", "reentrancy", "front-running", "access control violations", and "bad randomness" in decentralized applications.

13. **Invariant-Based Scarcity Defect Detector–detector** [132]: Published in 2022, this method uses invariant analysis to detect scarcity defects in blockchain digital assets. It defines transfer and swap invariants as test oracles and employs the ConFuzzius fuzzing tool to generate test inputs, effectively identifying "scarcity defects" in SCs.

14. **ILF–detector** [133]: ILF, developed in 2019, is an imitation learning-based fuzzer for Ethereum SCs, combining symbolic execution and fuzzing techniques. ILF effectively detects vulnerabilities such as "leaking," "suicidal," "locking," "block dependency," "unhandled exception," and "controlled delegatecall." It learns a fuzzing policy from inputs generated by a symbolic execution expert to improve code coverage and vulnerability detection.

15. **ItyFuzz–detector** [134]: Published in 2023, ItyFuzz is a snapshot-based fuzzer for Ethereum SCs. It replaces transaction sequences with state snapshots to reduce re-execution overhead. The tool employs dataflow and comparison waypoints to prioritize interesting states, enabling fast and efficient vulnerability detection. ItyFuzz can identify vulnerabilities such as "reentrancy", "integer overflow", "timestamp dependence", "out-of-gas", "unhandled exceptions", and "short address attack".

16. **Language-Agnostic Fuzzing Framework–detector** [135]: This framework, introduced in 2022, uses a single fuzzer to detect vulnerabilities in SCs written in different programming languages by converting them to Low-Level Virtual Machine (LLVM) IR. It employs American Fuzzy Lop++ (AFL++), Honggfuzz, and libFuzzer to perform the fuzzing, enhancing scalability and maintainability for enterprise Development and Operations (DevOps) setups.

17. **MagicMirror–detector** [136]: MagicMirror is a high-coverage fuzzing tool designed in 2023 for SCs. It integrates dynamic symbolic execution with traditional fuzzing to enhance the detection of vulnerabilities. By leveraging dynamic taint analysis, MagicMirror ensures comprehensive vulnerability detection and effectively mitigates security issues such as "integer overflow", "timestamp dependency", "reentrancy", "unhandled exceptions", "out-of-gas", and "short address attacks" in SCs.

18. **ReDefender–detector** [137]: ReDefender is a fully automated dynamic analysis tool designed to detect "reentrancy vulnerabilities" in Ethereum SCs introduced in 2022. Using fuzz testing, it preprocesses contracts to create a candidate pool, generates fuzzing inputs to simulate attacks, and analyzes execution logs to verify vulnerabilities.

19. **SeqFuzz–detector** [138]: It is a guided mutation fuzzer published in 2023 for SCs testing that combines Dynamic Dependency Learning (DDL) and Dynamic Variables Analysis (DVA). DDL generates transaction sequences by analyzing state variable dependencies, while DVA handles external parameters using dynamic taint analysis. SeqFuzz enhances branch coverage and bug detection by focusing on mutations that are relevant to the code. The vulnerabilities detected by SeqFuzz include "arbitrary write," "block state dependency," "control hijack," "ether leak," "integer bug," "mishandled exception," "multiple send," "reentrancy," "suicidal contract," and "transaction origin use."

20. **SynTest-Solidity–detector** [139]: Developed in 2022, it is an automated test case generation and fuzzing framework for Solidity SCs. It employs various metaheuristic search algorithms, including random search and genetic algorithms like Non-dominated Sorting Genetic Algorithm II (NSGA-II) and Dynamic Many-Objective Sorting Algorithm (DynaMOSA), to optimize test case generation. The tool provides both a command-line interface and a web service, making it accessible and easy to use for developers.

21. **ContractFuzzer–identifier** [140]: Introduced in 2018, ContractFuzzer uses fuzzing techniques to test Ethereum SCs for vulnerabilities. It generates random inputs to execute contracts and applies predefined rules to identify vulnerabilities such as "reentrancy", "timestamp dependence", "integer overflow", "unchecked call", "exception disorder", and "out-of-gas". This approach helps identify potential security issues by simulating various attack scenarios.

22. **Etherolic–identifier** [141]: Etherolic, introduced in 2020, is a robust and efficient fuzzing tool for the security analysis of Ethereum SCs. It combines dynamic taint tracking and concolic testing to analyze bytecode, identify vulnerabilities, and generate exploits. Etherolic is capable of detecting a wide range of vulnerabilities such as "integer overflow and underflow", "bad randomness", "re-entrancy", "locked ether", "unhandled exceptions", "denial of service", "short address attack", "race condition", and "shadow memory", as well as zero-day attacks at the bytecode level, even without access to the source code, while minimizing false positives by recognizing protection methods in the code.

23. **GFuzzer–identifier** [142]: Released in 2022, GFuzzer is a grey-box fuzzing tool for EOSIO SCs. It uses execution feedback to guide fuzzing and improve branch coverage. GFuzzer detects vulnerabilities such as "fake EOS transfer," "forged transfer notification," and "block information dependency."

Employing a distance-based mutation strategy generates test cases that cover hard-to-reach branches, ensuring comprehensive security analysis.

24. **Harvey–identifier** [143]: Harvey, a grey-box fuzzier designed for SCs, was introduced in 2020. It enhances standard grey-box fuzzing by predicting new inputs likely to cover new paths or reveal vulnerabilities. Harvey also fuzzes transaction sequences to explore different contract states, aiming to detect vulnerabilities such as "assertion violations" and "memory-access errors." Public datasets were used to demonstrate their effectiveness.

25. **NeoDiff–identifier** [144]: NeoDiff is a differential fuzzing framework for SCs virtual machines. Released in 2021, it uses coverage-guided and state-guided fuzzing to explore VM behavior and uncover critical differences. NeoDiff has been applied to various blockchain platforms, including Ethereum and Neo, finding discrepancies and "memory corruption issues." The tool can be ported to new SCs platforms with ease.

26. **ReGuard–identifier** [145]: ReGuard is a fuzzing-based analyzer for automatically detecting "reentrancy bugs" in Ethereum SCs. It performs fuzz testing by generating random and diverse transactions, then dynamically identifies reentrancy vulnerabilities based on runtime traces. Released in 2018, ReGuard translates SCs code to C++ for comprehensive analysis.

27. **RLF–identifier** [146]: Reinforcement Learning Fuzzer (RLF), developed in 2022, is a reinforcement learning-guided fuzzing tool designed to generate vulnerable transaction sequences in Ethereum SCs. By modeling the fuzzing process as a Markov decision process, RLF effectively identifies complex vulnerabilities requiring specific sequences of transactions. It integrates both vulnerability and code coverage rewards to detect multiple types of vulnerabilities, including "ether leaking", "suicidal contract", "block state dependency", "unhandled exception", "dangerous delegatecall", and "ether freezing", especially those involving interactions across multiple functions.

28. **SMARTIAN–identifier** [147]: Symbolic Model Analysis Reasoning Tool for IoT and Applications in Networks (SMARTIAN) is a SCs fuzzer that enhances fuzzing through both static and dynamic data-flow analyses. Introduced in 2021, it systematically generates critical transaction sequences to find vulnerabilities such as "assertion failure", "arbitrary write", "block state dependency", "control-flow hijack", "ether leak", "freezing ether", "integer bug", "mishandled exception", "multiple send", "reentrancy", "requirement violation", and "suicidal contract" in SCs, utilizing data dependencies to guide fuzzing. This hybrid approach significantly improves bug detection and code coverage.

29. **sFuzz–identifier** [148]: sFuzz is an adaptive fuzzing tool for Solidity SCs on the Ethereum platform. It integrates AFL-based fuzzing with a lightweight, adaptive strategy to achieve high code coverage and discover vulnerabilities. This tool, which was introduced in 2020, effectively identifies issues including "gasless send", "exception disorder", "reentrancy", "timestamp dependency", "block number dependency", "dangerous delegatecall", "integer overflow/underflow", and "freezing Ether" by generating and executing numerous test cases through an optimized, feedback-guided process.

30. **WASAIUP–identifier** [149]: WASAIUP, introduced in 2023, is a demand-driven concolic fuzzer for EOSIO SCs. It detects specific vulnerabilities, including "fake EOS", "fake notification", "rollback", "missing authorization verification", and "blockinfo dependency". This hybrid tool combines symbolic and concrete execution to improve detection efficiency and accuracy.

Overall, fuzz testing tools offer an adaptive approach to identifying vulnerabilities in SCs by generating diverse and unpredictable inputs. Although fuzzing has limitations, such as requiring significant computational resources and potential gaps in coverage, it remains a potent method for testing contracts under unexpected conditions and edge cases. The tools highlighted here demonstrate the value of fuzz testing in improving the security of SCs.

3.1.8 "Model-Based Testing" Tools

The eighth category uses model-based testing. This approach involves creating abstract models of SCs to guide the generation of test cases and validate contract behaviors. Using formal models and state machines, these tools can systematically explore different execution paths and detect potential issues, ensuring thorough testing and validation of SCs. In the list of tools below, the first five tools leverage abstract interpretation for vulnerability detection, whereas the remaining six tools are used for vulnerability identification in SCs.

1. **ADF-GA–detector** [150]: Adaptive Fuzzing with Genetic Algorithm (ADF-GA) is a genetic algorithm-based approach for generating test cases for Solidity SCs. ADF-GA, introduced in 2020, constructs control flow graphs, performs data flow analysis to identify variable uses, and optimizes test generation with an improved fitness function. ADF-GA detects vulnerabilities such as "reentrancy bugs", "incorrect state transitions", and "integer overflow/underflow issues".

2. **FSolidM–detector** [151]: FSolidM, developed in 2018, is a framework designed for creating secure Ethereum SCs using Finite State Machines (FSMs). It includes a graphical editor for specifying Finite State Machines (FSMs), a code generator for producing Solidity code, and plugins to enhance security and functionality. However, since FSolidM focuses on helping developers create secure contracts rather than detecting vulnerabilities in existing contracts, it is not included in this thesis's theoretical and practical analysis of tools. FSolidM helps developers by providing a formal model, reducing manual coding errors, and integrating common security patterns like "reentrancy," "transaction ordering," and "access control."

3. **ModCon–detector** [152]: It is a model-based testing platform for SCs on permissionless and permissioned blockchains. It uses user-specified models to define test oracles, guide test generation, and measure test adequacy. With a web-based interface, ModCon supports customized testing processes, enabling thorough validation of complex statecharts (SCs) by specifying state definitions, transition relations, and pre- and post-conditions. ModCon, developed in 2020, detects vulnerabilities such as "state transition errors", "functional correctness issues", and security breaches in complex SCs applications.

4. **SCs modeling and behavior verification–detector** [153]: This tool uses a formal modeling approach based on behavior interaction priorities to verify SCs behaviors. Introduced in 2018, It models SCs, users, and blockchain interactions to identify potential vulnerabilities through statistical model checking. Simulating different attack scenarios provides insights into security breaches and suggests design improvements for robust SC implementations. It detects vulnerabilities, including "transaction reordering attacks", "pending transaction data leakage", and "network eavesdropping".

5. **SmartInspect–detector** [154]: It is a Solidity SCs inspector that uses decompilation techniques and a mirror-based architecture to represent and interpret contract states. Developed in 2018, it decompiles the contract's binary structure and maps it to its source code, allowing developers to inspect and understand the contract state without redeployment or additional code.

6. **Elysium–identifier** [155]: Elysium is a bytecode-level patching tool introduced in 2022 for automatically fixing vulnerabilities in SCs. It integrates Osiris, Oyente, and Mythril to detect and patch issues like "integer overflows", "reentrancy", and "unhandled exceptions". Using CFG and taint analysis, Elysium generates effective patches. The tool addresses seven types of vulnerabilities, including "same-function reentrancy", "cross-function reentrancy", "delegatecall misuse", "create-based reentrancy", "transaction origin", "suicidal contracts", and "integer overflows and underflows" and is available on GitHub.

7. **ESAF–identifier** [156]: Introduced in 2021, Ethereum Security Analysis Framework (ESAF) is a comprehensive framework that integrates multiple existing SCs vulnerability analysis tools. It simplifies the analysis process by unifying the output formats and managing dependencies through containerization. ESAF supports persistent security monitoring and individual vulnerability analysis, leveraging the combined capabilities of tools like Oyente, Mythril, and Securify.

8. **SecSEC–identifier** [157]: Introduced in 2024, Securing Smart Ethereum Contracts (SecSEC) integrates multiple SCs analysis tools using logistic regression to enhance vulnerability detection. This hybrid method combines the strengths of various tools and trains on a large dataset of real-world contracts labeled with vulnerabilities.

9. **SmartBugs–identifier** [158]: SmartBugs is an extensible execution framework designed to simplify the analysis of Solidity SCs. Developed in 2020, it supports multiple analysis tools and datasets, enabling reproducible research and comprehensive vulnerability detection. SmartBugs integrates tools like HoneyBadger, Maian, and Mythril and provides detailed reports on various vulnerability types such as "reentrancy", "integer overflow and underflow", "unchecked send", "denial of service (DoS)", "timestamp dependence", "TOD/front-running", "uninitialized storage pointers", "tx.origin authentication", "locked ether", "access control issues", and "bad randomness" using Docker images and a user-friendly command-line interface.

10. **SmartBugs 2.0–identifier** [158]: SmartBugs 2.0 is an execution framework for the automated analysis of Ethereum SCs. Released in 2023, it integrates 19 analysis tools and supports Solidity source code

and EVM bytecode. The framework detects a variety of vulnerabilities, including "reentrancy," "arithmetic issues (overflow/underflow)," "access control violations," "unchecked low-level calls," "denial of service," "bad randomness," "timestamp dependence," "front-running," "short address attacks," "unchecked return values," "unsafe delegatecall usage," "integer bugs," "gas consumption issues," "authentication bypass," "assertion violations," "transaction origin usage," "race conditions," "shadowing variables," and "suicidal contracts." The framework standardizes output formats and maps findings to the SWC taxonomy, facilitating large-scale, reproducible analysis.

11. **SoliAudit–identifier** [159]: SoliAudit is a SCs vulnerability assessment tool that combines machine learning and fuzz testing. Introduced in 2019, it detects 13 types of vulnerabilities, including "reentrancy," "arithmetic overflow," "access control," "unchecked low-level calls," "denial of service," "bad randomness," "front running," "time manipulation," "short address," "call depth," "transaction origin," "inline assembly," and "self-destruct," without requiring expert knowledge or predefined patterns. SoliAudit employs static machine learning classifiers using Solidity machine code features and a dynamic fuzzer for online transaction verification.

Model-based testing tools provide a structured approach to analyzing SCs using abstract models and state machines. While building these models can be time-consuming, they help ensure SCs work as intended by exploring various execution paths. The tools discussed illustrate how model-based testing can improve SC reliability.

3.1.9 "Mutation Testing" Tools

The ninth category uses mutation testing. This technique involves introducing small, deliberate changes, or mutations, into the source code of SCs to simulate potential errors. The modified contracts, or mutants, are then tested to determine if the existing test suites can detect and handle these errors. This process helps improve the effectiveness and thoroughness of the test suites. The tools listed below utilize mutation testing, with the first three being detection tools and the remaining two serving identification purposes in SCs.

1. **RegularMutator–detector** [160]: RegularMutator, introduced in 2020, is a mutation testing tool for Solidity SCs. It uses regular expressions to create mutants by introducing common errors into the source code. Then, the effectiveness of test suites is evaluated based on their ability to detect these mutants. The tool was proven effective in improving the quality of test suites by detecting additional defects and providing a more reliable assessment than line coverage metrics.

2. **ReSuMo–detector** [161]: ReSuMo, published in 2023, is a regression mutation testing tool for Solidity SCs. It uses a static and file-level technique to select a subset of SCs to mutate and a subset of test files to re-run during a regression mutation testing campaign. By incrementally updating the results using previous test outcomes, ReSuMo speeds up the mutation testing process for evolving projects while ensuring comprehensive adequacy assessment.

3. **SuMo–detector** [162]: SuMo is a mutation testing tool for SCs written in Solidity. It supports a stand-alone command-line interface and a web service with a REST API, enabling automated mutation

testing with minimal effort. SuMo, published in 2022, employs 44 different mutation operators and integrates with Truffle and Ganache to compile and test SCs, providing comprehensive reports on the mutation testing process and results.

4. **MuRE–identifier** [163]: Mutation testing approach to Reentrancy detection tools Evaluation (MuRE), introduced in 2023, employs mutation testing to generate classified sets of "reentrancy" vulnerabilities in Ethereum SCs. It uses symbolic execution to identify potential reentrancy paths and applies mutation operators to generate reentrancy mutants. MuRE evaluates the effectiveness of reentrancy detection tools by comparing their ability to detect these classified mutants.

5. **MuSC–identifier** [164]: Mutation testing Tool for Ethereum SCs (MuSC) was introduced in 2019 and focuses on mutation testing to evaluate the robustness of SCs testing. It generates mutants using a set of novel mutation operators specifically tailored for Solidity and performs automated operations, such as creating test nets, deploying, and executing tests. The tool helps expose various defects in SCs by modifying specific parts of the code to assess test coverage and effectiveness.

In summary, mutation testing tools provide a framework for evaluating and improving the quality of test suites for software components (SCs). Mutation testing introduces controlled modifications and assesses if the tests can catch these errors. This approach measures test coverage and reveals gaps in testing, enhancing the overall reliability and security of SCs.

3.1.10 "Pattern Matching and Syntactical Analysis" Tools

The tenth category uses pattern matching and syntactical analysis. This methodology involves analyzing the structure and syntax of SCs to identify patterns and detect potential vulnerabilities. By leveraging predefined patterns and syntactical rules, these tools can efficiently uncover security risks and coding issues within the contracts. Among the tools utilizing pattern matching and syntactical analysis listed below, the first fourteen are aimed at vulnerability detection, while the remaining twenty-one focus on vulnerability identification within software components (SCs).

1. **DeFiRanger–detector** [165]: It is a detection tool for price manipulation attacks in Decentralized Finance (DeFi) applications. Published in 2023, It constructs a Cash Flow Tree (CFT) from raw transactions, lifts low-level semantics to high-level ones, and applies predefined patterns to detect "price manipulation attacks."

2. **Ethlint–detector** [166]: Ethlint, initially known as Solium and developed in 2016, is a static analysis tool implemented in JavaScript for checking Solidity code for style and security issues. It standardizes SCs practices across organizations and integrates seamlessly with build systems. Ethlint offers a user-friendly Command-Line Interface (CLI) for linting Solidity files, featuring multiple output formats and automatic code fixes.

3. **Horus–detector** [167]: Developed in 2021, it automates detecting and analyzing attacks on Ethereum SCs by leveraging logic-driven and graph-driven methods. It extracts execution-related information from transactions, identifies attacks using Datalog queries, and traces stolen assets using graph

databases. Implemented in Python, Horus provides detailed insights into security issues, facilitating attack identification and analysis.

4. **MSmart–detector** [168]: MSmart enhances the Smartcheck static analysis tool by introducing additional rules and improving existing ones to detect vulnerabilities in Solidity SCs, developed in 2023. It identifies issues like "integer overflow", "timestamp dependence", "self-destruct", "delegatecall", and "DOS" while supporting batch detection to streamline the analysis of large datasets.

5. **Naga–detector** [169]: Naga is a tool designed in 2023 to detect "centralized security risks" in decentralized ecosystems, specifically targeting Ethereum crypto wallets and SCs. It identifies and analyzes seven centralized security risks using Information Risks (IRs) for data dependency analysis.

6. **SafelyAdministrated–detector** [170]: This tool, introduced in 2021, uses pattern recognition based on nine syntactic features to identify administrated Ethereum Request for Comments 20 (ERC20) tokens on the Ethereum Mainnet. The tool comprehensively analyzes and classifies administrated tokens to highlight their ubiquity and associated dangers. The specific vulnerabilities detected by SafelyAdministrated include "self-destruction," "contract deprecation," "change of address," "change of parameters," and "minting and burning capabilities."

7. **SCGraphs–detector** [171]: Semantic Contract Graphs (SCGraphs), published in 2023, is a SCs vulnerability detection tool that uses semantic contract graphs and approximate graph matching based on the Weisfeiler-Lehman graph kernel. It constructs SCGraphs from SCs and then calculates similarity matrices to detect vulnerabilities such as "reentrancy", "timestamp dependence", "delegatecall misuse", and "integer overflow". The tool builds a library of vulnerability SCGraphs from manually audited contracts, which helps identify new vulnerabilities.

8. **SIF–detector** [172]: Solidity Instrumentation Framework (SIF) is a framework for Solidity contract analysis and instrumentation that operates on the AST. It enables querying, modification, and code generation of Solidity contracts. SIF includes tools for function listing, call graph generation, control flow graph generation, and fault seeding, facilitating in-depth security and optimization analysis. Published in 2019, SIF detects vulnerabilities including "division by zero," "overflow," and "underflow," among others.

9. **SolChecker–detector** [173]: Developed in 2022, SolChecker is an automated static analysis tool for finding vulnerabilities in Solidity SCs. It uses an AST to analyze the code's control and data flow. By examining over 47,000 real-world contracts, SolChecker has shown its effectiveness in detecting common security issues including "Uninitialized variables", "malicious address", "function return default value", "change array length directly", "abuse of block information", "reentrancy", "uninitialized storage variables", "unchecked send and call", "divide before multiply", "constructor name error", "balance strict equality", and "abuse of tx.origin".

10. **Solhint–detector** [174]: Solhint is an open-source linting tool for Solidity code, providing both security and style guide validations. It offers a range of rules that can be customized via configuration files

to enforce coding standards and detect potential issues. Solhint, introduced in 2017, can automatically fix specific problems.

11. **SolGuard–detector** [175]: SolGuard is a security analysis plugin integrated with the Solhint linter to detect "external call issues" in Solidity SCs. Developed in 2021, it defines three security rules: checking the order of state variable declarations, avoiding address parameters in constructors, and ensuring the presence of a fallback function.

12. **TokenScope–detector** [176]: TokenScope is a tool designed in 2019 for detecting inconsistent behaviors in Ethereum cryptocurrency tokens. It examines the interactions between tokens, users, and third-party tools by contrasting behaviors derived from core data structures, standard interfaces, and standard events. TokenScope's analysis covers 7,472 tokens and identifies 3,259,001 transactions triggering inconsistencies, revealing flaws such as "integer overflow", "fake deposit", "lack of standard events", "incorrect balance update", and "mismatched token transfer behaviors".

13. **UBF-ChaincodeScan–detector** [177]: Universal Blockchain Framework Chaincode Scanner (UBF ChaincodeScan) detects vulnerabilities in Node.js-based SCs for Hyperledger Fabric published in 2023. It features a two-phase architecture: validation and scanning. The validation phase includes script command validation, SCs language identification, and syntax validation, while the scanning phase executes the scanning and generates a report in JSON format. UBF-ChaincodeScan identifies issues like "global variable", "random number generator", "system command execution", "external file libraries", "unchecked errors", "range query risks", "read write conflict", and "external API".

14. **Vulpedia–detector** [178]: Developed in 2022, Vulpedia is a static analysis tool that uses abstracted vulnerability signatures to detect security issues such as "Reentrancy", "Abuse of tx.origin", "Unexpected Revert", and "Self-destruct Abusing" in Ethereum SCs. It combines vulnerable and benign signatures to formulate detection rules.

15. **AChecker–identifier** [179]: AChecker, introduced in 2023, is a tool designed for statically detecting "access control vulnerabilities" in SCs. Unlike previous methods, AChecker infers access control mechanisms via static data-flow analysis and uses symbolic-based analysis to distinguish between intended functionalities and actual vulnerabilities.

16. **CloudAudit–identifier** [180]: Proposed in 2020, this tool uses static analysis based on XPath patterns to detect "integer overflow vulnerabilities" in Solidity SCs. By defining 83 XPath patterns across 11 types of integer overflow features, the tool effectively identifies vulnerabilities such as "multiplication," "addition," and "subtraction overflows."

17. **EOSIOAnalyzer–identifier** [181]: Proposed in 2022, EOSIOAnalyzer is a static analysis framework for detecting vulnerabilities in EOSIO SCs. It identifies vulnerabilities like "fake EOS transfer," "forged transfer notification," and "block information dependency". The tool transforms Wasm bytecode into a high-level intermediate representation and applies a context-sensitive data flow analysis algorithm.

18. **Graph Embedding-Based Bytecode Matching Tool–identifier** [182]: Introduced in 2021, this tool detects vulnerabilities in SCs using graph embedding and bytecode matching techniques. It normalizes and slices contract bytecodes to enable accurate similarity measurements between known vulnerabilities and target contracts. The tool identifies vulnerabilities such as "integer overflow," "reentrancy," "bad randomness," "unprotected ownership," and "mishandled exceptions."

19. **Matching Rules-Based SCs Audit Tool–identifier** [183]: This audit tool, developed in 2020, leverages matching rules to evaluate SCs vulnerabilities. It matches each vulnerability type with a corresponding threat level (high, medium, or low) and generates a comprehensive audit report. The extensible tool allows auditors to customize and add new rules to address emerging threats. It detects vulnerabilities such as "re-entrancy," "integer overflow," "timestamp dependency," and "permission theft" by matching specific code patterns in SCs.

20. **NeuCheck–identifier** [184]: Introduced in 2019, NeuCheck uses a syntax tree-based approach to transform source code into an intermediate representation, avoiding semantic loss. It detects various vulnerabilities, including "access control vulnerability", "reentrancy vulnerability", "hash collision vulnerability", "integer overflow vulnerability", and "dependence on predictable variables vulnerability". NeuCheck enhances analysis speed and facilitates cross-platform deployment, surpassing other tools like Securify and Mythril.

21. **Remix Solidity Static Analysis Plugin–identifier** [185]: This static analysis tool is integrated into the Remix IDE and was developed to detect vulnerabilities in Solidity SCs. It supports 21 analysis modules across security, gas, economy, and other categories. Evaluated in 2023, it can detect "access control", "arithmetic", "bad randomness", "denial of service", "front running", "reentrancy", "short addresses", "time manipulation", and "unchecked low-level calls" with moderate performance in other vulnerabilities. The tool's Graphical User Interface (GUI) makes it user-friendly compared to other command-line tools.

22. **SafeCheck–identifier** [186]: SafeCheck is a static analysis tool introduced in 2024, designed for detecting six types of vulnerabilities in Ethereum SCs, including "reentrancy", "timestamp dependency", "dangerous delegatecall", "DoS", "self-destructible", and "TOD". It converts contract bytecode into an intermediate representation to extract semantic information and employs Datalog-based rules for vulnerability detection.

23. **SASC–identifier** [187]: Static Analysis for SCs (SASC), developed in 2018, is a static analysis tool for Ethereum SCs focusing on topological analysis and logic risk detection. It identifies risks such as "call stack risks", "transaction order risks", "reentrancy", "timestamp risks", "tx.origin risks", and "zero division risks". SASC provides a detailed report in HTML format, helping developers mitigate vulnerabilities.

24. **SmartAnvil–identifier** [188]: SmartAnvil, introduced in 2019, is an open-source platform for SCs analysis. It utilizes static analysis, deployed contract binary analysis, and blockchain navigation to

detect vulnerabilities in Solidity SCs. The platform supports various components like SmaccSol for parsing, SmartGraph for semantic analysis, and SmartInspect for live state inspection, effectively identifying issues such as "semantic bugs".

25. **SmartCheck–identifier** [189]: SmartCheck is a static analysis tool for Ethereum SCs. Introduced in 2018, the tool translates Solidity code into an XML-based intermediate representation and checks it against XPath patterns to detect various code issues, including "balance equality", "unchecked external call", "DoS by external contract", "send instead of transfer", "reentrancy", "malicious libraries", "using tx.origin", "transfer forwards all gas", "integer division", "locked money", "unchecked math", "timestamp dependence", "unsafe type inference", "byte array", "costly loop", "token API violation", "compiler version not fixed", "private modifier", "redundant fallback function", and "style guide violation".

26. **SmartDagger–identifier** [190]: SmartDagger is a bytecode-based static analysis tool designed for detecting "cross-contract vulnerabilities" in Ethereum SCs. Released in 2022, it integrates novel mechanisms to recover contract attribute information from bytecode, accurately identifying vulnerabilities during cross-contract interactions. SmartDagger selectively analyzes a subset of functions, reusing data-flow results to improve efficiency.

27. **Smartmuv–identifier** [191]: Smartmuv is an automatic source-code-based static analysis tool for analyzing and extracting the storage state from the storage-trie of Ethereum SCs. Developed in 2022, it employs ASTs and CFGS to analyze state variables, including mapping types along the inheritance hierarchy. The tool ensures accurate extraction of storage state, facilitating seamless upgrades and migrations of SCs.

28. **SoliDetector–identifier** [192]: SoliDetector is a static defect detection tool for Solidity SCs, leveraging a knowledge graph to capture syntactic and logical relationships within the code. Introduced in 2023, it supports the detection of 20 kinds of defects, including "reentrancy", "integer overflow", "mishandled exception", "DoS by external contract", "using tx.origin for authentication", "missing constructor", "locked money", "unsafe type inference", "timestamp dependence", "token API violation", "private modifier", "redundant refusal of payment", "compiler version problem", "style guide violation", "integer division", "implicit visibility level", "balance equality", "costly loop", "using fixed point number type", and "byte[]", using SPARQL queries to infer complex relationships and localize defects accurately.

29. **SolidityCheck–identifier** [193]: SolidityCheck is a tool designed to detect a wide range of problems in Solidity SCs using regular expressions. Introduced in 2019, it identifies 20 issues, including security vulnerabilities such as re-entrancy and integer overflow. SolidityCheck uses regular matching and program instrumentation to quickly and accurately locate problematic statements in the source code.

30. **Solidity Vulnerability Scanner–identifier** [194]: The Solidity Vulnerability Scanner is a static analysis tool designed to identify and categorize vulnerabilities in Solidity SCs. Released in 2022, it uses

parsing and pattern matching to detect issues including “reentrancy”, “integer overflow and underflow”, “floating pragma”, “denial of service”, “bad randomness”, “unprotected function”, “unchecked external call”, “variable shadowing”, “race condition”, “incorrect interface”, and “forced ether reception”. The tool provides mitigation suggestions line by line, making it user-friendly for developers new to Web3 security.

31. **SPCon–identifier** [195]: Security Policy Conformance (SPCon) is a tool designed in 2023 to find permission bugs in SCs using role mining and security policy validation. It mines past contract transactions to recover a likely access control model and checks against various information flow policies to identify potential user permission bugs. SPCon demonstrates superior accuracy in identifying user roles and effectively detects vulnerabilities such as “unauthorized access”, “incorrect role assignment”, and “permission bugs related to access control policies”.

32. **Static Analysis-Based Vulnerability Detection Approach–identifier** [196]: Introduced in 2022, this study focuses on expanding security vulnerability detection in SCs through static analysis. It identifies the weaknesses of current static analysis tools. It addresses issues such as “gas-related vulnerabilities” and “access control vulnerabilities”, using new detection methods that do not rely on pre-existing code patterns and rules.

33. **Static Analyzer for Price Gouging TOD Vulnerabilities–identifier** [197]: This prototype tool is designed to detect and rectify “price-gouging TOD vulnerabilities” in SCs. Developed in 2022, it uses a static analysis approach to locate these vulnerabilities by extracting data dependencies and modifying the contract code. The tool implements this approach using Slither, a static analyzer for Solidity.

34. **SWAT–identifier** [198]: SWAT (SWC-based Analysis Tool) is a static analysis tool designed to detect SCs vulnerabilities as categorized by the SCs Weakness Classification (SWC) standard. It uses pattern matching on Solidity code to identify vulnerabilities such as “SWC-100 function default visibility”, “SWC-102 outdated compiler version”, “SWC-103 floating pragma”, “SWC-111 use of deprecated Solidity functions”, “SWC-129 typological error”, and “SWC-134 message call with the hardcoded gas”.

35. **VULTRON–identifier** [199]: VULTRON is a vulnerability detection tool designed to catch irregular transactions in SCs. It focuses on mismatches between actual transferred amounts and internal contract bookkeeping, identifying transaction-related vulnerabilities, including “reentrancy”, “exception disorder”, “gasless send”, and “integer overflow/underflow”. Introduced in 2019, VULTRON provides a general approach that can be adapted across different SCs platforms.

Pattern-matching and syntactical analysis tools systematically detect vulnerabilities in SCs by analyzing their structure and syntax. While these methods rely on predefined patterns and rules, they effectively uncover security risks and ensure coding consistency. The tools outlined in this section demonstrate how pattern matching and syntactical analysis can contribute to improving SC security.

“Runtime Verification” Tools

The eleventh tool category focuses on runtime verification. This methodology involves monitoring and checking the execution of SCs at runtime to ensure they adhere to specified behaviors and properties. By continuously observing contract interactions, these tools can detect and mitigate issues in real-time, ensuring the security and correctness of SCs as they execute. Below are tools that use runtime verification: the first six are detection tools, and the others are identification tools in SCs.

1. **ContractLarva–detector** [200]: ContractLarva is a runtime verification tool for Ethereum SCs written in Solidity and published in 2018. It instruments additional code into SCs based on formal specifications to monitor and enforce correct behavior. ContractLarva captures control-flow and data-flow events to detect violations and supports custom reparation strategies for handling detected issues, including "unauthorized access", "incorrect function execution", and "data integrity issues".

2. **DappGuard–detector** [201]: It is an active monitoring and defense system developed in 2017 for Solidity SCs. It analyzes blockchain transactions for signs of attacks, such as high gas usage and exception rates, and employs a rules engine to detect and mitigate known vulnerabilities including "call to the unknown", "gasless send", "exception disorder", "type casts", "reentrancy", "keeping secrets", "immutability", "ether lost in transfer", "TOD", "stack size limit", "generating randomness", "timestamp dependence", and "integer overflow/underflow" in real-time.

3. **ECFChecker–detector** [202]: It is a dynamic monitor for verifying Effectively Callback Free (ECF) properties in Ethereum SCs, introduced in 2017. It detects non-ECF executions using a polynomial-time online algorithm integrated into the EVM. The tool prevents vulnerabilities like the "DAO" bug by ensuring callback-free execution paths.

4. **EVM–detector** [203]: It is a reinforced EVM designed in 2019 to prevent dangerous transactions in real-time. It uses monitoring strategies, opcode-structure maintenance, and EVM instrumentation to detect and stop operations that could exploit vulnerabilities, including "overflow bugs" and "timestamp bugs".

5. **EVM-Shield–detector** [204]: EVM-Shield is a novel runtime tool that enforces fine-grained access control over sensitive states within SCs. It utilizes a hybrid storage analyzer to identify storage locations dynamically and employs a multi-stage cache-based filter to prevent unexpected state access efficiently. Developed in 2024, EVM-Shield can detect various vulnerabilities, including "reentrancy", "integer overflow", and "unauthorized state modifications". This tool enhances security by preventing transactions that attempt to access sensitive states or Ether without proper permissions.

6. **Xscope–detector** [205]: Published in 2023, Xscope is an automatic tool designed to detect security violations in cross-chain bridges. It addresses emerging security issues by providing runtime monitoring and offline analysis functionalities. Xscope identifies three new security bug classes in cross-chain bridges: "unrestricted deposit emitting", "inconsistent event parsing", and "unauthorized unlocking". The tool uses security properties and patterns to characterize and detect these vulnerabilities.

7. **ÆGIS–identifier** [206]: Introduced in 2020, ÆGIS is a dynamic analysis tool that can detect vulnerabilities such as "Same-Function Reentrancy," "Cross-Function Reentrancy," "Delegated Reentrancy," "Create-Based Reentrancy," "Parity Wallet Hack 1," and "Parity Wallet Hack 2". It protects SCs from being exploited during runtime. It uses attack patterns described in a domain-specific language tailored to Ethereum SCs, enabling real-time detection and reversion of malicious transactions. The tool supports decentralized and transparent security updates through a SCs-based voting mechanism.

8. **ContractGuard–identifier** [207]: ContractGuard, introduced in 2020, is an anomaly-based Intrusion Detection System (IDS) designed to protect Ethereum SCs after deployment. It profiles context-tagged acyclic paths and detects abnormal control flow to identify intrusions. The tool targets vulnerabilities like "Reentrancy explicitly," "Dangerous delegate call," "Arithmetic Over/Under Flows," "Default visibilities," "Unchecked send," "Tx.Origin Authentication", "Denial Of Service," and "Logic error," providing real-time defense against attacks by rolling back transactions when anomalies are detected.

9. **Gas Gauge–identifier** [208]: Gas Gauge, released in 2023, is a security analysis tool designed to detect "Out-of-Gas (OOG) DoS vulnerabilities" in Ethereum SCs. It combines static analysis, white-box fuzzing, and runtime verification to identify, summarize, and correct loops that could cause OOG errors. Gas Gauge efficiently analyzes contract loops, generates inputs to trigger OOG errors, and suggests repairs to prevent such vulnerabilities.

10. **SODA–identifier** [209]: SODA is a generic online detection framework for SCs on EVM-compatible blockchains. It can detect vulnerabilities including "reentrancy", "unexpected function invocation", "invalid input data", "incorrect check for authorization", "no check after contract invocation", "missing the Transfer event", "strict check for balance", and "dependency of block number and timestamp". Introduced in 2020, it separates information collection and attack detection, allowing users to develop apps for various attack detections quickly. SODA provides unified interfaces and instruments EVM to collect necessary information, facilitating quick responses to new attacks.

11. **Solitor–identifier** [210]: Solitor is a runtime verification tool for Ethereum SCs, developed in 2018. It uses annotations to specify contract behavior, which are checked at runtime. This method enhances security by allowing developers to define and verify properties such as invariants, preconditions, and postconditions directly in the Solidity code.

12. **TaintGuard–identifier** [211]: TaintGuard is a static analysis tool designed to prevent "implicit privilege leakage" in SCs through taint tracking at the AST level. Introduced in 2023, it analyzes Solidity contracts to identify and mitigate vulnerabilities related to delegate calls for cross-contract calls, which can tamper with contract privileges. TaintGuard integrates code instrumentation to monitor contract state at runtime, ensuring security against malicious privilege modifications.

To conclude, runtime verification tools can enhance the security of SCs by monitoring their execution in real-time. Even though runtime verification requires additional computational resources, it offers advantages by allowing immediate handling of vulnerabilities. The tools presented in this section show how necessary runtime verification is for keeping SCs systems secure.

3.1.11 "Symbolic Execution" Tools

The twelfth tools category focuses on the Symbolic Execution method. This methodology analyzes programs by executing them symbolically rather than with actual inputs. It allows exploring multiple execution paths simultaneously, particularly useful for detecting vulnerabilities in complex systems like Ethereum SCs. These tools often use decompilation, constraint solving, and control flow analysis techniques to identify reentrancy, gas inefficiency, and concurrency exploits. The first thirty tools listed below are detection tools using symbolic execution, while the remaining twenty-six are identification tools.

1. **Abbe–detector** [212]: Abnormal BEhavior detection in Ethereum (ABBE), developed in 2019, detects abnormal behaviors in Ethereum SCs by identifying attack vectors through transaction analysis. It uses decompilation, symbolic execution, and heuristic-based methods to detect various vulnerabilities, including "reentrancy", "gasless send", "force-sending ether by suicide", "integer overflow", "array overflow", "uninitialized storage pointer" and "overridden by delegate call".

2. **ConSym-OSIRIS–detector** [213]: ConSym-OSIRIS is a tool introduced in 2022 for the combined symbolic execution and mutation testing of software systems. It aims to identify and analyze potential vulnerabilities by generating symbolic inputs to explore different execution paths. The tool enhances traditional symbolic execution by integrating mutation testing techniques and systematically introducing and testing code variations to uncover hidden bugs, including "integer overflow and underflow", "reentrancy", "arithmetic errors" and "incorrect handling of state variables".

3. **CESC–detector** [214]: Concurrency Exploit SCs (CESC) detects "concurrency exploits" in Ethereum SCs using symbolic execution, developed in 2019. It traces storage operations, merges access flows, and verifies potential concurrency exploits. CESC effectively identifies "TOD", "unauthorized Ether transfers", "state pollution" and "contract self-destruction" , improving detection scope and reducing false positives.

4. **EtherSolve–detector** [215]: It reconstructs precise CFGs from Ethereum bytecode using symbolic execution. It resolves jump destinations by symbolically executing the operands stack and has been validated on real-world contracts. EtherSolve, launched in 2023, detects "reentrancy" and "Tx.origin" vulnerabilities more accurately than other bytecode analysis tools.

5. **EthChecker–detector** [216]: It is a SCs vulnerability detection tool combining fuzzing and symbolic execution, developed in 2024. It employs a context-guided genetic programming algorithm to enhance code coverage and detect vulnerabilities efficiently. It can detect vulnerabilities including "reentrancy", "timestamp-dependency", "integer overflow and underflow" and "unhandled exceptions".

6. **Ethainter–detector** [217]: It is a security analyzer created in 2020 that detects composite information flow violations in Ethereum SCs. It models tainted information flow and evaluates the effectiveness of guard conditions to identify vulnerabilities, including "tainted owner variable", "tainted delegatecall", "accessible self-destruct", "tainted self-destruct" and "unchecked tainted static call" that escalate through multiple transactions.

7. **EthRacer–detector** [218]: This tool was developed in 2019 to identify "event-ordering bugs" in Ethereum SCs using dynamic symbolic execution and happens-before relations. It employs symbolic execution, fuzzing, and HB analysis to reduce the search space and flags contracts with differing outputs under reordered events for efficient detection.

8. **EXGEN–detector** [219]: Exploit Generation (EXGEN), released in 2022, is a cross-platform framework for automated exploit generation in SCs. It translates Ethereum and EOS contracts to an intermediate representation, generates symbolic attack contracts, and executes them to find vulnerabilities, including "reentrancy", "integer overflow and underflow", "suicidal", "call injection" and "arbitrary value transfer".

9. **FSFC–detector** [220]: Filter-based Secure Framework for Ethereum SCs (FSFC) is a framework for SCs fuzzing that combines fuzz testing with symbolic execution, launched in 2020. It generates initial inputs through fuzzing and refines them using symbolic execution to explore deeper contract states and identify hidden vulnerabilities, including "reentrancy", "integer overflow", "integer underflow", "access control issues", "self-destruct vulnerabilities" and "DoS".

10. **GasChecker–detector** [221]: GasChecker detects gas-inefficient patterns in Ethereum SCs, such as "opaque predicates," "dead code," "expensive operations in a loop," "fusible loops," "repeated computation in a loop," "unilateral comparison in a loop," "redundant SSTORE," "SWAP1=DUP2=SWAP1 pattern," "PUSHx=POP pattern," and "PUSH1=NOT pattern," using symbolic execution. Published in 2020, GasChecker scales to analyze millions of contracts by parallelizing execution with a MapReduce model and employs a feedback-based load-balancing strategy to optimize resource use.

11. **GASPER–detector** [222]: GASPER is a symbolic execution tool designed in 2017 to discover gas-costly patterns in Ethereum SCs bytecode automatically. It targets patterns such as "dead code," "opaque predicates," "expensive operations in a loop," "constant outcome of a loop," "loop fusion," "repeated computations in a loop," and "comparison with unilateral outcome in a loop." GASPER helps developers optimize their contracts to reduce unnecessary gas consumption.

12. **HoneyBadger–detector** [223]: It is a tool developed in 2019 that detects "honeypots" in Ethereum SCs using symbolic execution and heuristics. It constructs control flow graphs, performs cash flow analysis, and identifies honeypot techniques by analyzing bytecode.

13. **Jyane–detector** [224]: It detects "reentrancy" vulnerabilities in SCs using path profiling. It was introduced in 2021 and constructs CFG from EVM bytecode, employs an improved Ball-Larus Path Profiling algorithm to generate unique path IDs, and identifies suspicious paths using Deterministic Finite Automata (DFA).

14. **MPro–detector** [225]: MPro is a tool developed in 2019 that combines static and symbolic analysis to enhance the scalability of SCs testing. It uses static analysis to identify potential vulnerabilities and symbolic execution to generate test inputs that thoroughly explore these sites. MPro effectively

detects depth-n vulnerabilities, such as "reentrancy" and "unrestricted suicide", which require specific sequences of function invocations to exploit.

15. **Mythril Extension for Gasless Send issue detection–detector** [226]: This framework, published in 2019, enhances the Mythril security analysis tool to model gas usage and detect "gasless send" vulnerabilities in Ethereum SCs. By simulating gas consumption during symbolic execution, it identifies contracts whose fallback functions exceed the 2,300 gas limit, preventing them from receiving Ether.

16. **NPChecker–detector** [227]: NPChecker targets vulnerabilities such as "nondeterministic payment bugs," "reentrancy," "transaction-order dependence," "failed external calls," and "system property dependence" in Ethereum SCs. Launched in 2019, it systematically models and detects nondeterministic factors affecting contract payments. Using information flow tracking and model checking, NPChecker identifies contracts with vulnerabilities caused by unpredictable transaction scheduling and external callee behavior.

17. **Pluto–detector** [228]: Developed in 2022, Pluto detects vulnerabilities in inter-contract scenarios by constructing an Inter-contract Control Flow Graph (ICFG). It symbolically explores the ICFG and deduces Inter-Contract Path Constraints (ICPCs) to accurately check the reachability of execution paths. Pluto can detect vulnerabilities including "integer overflow", "timestamp dependency" and "reentrancy"

18. **Rattle–detector** [229]: Rattle is a symbolic execution tool developed by Trail of Bits to analyze Ethereum SCs. It targets vulnerabilities such as "storage access patterns," "external calls," and "control flow hijacking." Rattle translates EVM bytecode into SSA form, enabling detailed control and data flow analysis within contracts. Introduced in 2019, it facilitates the detection of vulnerabilities by providing an intermediate representation that simplifies the complexity of the original bytecode.

19. **ReDetect–detector** [230]: ReDetect is a symbolic execution-based tool for detecting "reentrancy" vulnerabilities in Ethereum SCs at the EVM bytecode level, published in 2021. It preprocesses source files into EVM assembly code, constructs a control flow graph, and employs symbolic execution to analyze potential reentrancy paths. ReDetect significantly reduces false positives by implementing five effective path filters.

20. **Reentrancy Vulnerability Identification Framework–detector** [231]: This framework was introduced in 2020 to combine static and dynamic analysis to detect "reentrancy" vulnerabilities in Ethereum SCs. It utilizes Tree Transformation Language (TXL) grammar for parsing Solidity code and generates an attacker contract based on Application Binary Interface (ABI) specifications to simulate reentrancy attacks.

21. **SAILFISH–detector** [232]: SCs stAte-Inconsistency anaLysis Framework leveraging Incremental and Symbolic evaluation Heuristics (SAILFISH), developed in 2022, is a scalable tool designed to detect state-inconsistency bugs in Ethereum SCs. It employs a hybrid approach that combines a lightweight exploration phase with a precise refinement phase, utilizing symbolic evaluation guided

by a novel value-summary analysis. SAILFISH efficiently detects vulnerabilities such as “reentrancy” and “TOD”.

22. **Sereum–detector** [233]: Sereum is a run-time monitoring tool designed to protect existing deployed SCs against reentrancy attacks. Using dynamic taint tracking, Sereum monitors data flows from storage variables to control-flow decisions, preventing state inconsistencies during reentrant calls. Launched in 2019, It effectively detects and mitigates “reentrancy attacks” with a negligible run-time overhead and a low false positive rate.

23. **Seraph–detector** [234]: Developed in 2020, Seraph is a cross-platform security analyzer for blockchain SCs, supporting EVM and WASM runtimes. It targets vulnerabilities such as “integer overflow,” “pseudo-random number generator (PRNG) issues,” “insecure message calls,” and “DoS.” Seraph uses a symbolic semantic graph (SSG) to model critical dependencies and employs connector APIs to abstract interactions between virtual machines and blockchains. It enables automated security analysis and generates comprehensive security reports by exploring symbolic execution paths.

24. **SMARTEST–detector** [235]: SMARTEST integrates deep learning language models with symbolic execution to enhance SCs testing. It specifically targets vulnerabilities such as “integer overflow and underflow,” “division by zero,” “assertion violations,” and “ERC20 standard violations.” Developed in 2023, it utilizes neural network architectures, including Transformer, GRU, and RNN, to train models on vulnerable transaction sequences from the Common Vulnerabilities and Exposures (CVE) Benchmark.

25. **SoMo–detector** [236]: SoMo, published in 2023, detects insecure modifiers in Ethereum contracts. It constructs a Modifier Dependency Graph (MDG) to analyze control and data flows and uses symbolic execution to identify bypassable modifiers.

26. **Static Analyzer for Solidity Vulnerability Detection–detector** [237]: This tool employs static analysis to identify security flaws in Solidity SCs. It detects various vulnerabilities, including “reentrancy”, “integer overflow and underflow”, “timestamp dependence”, “unprotected selfdestruct”, “unchecked call return values”, “denial of service” and “transaction-ordering dependence”, using techniques like symbolic execution and data flow analysis. The tool was introduced in 2020 and aims to enhance the security of Ethereum smart contracts (SCs) by providing detailed reports on identified vulnerabilities and recommending mitigation strategies.

27. **Symbolic Execution-Based Vulnerability Detector–detector** [238]: This system enhances the Mythril symbolic execution tool by optimizing the pruning algorithm to reduce execution time and introducing a detection algorithm for vulnerabilities such as “integer overflow and underflow,” “unchecked call return value,” “reentrancy,” “TOD,” “authorization through tx.origin,” and “block values as a proxy for time.” Developed in 2020, it also integrates a machine-learning model based on LSTM networks for preliminary vulnerability detection.

28. **Teether–detector** [239]: Teether, published in 2018, is a tool designed to automatically exploit vulnerabilities in Ethereum SCs by leveraging symbolic execution. It identifies critical paths that lead to executing exploitable instructions, such as CALL, CALLCODE, DELEGATECALL, and SELFDESTRUCT. Teether generates exploits by resolving constraints related to these paths, demonstrating the severity of vulnerabilities through practical exploits. The tool detects vulnerabilities such as "arbitrary ether extraction", "self-destruct vulnerabilities", "code injection via callcode and delegatecall" and "execution of unauthorized code".

29. **TransRacer–detector** [240]: TransRacer is an automated tool designed to detect "transaction races" in Ethereum SCs. It uses symbolic execution to analyze function dependencies and identify races that arise from the nondeterministic execution sequence of transactions. TransRacer, developed in 2023, efficiently detects race conditions by pruning function pairs with no read/write conflicts and validating the identified races through concrete execution.

30. **VerX–detector** [241]: VerX is an automated verifier designed in 2023 to prove the functional properties of Ethereum SCs. It combines symbolic execution and delayed predicate abstraction to verify temporal safety properties. VerX can handle an unbounded number of transactions and external contract interactions by reducing temporal property verification to reachability checking, ensuring precise verification of real-world SCs.

31. **Annotary–identifier** [242]: Annotary, introduced in 2019, is a concolic execution framework designed to analyze SCs for vulnerabilities. It supports annotations that developers write directly in the Solidity source code, enabling the analysis of inter-transactional and inter-contract control flows. Annotary combines symbolic execution of EVM bytecode with the resolution of concrete values from the Ethereum blockchain to identify vulnerabilities including "reentrancy", "arithmetic overflows", "unchecked send returns", "uninitialized state variables", "unreachable code" and "visibility modifier errors".

32. **Conkas–identifier** [243]: Conkas is a modular static analysis tool for EVM based on symbolic execution, introduced as part of a master's thesis. It analyzes Ethereum smart contracts (SCs) written in Solidity or compiled runtime bytecode. Conkas uses Z3 as the SMT Solver and a modified version of Rattle as the IR. It detects vulnerabilities such as "arithmetic issues", "reentrancy", "time manipulation", "transaction ordering dependence", and "unchecked low-level calls".

33. **DefectChecker–identifier** [244]: DefectChecker, introduced in 2020, uses symbolic execution to detect vulnerabilities in Ethereum SCs by analyzing EVM bytecode. It identifies issues including "transaction state dependency", "DoS under external influence", "strict balance equality", "reentrancy", "nested call", "greedy contract", "unchecked external calls" and "block info dependency". The tool and its dataset are publicly available for community use.

34. **ETHBMC–identifier** [245]: Ethereum Bounded Model Checker (ETHBMC), released in 2020, is a bounded model checker for Ethereum SCs that uses symbolic execution to provide precise modeling

of the Ethereum network. It supports inter-contract analysis, memory modeling, and keccak256 hash functions. ETHBMC automatically generates concrete inputs that demonstrate vulnerabilities, including “arbitrary ether extraction”, “suicidal contracts”, “control flow hijacking” and “memory handling issues”.

35. **GASOL–identifier** [246]: Released in 2020, GASOL is a gas analysis and optimization tool for Ethereum SCs. It employs a hybrid approach that combines static analysis and symbolic execution to identify and optimize gas consumption in smart contracts (SCs), specifically targeting vulnerabilities such as ”out-of-gas vulnerabilities,” ”under-optimized storage patterns,” and ”gas-expensive operations.” GASOL optimizes loops and arithmetic operations to reduce the overall gas cost.

36. **HFCCT–identifier** [247]: Hyperledger Fabric Contract Code Tester (HFCCT), created in 2022, is an open-source tool that combines dynamic symbolic execution and static AST analysis to detect vulnerabilities in Hyperledger Fabric SCs. The tool identifies issues including “global variable misuse”, “random number generation”, “system timestamp”, “map structure iteration”, “reified object addresses”, “concurrency issues”, “web service risks”, “external library calling”, “system command execution”, “external file accessing”, “range query risks”, “field declarations”, “cross-channel chaincode invocation”, “read-write conflict”, “unchecked input arguments”, “unhandled errors” and “Golang grammar errors”.

37. **Honeytoken-Detector–identifier** [248]: Developed in 2023, this symbolic execution-based, open-source tool is designed for detecting honeypot tokens in Ethereum SCs. It identifies six common honeypot issues, including “token transfer quantity inconsistency”, “token transfer restriction”, “fake logs”, “infinite minting”, “balance manipulation” and “proxy token”. By parsing bytecode and generating a control flow graph, it explores paths and performs symbolic execution to analyze SCs effectively.

38. **Improved Symbolic Execution-Based Vulnerability Detector–identifier** [238]: This enhanced system integrates Mythril’s symbolic execution with an LSTM network. Developed in 2022, it supports both source code and bytecode forms, automatically detecting six types of SCs vulnerabilities. With public code available on GitHub, it employs a machine-learning approach for initial detection, followed by symbolic execution for precise vulnerability localization. This hybrid method leverages machine learning’s speed and symbolic execution’s accuracy to detect “integer overflow and underflow”, “unchecked call return value”, “reentrancy”, “TOD”, “authorization through tx.origin” and “block values as a proxy for time”.

39. **KEVM–identifier** [249]: This open-source tool, introduced in 2017, provides a complete formal semantics of the EVM using the K framework. KEVM enables formal analysis and verification of SCs and has successfully passed the official EVM test suites. It addresses vulnerabilities including “stack overflow”, “out-of-gas exceptions”, “arithmetic overflow”, “division by zero”, “invalid opcodes”, “access to non-existent account data” and “call stack limit exceeded”.

40. **KEVM Verifier–identifier** [250]: This open-source tool, developed using the K framework, was introduced in 2018. It provides comprehensive formal semantics of the EVM, enabling rigorous formal analysis and verification of SCs, including high-profile ones like ERC20 tokens, Ethereum Casper, and DappHub MakerDAO. The tool addresses vulnerabilities including "arithmetic overflow", "hash collisions", "byte manipulation errors" and "incorrect implementation of functional specifications".

41. **MAIAN–identifier** [251]: This tool uses symbolic analysis to identify vulnerabilities in SCs execution traces, explicitly targeting "greedy," "prodigal," and "suicidal" contracts. It analyzes bytecode and flags contracts with these vulnerabilities, utilizing a custom Ethereum Virtual Machine for symbolic execution. MAIAN, created in 2018, can identify vulnerabilities such as "locking funds indefinitely", "unauthorized fund transfers" and "arbitrary contract termination". Its open-source code is available on GitHub for further research and development.

42. **Manticore–identifier** [252]: Manticore is a dynamic symbolic execution framework designed for both binaries and Ethereum SCs. Introduced in 2019, this open-source tool enables the systematic exploration of program state spaces to identify vulnerabilities without generating false positives. The tool supports various execution environments, including Ethereum, and can detect "reentrancy", "integer overflows", "assertion failures", "unhandled exceptions" and "memory safety violations".

43. **MOPS–identifier** [253]: Multi-Objective Oriented Path Search (MOPS), developed in 2019, optimizes vulnerability detection for SCs by combining static and dynamic analysis. It focuses on critical paths involving Ether transfer, thus improving efficiency and reducing false positives. MOPS employs dynamic symbolic execution and taint analysis to identify security issues, including "reentrancy", "integer overflow", "delegatecall abuse", "transaction-order dependency", "suicide contract vulnerability", "predictable variable dependency", "mishandled exceptions" and "unchecked return values".

44. **Mythril–identifier** [254]: Mythril, a security analysis tool introduced in 2018, analyzes EVM bytecode to detect vulnerabilities in SCs across multiple EVM-compatible blockchains. Using symbolic execution, SMT solving, and taint analysis, Mythril identifies issues including "reentrancy", "integer overflows and underflows", "unrestricted Ether flows", "unprotected selfdestruct", "arbitrary storage writes", "unhandled exceptions", "timestamp dependence", "transaction-ordering dependence" and "use of tx.origin". It supports both on-chain and off-chain contract analysis, providing flexible options for security assessment.

45. **MythX–identifier** [255]: MythX is a comprehensive SCs security analysis API supporting Ethereum and other EVM-compatible blockchains. Introduced in 2020, it combines static analysis, symbolic execution, and input fuzzing to detect security vulnerabilities and verify SCs correctness. MythX integrates with various development tools, including Remix IDE, Truffle, and Visual Studio Code, and offers both command-line tools and continuous integration support. It identifies vulnerabilities including "reentrancy", "integer overflow and underflow", "timestamp dependence", "unprotected selfdestruct", "unprotected ether withdrawal", "weak randomness", "assert violation", "write to arbitrary storage location", "jump to arbitrary destination" and "uninitialized storage pointer".

46. **NFTGuard–identifier** [256]: NFTGuard is a symbolic execution-based tool designed to detect defects in Non-Fungible Token (NFT) SCs. Introduced in 2023, it identifies five specific defects: "Risky Mutable Proxy", "ERC-721 Reentrancy", "Unlimited Minting", "Missing Requirements", and "Public Burn". NFTGuard combines source code-level information with bytecode analysis to locate and report these defects in SCs effectively.

47. **Osiris–identifier** [257]: Osiris is a symbolic execution tool designed to detect various integer bugs in Ethereum SCs. It combines symbolic execution and taint analysis to identify "arithmetic", "truncation", and "signedness bugs". Released in 2018, Osiris effectively locates vulnerabilities in EVM bytecode, providing detailed analysis to ensure SCs security.

48. **Oyente–identifier** [258]: Oyente is a symbolic execution tool designed to analyze Ethereum SCs. It detects potential security bugs by analyzing EVM bytecode directly, without requiring a high-level representation. Oyente, launched in 2016, identifies vulnerabilities such as "transaction-ordering dependence", "timestamp dependence", "mishandled exceptions" and "reentrancy vulnerability".

49. **Pakala–identifier** [259]: Pakala is an open-source symbolic execution tool for the EVM, introduced in 2018. Implemented in Python, it uses Z3 with an added SHA3 layer to analyze bytecode for vulnerabilities. Pakala employs a two-step process: it first executes the bytecode to find outcomes and then analyzes these outcomes to detect vulnerabilities like "calls to suicide()" and "excessive ether transfers".

50. **Park–identifier** [260]: Park is a symbolic execution framework for SCs, introduced in 2022. It uses parallel-fork symbolic execution to enhance the efficiency of vulnerability detection by leveraging multiple CPU cores. Park implements a dynamic forking algorithm and adaptive process restriction to address performance issues, and it integrates with existing tools like Oyente and Mythril to speed up vulnerability detection. It can detect "transaction-ordering dependence", "timestamp dependence", "mishandled exceptions", and "reentrancy vulnerability".

51. **RA–identifier** [261]: Re-entrancy Analyzer (RA) is a static analysis tool developed in 2020 to detect "re-entrancy vulnerabilities" in Ethereum SCs. It combines symbolic execution and SMT solver techniques to analyze EVM bytecodes without requiring prior knowledge of attack patterns. RA supports inter-contract behavior analysis, targeting vulnerabilities such as "re-entrancy attacks", "exploitation of fallback functions" and "cross-function calls".

52. **sCompile–identifier** [262]: sCompile is a tool implemented in C++ and designed to identify critical program paths in SCs. Launched in 2019, it combines control flow graph construction and symbolic execution to prioritize and analyze paths involving monetary transactions. It focuses on conditions like avoiding non-existing addresses, ensuring contracts are not black holes that can only receive but not send Ether, guarding against improper self-destruction of contracts, and preventing transactions that exceed transfer limits.

53. **SolSEE–identifier** [263]: SolSEE is a source-level symbolic execution engine for Solidity SCs. Launched in 2022, it performs symbolic execution on the source code rather than bytecode, retaining high-level semantic information. SolSEE supports advanced Solidity language features and provides a web-based user interface for interactive analysis tasks such as visualization and debugging. It can detect vulnerabilities, including "integer underflow", "integer overflow", and "assertion violations".

54. **Solar–identifier** [264]: Solar is a system for automatically synthesizing adversarial contracts that exploit vulnerabilities in victim SCs. Introduced in 2020, this technique employs a summary-based symbolic evaluation approach to significantly reduce the number of instructions evaluated symbolically while maintaining precision in detecting vulnerabilities. Solar encodes common vulnerabilities, including "reentrancy", "time manipulation", "malicious access control" and "batch overflow bugs", and efficiently synthesizes attack programs.

55. **Symvalic–identifier** [265]: Symvalic is a static analysis approach combining concrete values and symbolic expressions to model SCs behavior. Introduced in 2021, it achieves deep modeling of program semantics through a symbiotic relationship between a traditional static analysis fixpoint computation and a symbolic solver. Symvalic is particularly effective for high-value Ethereum SCs, identifying vulnerabilities such as reentrancy and overflow.

56. **WANA–identifier** [266]: WebAssembly Analysis (WANA) is a symbolic execution engine designed to detect vulnerabilities in Smart Contracts (SCs), explicitly targeting the WebAssembly (Wasm) bytecode used in platforms such as EOSIO. Introduced in 2021, it effectively detects vulnerabilities including "fake EOS transfer", "forged transfer notification", "block information dependency", "greedy", "dangerous delegatecall", "mishandled exception" and "reentrancy vulnerability" by handling the complete Wasm instruction set. WANA is also compatible with other blockchain platforms, such as Ethereum.

Ultimately, symbolic execution tools provide a powerful approach to identifying vulnerabilities in SCs by exploring multiple execution paths. By simulating various scenarios, these tools can detect subtle and complex issues. The tools highlighted here demonstrate the value of symbolic execution in enhancing SCs' reliability.

3.1.12 "Taint Analysis" Tools

The thirteenth tool category focuses on Taint Analysis. This technique tracks the flow of data through SCs to identify potential vulnerabilities and security risks. Taint analysis helps in understanding how data propagates within the contract, enabling the detection of issues such as reentrancy, overflow, and bad randomness. These tools can effectively identify and mitigate risks by marking and tracking tainted data. Here are seven notable identification tools developed for taint analysis in SCs:

1. **Clairvoyance–identifier** [267]: Clairvoyance, introduced in 2020, employs cross-contract static taint analysis to detect "reentrancy vulnerabilities" in SCs. It reduces false positives and negatives by summarizing Path Protection Techniques (PPTs) and performing lightweight symbolic analysis.

2. **DoSChecker–identifier** [268]: Introduced in 2023, DoSChecker is a tool for detecting "DoS vulnerabilities" in SCs. It defines four patterns of DoS vulnerabilities: "Loops With Exceptional Instructions (LWEI)", "Unbounded Batch High Gas Instructions (UBHGI)", "Transfer Procedures with Non-Payable Fallback Functions (TPWNPFF)", and "Strict Balance Equality (SBE)". Using symbolic execution, DoSChecker analyzes SCs bytecodes to identify these vulnerabilities.

3. **EASYFLOW–identifier** [269]: Introduced in 2019, EASYFLOW is designed to detect "overflow vulnerabilities" in Ethereum SCs. It uses taint analysis to track data propagation during transaction execution, identifying manifested, protected, and potential overflows. EASYFLOW generates transactions to trigger potential overflows and supports dynamic analysis by extending the Ethereum Virtual Machine (EVM) interpreter.

4. **RNVulDet–identifier** [270]: Randomness Vulnerability Detection (RNVulDet) is a taint analysis tool developed in 2023 for identifying "bad randomness vulnerabilities" in Ethereum SCs. It simulates the EVM runtime environment by examining stack state, memory segmentation, storage key-value pair comparisons, and transaction replay to detect vulnerabilities. RNVulDet excels in pinpointing vulnerabilities in random number generation.

5. **SESCon–identifier** [271]: SESCon is a static analysis tool that utilizes taint analysis and XPath queries to detect vulnerabilities in Ethereum SCs. Introduced in 2021, it follows standard patterns defined by the Ethereum community to identify issues including "reentrancy", "TOD", "tx.origin usage", "block.timestamp usage", "insecure use of SelfDestruct instruction" and "DAO vulnerabilities".

6. **SIGUARD–identifier** [272]: Introduced in 2023, this tool identifies signature-related vulnerabilities in SCs using symbolic execution and taint analysis. It constructs a CFG for the bytecode, simulates the EVM, and tracks signature-related dataflows. By checking external call destinations and signature verification processes, SIGUARD detects and validates potential vulnerabilities, including "stateless signature verification" and "unseparated signing domain".

7. **SmartScan–identifier** [273]: SmartScan is a tool designed to detect "DoS vulnerabilities" in Ethereum SCs. Combining static and dynamic analysis, it identifies patterns of potential vulnerabilities and confirms their exploitability through dynamic interaction. Introduced in 2021, SmartScan targets DoS vulnerabilities caused by unexpected reverts.

Overall, taint analysis tools offer an effective method for tracing data flows and detecting vulnerabilities in SCs. By tagging and monitoring tainted data, these tools can reveal security issues that might be overlooked by conventional analysis. The tools listed here demonstrate how taint analysis can contribute to maintaining secure SCs.

3.1.13 "Visualization Analysis" Tools

The fourteenth tool category employs visualization analysis. This methodology involves creating visual representations of SCs to facilitate the understanding of their structure, behavior, and potential vulnerabilities.

Translating code into graphical formats, these tools help developers and auditors gain insights into the contract's operations and identify issues more effectively. The first six tools in the list below apply visualization analysis for vulnerability detection, while the last tool is used for vulnerability identification within SCs.

1. **E-EVM–detector** [274]: Published in 2018, E-EVM emulates and visualizes the execution of Ethereum SCs on the EVM. It visually represents the contract's control flow graph, opcodes, and stack state for each execution step, helping users understand EVM operations and contract behaviors. The tool is handy for identifying loops, optimization candidates, and tracing execution paths, thus aiding in detecting vulnerabilities such as "re-entry issues" and "invalid jump destinations".

2. **Erays–detector** [275]: Developed in 2018, It is a reverse engineering tool implemented in Python, designed to decompile Ethereum SCs from EVM bytecode to high-level pseudocode. Erays enhance understanding of opaque SCs, providing insights into code complexity and code reuse and enabling partial source code recovery for contracts without publicly available source code.

3. **EthIR–detector** [276]: EthIR (Ethereum Intermediate Representation) is a framework for high-level analysis of Ethereum bytecode, introduced in 2018. It analyzes Ethereum bytecode using CFGs generated by Oyente and produces a Rule-Based Representation (RBR) of the bytecode. EthIR detects vulnerabilities such as "gas-related inefficiencies", "unbounded loops" and "potential denial-of-service attacks"

4. **InvCon–detector** [277]: InvCon is a dynamic invariant detection tool for Ethereum SCs. It uses historical transaction data to infer invariants, aiding in reverse engineering and compliance checking. InvCon, published in 2022, features a web-based interface and a backend in Python and Java for data-trace generation and invariant detection. InvCon helps identify vulnerabilities such as invariant violations, inconsistencies with ERC20 standard specifications, and potential non-compliance issues in SCs implementations.

5. **Smart-Graph–detector** [278]: Smart-Graph is a web-based tool developed in 2021 that generates augmented Unified Modeling Language (UML) class diagrams for Solidity SCs. It utilizes a backend API to fetch the contract source code and create visual representations that include Solidity-specific features like function modifiers and fallback functions. Smart-Graph aids developers in visualizing the contract's architecture, helping to identify issues such as ""excessive code complexity" and "potential interaction flow problems".

6. **SolGraph–detector** [279]: SolGraph, published in 2018, generates DOT graphs to visualize the control flow of functions in Solidity contracts and highlights potential security vulnerabilities. It labels functions and operations with different colors to represent their types, such as external sends, constant functions, and transfers. SolGraph helps developers understand contract behavior and identify vulnerabilities through graphical representations.

7. **Slither–identifier** [280]: Slither is a static analysis framework for Ethereum SCs that converts Solidity code into an intermediate representation called SlithIR. This representation uses the SSA form to

preserve semantic information, facilitating dataflow and taint tracking analyses. Introduced in 2019, Slither offers automated vulnerability detection, code optimization, and enhanced code understanding. It can detect “reentrancy”, “shadowing of variables”, “uninitialized variables”, “suicidal contracts”, “locked Ether”, and “arbitrary sending of Ether”.

Visualization analysis tools provide a valuable approach for examining and interpreting SCs by transforming code into visual representations. This conversion helps identify potential flaws and gain insights into contract logic and execution flow. The tools listed here demonstrate how visualization can enhance the analysis of SCs, making complex code structures more comprehensible.

Table 34: Overview of SCs Analysis Tools by Methodology

Methodology	**Detection Tools**	**Identification Tools**	**Total Tools**
Abstract Interpretation	2	1	3
AI-based Methods	24	33	57
Code Instrumentation	3	0	3
Control Flow Analysis	2	0	2
Disassembly Analysis	2	1	3
Formal Verification	8	17	25
Fuzzing	20	10	30
Model-Based Testing	5	6	11
Mutation Testing	3	2	5
Pattern Matching and Syntactical Analysis	14	21	35
Runtime Verification	6	6	12
Symbolic Execution	30	26	56
Taint Analysis	0	7	7
Visualization Analysis	6	1	7
Total	**125**	**131**	**256**

In summary, this section has explored various tools and methodologies for analyzing and securing SCs, including runtime verification, symbolic execution, and fuzz testing. As detailed in Table 34, AI-based methods and symbolic execution stand out with the highest number of tools—57 and 56, respectively—demonstrating the field’s strong emphasis on their flexibility and effectiveness in addressing complex SCs vulnerabilities. In contrast, categories such as code instrumentation and control flow analysis include fewer tools, indicating either limited adoption or opportunities for further research and development. This analysis highlights both the dominant role of AI and symbolic execution and the potential for innovation in less-explored methodologies to create a more balanced and comprehensive security toolkit.

To support a clearer understanding of the landscape, we visualized the methodology distribution in Figure 13. This structured overview serves as a practical reference for developers and researchers, allowing them to quickly identify tools aligned with specific security needs and recognize areas where current approaches may fall short. By providing this organized framework, we aim to support more informed decision-making in the selection and development of SCs security tools.

Additionally, we examined the wide range of vulnerabilities addressed by the tools. Given the breadth of coverage, we grouped vulnerabilities into broader categories for clarity. Table 36 focuses on functional vul-

nerabilities, while Table 37 outlines structural or design-level issues. This classification provides a structured approach to evaluating which tools and methodologies are best suited to detect specific types of vulnerabilities, supporting more targeted and effective security strategies.

Moreover, we provide in Table 35 an overview of how different methodologies correspond to various categories of vulnerabilities. Approaches such as AI-based methods, fuzzing, runtime verification, symbolic execution, formal verification, and pattern matching and syntactic analysis demonstrate notable adaptability. Because these methodologies are capable of addressing multiple vulnerability categories, they are especially effective when a comprehensive detection strategy is required. Such methods are particularly valuable in scenarios where different types of risks must be mitigated simultaneously, offering more holistic protection for SCs.

In contrast, methodologies such as abstract interpretation, control flow analysis, disassembly analysis, and visualization analysis exhibit more limited applicability, typically addressing a narrower range of vulnerability types. These approaches may serve more specialized roles in vulnerability detection, offering analytical depth rather than broad coverage. Our overall assessment of methodology coverage indicates a clear focus on common vulnerabilities, including gas-related issues, reentrancy and call problems, access control, and logic or data flow weaknesses. However, categories such as address and function call issues, visibility and scope concerns, and mathematical or computational errors receive relatively little attention across methodologies. This reveals potential gaps where existing tools may be less effective, suggesting opportunities for further research and the development of new tools to enhance detection coverage in these underrepresented areas.

While some methodologies offer broad coverage, no single approach is sufficient to comprehensively address all categories of vulnerabilities. This underscores the need for integrated solutions that combine multiple methodologies to effectively detect both code-level and system-level vulnerabilities in SCs. By understanding the strengths and limitations of each approach, we can guide future efforts toward enhancing SCs security, addressing the identified gaps, and fostering innovation in underexplored areas of vulnerability detection.

Table 35: Summary of Vulnerabilities in Each Category

Methodology	Integer Issues	Gas-Related Vulnerabilities	Reentrancy and Call Issues	Timestamp and Time-Related Vulnerabilities	Access Control and Security	Logic and Data Flow Issues	Error Handling and Exception Issues	Contract Design and Standards	Mathematical and Computational Errors	Execution and Performance Issues	Miscellaneous Issues	Visibility and Scope Issues	Transaction and Order Dependencies	Address and Function Call Issues	Specific Contract and Code Issues	Randomness and Predictability Issues	Specific Functionality Issues
Abstract Interpretation		✓			✓												
AI-based Methods	✓	✓	✓	✓	✓	✓	✓					✓	✓	✓	✓		✓
Code Instrumentation	✓	✓	✓	✓	✓	✓											
Control Flow Analysis	✓	✓			✓												
Disassembly Analysis			✓	✓	✓	✓											
Formal Verification	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
Fuzzing	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
Model-Based Testing	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓						
Mutation Testing			✓														
Pattern Matching and Syntactical Analysis	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
Runtime Verification	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
Symbolic Execution	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
Taint Analysis	✓	✓	✓	✓	✓	✓										✓	
Visualization Analysis		✓				✓		✓									

3.2 Evaluation Criteria, Analysis, and Discussion

Evaluating SCs detection tools requires a methodical approach to ensure the results are accurate and useful. In this thesis, we conduct a systematic evaluation after identifying a comprehensive set of tools through a defined search and screening process. The tools are divided into two main groups: academic and industry tools. This separation enables us to evaluate them more effectively, as the goals and needs within each group

Table 36: Classification of Functional Vulnerabilities

Integer Issues	Gas-Related Vulnerabilities	Reentrancy and Call Issues	Timestamp and Time-Related Vulnerabilities	Logic and Data Flow Issues	Error Handling and Exception Issues	Mathematical and Computational Errors	Execution and Performance Issues
Array overflow	Call with hardcoded gas amounts	Arbitrary contract termination	Abuse of block information	Array access and bounds issues	Assertion violations	Array access verifications	Assembly and low-level operations
Batch overflow bugs	ETH transfer inside the loop	Arbitrary ether extraction	Block information dependency	Assertion failures	Deadlock detection	Arbitrary write	Bad randomness
Divide before multiply	Excessive gas consumption in loops	Arbitrary sending of Ether	Block number dependency	Balance equality and update issues	Error handling and recovery	Mathematical errors	Code quality and readability
Division by zero	Extra gas consumption	Arbitrary value transfer	Block timestamp	Constructor name error	Exception disorder	Arithmetic bugs (specific to Osiris)	Data leakage and network eavesdropping
Integer division	Gas consumption issues	Call depth issues	Block values as a proxy for time	Contract deprecation	Exception handling		Denial Of Service
Integer overflow and underflow	Gas limit underestimation	Call graph issues	Dependence on predictable variables vulnerability	Control flow issues	Function return default value		External API
Loop overflows	Gas optimization and cost management	Call injection	Dependency of block number and timestamp	Cross-channel chaincode invocation	Inline assembly		Front-running
Missing checks for out-of-bound arithmetic operations	Gas-related inefficiencies	Call stack risks	Price-gouging TOD vulnerabilities	Cross-function calls	Mishandled exception		Liveness
Range query risks	Gasless send	Call to the unknown	System timestamp	DAO bug	Missing the Transfer event		Price manipulation attacks
Signedness bugs	Out-of-Gas DoS vulnerabilities	Code injection via callcode and delegatecall	Time manipulation	Data flow vulnerabilities	Requirement violation		Race condition
Truncation bugs	Out-of-gas exceptions	Compositional reentrancy	Timestamp dependency	Data handling and logic flow	SWC-129 typological error		Safety properties
Type conversion errors	Potential denial-of-service attacks	Controlled delegatecall	Transaction environment dependency	Data integrity issues	Unbounded safety properties		Scarcity defects
Unchecked math	Redundant SSTORE	Create-based reentrancy	TOD	Effectively Callback Free (ECF) violations	Unhandled and unchecked errors		
Unsafe type inference	Transfer forwards all gas	Cross-contract vulnerabilities		Excessive code complexity	Unexpected revert		
	Unbounded mass operations	Cross-function reentrancy		Execution and functionality			
	Under-optimized storage patterns	Delegatecall misuse		Functional and computational correctness			
		Delegated reentrancy		Invariant violations			
		DoS by external contract		Invariants, preconditions, and postconditions verification			
		ERC-721 reentrancy		Invalid jump destinations			
		Execution of unauthorized code		Logic and computation issues			
		Exploitation of fallback functions		Memory corruption and access errors			
		External call issues		Memory and variable shadowing issues			
		External file accessing		Memory safety violations			
		External library calling		No check after contract invocation			
		Failed external calls		Potential interaction flow problems			
		Fake deposit		Pseudo-random number generator (PRNG) issues			
		Insecure message calls		Read-write conflict			
		Insecure modifiers		Redundant fallback function			
		Missing checks for failing external calls		Redundant refusal of payment			
		Multiple send		Reified object addresses			
		Nested call		Semantic bugs			
		Reentrancy		State management and transition issues			
		Repetitive call functions		State pollution			
		Same-function reentrancy		State reachability			
		Send instead of transfer		Storage access patterns			
		Single-entrancy		Tainted state variables			
		System command execution		Token handling issues			
		Unchecked external call		Transaction races			
		Unchecked low-level calls		Uninitialized storage pointers			
		Unchecked send and call		Unrestricted Deposit Emitting (UDE)			
		Unsafe delegate call		Visibility modifier errors			

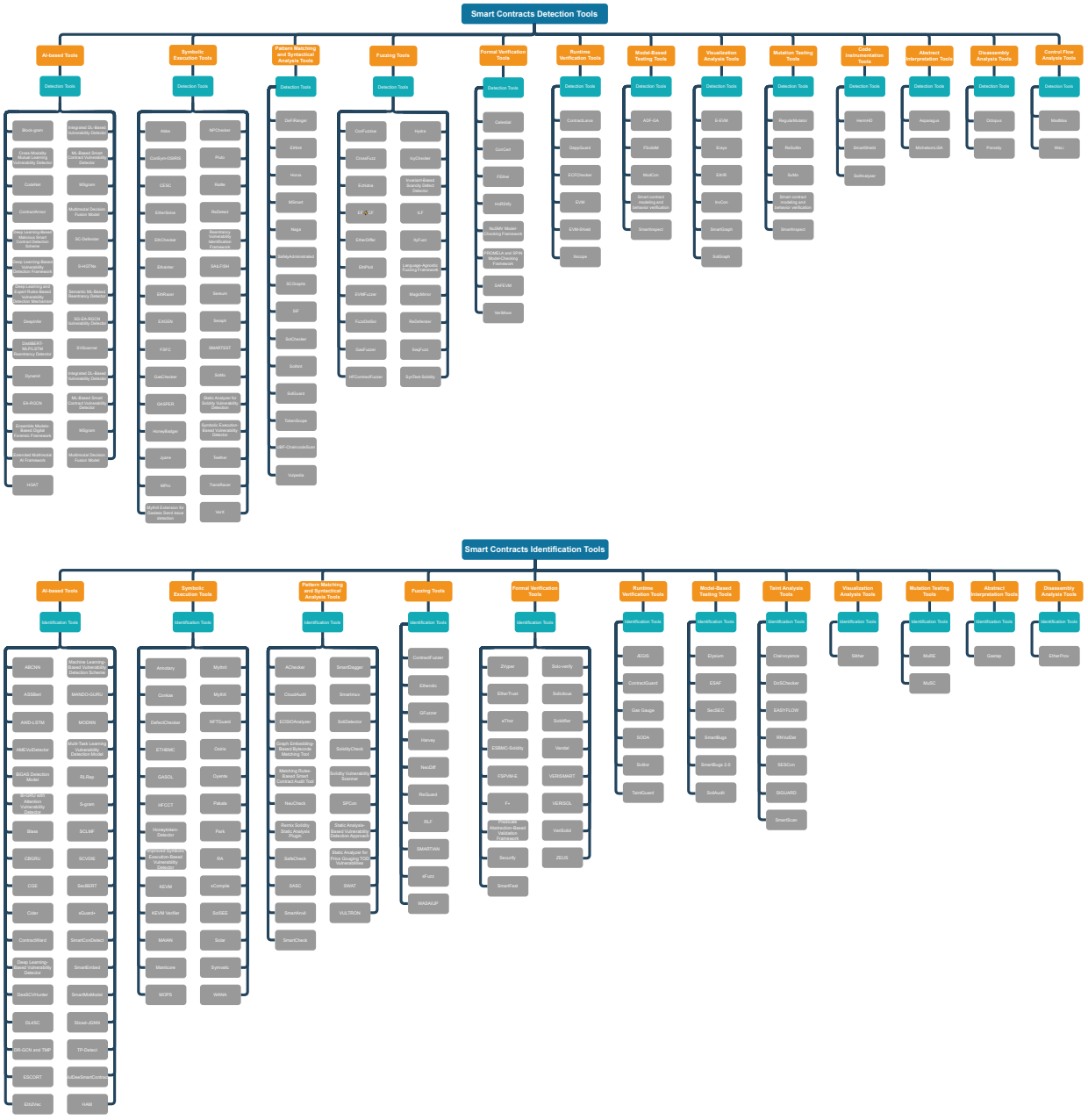

Figure 13: Taxonomy of Smart Contract Vulnerability Detection (top) and Identification (bottom) Tools

are often quite distinct. Each group is assessed using specific criteria that reflect their intended use and context. The evaluations in this section are based on the technical documentation provided by the original authors of the tools. In the following subsections, we analyze both academic and industry tools by grouping them under static, dynamic, and hybrid categories. We explain the criteria used to compare tools in each

Table 37: Classification of Structural or Design Vulnerabilities

Access Control and Security	Contract Design and Standards	Miscellaneous Issues	Visibility and Scope Issues	Address and Function Call Issues	Specific Contract and Code Issues	Specific Functionality Issues
Access control permission bugs and role assignment issues	Change of address	Balance equality and checks	Data leakage when using private	Has-short-address	Deprecated constructions	Array length manipulation
Arbitrary jump with function type variable	Change of parameters	Byte array	Misuse of approve function in ERC20 library	Short addresses	False return of ERC20	Checking for strict balance equality
Authentication bypass	Compiler version issues	Concurrency and race condition issues	Misuse of pure function	Unknown addresses	Misuse of multiple return values in internal and private function	Redundant fallback reject
Centralized security risks	Contract deprecation	Control hijack and unauthorized code execution	Misuse of revert require		Misuse of transfer function in loop	This.balance equality checkpoint
Destroyable contract	Contracts as black holes	Ether handling issues (includes Liquid ether, Locked ether, Leaking ether, Ether freezing)	Misuse of view functions			Use of insecure math functions
Fake logs	Custom property violations	Fake transfers and notifications	Misuse of visibility			
Malicious addresses and libraries	ERC20 standard violations	Force-sending ether by suicide				
Missing authorization verification	Immutability	Global variable				
Proxy token	Implicit visibility level	Greedy contract				
Public Burn	Incorrect implementation of functional specifications	Hash collisions				
Restricted and unrestricted writes	Locked Ether	Honeypots				
Security breaches	Minting and burning capabilities	Implicit privilege leakage				
Stateless signature verification	Missing constructor	Invalid input data				
Suicide contract	Missing Requirements	Keeping secrets				
Tainted delegatecall	Parity Wallet Hacks	Lack of standard events				
Tx.origin misuse and vulnerabilities	Potential non-compliance issues	Risky Mutable Proxy				
Unauthorized access and actions	Security best practices	Stack overflow and size limit issues				
Unprotected function and visibility issues	Use of deprecated Solidity functions	Strict check for balance				
Unprotected or unsafe self-destruct usage	Using fixed point number type	Style guide violation				
Unprotected ownership and permission theft	Write to arbitrary storage location	Type casts				
Unsecured balance						
Unseparated signing domain						
Unlimited Minting						
Untrusted cross-contract invocations						
Wallet griefing						
Weak randomness						

group. This classification is used because it aligns with the primary techniques in SCs analysis and provides a clear structure for our evaluation.

3.2.1 Evaluating Academic Tools

In SCs security, academic tools play a key role in advancing research and analysis, as they often introduce new ideas and offer deeper insight into emerging vulnerabilities. To evaluate the effectiveness and capabilities of these tools, we apply a set of criteria tailored to the specific needs of academic research. These criteria assess whether the tools are capable of accurately detecting vulnerabilities, adaptable to a variety of research contexts, well-documented to support transparency and reproducibility, and able to integrate with existing research infrastructures. The following criteria are defined and used in this evaluation to assess the academic tools:

- **Flexibility**: The tool's adaptability to various SCs platforms is evaluated, as this is crucial for its applicability in diverse research scenarios.
- **Detailed Documentation and Publication**: The availability and comprehensiveness of documentation and publications are assessed, reflecting the tool's transparency and reliability.
- **Integration Capabilities**: The tool's ability to integrate with existing systems and tools is considered, facilitating seamless incorporation into research workflows.
- **Automation**: The tool is assessed to determine whether it offers automated functionalities, which can significantly impact the efficiency of academic research processes.

- **Comprehensive Coverage**: The tool's ability to effectively handle a wide range of SCs vulnerabilities is assessed, ensuring thorough analysis.
- **Reliable Testing and Evaluation**: Using real-world SCs datasets for testing and validation is assessed. This evaluation demonstrates the tool's applicability to practical scenarios.
- **Extensibility**: The tool's potential for extension and customization is considered. This highlights its adaptability to evolving research needs and future applications.

As mentioned above, academic tools are categorized into three subsections (3.2.1.1, 3.2.1.2, and 3.2.1.3), each accompanied by a table for tool comparison (Tables 38, 39, and 40). These tools are evaluated using a binary scale (Yes/No) across the selected criteria, focusing on whether each feature is present or absent. The evaluation results are then used to sort the tools within each table according to their cumulative scores. This approach ensures a consistent and fair basis for comparison, allowing us to assess and rank the tools based on their overall capabilities. By presenting the results in this structured way, we provide a clear overview of each tool's strengths and limitations and offer practical insights to guide future research, development, and tool selection.

3.2.1.1 Evaluation of Static Analysis Tools

Static analysis tools are particularly significant in evaluating academic tools for SCs security, as they are capable of analyzing code without the need for execution. This section focuses specifically on static academic tools, assessing their capabilities and effectiveness in identifying vulnerabilities within SCs. Evaluating these tools is essential for understanding both their strengths and the areas where they may require improvement. The comparative details of the tools are presented in Table 38, while a visual summary of feature support across the evaluated tools is provided in Figure 14.

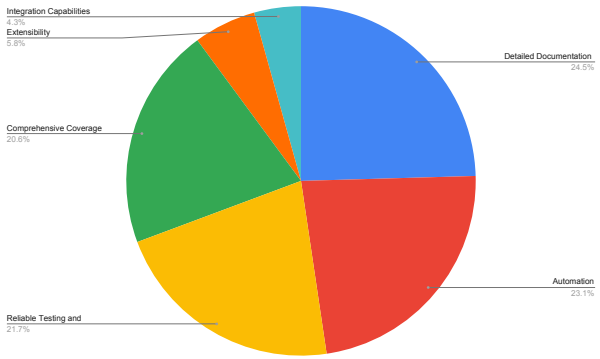

Figure 14: Percentage of Static Academic Tools Supporting Each Feature

The top 20% of tools in this evaluation consistently satisfy most or all of the defined criteria, indicating high overall quality and maturity. Tools such as ESCORT and SoliDetector stand out due to their strong flexibility, well-documented implementations, and seamless integration with research and development workflows. Additionally, they excel in areas such as automation and extensibility, making them suitable for a broad

range of academic and practical applications. The consistent performance across multiple criteria suggests that these tools are robust, thoughtfully designed, and capable of supporting advanced SCs analysis tasks.

The middle 60% of tools demonstrate moderate alignment with the evaluation criteria. Tools in this group, including sCompile and Zeus, typically demonstrate solid flexibility and acceptable integration support, but often fall short in terms of extensibility. This inconsistency suggests that while these tools can be helpful in specific scenarios, they may not fully meet the needs of more complex research or development efforts. The mixed performance highlights the need for targeted improvements, especially in areas such as modularity, automation, or broader use-case adaptability.

The bottom 20% of tools meet the fewest evaluation criteria, indicating significant limitations in their current form. Tools like Porosity and AChecker often lack core features such as automation, comprehensive documentation, and extensibility, which restrict their practical applicability—particularly in more demanding or large-scale environments. Our analysis suggests that these tools would benefit from substantial refinement and feature expansion to align more closely with user needs and become viable options in academic research or production-level use.

Several key insights emerge from this evaluation. First, we observe strong performance across many tools in flexibility, documentation, and integration, which suggests that developers are prioritizing usability and adaptability—an encouraging trend for academic workflows. However, the widespread absence of automation and limited coverage across many tools remains a concern. These gaps suggest that users may encounter considerable manual effort and reduced analytical scope, ultimately affecting overall efficiency and limiting practical applicability. In addition, the lack of extensibility in a significant number of tools restricts their ability to evolve in tandem with emerging research requirements and technologies.

To address these shortcomings, future development efforts should focus on improving automation and broadening the scope of vulnerability coverage. Incorporating more automated processes can reduce user burden, increase consistency, and enhance analysis speed. Expanding coverage enables tools to handle a broader range of vulnerabilities and use cases, resulting in more comprehensive and reliable outcomes. Furthermore, enhancing tool extensibility will support better scalability and integration with other platforms, enabling continued adaptation as SCs evolve. By addressing these areas, we can move toward more robust, versatile, and user-centered academic tools that better align with the ongoing needs of the research community.

Table 38: Static Academic Tools Evaluation

Tool	Flexibility	Detailed Documentation	Integration Capabilities	Automation	Comprehensive Coverage	Reliable Testing and Evaluation	Extensibility
ESCORT	Y	Y	Y	Y	Y	Y	Y
SoliDetector	N	Y	Y	Y	Y	Y	Y

Continued on next page

SIF	N	Y	Y	Y	Y	Y	Y
Solc-Verify	N	Y	Y	Y	Y	Y	Y
Vandal	N	Y	Y	Y	Y	Y	Y
Predicate Abstraction-Based Validation Framework (Unnamed Tool)	N	Y	N	Y	Y	Y	Y
Eth2Vec	N	Y	N	Y	Y	Y	Y
Gastap	N	Y	Y	Y	Y	Y	N
sCompile	Y	Y	Y	N	Y	Y	N
SolGuard	N	Y	Y	Y	N	Y	Y
VeriSol	N	Y	Y	Y	Y	Y	N
Zeus	Y	Y	Y	Y	Y	N	N
Erays	N	Y	Y	Y	Y	Y	N
eThor	N	Y	Y	Y	Y	Y	N
Gasper	N	Y	N	Y	Y	Y	Y
MadMax	N	Y	N	Y	Y	Y	Y
Oyente	N	Y	N	Y	Y	Y	Y
VerX	N	Y	N	Y	Y	Y	Y
Conkas	N	Y	N	Y	Y	Y	Y
SAFEVM	N	Y	Y	Y	Y	Y	N
EthBMC	N	Y	Y	Y	Y	Y	N
Multi-Task Learning Vulnerability Detection Model (Unnamed Tool)	N	Y	N	Y	Y	Y	Y
CBGRU	N	Y	N	Y	Y	Y	Y
S-HGTNs	Y	Y	Y	Y	Y	N	N
sGuard+	N	Y	Y	Y	Y	Y	N
ESBMC-Solidity	N	Y	N	Y	Y	Y	Y
SmartDagger	N	Y	N	Y	Y	Y	Y
Symvalic	N	Y	N	Y	Y	Y	Y
Remix Solidity Static Analysis Plugin (Unnamed Tool)	N	Y	Y	Y	Y	Y	N
SecBERT	N	Y	N	Y	Y	Y	Y
SmartFast	N	Y	Y	Y	Y	Y	N
PROMELA and SPIN Model-Checking Framework (Unnamed Tool)	N	Y	N	Y	Y	Y	Y
NFTGuard	N	Y	N	Y	Y	Y	Y
Ethainter	N	Y	N	Y	Y	Y	N
SESCon	N	Y	N	Y	Y	Y	N
SmartAnvil	N	Y	Y	N	Y	Y	N

Continued on next page

SmartEmbed	N	Y	N	Y	Y	Y	N
Solidifier	N	Y	N	Y	Y	Y	N
Solidityche ck	N	Y	N	Y	Y	Y	N
VeriSmart	N	Y	N	Y	Y	Y	N
VeriSolid	N	Y	N	Y	Y	Y	N
DefectChecker	N	Y	N	Y	Y	Y	N
S-GRAM	N	Y	N	Y	Y	Y	N
EtherTrust	N	Y	Y	Y	N	Y	N
EthIR	N	Y	Y	Y	N	Y	N
GasChecker	N	Y	Y	Y	Y	N	N
Osiris	N	Y	N	Y	Y	Y	N
SmartShield	N	Y	N	Y	Y	Y	N
GASOL	N	Y	Y	Y	Y	N	N
MODNN	N	Y	N	Y	Y	Y	N
ContractWard	N	Y	N	Y	Y	Y	N
CGE	Y	Y	N	Y	Y	N	N
SWAT	N	Y	N	Y	Y	Y	N
CodeNet	N	Y	N	Y	Y	Y	N
SCVDIE	N	Y	N	Y	Y	Y	N
Deep Learning-Based Malicious SCs Detection Scheme (Unnamed Tool)	N	Y	N	Y	Y	Y	N
ABCNN	N	Y	N	Y	Y	Y	N
AWD-LSTM	N	Y	N	Y	Y	Y	N
Machine Learning-Based Vulnerability Detection Scheme (Unnamed Tool)	N	Y	N	Y	Y	Y	N
MSmart	N	Y	N	Y	Y	Y	N
Blass	N	Y	N	Y	Y	Y	N
SafeCheck	N	Y	N	Y	Y	Y	N
Sliced-JGNN	N	Y	N	Y	Y	Y	N
Extended Multimodal AI Framework (Unnamed Tool)	N	Y	N	Y	Y	Y	N
SVScanner	N	Y	N	Y	Y	Y	N
Multimodal Decision Fusion Model (Unnamed Tool)	N	Y	N	Y	Y	Y	N
Deep Learning-Based Vulnerability Detection Framework (Unnamed Tool)	N	Y	N	Y	Y	Y	N
Block-gram	N	Y	N	Y	Y	Y	N
Ensemble Models-Based Digital Forensic Framework (Unnamed Tool)	N	Y	N	Y	Y	Y	N
EtherSolve	N	Y	N	Y	Y	Y	N

Continued on next page

TaintGuard	N	Y	N	Y	Y	Y	N
ContractArmor	N	Y	N	Y	Y	Y	N
CrossFuzz	N	Y	N	Y	Y	Y	N
EA-RGCN	N	Y	N	Y	Y	Y	N
SG-EA-RGCN Vulnerability Detector (Unnamed Tool)	N	Y	N	Y	Y	Y	N
Vulpedia	N	Y	N	Y	Y	Y	N
MichelsonLiSA	N	Y	N	Y	Y	Y	N
ExGen	N	Y	N	Y	Y	Y	N
Xscope	N	Y	N	Y	Y	Y	N
Naga	N	Y	N	Y	Y	Y	N
ConSym-OSIRIS	N	Y	N	Y	Y	Y	N
MANDO-GURU	N	Y	N	Y	Y	Y	N
SolSEE	N	Y	N	Y	Y	N	Y
SPCon	N	Y	N	Y	Y	Y	N
Matching Rules-Based SCs Audit Tool (Unnamed Tool)	N	Y	N	Y	Y	N	Y
Honeytoken-Detector	N	Y	N	Y	Y	Y	N
Improved Symbolic Execution-Based Vulnerability Detector (Unnamed Tool)	N	Y	N	Y	Y	N	Y
EOSIOAnalyzer	N	Y	N	Y	Y	Y	N
SC-Defender	N	Y	N	Y	Y	Y	N
SolChecker	N	Y	N	Y	Y	Y	N
SCGraphs	N	Y	N	Y	N	Y	Y
VeriMove	N	Y	N	Y	Y	Y	N
UBF-ChaincodeScan	Y	Y	N	Y	Y	N	N
DL4SC	N	Y	N	Y	Y	Y	N
TP-Detect	N	Y	N	Y	Y	Y	N
WaLi	N	Y	N	Y	Y	Y	N
SmartConDetect	N	Y	N	Y	Y	Y	N
Clairvoyance	N	Y	N	Y	Y	Y	N
Graph Embedding-Based Bytecode Matching Tool (Unnamed Tool)	N	Y	N	Y	Y	Y	N
ConCert	Y	Y	N	N	Y	N	Y
F* Verification	N	Y	N	N	Y	Y	Y
DR-GCN and TMP	N	Y	N	Y	Y	Y	N
KEVM Verifier	N	Y	N	Y	Y	Y	N
NuSMV Model-Checking Framework (Unnamed Tool)	N	Y	N	Y	Y	Y	N

Continued on next page

MSgram analysis	N	Y	N	Y	Y	Y	N
NPChecker	N	Y	N	Y	Y	Y	N
RegularMutator	N	Y	N	Y	Y	Y	N
Static Analyzer for Solidity Vulnerability Detection	N	Y	N	Y	Y	Y	N
SCs Modeling and Behavior Verification (Unnamed Tool)	N	Y	N	Y	Y	Y	N
Solicitous	N	Y	N	Y	Y	Y	N
VulDeeSmartContract	N	Y	N	Y	N	Y	Y
Siguard	N	Y	N	Y	Y	Y	N
Cross-Modality Mutual Learning Vulnerability Detector (Unnamed Tool)	N	Y	N	Y	N	Y	Y
Smart-Graph	N	Y	N	N	Y	Y	N
SmartInspect	N	Y	N	Y	N	N	Y
RA	N	Y	N	Y	N	Y	N
HermHD	N	Y	N	Y	N	Y	N
Static Analysis-Based Vulnerability Detection Approach (Unnamed Tool)	N	Y	N	N	Y	Y	N
TransRacer	N	Y	N	Y	N	Y	N
Smartmuv	N	Y	N	Y	N	Y	N
Asparagus	N	Y	N	Y	N	Y	N
Elysium	N	Y	N	Y	Y	N	N
2Vyper	N	Y	N	Y	Y	Y	N
DoSChecker	N	Y	N	Y	N	Y	N
MuRE	N	Y	N	Y	N	N	Y
SCLMF	N	Y	N	Y	Y	N	N
HoRStify	N	Y	N	Y	Y	N	N
Static Analyzer for Price Gouging TOD Vulnerabilities (Unnamed Tool)	N	Y	N	Y	N	Y	N
Semantic ML-Based Reentrancy Detector (Unnamed Tool)	N	Y	N	Y	N	Y	N
Invariant-Based Scarcity Defect Detector (Unnamed Tool)	N	Y	N	Y	N	Y	N
ReDetect	N	Y	N	Y	N	Y	N
Transaction-based analysis with LSTM network	N	Y	N	Y	N	Y	N
SMARTEST	N	Y	N	Y	N	Y	N
AChecker	N	Y	N	N	N	Y	N
Porosity	N	Y	N	N	N	N	N

3.2.1.2 Evaluation of Dynamic Analysis Tools

Evaluating dynamic academic tools for SCs security is essential for identifying their strengths and areas needing improvement, as these tools can detect runtime vulnerabilities. In this thesis, we analyze the ca-

pabilities of dynamic tools to understand their effectiveness in identifying SCs vulnerabilities. Detailed comparisons of these tools can be found in Table 39 with a pie chart summarizing the supported features shown in Figure 15.

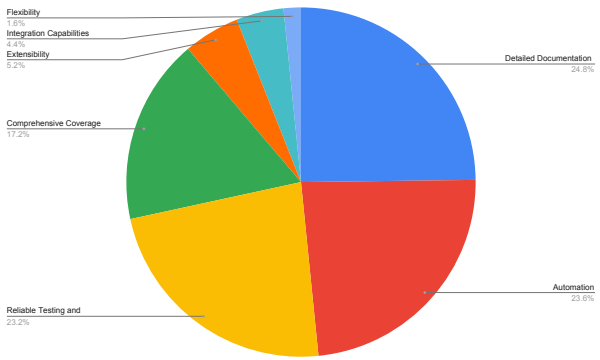

Figure 15: Percentage of Dynamic Academic Tools Supporting Each Feature

The top 25% of tools in this category, including ContractLarva, ModCon, and SODA, demonstrate strong performance across nearly all evaluation criteria. These tools exhibit a high degree of flexibility and support a wide range of SCs platforms, making them well-suited for diverse research contexts. They are backed by thorough documentation and academic publications, ensuring transparency, reproducibility, and ease of adoption. Their integration capabilities allow for seamless incorporation into existing research workflows, and they include automated functionalities that enhance research efficiency. Furthermore, their comprehensive vulnerability coverage allows for in-depth and wide-ranging analysis, making these tools especially valuable for detailed academic studies.

The middle 50% of tools, such as GFuzzer, DappGuard, and EVMFuzzer, display a combination of strengths and weaknesses across the evaluation dimensions. These tools generally perform well in terms of flexibility and vulnerability coverage but often fall short in areas such as testing reliability, extensibility, and evaluation support. Documentation quality is inconsistent—some tools provide adequate usage guidance, while others lack essential technical details. Integration capabilities are present but less mature than those found in the top-tier tools. Automation is implemented to varying degrees, which affects the consistency and overall efficiency of these tools. While this group can address many SCs vulnerabilities, targeted improvements in documentation, extensibility, and evaluation would enhance their utility in academic research.

The bottom 25% of tools, including RLRep, SeqFuzz, and EVM, exhibit limited performance across several critical criteria. These tools lack flexibility and offer minimal integration support, which limits their applicability in broader research environments. Documentation is often sparse or incomplete, creating challenges for adoption and effective use. Automation features are either missing or underdeveloped, resulting in greater manual effort and increased chances of error. In addition, the scope of vulnerability coverage is narrow, and the lack of extensibility makes it difficult for these tools to adapt to evolving research requirements. These limitations significantly reduce their usefulness for complex or long-term SCs research projects.

Based on our evaluation, we propose several recommendations to improve the overall utility and effectiveness of dynamic academic tools. First, integrating real-world datasets more extensively would improve

the practical applicability and reliability of these tools, making them more reflective of real deployment scenarios. Second, enhancing documentation across all tools is essential to support user adoption, reduce learning curves, and ensure accurate implementation. Third, strengthening integration capabilities will allow for smoother adoption within existing research workflows and enable broader applicability. In addition, expanding automation features will significantly reduce manual effort and increase the overall efficiency of vulnerability analysis. Finally, improving extensibility will ensure that tools can evolve with emerging research challenges and remain valuable for future academic and practical applications. By addressing these areas, we can support the development of more robust, usable, and scalable dynamic tools for SCs security analysis.

Table 39: Dynamic Academic Tools Evaluation

Tool	Flexibility	Detailed Documentation	Integration Capabilities	Automation	Comprehensive Coverage	Reliable Testing and Evaluation	Extensibility
SODA	Y	Y	Y	Y	Y	Y	Y
ContractLarva	Y	Y	Y	Y	N	Y	Y
ModCon	Y	Y	Y	Y	Y	N	Y
sFuzz	N	Y	Y	Y	Y	Y	Y
ReGuard	N	Y	Y	Y	Y	Y	Y
GFuzzer	N	Y	Y	Y	Y	Y	Y
DappGuard	N	Y	Y	Y	Y	Y	Y
SynTest-Solidity	N	Y	Y	Y	Y	Y	N
Language-Agnostic Fuzzing Framework (Unnamed Tool)	N	Y	Y	Y	Y	N	Y
Bi-GRU with Attention Vulnerability Detector (Unnamed Tool)	N	Y	N	Y	Y	Y	Y
EVM-Shield	N	Y	N	Y	Y	Y	Y
EFCF	N	Y	N	Y	Y	Y	Y
AEGIS	N	Y	N	Y	Y	Y	Y
Hydra	N	Y	Y	Y	Y	Y	N
Solitor	N	Y	N	Y	N	Y	Y
ContractFuzzer	N	Y	N	Y	Y	Y	N
Etherolic	N	Y	N	Y	Y	Y	N
EVMFuzzer	N	Y	N	Y	Y	Y	N
Easyflow	N	Y	N	Y	Y	Y	N
ASSBert	N	Y	N	Y	Y	Y	N

Continued on next page

RNVulDet	N	Y	N	Y	Y	Y	N
SuMo	N	Y	N	Y	Y	Y	N
RLRep	N	Y	N	Y	Y	Y	N
InvCon	N	Y	N	Y	Y	Y	N
AMEVulDetector	N	Y	N	Y	Y	Y	N
FuzzDelSol	N	Y	N	Y	Y	Y	N
EtherDiffer	N	Y	N	Y	Y	Y	N
NeoDiff	N	Y	Y	Y	Y	N	N
SeqFuzz	N	Y	N	Y	Y	Y	N
MagicMirror	N	Y	N	Y	Y	Y	N
Ethchecker	N	Y	N	Y	Y	Y	N
HGAT	N	Y	N	Y	Y	Y	N
Symbolic Execution-Based Vulnerability Detector (Unnamed Tool)	N	Y	N	Y	Y	Y	N
Integrated DL-Based Vulnerability Detector (Unnamed Tool)	N	Y	N	Y	Y	Y	N
ABBE	N	Y	N	Y	Y	Y	N
ADF-GA	N	Y	N	Y	Y	Y	N
CESC	N	Y	N	Y	Y	Y	N
EthRacer	N	Y	N	Y	Y	Y	N
FSFC	N	Y	N	Y	Y	Y	N
Gasfuzzer	N	Y	N	Y	Y	Y	N
ILF	N	Y	N	Y	Y	Y	N
Mythril extension for ’gasless send’ issue detection	N	Y	N	Y	Y	Y	N
ItyFuzz	N	Y	N	Y	Y	Y	N
E-EVM	Y	Y	N	N	N	Y	N
Sereum	N	Y	N	Y	N	Y	N
Park	N	Y	N	N	Y	Y	N
BiGAS	N	Y	N	Y	N	Y	N
DistilBERT-MLP/LSTM Reentrancy Detector	N	Y	N	Y	N	Y	N
DeepInfer	N	Y	N	Y	N	Y	N
TokenScope	N	Y	N	Y	N	Y	N
DeFiRanger	N	Y	N	Y	N	Y	N
Horus	N	Y	N	Y	N	Y	N
IcyChecker	N	Y	N	Y	N	Y	N
HFContractFuzzer	N	Y	N	Y	Y	N	N

Continued on next page

Dynamit	N	Y	N	Y	N	Y	N
ML-Based SCs Vulnerability Detector (Unnamed Tool)	N	Y	N	Y	N	Y	N
ReDefender	N	Y	N	Y	N	Y	N
Jyane	N	Y	N	Y	N	Y	N
Celestial	N	Y	N	Y	N	Y	N
ECFChecker	N	Y	N	Y	N	Y	N
EVM	N	Y	N	Y	N	Y	N
RLF	N	Y	N	N	N	Y	N

3.2.1.3 Evaluation of Hybrid Analysis Tools

Hybrid academic tools play a key role in the analysis of SCs, as they combine both static and dynamic analysis techniques to detect vulnerabilities and improve overall security. By leveraging the strengths of these complementary approaches, hybrid tools provide a more comprehensive solution for identifying and addressing security issues within SCs. In this thesis, we evaluate hybrid tools to understand their relative strengths and weaknesses, with the goal of informing future development. The comparison of these tools can be found in Table 40, and a visual summary of feature support across these tools is presented in Figure 16.

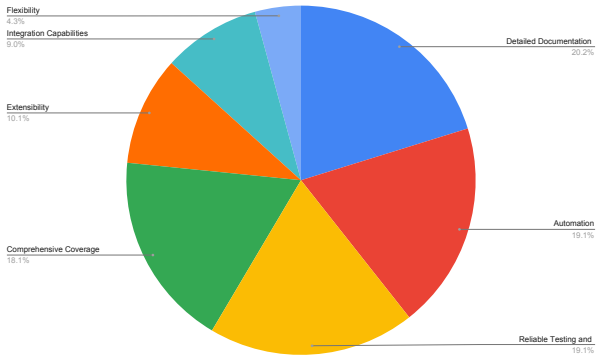

Figure 16: Percentage of Hybrid Academic Tools Supporting Each Feature

The top 30% of hybrid academic tools—including SmartBugs, KEVM, Vultron, WANA, and FSPVM-E—perform exceptionally well across nearly all evaluation criteria. These tools demonstrate strong flexibility and are adaptable across different SCs platforms. They are supported by detailed documentation and academic publications, ensuring reliability and ease of use. Their integration capabilities enable them to seamlessly integrate into existing research workflows, while their automation features enhance research efficiency. These tools offer comprehensive vulnerability coverage and utilize real-world datasets, which enhances their practical relevance. Their extensibility also enables continued adaptation to emerging research requirements, making them well-suited for long-term academic use.

The middle 50% of tools, such as MuSc, ConFuzzius, ESAF, SecSEC, and Smartian, show a balance of strengths and weaknesses. While they generally offer flexibility and include usable documentation, they

often fall short of top-tier tools in areas such as integration and automation. Although they address many SCs vulnerabilities, their coverage tends to be narrower. Support for reliable testing and evaluation is inconsistent, which can limit its practical applicability. In addition, extensibility is frequently underdeveloped, restricting their ability to adapt to new research needs or technological shifts.

The bottom 20% of tools—including TEEther, SoliAudit, and HAM—demonstrate limited capabilities across several key areas. These tools typically lack flexibility, offer minimal documentation, and have limited integration capabilities, making them more challenging to incorporate into research workflows. Automation features are often absent or underdeveloped, requiring more manual effort and increasing the potential for inefficiencies. Their coverage of SCs vulnerabilities is limited, and they rarely incorporate real-world datasets into their testing. A lack of extensibility further restricts their adaptability, reducing their value in evolving research environments.

Based on the above analysis, we identify several areas for improving the effectiveness and applicability of hybrid academic tools. First, flexibility should be strengthened to ensure that these tools can adapt to a variety of SCs platforms and environments. Improving documentation is also essential to support user adoption by providing clear, accessible, and well-structured guidance. Integration capabilities should be enhanced to allow for smoother incorporation into academic and research workflows. Ultimately, incorporating reliable testing and evaluation mechanisms will enhance the practical value and trustworthiness of these tools, thereby supporting their broader adoption in real-world and experimental contexts.

Table 40: Hybrid Academic Tools Evaluation

Tool	Flexibility	Detailed Documentation	Integration Capabilities	Automation	Comprehensive Coverage	Reliable Testing and Evaluation	Extensibility
Seraph	Y	Y	Y	Y	Y	Y	Y
Vultron	Y	Y	Y	Y	Y	Y	Y
WANA	Y	Y	Y	Y	Y	Y	Y
KEVM	N	Y	Y	Y	Y	Y	Y
SmartBugs	N	Y	Y	Y	Y	Y	Y
FSPVM-E	Y	Y	Y	Y	Y	N	Y
MuSc	N	Y	Y	Y	Y	Y	Y
ConFuzzius	N	Y	Y	Y	Y	Y	Y
Deep Learning and Expert Rules-Based Vulnerability Detection (Unnamed Tool)	N	Y	Y	Y	Y	Y	Y
ESAF	N	Y	Y	Y	Y	Y	Y
SecSEC	N	Y	Y	Y	Y	Y	Y
Smartian	Y	Y	N	Y	Y	Y	Y

Continued on next page

SmartBugs 2.0	N	Y	Y	Y	Y	Y	Y
WASAIUP	N	Y	Y	Y	Y	Y	Y
GasGauge	N	Y	Y	Y	Y	Y	N
NeuCheck	N	Y	Y	Y	Y	Y	N
HoneyBadger	N	Y	N	Y	Y	Y	Y
HFCCT	Y	Y	N	Y	Y	Y	N
SmartMixModel	N	Y	N	Y	Y	Y	Y
Solar	Y	Y	N	Y	Y	Y	N
FEther	N	Y	N	N	Y	Y	Y
SmartScan	N	N	Y	N	Y	Y	Y
EthPloit	N	Y	N	Y	Y	Y	N
TEEther	N	Y	N	Y	Y	Y	N
ContractGuard	N	Y	N	Y	Y	Y	N
MAIAN	N	Y	N	Y	Y	Y	N
SoliAudit	N	Y	N	Y	Y	Y	N
HAM	N	Y	N	Y	Y	Y	N
CloudAudit	N	Y	Y	Y	Y	N	N
Deep Learning-Based Vulnerability Detector (Unnamed Tool)	N	Y	N	Y	Y	Y	N
Pluto	N	Y	N	Y	Y	Y	N
ReSuMo	N	Y	N	Y	Y	Y	N
Mops	N	Y	N	Y	Y	Y	N
Reentrancy Vulnerability Identification Framework	N	Y	N	Y	Y	Y	N
SolAnalyser	Y	Y	N	N	N	N	Y
SafelyAdministrated	N	Y	N	Y	N	Y	N
SAILFISH	N	Y	N	Y	N	Y	N
EtherProv	N	Y	N	Y	N	Y	N
DeeSCVHunter	N	Y	N	Y	N	Y	N

In this thesis, we conduct a comprehensive analysis of academic SCs vulnerability tools, evaluating each based on seven key characteristics. Our evaluation reveals substantial variation in the presence of these qualities across different tools. As illustrated in Figure 17, most academic tools perform strongly in areas such as detailed documentation and publication (237 tools), automation (223 tools), reliable testing and evaluation (214 tools), and comprehensive coverage (192 tools). In contrast, significant shortcomings are evident in flexibility (24 tools) and integration capabilities (50 tools). Additionally, extensibility remains limited, with only 62 tools satisfying this criterion.

In conclusion, while academic tools generally offer robust documentation and automation, we identify a pressing need to enhance their flexibility, integration capabilities, and extensibility. Future development of

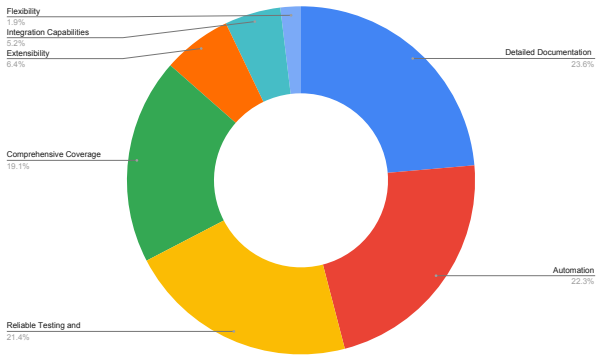

Figure 17: Percentage of All Academic Tools Supporting Each Feature

academic SCs vulnerability tools should address these limitations to ensure broader applicability, seamless integration into research workflows, and improved adaptability to evolving research requirements.

3.2.2 Evaluating Industry Tools

In this thesis, we extend the evaluation of SCs vulnerability detection tools beyond academic interest to address the practical requirements of real-world applications. Within the industry context, here security and reliability are paramount, conducting a comprehensive assessment necessitates a broader set of evaluation criteria. Our goal is to ensure that the selected tools not only effectively identify vulnerabilities but also align with the operational needs of businesses. The following criteria are defined and employed in this subsection:

- **Flexibility**: The tool's adaptability to various SCs platforms is evaluated, as this is essential for its applicability in diverse industry settings.
- **Usability and User Experience**: The tool's ease of use and overall user experience are assessed, as these factors are crucial for adoption and effective utilization in professional environments.
- **Scalability and Performance**: The tool's capacity to handle large-scale SCs analysis efficiently is evaluated, a key requirement for enterprise-level applications.
- **Integration Capabilities**: The tool's ability to integrate seamlessly with existing systems and workflows is assessed, which can save time and improve security operations.
- **Support**: The availability of user support, including documentation, tutorials, and customer service, is examined. Effective user support is crucial for troubleshooting and optimizing the tool's effectiveness in industrial settings.
- **Maintenance and Update**: The presence of a regular update schedule is evaluated, ensuring the tool remains effective and up-to-date.
- **Detailed Documentation and Publication**: The availability and clarity of documentation are assessed. Clear documentation promotes user understanding, facilitates adoption, and enhances the tool's credibility.

- **Automation**: The tool is assessed to determine whether it offers automated functionalities that can expedite vulnerability detection processes in industry settings.
- **Comprehensive Coverage**: The tool's ability to detect a wide range of SCs vulnerabilities is assessed, ensuring its effectiveness in identifying potential risks.
- **Reliable Testing and Evaluation**: The tool is evaluated to determine if it uses real-world datasets for validation and testing. This assessment indicates the tool's applicability to real-world scenarios.
- **Availability and Open Source**: The tool's availability, whether open or commercially licensed, is considered, as this can impact accessibility and transparency.
- **Detailed Vulnerability Reporting**: The tool's ability to provide actionable suggestions for repairing or mitigating detected vulnerabilities is assessed, which can significantly help in remediation efforts.
- **Repair/Mitigation Suggestion**: The tool's ability to provide practical suggestions for repairing or mitigating detected vulnerabilities is assessed, as this can significantly expedite the remediation process.
- **Extensibility**: The tool's potential for incorporating additional functionalities or features is considered, ensuring its adaptability to future needs and advancements in the field.

Similar to the approach used for academic tools, we apply the same methodology to compare industry tools. The tools are divided into three categories: static (3.2.2.1), dynamic (3.2.2.2), and hybrid (3.2.2.3), each accompanied by a comparison table (Tables 41, 42, and 43). We evaluate the tools using a binary scale (Yes/No) based on the previously defined criteria, identifying the presence or absence of specific features. The tools are then ranked according to the number of criteria they satisfy, with higher-ranking tools meeting more requirements. This approach ensures a fair and transparent comparison, effectively highlighting the strengths and limitations of each tool and offering valuable insights for future improvements.

3.2.2.1 Evaluation of Static Analysis Tools

Our evaluation of static industry tools for SCs vulnerability reveals a landscape characterized by both strengths and limitations. As presented in Table 41 and illustrated in Figure 18, we observe that detailed documentation and automation are widely available across many tools. This is a promising trend, as thorough documentation enables users to understand and utilize tools effectively, while automation reduces manual effort and streamlines workflows. Tools such as Slither, SoMo, and SmartCheck exemplify these qualities, offering both user guidance and automated functionalities.

However, notable gaps persist in other areas. Flexibility and integration capabilities are often lacking in many tools, which limits their adaptability to diverse user needs and operational environments. While SmartCheck demonstrates strong integration support, most tools fall short in this regard, which may impact their practical applicability. Furthermore, aspects such as scalability and performance are emphasized by only a few tools—SoMo being a key example—highlighting the need for more robust solutions capable of handling large-scale applications efficiently.

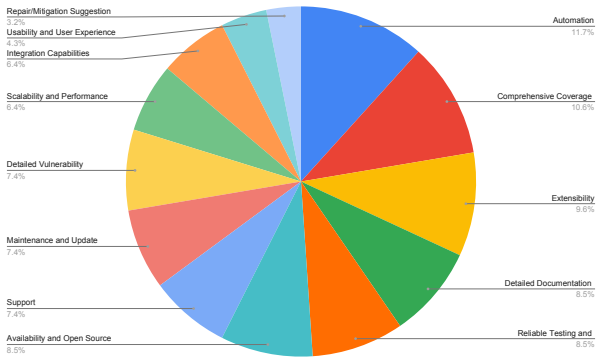

Figure 18: Percentage of Static Industry Tools Supporting Each Feature

Another area of concern identified in our evaluation is the limited emphasis on usability and user experience. Although tools such as Ethlint and Solhint demonstrate notable efforts in this regard, the majority do not prioritize user-friendly interfaces or intuitive interactions. This shortcoming can hinder broader adoption, particularly among users with limited technical expertise. Additionally, features such as comprehensive coverage and access to real-world vulnerability databases are not commonly supported, with only Slither and Cider standing out in these areas. The absence of these capabilities reduces the ability of many tools to address a full range of vulnerabilities and practical scenarios, thereby limiting their overall effectiveness.

Support and maintenance also represent critical areas for improvement. While tools like Rattle and Solhint provide solid support and receive regular updates—allowing users to depend on them for long-term use—many others fall short in maintaining ongoing reliability. This lack of consistent support can result in outdated tools and user frustration due to insufficient assistance and a failure to keep pace with emerging security threats.

Addressing the identified shortcomings requires a comprehensive and strategic approach. First, we emphasize the importance of enhancing flexibility and integration capabilities to increase the adaptability and utility of these tools across diverse environments. Developers should adopt modular architectures and provide well-documented APIs to support seamless integration with other systems. Second, improving usability and performance is essential for broader adoption. Regular interface enhancements and performance optimizations should be prioritized to ensure accessibility and efficiency.

Furthermore, expanding coverage and incorporating real-world vulnerability databases will increase the accuracy and practical relevance of the tools' analyses. Continued maintenance of documentation and the provision of reliable user support are also vital to ensure user satisfaction and tool dependability. Ultimately, fostering the development of open-source tools can encourage community engagement and contributions, resulting in ongoing improvements. By addressing these areas, we can significantly enhance the development and adoption of static industry tools, ultimately strengthening security practices across the field.

Table 41: Static Industry Tools Evaluation

Tool	Flexibility	Usability and User Experience	Scalability and Performance	Integration Capabilities	Support	Maintenance and Update	Detailed Documentation	Automation	Comprehensive Coverage	Reliable Testing and Evaluation	Availability and Open Source	Detailed Vulnerability Reporting	Repair/Mitigation Suggestion	Extensibility
Slither	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y
Securify	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y
SoMo	N	N	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	N	Y
Solhint	N	Y	N	Y	Y	Y	Y	Y	Y	N	Y	N	Y	Y
Ethlint	N	Y	N	Y	Y	Y	Y	Y	Y	N	Y	N	Y	Y
SmartCheck	N	N	Y	Y	N	N	Y	Y	Y	Y	Y	Y	N	Y
Cider	N	N	Y	N	N	Y	Y	Y	Y	Y	N	Y	N	Y
Solidity Vulnerability Scanner	N	Y	N	N	N	N	N	Y	Y	Y	N	Y	Y	Y
Rattle	N	N	N	N	Y	Y	N	Y	Y	Y	Y	N	N	Y
SASC	N	N	Y	N	N	N	Y	Y	Y	Y	N	Y	N	N
Solgraph	N	Y	N	N	Y	N	N	Y	Y	N	Y	N	N	N

3.2.2.2 Evaluation of Dynamic Analysis Tools

In our evaluation of dynamic industry tools for SCs vulnerability assessment, we observe a combination of notable strengths and some limitations. As summarized in Table 42 and illustrated in Figure 19, the four tools—Manticore, Echidna, Harvey, and Pakala—demonstrate several consistently present qualities that highlight their effectiveness. All tools provide detailed documentation and are both available and open source, ensuring transparency, accessibility, and comprehensive user guidance.

Manticore, Echidna, and Harvey exhibit a broad range of features including scalability and performance, integration capabilities, ongoing maintenance and updates, automation, comprehensive vulnerability coverage, reliable testing and evaluation, and extensibility. These capabilities reflect the tools' capacity to perform large-scale analyses, integrate smoothly with other systems, receive regular enhancements, operate efficiently with minimal manual effort, and adapt to a variety of real-world scenarios. Their extensibility further enables future improvements, positioning them as robust and versatile solutions in the dynamic analysis landscape.

Despite these strengths, our evaluation also highlights several critical shortcomings in current dynamic industry tools. The most significant limitation is the absence of repair or mitigation suggestions—none of the evaluated tools provide this functionality. This feature is essential for helping users address and resolve the vulnerabilities identified during analysis. Additionally, the support category is lacking in both Harvey and Pakala, which undermines users' ability to obtain assistance and troubleshoot effectively. Echidna and Harvey also fall short in delivering detailed vulnerability reports, limiting their ability to provide in-depth assessments. Furthermore, flexibility is a concern, as Echidna, Harvey, and Pakala do not support this feature,

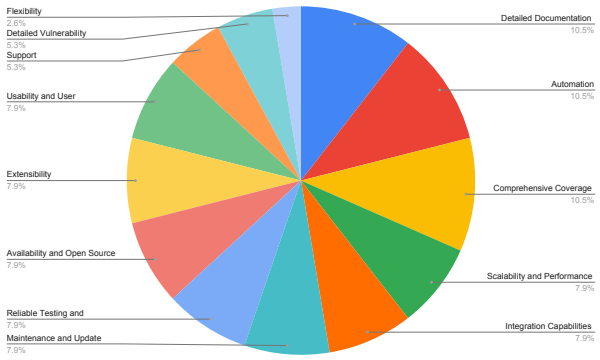

Figure 19: Percentage of Dynamic Industry Tools Supporting Each Feature

reducing their adaptability across different platforms and use cases.

It is essential to address these deficiencies to enhance the utility and impact of future dynamic industry tools. Incorporating repair and mitigation suggestions would significantly increase the practicality of these tools by offering actionable remediation steps. Improving support services would ensure users receive timely assistance, contributing to a more effective user experience. Detailed vulnerability reporting should be standardized to deliver deeper insights into detected issues. Lastly, enhancing flexibility would allow these tools to better adapt to a variety of platforms and operational environments. By targeting these areas, future dynamic tools can become more robust, user-friendly, and effective in securing SCs.

Table 42: Dynamic Industry Tools Evaluation

Tool	Flexibility	Usability and User Experience	Scalability and Performance	Integration Capabilities	Support	Maintenance and Update	Detailed Documentation	Automation	Comprehensive Coverage	Reliable Testing and Evaluation	Availability and Open Source	Detailed Vulnerability Reporting	Repair/Mitigation Suggestion	Extensibility
Manticore	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y
Echidna	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Y
Harvey	N	N	Y	Y	N	Y	Y	Y	Y	Y	N	N	N	Y
Pakala	N	Y	N	N	N	N	Y	Y	Y	N	Y	Y	N	N

3.2.2.3 Evaluation of Hybrid Analysis Tools

In our evaluation of hybrid industry tools for SCs analysis, we identify several key strengths across the four tools examined. As shown in Table 43 and Figure 20, all four tools demonstrate strong performance in flexibility, automation, comprehensive coverage, and extensibility. These capabilities suggest that the tools are well-suited for a range of SCs analysis tasks.

Additionally, three of the tools stand out due to their scalability, performance, integration capabilities, user support, and open-source availability. These features indicate that the tools are capable of handling large-

scale analyses, integrating effectively with other systems, providing meaningful assistance to users, and maintaining transparency through open-source distribution. Collectively, these strengths make hybrid tools reliable and effective solutions for addressing vulnerabilities in various industry contexts.

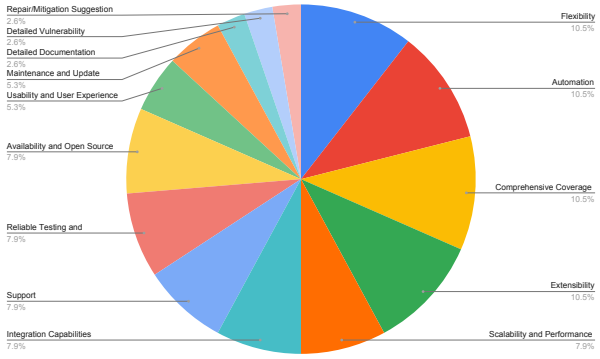

Figure 20: Percentage of Hybrid Industry Tools Supporting Each Feature

Despite the noted strengths, our evaluation of hybrid industry tools also uncovers several critical gaps that may limit their overall effectiveness. Some tools lack detailed documentation, which hinders user understanding and reduces the ease of adoption. Additionally, features such as detailed vulnerability reporting and repair suggestions are absent in certain tools—both of which are essential for enabling users to promptly and effectively address identified issues. Usability and user experience also require attention, as poor interface design can negatively impact user efficiency and satisfaction. These shortcomings underscore the need for targeted improvements to ensure that hybrid tools meet the practical standards demanded in industry settings. To address these issues, we propose several recommendations aimed at enhancing the utility and impact of SCs analysis tools. First, improving the quality and comprehensiveness of documentation will support user learning and facilitate effective tool usage. Second, integrating detailed vulnerability reporting and actionable repair suggestions will empower users to resolve issues more efficiently. Finally, enhancing the user interface and overall experience will make the tools more intuitive and accessible. By prioritizing these improvements, hybrid industry tools can become more robust, user-centered, and effective in supporting comprehensive SCs vulnerability analysis.

Table 43: Hybrid Industry Tools Evaluation

Tool	**Flexibility**	**Usability and User Experience**	**Scalability and Performance**	**Integration Capabilities**	**Support**	**Maintenance and Update**	**Detailed Documentation**	**Automation**	**Comprehensive Coverage**	**Reliable Testing and Evaluation**	**Availability and Open Source**	**Detailed Vulnerability Reporting**	**Repair/Mitigation Suggestion**	**Extensibility**
Mythril	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y
MPro	Y	N	Y	Y	Y	N	Y	Y	Y	Y	Y	N	N	Y

Table 43: Hybrid Industry Tools Evaluation (Continued)

Tool	Flexibility	Usability and User Experience	Scalability and Performance	Integration Capabilities	Support	Maintenance and Update	Detailed Documentation	Automation	Comprehensive Coverage	Reliable Testing and Evaluation	Availability and Open Source	Detailed Vulnerability Reporting	Repair/Mitigation Suggestion	Extensibility
MythX	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	N	N	N	Y
Octopus	Y	N	N	N	N	N	N	Y	Y	N	Y	N	N	Y

In our analysis of industry SCs vulnerability tools, we examine several critical categories relevant to professional and enterprise-level applications. As illustrated in Figure 21, most industry tools demonstrate strong performance in automation (19 tools), detailed documentation and publication (16 tools), and comprehensive coverage (15 tools). However, notable gaps persist in flexibility (4 tools), usability and user experience (8 tools), repair or mitigation suggestions (8 tools), and extensibility (8 tools).

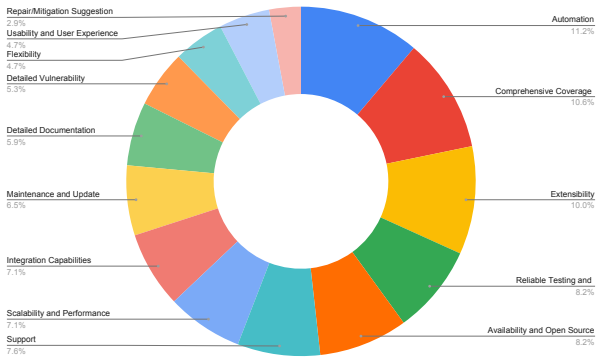

Figure 21: Percentage of All Industry Tools Supporting Each Feature

In conclusion, while industry tools generally excel in automation and documentation, our findings highlight a critical need for improvements in flexibility, user experience, and the provision of actionable repair suggestions. Future development should prioritize enhancing adaptability, user-centered design, and the ability to address identified vulnerabilities effectively. By focusing on these areas, industry tools can better support the demands of real-world applications and contribute to more robust SCs security solutions.

3.3 Practical Analysis of SCs Vulnerability Tools

This section presents a practical analysis of vulnerability detection tools for SCs, focusing on a representative subset selected based on their evaluation scores from the previous section. For industry tools, we selected those that achieved the highest scores—13 and 11 out of a possible 14. For academic tools, we included those that scored 6 and 7 out of 7. The industry tools considered included Mythril, Manticore, Echidna, Slither, and Securify, and the academic tools considered were SoliDetector, SIF, SMARTBUGS, KEVM,

Vultron, Solc-Verify, WANA, FSPVM, ESCORT, FSolidM, MuSc, ConFuzzius, ESAF, SecSEC, SmartBugs 2.0, ModCon, sFuzz, SODA, WASAIUP, Vandal, and Seraph.

During testing, we encountered several technical limitations that restricted the execution of many tools. For example, Echidna requires Solidity version 0.4.25 or higher, which is incompatible with numerous contracts in our dataset that use earlier versions. Additionally, tools such as SoliDetector, Solc-Verify, ESAF, SecSEC, SmartBugs 2.0, WASAIUP, Seraph, SODA, Vultron, and ESCORT were excluded due to the lack of publicly available code or inadequate documentation. Tools like SIF, KEVM, MuSc, and ModCon were not tested as their focus does not directly align with vulnerability detection, which is the central objective of this analysis. WANA, although theoretically flexible, was excluded due to its reliance on specific Solidity versions for WASM conversion, which were incompatible with our dataset. Ultimately, the tools tested—four industry tools (Manticore, Mythril, Slither, and Securify) and three academic tools (ConFuzzius, sFuzz, and Vandal)- were selected based on their availability, compatibility with the Solidity versions in our dataset, and alignment with the core goal of SCs vulnerability detection.

In addition to evaluating existing tools, we also developed a custom vulnerability detection model tailored to the structure and behavior of Solidity smart contracts. Our approach is graph-based and combines both code semantics and contract structure to predict vulnerability types. The pipeline is designed to operate at the function level, which enables finer-grained learning and better generalization across varying contract styles. The first step involves flattening each Solidity file to extract all top-level functions, including those nested within inherited contracts. Functions without a body—such as interface declarations or abstract functions—are excluded. Each function is then passed through a GraphCodeBERT encoder to generate a semantic embedding vector. These embeddings are intended to capture the functional meaning of the code beyond syntactic patterns. The embeddings are stored in per-contract batches, aligned with function names stripped of their contract prefixes for consistency.

To represent interactions between functions, we construct function-level call graphs for each contract. Nodes in the graph correspond to individual functions, while directed edges represent explicit function calls. In addition to call relationships, we also include inheritance-based connections between functions defined in different contracts but related through inheritance hierarchies. This captures Solidity-specific behaviors, such as overridden functions or parent contract logic that affects vulnerability flow.

Each edge is further enriched with a 7-dimensional feature vector. These edge features include indicators such as whether the edge originates from an external call, whether it bridges contracts through inheritance, and whether the call is conditional or reentrant. Incorporating edge features allows the model to learn richer message-passing patterns that go beyond simple node connectivity.

The graph and features are converted into PyTorch Geometric Data objects, where each graph (i.e., contract) is labeled with one of nine vulnerability classes. This forms the input to our graph neural network models. To identify the most effective architecture, we experimented with several GNN variants, including a baseline Graph Attention Network (GAT), a Message Passing Neural Network (MPNN), a custom Edge-aware Graph Attention Network (EGAT), and the more recent GATv2. While GAT, MPNN, and EGAT showed competitive results, GATv2 achieved the best performance, benefiting from its improved attention mechanism that better captures dependencies between functions.

The final model consists of seven stacked GATv2 layers, each using multi-head attention and residual connections. Between layers, we apply batch normalization and dropout to improve generalization and training stability. A global add pooling layer aggregates the learned node features into a graph-level representation, which is then passed to a final MLP classifier. During training, we use label smoothing to improve robustness and ReduceLROnPlateau for dynamic learning rate adjustment. The model was trained in a supervised manner using cross-entropy loss, and an optional Exponential Moving Average (EMA) mechanism was tested to stabilize the optimization further.

This tool was integrated into the same evaluation setup as the existing academic and industry tools. By combining semantic embeddings, function-level graphs, and a carefully designed GNN architecture, our model addresses several key limitations observed in traditional tools, particularly in handling structural complexity and code variation across contracts.

The dataset used for testing in this thesis is based on the dataset introduced by HajiHosseinkhani et al. [24], which includes 1998 randomly selected secure contracts and 1998 randomly selected vulnerable contracts. For the vulnerable contracts, a larger pool is available, with each contract categorized under a specific type of vulnerability. From this pool, an equal number of unique random contracts are selected from each of the nine vulnerability categories: ExternalBug, GasException, MishandledException, Timestamp, TOD, UnusedReturn, CallToUnknown, Integer Under/Overflow (IntegerUO), and Reentrancy. These are combined to form the final set of vulnerable contracts. Testing is conducted on two systems—one with an Intel Core i7-1370P CPU and 32 GB of RAM, and another with an Intel Core i5-1240P CPU and 30 GB of RAM—both running Ubuntu 22.04. All tools are executed within a Docker environment (version 24.0.7) to ensure consistent performance and mitigate system-specific variations.

The performance of each tool is evaluated using standard vulnerability detection metrics: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). TP indicates the number of correctly identified vulnerabilities, while FN captures the vulnerabilities that the tool failed to detect. TN refers to secure contracts accurately classified as non-vulnerable, and FP accounts for incorrect vulnerability detections. This thesis uses accuracy to represent the proportion of correct vulnerability detections, offering a general measure of effectiveness. Precision is used to evaluate the correctness of positive identifications, whereas recall reflects the proportion of actual vulnerabilities successfully detected. The F1-Score, calculated as the harmonic mean of precision and recall, provides a balanced view of a tool's detection capability. Additionally, execution time—the duration required by each tool to analyze the contracts—is recorded as a measure of performance efficiency.

The results of the tool evaluations, including our proposed model, are presented in Table 44 and Table 45, which reflect two distinct strategies for assessing vulnerability detection performance. Table 44 reports on overall detection capability without differentiating between types of vulnerabilities. In this evaluation, each tool was applied to the dataset to determine whether it could detect the presence of any vulnerability, regardless of category. A file was considered a True Positive (TP) if the tool flagged it as vulnerable, even if it did not specify the exact vulnerability type. For instance, if a file contained a reentrancy vulnerability and the tool simply reported it as vulnerable—without labeling it as reentrancy—it was still included in the TP count. False Negatives (FN) in this context represent vulnerable files that the tool failed to detect

entirely. The True Negatives (TN) and False Positives (FP) remain unchanged between the two tables, as the identification of secure files and incorrect detections is unaffected by the classification of vulnerability types. In contrast, Table 45 adopts a more granular evaluation approach based on multi-class classification, focusing on detection accuracy across specific vulnerability types. In this case, the dataset was organized to include an equal number of contracts for each of the nine vulnerability categories. A True Positive (TP) in this context required the tool not only to detect the presence of a vulnerability but also to correctly identify its specific type. For example, if a contract contained a GasException vulnerability, it was only counted as a TP if the tool explicitly recognized it as GasException. If the tool detected a vulnerability but misclassified its type, the result was not considered a TP and was instead recorded as a False Negative (FN). This evaluation method offers a more precise analysis of each tool's ability to accurately identify and classify different types of vulnerabilities.

Our comparison of the seven vulnerability detection tools reveals substantial differences in their capabilities, with each tool exhibiting distinct strengths and weaknesses. These tools involve trade-offs across detection accuracy, execution speed, and the rates of false positives and false negatives, underscoring the difficulty of developing a comprehensive and universally effective vulnerability detection solution.

Table 44: Overall Detection Performance of Vulnerability Detection Tools (Binary Classification)

Metrics	**ConFuzzius**	**Manticore**	**Mythril**	**sFuzz**	**Securify**	**Slither**	**Vandal**	**Our GNN Tool**
True Positives	1063	375	953	1348	677	1843	509	**1800**
False Negatives	935	1623	1045	650	1321	155	1491	**198**
False Positives	1022	412	870	1451	835	1986	387	**268**
True Negatives	976	1586	1127	547	1163	12	1611	**1730**
Accuracy	53.3%	18.8%	47.7%	67.5%	33.9%	92.3%	25.5%	**88.3%**
Precision	0.51	0.48	0.52	0.48	0.45	0.55	0.57	**0.87**
Recall	0.53	0.19	0.48	0.67	0.34	0.29	0.25	**0.90**
F1-Score	0.52	0.27	0.50	0.56	0.39	0.38	0.35	**0.88**
Total Average Execution Time	3708.1	4439.0	3043.6	3495.7	357.7	2.5	4.5	**75.0**

Among the evaluated tools, Slither performs best in binary classification, achieving the highest accuracy of 92.3% as shown in Table 44 and Figure 22. This makes Slither effective for broadly identifying vulnerabilities, mainly when time is a constraint, as evidenced by its fast average execution time of just 2.5 seconds. However, the high rate of false positives means that users must spend additional time manually verifying detected vulnerabilities, diminishing their effectiveness. This limitation suggests that although Slither is helpful for a first-pass assessment, its outputs require significant follow-up, making it less practical for environments where precision is crucial. Figure 22 confirms our proposed model's strong binary (88.3%) and multi-class (83.1%) performance, as shown in Tables 44–45, where it outperforms all other tools. This result highlights the effectiveness of combining code-aware embeddings with attention mechanisms in a graph-based model

for smart contract vulnerability detection.

Table 45: Detection Performance of Vulnerability Detection Tools (Multi-Class Classification)

Metrics	**ConFuzzius**	**Manticore**	**Mythril**	**sFuzz**	**Securify**	**Slither**	**Vandal**	**Our GNN Tool**
True Positives	229	235	559	143	61	535	7	**1602**
False Negatives	1769	1751	1439	1855	1937	1463	1991	**396**
False Positives	1022	412	870	1451	835	1986	387	**279**
True Negatives	976	1586	1127	547	1163	12	1611	**1719**
Accuracy	11.5%	12.0%	28.1%	6.9%	3.0%	26.8%	0.3%	**83.1%**
Precision	0.18	0.36	0.39	0.09	0.07	0.21	0.02	**0.85**
Recall	0.11	0.12	0.28	0.07	0.03	0.27	0.00	**0.80**
F1-Score	0.14	0.18	0.33	0.08	0.04	0.24	0.01	**0.83**
Total Average Execution Time	3708.1	4439.0	3043.6	3495.7	357.7	2.5	4.5	**75.0**

Compared to Slither, Manticore, and Securify perform poorly in binary classification tasks. Manticore achieves only 18.8% accuracy, and Securify reaches 33.9%. Both tools also have higher rates of false negatives and false positives, which impacts their reliability for detecting vulnerabilities. Manticore's performance is further limited by its long execution time of 4439.0 seconds, making it unsuitable for quick or large-scale analysis. Security, with a faster execution time of 357.7 seconds, is closer to Slither's speed, but its low recall indicates that it misses many vulnerabilities, reducing its effectiveness as a standalone tool. Although Manticore's performance may seem weak in these measures, its symbolic execution approach offers a more in-depth analysis, which can be valuable in specific, detailed audit scenarios.

Moreover, sFuzz offers a more balanced performance, achieving a reasonable accuracy of 67.5% in binary classification while maintaining moderate precision and recall scores. This balance suggests that sFuzz may be suitable for use cases where detection capability and overall accuracy are essential. In contrast, Mythril performs with relatively low accuracy (47.7%) and suffers from a longer execution time, which is visualized in Figure 23 and may limit its practical value, especially in scenarios requiring quick analysis. The high rate of false negatives further indicates that Mythril often fails to identify many vulnerabilities, reducing its reliability. Vandal, with an accuracy of only 25.5%, shows limited utility across all significant metrics, offering poor detection quality and low speed, making it less effective than other tools. In this comparison of accuracy and runtime, our approach shows the highest accuracy among all tools while maintaining a moderate analysis time, supporting its suitability for both precision-focused and practical large-scale use.

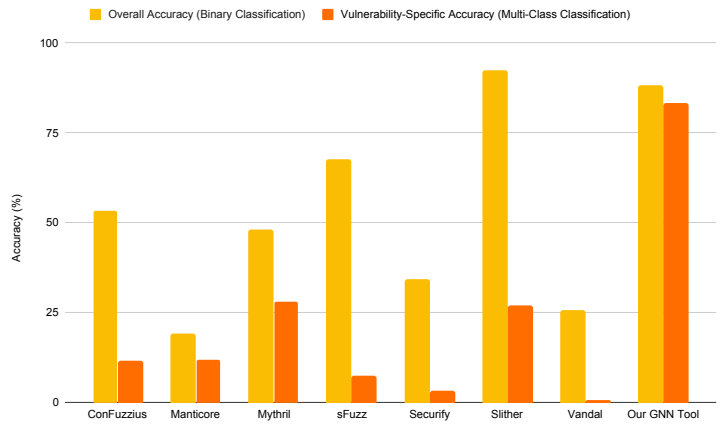

Figure 22: Comparison of Binary and Multi-Class Accuracy Across Tools

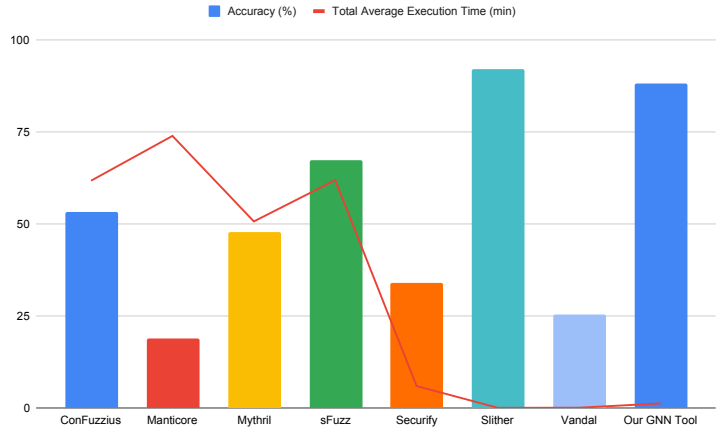

Figure 23: Comparison of Tool Accuracy and Total Average Execution Time (Minutes) in Binary Classification

In the more complex scenario of multi-class classification, as presented in Table 45 and illustrated in Figure 22, we observe a significant decline in the performance of all tools. Slither, which excelled in binary classification, sees a major drop in accuracy, falling to 26.8%, indicating its difficulty in distinguishing between specific types of vulnerabilities. Manticore's accuracy falls even further to 12.0%, highlighting its particular struggles with increased classification complexity. Securify also exhibits a poor performance here, with an accuracy of just 3.0%, among the lowest of all tools. Mythril achieves the highest accuracy in multi-class classification with 28.1%, but this number remains low, showing that even the best-performing tool struggles significantly in identifying particular types of vulnerabilities with precision. The decline across all tools points to a broader limitation in current vulnerability detection technology—the inability to maintain accuracy as complexity increases.

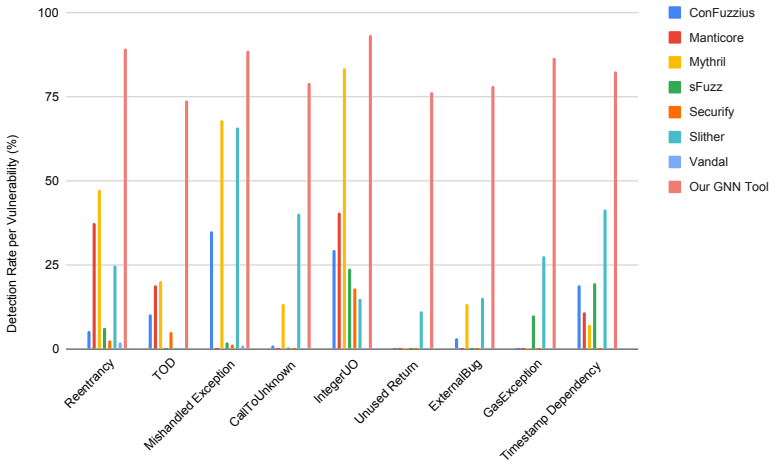

Figure 24: Detection rates of Analysis Tools Across Vulnerability Types

A detailed examination of detection rates for specific vulnerabilities, as shown in Table 46 and illustrated in Figure 24, shows variations in tool specialization. Mythril excels in identifying Integer Under/Overflow vulnerabilities (83.3%) and performs moderately well with Mishandled Exceptions (68.0%), making it suitable for use cases focused on these issues. Slither shows comparatively strong detection rates for Timestamp Dependency (41.4%), GasException (27.5%), and Mishandled Exceptions (65.8%), indicating its usefulness for targeting these vulnerabilities even if its overall coverage is limited. Manticore achieves decent performance with Integer Under/Overflow (40.5%) but has generally low detection rates for other vulnerabilities, pointing to a narrower specialization. By contrast, the proposed detection model delivers consistently high detection rates across all nine vulnerability types, ranging from 74% to over 93%, indicating strong generalization rather than narrow specialization.

Table 46: Detection Rates per Vulnerability Type for Smart Contract Analysis Tools

Analysis Tools	Reentrancy	TOD	Mishandled Exception	CallToUnknown	IntegerUO	Unused Return	ExternalBug	GasException	Timestamp Dependency	Average Detection Rate
ConFuzzius	5.4%	10.4%	35.1%	0.9%	29.3%	—	3.2%	—	18.9%	11.5%
Manticore	37.4%	18.9%	—	—	40.5%	—	—	—	10.8%	12.0%
Mythril	47.3%	20.3%	68.0%	13.5%	83.3%	—	13.5%	—	7.2%	28.1%
sFuzz	6.3%	—	1.8%	0.5%	23.9%	—	0%	9.9%	19.4%	6.9%
Securify	2.7%	5.0%	1.4%	—	18.0%	—	0%	—	—	3.0%
Slither	24.8%	0%	65.8%	40.1%	14.9%	11.3%	15.3%	27.5%	41.4%	26.8%
Vandal	1.8%	0%	0.9%	0%	—	—	0%	—	—	0.3%
Our GNN Tool	**89.2%**	**74.0%**	**88.6%**	**79.1%**	**93.4%**	**76.3%**	**78.1%**	**86.6%**	**82.6%**	**83.1%**

Note: The symbol '—' indicates that the tool cannot detect the corresponding vulnerability.

In contrast, Securify and other tools, including Confuzzius, sFuzz, and Vandal, show limited effectiveness, with consistently low detection rates across most categories. Securify, for instance, does not exceed a 5% detection rate in any vulnerability type, which restricts its practicality as a standalone detection tool. Across all tools, specific vulnerabilities, such as ExternalBug and Unused Return, have particularly low or unsupported detection rates, highlighting the overall limitations in comprehensive coverage. This analysis suggests that

while some tools have specific strengths, combining multiple tools or enhancing detection capabilities would be necessary to address a broader range of vulnerabilities.

The execution times, visualized in Figure 25, provide further insight into the practical applicability of these tools. Slither and Vandal are significantly faster than their counterparts, making them potentially suitable for processing larger datasets or in situations where speed is critical. Although not as fast as the quickest tools, our model's 75-second analysis time represents a reasonable trade-off given its substantial accuracy gains. In Figure 26, our model is positioned in the upper-right, reflecting its combination of the highest multi-class accuracy with a moderate runtime. However, according to Figures 23 and 26, the speed advantage of the fastest tools does not translate to strong detection performance, as both have low accuracy in both classifications. In contrast, Manticore, ConFuzzius, sFuzz, and Mythril have much longer execution times. Mythril achieves moderate accuracy, but like the others, its slower speed does not result in significantly better detection rates, indicating a limited trade-off between speed and accuracy. Securify occupies a middle ground with moderate speed but low accuracy, underscoring the difficulty of balancing efficiency with effective vulnerability detection. Among the established tools, none achieves an ideal balance. Our proposed model, however, combines top accuracy with moderate runtime, showing a more favorable trade-off while still leaving room for future refinements.

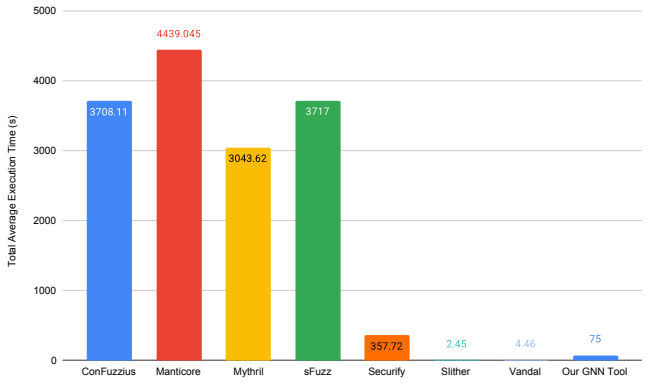

Figure 25: Average Detection Time (Seconds) of Tools

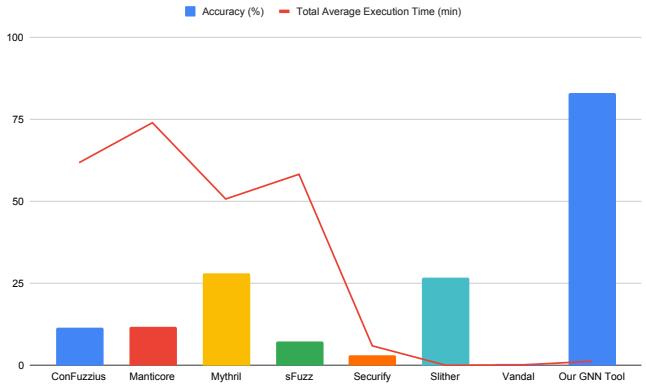

Figure 26: Comparison of Tool Accuracy and Total Average Execution Time (Minutes) in Multi-Class Classification

The comparison of precision, recall, and F1-scores for both binary and multi-class classifications, as presented in Figure 27, further highlights the limitations of these tools. ConFuzzius, which shows a relatively balanced F1-score in binary classification, experiences a sharp decline in performance during multi-class classification. This pattern suggests that while the tools may be capable of identifying general vulnerabilities, they lack the sophistication required for accurately distinguishing between specific types of vulnerabilities. The limited precision and high false-positive rates, especially in complex classifications, make it challenging to rely on these tools for detailed, multi-class vulnerability analysis. Our model attains the highest precision, recall, and F1-scores in both binary and multi-class evaluations, showing balanced performance across detection metrics.

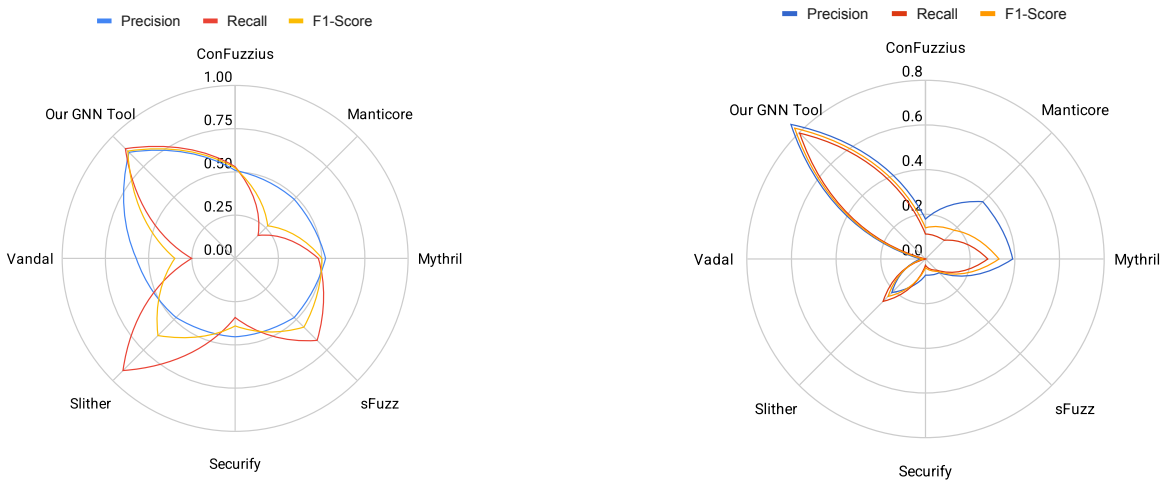

Figure 27: Precision, Recall, and F1-Score for each tool in Binary Classification (left) and Multi-Class Classification (right)

3.4 Concluding Remarks

This chapter highlights the strengths and weaknesses of current SCs vulnerability detection tools, revealing substantial room for improvement. Mythril proves to be the most effective in detecting specific vulnerabilities, such as Integer Under/Overflow, and achieves the highest accuracy in multi-class classification. However, its long execution time may reduce its practicality for rapid analysis. Slither demonstrates high accuracy in binary classification and offers fast execution speed, making it suitable for broad, time-sensitive assessments. Nevertheless, its high false-positive rate necessitates considerable manual verification, limiting its efficiency in precision-critical scenarios. Other tools, including sFuzz, ConFuzzius, and Vandal, exhibit limited effectiveness due to low detection rates and slower performance. Manticore offers value through symbolic execution but is hindered by long execution times and low overall accuracy in general vulnerability detection. Overall, no single tool achieves an optimal balance of accuracy, speed, and reliability for comprehensive SCs vulnerability assessment. Future advancements should prioritize improving detection precision, minimizing false positives, and enhancing classification accuracy to make these tools more robust and applicable in real-world security workflows. In line with these goals, our experimental GNN-based

model demonstrates promising results, outperforming existing tools in both binary and multi-class evaluations while maintaining reasonable execution time. This suggests that graph-based learning approaches offer a viable direction for future improvements in SCs vulnerability detection.

4 A Conceptual Framework for Test & Evaluation of SCs Tools

SCs security has become a major area of concern. Many tools have been developed in recent years to detect vulnerabilities in these systems. However, as shown in section 3, no single tool covers all of the essential features. Some focus only on static analysis, while others provide user-friendly interfaces but lack strong detection models. Many tools also fail to support runtime monitoring, learning-based methods, or flexible integration with other platforms. These gaps show the need for a more complete and structured way to evaluate existing tools.

This section introduces a six-layer conceptual framework for evaluating SCs vulnerability detection tools. The framework organizes key capabilities into functional layers, each of which reflects an important aspect of what a well-rounded tool should support. These layers help clarify which areas a tool covers and which parts are missing or underdeveloped. The structure also reflects common patterns observed across real tools and highlights where gaps are most likely to appear.

The goal of this framework is to enable consistent and meaningful evaluation. Each layer contains a set of components that can be checked using a binary scoring system. Every component contributes to the tool's total score based on its role and the overall weight of its layer. This approach makes it possible to compare tools based on what they actually support, without relying on vague metrics or subjective criteria.

Although the framework is designed primarily for evaluation, it can also support future tool development. By organizing the essential components into a clear structure, it provides guidance on what to include in a more complete and practical detection tool. It also allows developers and researchers to identify trends, track improvements, and focus on the areas that need more attention.

The rest of this section explains the framework in detail. It begins with an overview of its structure and layers, followed by a description of each layer's purpose and components. A workflow is then presented to show how the layers interact during tool operation. After that, the framework's evaluation method is described, and the section ends with a summary of the main ideas.

4.1 Overview of the Framework Structure

This framework is organized into six distinct layers. Each layer captures one important aspect of how a SCs vulnerability detection tool can be evaluated. Together, these layers form a complete structure for assessing the capabilities and completeness of existing tools.

The six layers are: (1) Code Analysis, (2) Detection Capability, (3) Real-Time and Historical Analysis, (4) Integration and Extensibility, (5) User Interaction and Accessibility, and (6) Deployment and Performance. These layers are not arranged as steps but as parts that can work in parallel or support each other. Some tools may start from one layer and grow into others, depending on their goals and design. Figure 28 provides a visual overview of the framework, showing how the layers are structured and how their components are grouped for evaluation purposes.

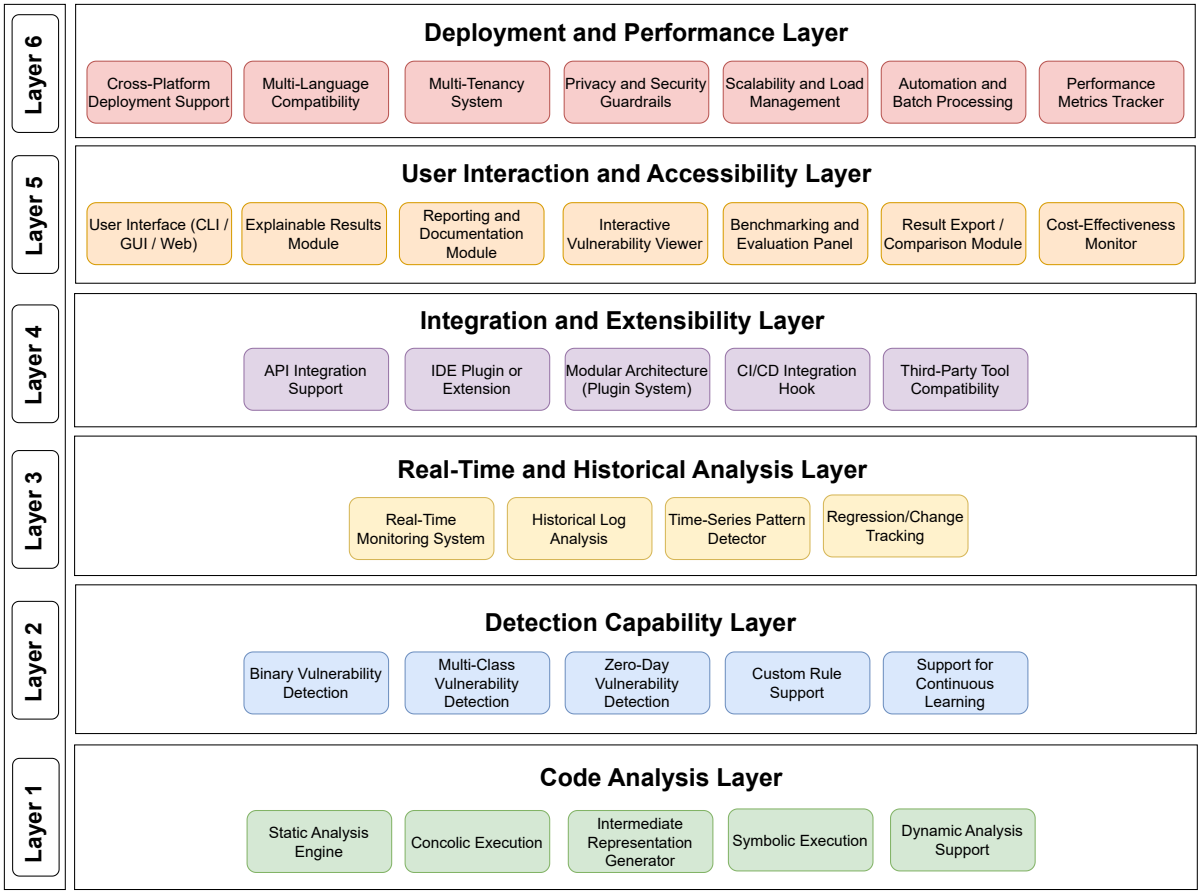

Figure 28: Layered framework for SCs tools.

Each layer includes key components that serve a specific role. For example, one layer may focus on how to understand SCs code, while another may manage how results are shared with users. The structure gives space for technical parts, user-facing parts, and operational concerns to be treated separately but still kept in one system.

This kind of separation helps to spot which areas are often ignored in current tools. From the previous section, we know many tools are strong in detection but weak in reporting or platform support. This framework helps fill those gaps by showing what a balanced tool should contain. It encourages the development of tools that are not only accurate but also usable, flexible, and ready for real-world use.

In the following subsections, each layer of the framework is explained in detail. For every layer, we describe its main function, list its components, and provide practical methods for how those components may be implemented. These components serve as the basis for tool evaluation, and the framework as a whole reflects areas that are often overlooked in current SCs vulnerability detection systems.

4.1.1 Code Analysis Layer

The Code Analysis Layer is the foundation of the framework. It focuses on how the tool reads and interprets SCs code. This step is essential because poor analysis at the beginning can lead to incorrect or incomplete detection later. A reliable understanding of the code ensures that the next layers work effectively and produce useful results. This layer helps solve common challenges such as fragmented analysis techniques, shallow code understanding, and limited early-stage vulnerability detection in many current tools.

This layer includes both static and dynamic analysis techniques. Static analysis examines code without executing it, helping to detect issues like unsafe patterns or unreachable code. Dynamic analysis, on the other hand, runs the contract in a controlled environment to observe actual behavior, such as gas use or state changes during execution.

Symbolic and concolic execution are also part of this layer. Symbolic execution replaces concrete values with symbolic variables, allowing the tool to reason through many possible execution paths. Concolic execution builds on this by combining symbolic tracking with real input values during execution. Together, these methods help the tool explore deep and conditional paths in the contract, making it possible to detect vulnerabilities that might not appear during regular testing.

To support these techniques, the tool generates intermediate representations of the contract. These include abstract syntax trees (ASTs), control flow graphs (CFGs), and static single assignment (SSA) form. These representations break down the contract into structures that are easier to analyze and process in later steps.

Table 47 shows how each component of this layer can be implemented using specific technical approaches. These examples include methods such as taint analysis, symbolic substitution, SMT-based constraint solving, and runtime instrumentation. Together, they form a strong foundation for detecting vulnerabilities in SCs.

Table 47: Method-Based Implementation Examples for Code Analysis Layer Components

Component	**Implementation Examples**
Static Analysis Engine	- Pattern matching over ASTs for insecure constructs - Control-flow validation to detect unreachable or unsafe paths - Taint analysis to trace user input propagation to critical operations
Concolic Execution	- Path constraint generation via symbolic tracking - Concrete execution with symbolic state updates - SMT-based input synthesis to trigger new execution paths
Intermediate Representation Generator	- AST construction from parsed Solidity code - CFG generation based on logical branching and jump structure - SSA form derivation using dominance frontier computation
Symbolic Execution	- Symbolic substitution for variable assignments - Branch condition solving with SMT logic - Path pruning using loop bounds and constraint merging
Dynamic Analysis Support	- Instrumented runtime tracking of state, gas, and execution trace - Fuzz-based execution with anomaly monitoring - Reentrancy detection via log inspection and behavior trace analysis

4.1.2 Detection Capability Layer

The Detection Capability Layer focuses on how the tool identifies vulnerabilities in SCs. This layer determines the kinds of results the tool can produce, ranging from simple vulnerability alerts to more advanced classifications. It builds directly on the code analysis results and defines how detection logic is applied.

There are several different ways to detect vulnerabilities in SCs. Some tools give a binary answer, simply stating whether a contract is vulnerable or not. Others are more detailed and identify the exact type of vulnerability, such as reentrancy or timestamp misuse. Advanced systems can also detect new or unknown vulnerabilities by learning from behavioral patterns or by modeling unseen threats.

In addition to built-in detection logic, this layer may support user-defined rules. These allow auditors or developers to write custom checks for their specific needs. Continuous learning is also possible, allowing the tool to improve over time as new contracts are scanned or as new types of vulnerabilities appear.

Table 48 lists common methods for implementing each component in this layer. The table presents approaches such as binary rule matching, multi-class classification, anomaly detection, and rule configuration systems. These methods show how detection capabilities can be designed to cover both known and unknown vulnerabilities while remaining adaptable to new threats.

Table 48: Method-Based Implementation Examples for Detection Capability Layer

Component	Implementation Examples
Binary Vulnerability Detection	- Boolean flags for presence or absence of known bugs - Threshold-based rule triggers (e.g., overflow risk indicators) - Simple pass/fail scoring for vulnerability presence
Multi-Class Vulnerability Detection	- Rule-based classifiers tied to specific vulnerability types - Multi-label ML models trained on annotated SCs - Decision trees or logic graphs that label vulnerability classes
Zero-Day Vulnerability Detection	- Anomaly detection using unsupervised learning - Representation learning on behavior traces or opcode sequences - Statistical outlier detection in symbolic path results
Custom Rule Support	- DSLs for writing detection logic or custom patterns - Visual rule builders for non-technical users - JSON/YAML schema support for rule configuration
Support for Continuous Learning	- Online model updates with new contract data - Feedback loops based on user-tagged results - Auto-retraining based on recent scan outcomes

4.1.3 Real-Time and Historical Analysis Layer

The Real-Time and Historical Analysis Layer focuses on tracking how SCs behave during and after deployment. Most tools stop their analysis once a contract is deployed, but this layer keeps monitoring the contract's behavior over time. Adding this time-based perspective is important for detecting issues that only appear during use, or after updates.

This layer includes components that handle both live data streams and historical contract behavior. Real-time monitoring helps the tool detect ongoing threats while the contract is active. It can capture events, transaction patterns, or gas spikes that point to misuse or attacks. This is useful for stopping incidents as they happen and reducing the impact of exploitation.

Historical analysis supports deeper inspection based on past behavior. It allows the tool to detect patterns, trace vulnerabilities that reappear, or understand the impact of contract upgrades. Time-series analysis can reveal trends or unexpected behavior changes, while regression tracking helps compare new versions with old ones. These features are helpful for auditing, reviewing contract lifecycles, and supporting long-term security assurance.

Table 49 shows examples of how each component in this layer can be implemented. The table highlights approaches such as event listeners, transaction log parsers, anomaly detection strategies, and version comparison techniques. Each method reflects a different way to detect behavioral patterns, track contract evolution, or identify security issues that occur over time.

Table 49: Method-Based Implementation Examples for Real-Time and Historical Analysis Layer

Component	**Implementation Examples**
Real-Time Monitoring System	- Event listeners on the blockchain to capture live interactions - WebSocket-based monitoring of contract transactions - Trigger-based alerts for abnormal on-chain activity
Historical Log Analysis	- Transaction/event log parsing using Web3 libraries - Querying historical contract behavior via indexed datasets - Batch inspection of previous deployments or contract versions
Time-Series Pattern Detector	- Anomaly detection in time-series of gas usage or calls - RNNs for modeling sequential activity patterns - Rule-based filters for spotting periodic interaction anomalies
Regression/Change Tracking	- Diff tools for contract version comparisons - Control/data flow analysis across upgrades - Regression testing frameworks to verify repeated vulnerabilities

4.1.4 Integration and Extensibility Layer

The Integration and Extensibility Layer focuses on how well the tool fits into other systems and adapts to new needs. Many SCs projects are part of larger development environments, such as IDEs, CI/CD pipelines, or security platforms. A detection tool should not work in isolation. Instead, it should be easy to connect to other platforms and flexible enough to grow over time.

This layer includes several components that help the tool stay useful in different settings. API integration allows external systems to trigger scans, retrieve results, or update detection rules. This is useful for automation and continuous monitoring. Support for IDE plugins gives developers feedback while they write or review code, which can improve workflow efficiency and reduce delays.

Modular architecture is another key feature. It allows the tool to load and run components independently, making it easy to swap in new detection engines or update logic without affecting the whole system. Integration with CI/CD pipelines helps teams include security scans in their standard deployment flow. Finally, compatibility with other tools ensures that results can be reused, shared, or extended as needed.

Table 50 shows how each component in this layer can be implemented. The table lists selected examples such as REST APIs, plugin loaders, configuration systems, and adapters. These methods show how the tool can remain flexible and connected within different development environments.

Table 50: Method-Based Implementation Examples for Integration and Extensibility Layer

Component	**Implementation Examples**
API Integration Support	- RESTful APIs for remote tool access - Webhooks to send real-time scan results - GraphQL endpoints for flexible querying
IDE Plugin or Extension	- VS Code plugins for inline detection - Linting support in Remix or IntelliJ - LSP-based editor extensions for SCs feedback
Modular Architecture (Plugin System)	- Modular engines integrated via plugin APIs - Plugin loaders using Python 'importlib' or Java services - Registry systems to dynamically load/unload modules
CI/CD Integration Hook	- GitHub/GitLab scripts for automated vulnerability scans - Docker containers in CI pipelines - Jenkins integration via wrappers or plugins
Third-Party Tool Compatibility	- JSON or SARIF export formats for compatibility - Bridge modules for Slither/Mythril result consumption - Standardized adapters for third-party security tools

4.1.5 User Interaction and Accessibility Layer

This layer focuses on how users interact with the detection tool and how clearly the results are presented. A tool may be technically advanced but still difficult to use if the interface is confusing or if the outputs are hard to interpret. Improving accessibility allows a wider group of users—including developers, auditors, and analysts—to benefit from the tool's capabilities.

Several components support interaction and usability in different ways. A user interface makes the tool easier to operate, whether through a command-line terminal, a graphical dashboard, or a browser-based platform. Clear reporting features help users understand what was found, where the issue is in the code, and why it matters for SCs security.

Explainable outputs allow users to trace how a vulnerability was detected and what logic the tool followed. This is important for building confidence in the tool's results and helping users take appropriate actions. Additional features like benchmarking panels, interactive result viewers, and export options allow users to compare results over time or across tools and to integrate findings into their workflows.

Table 51 presents examples of how each component in this layer can be implemented. The table highlights practical methods such as interactive dashboards, graph-based result visualizations, and export-ready reporting formats. These examples show how user-facing features can improve both the clarity of the output and the usability of the tool in real development workflows.

Table 51: Method-Based Implementation Examples for User Interaction and Accessibility Layer

Component	**Implementation Examples**
User Interface (CLI / GUI / Web)	- Command-line interface with scan flags - Web dashboards using React/Angular - GUI apps with drag-and-drop contract input
Explainable Results Module	- Visual CFG/call graph overlays - Natural language vulnerability summaries - Source highlighting with symbolic execution paths
Reporting and Documentation Tools	- PDF reports with auto-filled fields - Markdown/HTML export for wiki documentation - CWE-linked descriptions inline with findings
Interactive Vulnerability Viewer	- Expandable views for vulnerabilities - Clickable graph overlays - Simulated test triggers in-browser
Benchmarking and Evaluation Panel	- Result comparison across datasets - Charts of FP/FN rates - Leaderboard-style evaluation tracking
Result Export / Comparison Module	- CSV, JSON, SARIF export formats - Tool-to-tool or version diffing of findings - Audit trace support via Excel exports
Cost-Effectiveness Monitor	- Gas usage and runtime profiling - Cost-to-benefit scoring systems - Resource alerts for expensive scans

4.1.6 Deployment and Performance Layer

This layer focuses on how the detection tool is deployed and how well it performs in different environments. SCs tools are often used by teams working on varied platforms, such as local machines, cloud services, or continuous integration pipelines. For this reason, the tool should be designed to run reliably across different operating systems and environments with minimal setup.

Performance and scalability are key factors in this layer. The tool should support large-scale analysis, batch processing of multiple SCs, and automation for integration into secure development workflows. It should also provide basic infrastructure features like user isolation, language compatibility, and runtime monitoring to make the system easier to manage and extend.

Measuring performance is also important. Metrics such as runtime, memory use, and scan coverage help developers understand how the tool behaves under real-world conditions. Security and privacy considerations should also be built into the system through sandboxing, secure logging, and permission controls.

Table 52 gives examples of how the components in this layer can be implemented. The methods listed reflect

ways to improve deployment flexibility, maintain efficiency under load, and monitor how the tool behaves during use.

Table 52: Method-Based Implementation Examples for Deployment and Performance Layer

Component	**Implementation Examples**
Cross-Platform Deployment Support	- Docker containers for consistent execution - Platform-agnostic binaries via cross-compilation - Web interfaces to eliminate OS dependency
Multi-Language Compatibility	- Solidity and Vyper parsing modules - Language-agnostic intermediate representation - Pluggable backends for future language support
Multi-Tenancy System	- Kubernetes namespaces for job isolation - Database partitioning or tagging per tenant - Tenant-aware authentication and authorization
Privacy and Security Guardrails	- Sandboxed analysis environments - Encrypted storage and secure logs - Minimal permissions for runtime file/network access
Scalability and Load Management	- Distributed task queues (e.g., Celery) - Dynamic worker allocation based on load - Cloud autoscaling for compute resources
Automation and Batch Processing	- CLI batch scan modes - Scheduled jobs via CRON or CI/CD - Repo monitoring and auto-triggered scans
Performance Metrics Tracker	- Precision/recall logging per scan - Runtime and memory profiling - Trend charts of scan cost vs. coverage

4.2 Workflow of the Framework

The six layers of the framework work as a connected system. Each layer performs a specific role, but they also interact with each other to support a full vulnerability detection process. The flow of information between layers is flexible. While the system may start at the code analysis layer, later steps can move forward, loop back, or run in parallel, depending on the contract's context and the tool's configuration.

The process typically starts when a SCs is submitted for analysis. This may happen during development, before deployment, or after the contract has gone live. The Code Analysis Layer reads the source code and converts it into internal representations, such as syntax trees and CFGs. These outputs are then passed into the Detection Capability Layer, which determines whether the contract is vulnerable, what type of vulnerability it may have, and whether the issue resembles previously unseen patterns. This layer may use rule-based checks, classification strategies, or learning-based models to support both fixed and evolving detection logic.

If the contract is already deployed on a blockchain, the Real-Time and Historical Analysis Layer becomes active. It observes the contract's activity, such as transactions, gas usage, or function calls over time. New issues may emerge that were not present in the code alone. This layer also supports deeper investigation using logs or previous contract versions, helping analysts trace how vulnerabilities develop or resurface.

As the analysis runs, the Integration and Extensibility Layer connects the tool to outside systems. For example, if the tool is part of a CI/CD pipeline, it can automatically trigger scans for every contract update. If used by an audit firm, the tool might send reports to a dashboard or feed results into a larger risk management system. This layer ensures the framework works in different environments without needing to be rebuilt each time.

The outputs of the analysis are presented through the User Interaction and Accessibility Layer. This includes dashboards, CLI reports, PDF exports, or integration into development tools. Users can view the results, filter findings, and see clear explanations of each issue. This layer also supports feedback loops. Developers may tag results as false positives or manually mark issues as fixed, helping improve future analysis.

The final step is handled by the Deployment and Performance Layer, which ensures the system runs smoothly. It manages how the tool scales to large projects, adapts to various platforms, and tracks performance metrics. If the system is deployed as a service, this layer also ensures privacy protection, user isolation, and cost efficiency.

For example, imagine a developer uploading a Solidity contract to a cloud-based version of the tool. The tool first analyzes the code and finds potential overflow risks. Then, it checks the contract's history and sees a redeployment with the same vulnerability. The findings are pushed to a project dashboard via an API, and a report is emailed to the team. The tool runs in a secure cloud container and completes the scan within minutes, with memory use and runtime logged for performance review.

Together, these layers form a flexible and modular system that adapts to different contract scenarios, project goals, and user needs. Each layer contributes to the overall workflow by producing, processing, or presenting information. Rather than enforcing a fixed order, the framework allows components to interact based on context. This flexible structure supports the design of tools that are effective and responsive to the evolving demands of SCs security.

4.3 Validation Through Evaluation Criteria

This framework not only outlines the key features of an ideal SCs vulnerability detection tool, but also provides a way to evaluate real tools using those features. Each layer contains components that represent specific capabilities a tool may support. The evaluation method checks whether a tool includes each of these components, allowing for a structured and repeatable scoring process.

The evaluation is based on a binary scale. For each component, the tool either supports it (score of 1) or does not (score of 0). Each layer contributes a percentage to the final score, based on how many components it contains and how important the layer is in practice. The total score adds up to 100 if a tool supports all components across all layers.

Table 53 shows the final weight assigned to each layer. Layers that are foundational to detection, such as code analysis, and those that determine how the tool integrates and operates in practice, such as deployment and

extensibility, receive higher weight. This distribution reflects both the technical importance of early-stage analysis and the practical value of usability, adaptability, and real-world performance.

Table 53: Assigned Weights for Framework Layers

Layer No.	**Layer Name**	**Components**	**Weight (%)**
1	Code Analysis Layer	5	15
2	Detection Capability Layer	5	10
3	Real-Time and Historical Analysis Layer	4	10
4	Integration and Extensibility Layer	5	15
5	User Interaction and Accessibility Layer	7	20
6	Deployment and Performance Layer	7	30
Total		**33**	**100**

The total score is calculated using the following formula:

$$\text{Total Score} = \sum_{i=1}^{n} \left(\frac{W_i}{C_i} \sum_{j=1}^{C_i} x_{ij} \right) \tag{1}$$

Where:

- W_i is the total weight of layer i
- C_i is the number of components in layer i
- x_{ij} is the binary score (0 or 1) for component j in layer i
- n is the number of layers in the framework

Each component contributes proportionally to its layer's weight. The scores from all layers are then summed to give a final score out of 100. Based on this score, tools can be grouped into quality categories as shown in Table 54. This score allows tools to be compared in a fair and consistent way, based on the features they support. It also highlights strengths and gaps in each tool, helping both developers and users understand where a tool performs well and where improvement is needed.

Table 54: Tool Quality Based on Evaluation Coverage Score

Coverage Score	Tool Quality
90–100%	Excellent
82–89%	Strong
75–81%	Moderate
Below 75%	Needs Improvement

4.4 Concluding Remarks

This section presents a six-layer conceptual framework designed to evaluate SCs' vulnerability detection tools. Each layer represents a distinct aspect of what a well-rounded tool should support, from code analysis and detection to runtime monitoring, integration, user interaction, and performance. The layered structure helps clarify which features are essential and how different capabilities relate to each other.

The framework is designed to address common gaps observed in existing tools, including the absence of support for continuous analysis, limited extensibility, and a lack of meaningful user feedback. By breaking the ideal tool into concrete components across six functional layers, the framework offers a clear standard for evaluating how complete or capable a detection tool is.

Each component in the framework is assessed using a binary score, and each layer contributes to the final evaluation based on its weight and number of components. This structured approach allows tools to be compared fairly and consistently, while also making it easier to identify which features are missing or underdeveloped.

Although the framework is intended primarily for evaluation, it can also support the design of future tools. By highlighting key capabilities and their relationships, it offers guidance for building systems that are more effective and adaptable. In this way, the framework serves both as a benchmark for current tools and as a reference model for future development in SCs security.

5 Conclusion and Future Work

This thesis reviewed 256 SCs analysis tools developed between 2018 and 2024, categorizing them based on methodologies such as fuzzing, symbolic execution, formal verification, and hybrid approaches. Each tool was further classified according to its detection capabilities, providing insights into their specific applications and limitations. The analysis covered the types of vulnerabilities each tool can detect, highlighting their coverage gaps and strengths in supporting blockchain security.

The evaluation was conducted in two stages. The first stage involved a theoretical analysis that grouped tools by their origin (academic or industry) and approach (static, dynamic, or hybrid), applying tailored evaluation criteria. The second stage consisted of an experimental assessment, where selected tools were tested on real-world datasets, requiring considerable time and computational resources. This approach allowed for a comprehensive understanding of the tools' effectiveness across different contexts.

The findings revealed distinct strengths and weaknesses among the tools. Academic tools often prioritize documentation, flexibility, and adaptability, making them suitable for research and development. In contrast, industry tools are designed to achieve higher speeds and better integration within production environments. Tools like Slither and Mythril demonstrated accurate detection capabilities but also experienced issues with false positives and slower speeds. Despite certain tools excelling in specific areas, none achieved complete vulnerability coverage. This emphasizes the need for ongoing improvements in accuracy, execution speed, and reliability to address the wide range of vulnerabilities present in SCs effectively.

In addition to individual tool performance, this study identified several common limitations across existing vulnerability detection tools. These include high false positive rates, slow analysis speeds, low accuracy in detecting multiple types of vulnerabilities, and a lack of standard benchmarks for fair comparison. Many tools also have rigid designs, limited integration options, and poor user experience. As SCs vulnerabilities continue to evolve, most tools struggle to keep up, especially those relying on static detection methods. Moreover, few tools offer guidance on how to fix the problems they detect.

To bridge the gap between scattered tool capabilities and the demand for more comprehensive assessment criteria, this research proposed a six-layer conceptual framework. Each layer addresses a core functional domain, such as code analysis, detection capabilities, real-time monitoring, integration, user experience, and deployment performance. Components within each layer were scored using a binary evaluation system, combined with weighted layer importance. This allowed for the consistent and transparent evaluation of any detection tool, highlighting both strengths and deficiencies.

To explore new directions in tool development, this thesis also introduced a custom graph-based detection model that integrates semantic code embeddings with function-level call graphs. The model was tested under the same conditions as other tools and showed strong results, outperforming existing tools in both binary and multi-class vulnerability detection tasks. Its ability to combine learned representations with structural contract information addresses several limitations found in current tools, including low accuracy and poor generalization. While the tool does not cover all aspects of the proposed framework, such as user interface and deployment features, it demonstrates the potential of learning-based methods to improve detection accuracy and flexibility in SCs analysis.

The proposed framework serves not only as a theoretical model but also as a practical tool to guide the development of future SCs analyzers. By aligning evaluation metrics with both user needs and technical capabilities, it supports more informed decision-making in the selection, comparison, and creation of security tools within blockchain ecosystems. To advance this work, a future tool could be developed with the explicit goal of fulfilling all components and criteria defined in the framework. Ideally, such a tool, when evaluated using the proposed model, would achieve a full score—showing comprehensive, end-to-end support for SCs vulnerability detection across all functional dimensions.

Bibliography

[1] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, ACM, June 2020.

[2] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57037–57062, 2022.

[3] B. Lashkari and P. Musilek, "Evaluation of smart contract vulnerability analysis tools: A domain-specific perspective," *Information*, vol. 14, no. 10, p. 533, 2023.

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

[5] A. Vacca, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges," *Journal of Systems and Software*, vol. 174, p. 110891, 2021.

[6] G. Wu, H. Wang, X. Lai, M. Wang, D. He, and S. Chan, "A comprehensive survey of smart contract security: State of the art and research directions," *Journal of Network and Computer Applications*, p. 103882, 2024.

[7] T. Hewa, M. Ylianttila, and M. Liyanage, "Survey on blockchain based smart contracts: Applications, opportunities and challenges," *Journal of network and computer applications*, vol. 177, p. 102857, 2021.

[8] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li, "A survey on smart contract vulnerabilities: Data sources, detection and repair," *Information and Software Technology*, p. 107221, 2023.

[9] J. Liu and Z. Liu, "A survey on security verification of blockchain smart contracts," *IEEE Access*, vol. 7, pp. 77894–77904, 2019.

[10] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–38, 2021.

[11] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[12] B. Hu, Z. Zhang, J. Liu, Y. Liu, J. Yin, R. Lu, and X. Lin, "A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems," *Patterns*, vol. 2, no. 2, 2021.

[13] D. Harz and W. Knottenbelt, "Towards safer smart contracts: A survey of languages and verification methods," *arXiv preprint arXiv:1809.09805*, 2018.

[14] A. López Vivar, A. T. Castedo, A. L. Sandoval Orozco, and L. J. Garc´ia Villalba, "An analysis of smart contracts security threats alongside existing solutions," *Entropy*, vol. 22, no. 2, 2020.

[15] H. Rameder, "Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools," *PhD thesis*, 2021.

[16] M. di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pp. 69–78, 2019.

[17] H. Li, R. Dang, Y. Yao, and H. Wang, "A review of approaches for detecting vulnerabilities in smart contracts within web 3.0 applications," *Blockchains*, vol. 1, no. 1, pp. 3–18, 2023.

[18] M. Bartoletti, L. Benetollo, M. Bugliesi, S. Crafa, G. Dal Sasso, R. Pettinau, A. Pinna, M. Piras, S. Rossi, S. Salis, *et al.*, "Smart contract languages: A comparative analysis," *Future Generation Computer Systems*, vol. 164, p. 107563, 2025.

[19] D. Ressi, R. Romanello, C. Piazza, and S. Rossi, "Ai-enhanced blockchain technology: A review of advancements and opportunities," *Journal of Network and Computer Applications*, p. 103858, 2024.

[20] T. Durieux, J. Ferreira, R. Abreu, and P. Cruz, "Smartbugs: Smartbugs wild - dataset for evaluating ethereum smart contract analysis tools," 2020. Accessed: 2025-01-13.

[21] T. Durieux, J. Ferreira, and R. Abreu, "Solidifi benchmark," 2020. Accessed: 2025-01-13.

[22] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic review of security vulnerabilities in ethereum blockchain smart contract," *IEEE Access*, vol. 10, pp. 6605–6621, 2022.

[23] Etherscan, "Etherscan: Ethereum block explorer," 2023. Accessed: 2025-01-13.

[24] S. HajiHosseinKhani, A. H. Lashkari, and A. M. Oskui, "Unveiling smart contracts vulnerabilities: Toward profiling smart contracts vulnerabilities using ega and generating benchmark dataset," *Blockchain: Research and Applications*, Submitted July 2024.

[25] T. Górski, "Smarts: A java package for smart contract test suite generation and execution," *SoftwareX*, vol. 26, p. 101698, 2024.

[26] Z. Cai, S. Farokhnia, A. K. Goharshady, and S. Hitarth, "Asparagus: Automated synthesis of parametric gas upper-bounds for smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 882–911, 2023.

[27] L. Olivieri, L. Negrini, V. Arceri, T. Jensen, and F. Spoto, "Design and implementation of static analyses for tezos smart contracts," *Distributed Ledger Technologies: Research and Practice*, 2024.

[28] E. Albert, P. Gordillo, A. Rubio, and I. Sergey, "Running on fumes," in *Verification and Evaluation of Computer and Communication Systems* (P. Ganty and M. Kaâniche, eds.), (Cham), pp. 63–78, Springer International Publishing, 2019.

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

[30] P. Qian, Z. Liu, Y. Yin, and Q. He, "Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode," in *Proceedings of the ACM Web Conference 2023*, pp. 2220–2229, 2023.

[31] S.-J. Hwang, S.-H. Choi, J. Shin, and Y.-H. Choi, "Codenet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection," *IEEE Access*, vol. 10, pp. 32595–32607, 2022.

[32] F. Özdemir Sönmez and W. J. Knottenbelt, "Contractarmor: Attack surface generator for smart contracts," *Procedia Computer Science*, vol. 231, pp. 8–15, 2024. 14th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 13th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (EUSPN/ICTH 2023).

[33] R. Gupta, M. M. Patel, A. Shukla, and S. Tanwar, "Deep learning-based malicious smart contract detection scheme for internet of things environment," *Computers & Electrical Engineering*, vol. 97, p. 107583, 2022.

[34] K. L. Narayana and K. Sathiyamurthy, "Automation and smart materials in detecting smart contracts vulnerabilities in blockchain using deep learning," *Materials Today: Proceedings*, vol. 81, pp. 653–659, 2023.

[35] Z. Liu, M. Jiang, S. Zhang, J. Zhang, and Y. Liu, "A smart contract vulnerability detection mechanism based on deep learning and expert rules," *IEEE Access*, vol. 11, pp. 77990–77999, 2023.

[36] K. Zhao, Z. Li, J. Li, H. Ye, X. Luo, and T. Chen, "Deepinfer: Deep type inference from smart contract bytecode," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 745–757, 2023.

[37] B. Lê Hng, T. Lê c, T. oàn Minh, D. Trn Tun, D. Phan Th, and H. Phm Vuan, "Contextual language model and transfer learning for reentrancy vulnerability detection in smart contracts," in *Proceedings of the 12th International Symposium on Information and Communication Technology*, pp. 739–745, 2023.

[38] M. Eshghie, C. Artho, and D. Gurov, "Dynamic vulnerability detection on smart contracts using machine learning," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, pp. 305–312, 2021.

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

[40] L. JJ, K. Singh, and B. Chakravarthi, "Digital forensic framework for smart contract vulnerabilities using ensemble models," *Multimedia Tools and Applications*, pp. 1–44, 2023.

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

[42] C. Ma, S. Liu, and G. Xu, "Hgat: smart contract vulnerability detection method based on hierarchical graph attention network," *Journal of Cloud Computing*, vol. 12, no. 1, p. 93, 2023.

[43] V. K. Jain and M. Tripathi, "An integrated deep learning model for ethereum smart contract vulnerability detection," *International Journal of Information Security*, vol. 23, no. 1, pp. 557–575, 2024.

[44] A. Mughaid, I. Obeidat, A. Shdaifat, R. Alhayjna, and S. AlZu'bi, "Modelling and simulation for detecting vulnerabilities and security threats of smart contracts using machine learning," in *2023 Eighth International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 123–127, 2023.

[45] Z. Yang, J. Keung, M. Zhang, Y. Xiao, Y. Huang, and T. Hui, "Smart contracts vulnerability auditing with multi-semantics," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 892–901, 2020.

[46] W. Deng, H. Wei, T. Huang, C. Cao, Y. Peng, and X. Hu, "Smart contract vulnerability detection based on deep learning and multimodal decision fusion," *Sensors*, vol. 23, no. 16, p. 7246, 2023.

[47] A. Mittal, G. Widjaja, R. D. Cosme Pecho, R. Kiruba, J. M. Falcón Roque, and A. Chandra, "Blockchain based abstract syntax tree to detect vulnerability in iot-enabled smart contract," in *2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, pp. 270–275, 2023.

[48] L. Liu, W.-T. Tsai, M. Z. A. Bhuiyan, H. Peng, and M. Liu, "Blockchain-enabled fraud discovery through abnormal smart contract detection on ethereum," *Future Generation Computer Systems*, vol. 128, pp. 158–166, 2022.

[49] X. Yan, S. Wang, and K. Gai, "A semantic analysis-based method for smart contract vulnerability," in *2022 IEEE 8th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pp. 23–28, 2022.

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

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

[52] T. Hu, X. Liu, T. Chen, X. Zhang, X. Huang, W. Niu, J. Lu, K. Zhou, and Y. Liu, "Transaction-based classification and detection approach for ethereum smart contract," *Information Processing & Management*, vol. 58, no. 2, p. 102462, 2021.

[53] Y. Sun and L. Gu, "Attention-based machine learning model for smart contract vulnerability detection," in *Journal of physics: conference series*, vol. 1820, p. 012004, IOP Publishing, 2021.

[54] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, and Y. Wang, "Assbert: Active and semi-supervised bert for smart contract vulnerability detection," *Journal of Information Security and Applications*, vol. 73, p. 103423, 2023.

[55] A. K. Gogineni, S. Swayamjyoti, D. Sahoo, K. K. Sahu, and R. Kishore, "Multi-class classification of vulnerabilities in smart contracts using awd-lstm, with pre-trained encoder inspired from natural language processing," *IOP SciNotes*, vol. 1, no. 3, p. 035002, 2020.

[56] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21* (Z.-H. Zhou, ed.), pp. 2751–2759, International Joint Conferences on Artificial Intelligence Organization, 8 2021. Main Track.

[57] L. Zhang, Y. Li, R. Guo, G. Wang, J. Qiu, S. Su, Y. Liu, G. Xu, H. Chen, and Z. Tian, "A novel smart contract reentrancy vulnerability detection model based on bigas," *Journal of Signal Processing Systems*, pp. 1–23, 2023.

[58] V. K. Jain and M. Tripathi, "Multi-objective approach for detecting vulnerabilities in ethereum smart contracts," in *2023 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, pp. 1–6, 2023.

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

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

[61] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 2, pp. 1296–1310, 2023.

[62] J. Liu, Y. Chen, B. Tan, I. Dillig, and Y. Feng, "Learning contract invariants using reinforcement learning," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–11, 2022.

[63] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2021.

[64] Z. Wu, S. Li, B. Wang, T. Liu, Y. Zhu, C. Zhu, and M. Hu, "Detecting vulnerabilities in ethereum smart contracts with deep learning," in *2022 4th International Conference on Data Intelligence and Security (ICDIS)*, pp. 55–60, 2022.

[65] X. Yu, H. Zhao, B. Hou, Z. Ying, and B. Wu, "Deescvhunter: A deep learning-based framework for smart contract vulnerability detection," in *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2021.

[66] Y. Liu, C. Wang, and Y. Ma, "Dl4sc: a novel deep learning-based vulnerability detection framework for smart contracts," *Automated Software Engineering*, vol. 31, no. 1, p. 24, 2024.

[67] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20* (C. Bessiere, ed.), pp. 3283–3290, International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[68] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, and F. Koushanfar, "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning.," in *NDSS*, 2023.

[69] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts," in *Proceedings of the 3rd ACM international symposium on blockchain and secure critical infrastructure*, pp. 47–59, 2021.

[70] H. Wu, H. Dong, Y. He, and Q. Duan, "Smart contract vulnerability detection based on hybrid attention mechanism model," *Applied Sciences*, vol. 13, no. 2, p. 770, 2023.

[71] C. Xing, Z. Chen, L. Chen, X. Guo, Z. Zheng, and J. Li, "A new scheme of vulnerability analysis in smart contract with machine learning," *Wireless Networks*, pp. 1–10, 2020.

[72] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, "Mando-guru: vulnerability detection for smart contract source code by heterogeneous graph embeddings," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1736–1740, 2022.

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

[74] J. Huang, K. Zhou, A. Xiong, and D. Li, "Smart contract vulnerability detection model based on multi-task learning," *Sensors*, vol. 22, no. 5, p. 1829, 2022.

[75] H. Guo, Y. Chen, X. Chen, Y. Huang, and Z. Zheng, "Smart contract code repair recommendation based on reinforcement learning and multi-metric optimization," *ACM Transactions on Software Engineering and Methodology*, 2023.

[76] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: Towards semantic-aware security auditing for ethereum smart contracts," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 814–819, 2018.

[77] Z. Yang, W. Zhu, and M. Yu, "Improvement and optimization of vulnerability detection methods for ethernet smart contracts," *IEEE Access*, vol. 11, pp. 78207–78223, 2023.

[78] L. Zhang, J. Wang, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen, "A novel smart contract vulnerability detection method based on information graph and ensemble learning," *Sensors*, vol. 22, no. 9, p. 3581, 2022.

[79] D. Vu, T. Nguyen, V. Tong, and S. Souihil, "Enhancing multi-label vulnerability detection of smart contract using language model," in *2023 5th Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*, pp. 1–4, 2023.

[80] C. Gao, W. Yang, J. Ye, Y. Xue, and J. Sun, "sguard+: Machine learning guided rule-based automated vulnerability repair on smart contracts.," *ACM Transactions on Software Engineering and Methodology*, 2024.

[81] S. Jeon, G. Lee, H. Kim, and S. S. Woo, "Design and evaluation of highly accurate smart contract code vulnerability detection framework," *Data Mining and Knowledge Discovery*, vol. 38, no. 3, pp. 888–912, 2024.

[82] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 394–397, 2019.

[83] S. Shakya, A. Mukherjee, R. Halder, A. Maiti, and A. Chaturvedi, "Smartmixmodel: Machine learning-based vulnerability detection of solidity smart contracts," in *2022 IEEE International Conference on Blockchain (Blockchain)*, pp. 37–44, 2022.

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

[85] P. Srinivasan *et al.*, "Tp-detect: trigram-pixel based vulnerability detection for ethereum smart contracts," *Multimedia Tools and Applications*, vol. 82, no. 23, pp. 36379–36393, 2023.

[86] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19685–19695, 2020.

[87] Z. Hou, C. Dong, and Y. Shang, "Hermhd: Enhancing smart contract security based on code obfuscation," in *Proceedings of the 2023 11th International Conference on Information Technology: IoT and Smart City*, pp. 96–101, 2023.

[88] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 23–34, 2020.

[89] S. Akca, A. Rajan, and C. Peng, "Solanalyser: A framework for analysing and testing smart contracts," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 482–489, 2019.

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

[91] S. Yang, H. Li, and Z. Zheng, "Wali: Control-flow-based analysis of wasm smart contracts," in *International Conference on Blockchain and Trustworthy Systems*, pp. 322–335, Springer, 2022.

[92] FuzzingLabs, "Octopus: A neural fuzzer for finding unique bugs in smart contracts." `https://github.com/FuzzingLabs/octopus`, 2024. GitHub repository.

[93] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF con*, vol. 25, no. 11, 2017.

[94] S. Linoy, S. Ray, and N. Stakhanova, "Etherprov: Provenance-aware detection, analysis, and mitigation of ethereum smart contract security issues," in *2021 IEEE International Conference on Blockchain (Blockchain)*, pp. 1–10, 2021.

[95] S. Dharanikota, S. Mukherjee, C. Bhardwaj, A. Rastogi, and A. Lal, "Celestial: A smart contracts verification framework," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, pp. 133–142, 2021.

[96] E. H. Nielsen, D. Annenkov, and B. Spitters, "Formalising decentralised exchanges in coq," in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 290–302, 2023.

[97] Z. Yang and H. Lei, "Fether: An extensible definitional interpreter for smart-contract verifications in coq," *IEEE Access*, vol. 7, pp. 37770–37791, 2019.

[98] S. Holler, S. Biewer, and C. Schneidewind, "Horstify: Sound security analysis of smart contracts," in *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pp. 245–260, 2023.

[99] Z. Neha¨i, P.-Y. Piriou, and F. Daumas, "Model-checking of smart contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 980–987, 2018.

[100] T. Osterland and T. Rose, "Model checking smart contracts for ethereum," *Pervasive and Mobile Computing*, vol. 63, p. 101129, 2020.

[101] E. Albert, J. Correas, P. Gordillo, G. Román-D´iez, and A. Rubio, "Safevm: a safety verifier for ethereum smart contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 386–389, 2019.

[102] E. Keilty, K. Nelaturu, B. Wu, and A. Veneris, "A model-checking framework for the verification of move smart contracts," in *2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 1–7, 2022.

[103] C. Bräm, M. Eilers, P. Müller, R. Sierra, and A. J. Summers, "Rich specifications for ethereum smart contract verification," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

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

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

[106] K. Song, N. Matulevicius, E. B. de Lima Filho, and L. C. Cordeiro, "Esbmc-solidity: An smt-based model checker for solidity smart contracts," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 65–69, 2022.

[107] Z. Yang, H. Lei, and W. Qian, "A hybrid formal verification system in coq for ensuring the reliability and security of ethereum-based service smart contracts," *IEEE Access*, vol. 8, pp. 21411–21436, 2020.

[108] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pp. 91–96, 2016.

[109] J. Godoy, J. P. Galeotti, D. Garbervetsky, and S. Uchitel, "Predicate abstractions for smart contract validation," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, pp. 289–299, 2022.

[110] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 67–82, 2018.

[111] Z. Li, S. Lu, R. Zhang, R. Xue, W. Ma, R. Liang, Z. Zhao, and S. Gao, "Smartfast: an accurate and robust formal analysis tool for ethereum smart contracts," *Empirical Software Engineering*, vol. 27, no. 7, p. 197, 2022.

[112] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*, pp. 161–179, Springer, 2020.

[113] M. Marescotti, R. Otoni, L. Alt, P. Eugster, A. E. Hyvärinen, and N. Sharygina, "Accurate smart contract verification through direct modelling," in *Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III 9*, pp. 178–194, Springer, 2020.

[114] P. Antonino and A. Roscoe, "Solidifier: bounded model checking solidity using lazy contract deployment and precise memory modelling," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 1788–1797, 2021.

[115] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[116] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1678–1694, 2020.

[117] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer, "Formal specification and verification of smart contracts for azure blockchain," *arXiv preprint arXiv:1812.08829*, 2018.

[118] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," in *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*, pp. 446–465, Springer, 2019.

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

[120] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroSP)*, pp. 103–119, 2021.

[121] H. Yang, X. Gu, X. Chen, L. Zheng, and Z. Cui, "Crossfuzz: Cross-contract fuzzing for smart contract vulnerability detection," *Science of Computer Programming*, vol. 234, p. 103076, 2024.

[122] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pp. 557–560, 2020.

[123] M. Rodler, D. Paaßen, W. Li, L. Bernhard, T. Holz, G. Karame, and L. Davi, "Efcf: High performance smart contract fuzzing for exploit generation," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroSP)*, pp. 449–471, 2023.

[124] S. Kim and S. Hwang, "Etherdiffer: Differential testing on rpc services of ethereum nodes," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1333–1344, 2023.

[125] Q. Zhang, Y. Wang, J. Li, and S. Ma, "Ethploit: From fuzzing to efficient exploit generation against smart contracts," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 116–126, 2020.

[126] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "Evmfuzzer: detect evm vulnerabilities via fuzz testing," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 1110–1114, 2019.

[127] S. Smolka, J.-R. Giesen, P. Winkler, O. Draissi, L. Davi, G. Karame, and K. Pohl, "Fuzz on the beach: Fuzzing solana smart contracts," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1197–1211, 2023.

[128] I. Ashraf, X. Ma, B. Jiang, and W. K. Chan, "Gasfuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities," *IEEE Access*, vol. 8, pp. 99552–99564, 2020.

[129] M. Ding, P. Li, S. Li, and H. Zhang, "Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, pp. 321–328, 2021.

[130] L. Breidenbach, P. Daian, F. Tramer, and A. Juels, "The hydra framework for principled, automated bug bounties," *IEEE Security Privacy*, vol. 17, no. 4, pp. 53–61, 2019.

[131] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, "Detecting state inconsistency bugs in dapps via on-chain transaction replay and fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 298–309, 2023.

[132] J. Sun, S. Huang, X. Wang, M. Wang, and J. Du, "A detection method for scarcity defect of blockchain digital asset based on invariant analysis," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pp. 73–84, 2022.

[133] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 531–548, 2019.

[134] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 322–333, 2023.

[135] S. Pani, H. V. Nallagonda, S. Prakash, V. R, R. K. Medicherla, and R. M. A, "Smart contract fuzzing for enterprises: The language agnostic way," in *2022 14th International Conference on COMmunication Systems NETworkS (COMSNETS)*, pp. 1–6, 2022.

[136] H. Feng, X. Ren, Q. Wei, Y. Lei, R. Kacker, D. R. Kuhn, and D. E. Simos, "Magicmirror: Towards high-coverage fuzzing of smart contracts," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 141–152, 2023.

[137] B. Li, Z. Pan, and T. Hu, "Redefender: Detecting reentrancy vulnerabilities in smart contracts automatically," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 984–999, 2022.

[138] S. Ji, J. Dong, J. Wu, and L. Lu, "A guided mutation strategy for smart contract fuzzing," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 282–292, 2023.

[139] M. Olsthoorn, D. Stallenberg, A. Van Deursen, and A. Panichella, "Syntest-solidity: Automated test case generation and fuzzing for smart contracts," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 202–206, 2022.

[140] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 259–269, 2018.

[141] M. Ashouri, "Etherolic: a practical security analyzer for smart contracts," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 353–356, 2020.

[142] W. Li, M. Wang, B. Yu, Y. Shi, M. Fu, and Y. Shao, "Grey-box fuzzing based on execution feedback for eosio smart contracts," in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 1–10, 2022.

[143] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1398–1409, 2020.

[144] D. Maier, F. Fäßler, and J.-P. Seifert, "Uncovering smart contract vm bugs via differential fuzzing," in *Reversing and Offensive-oriented Trends Symposium*, pp. 11–22, 2021.

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

[146] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–12, 2022.

[147] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 227–239, 2021.

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

[149] M. Wang, Z. Wang, B. Yu, and K. Zhang, "Wasaiup: A demand-driven concolic fuzzer for eosio smart contracts," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pp. 150–161, 2023.

[150] P. Zhang, J. Yu, and S. Ji, "Adf-ga: data flow criterion based test case generation for ethereum smart contracts," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 754–761, 2020.

[151] A. Mavridou and A. Laszka, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," in *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*, pp. 270–277, Springer, 2018.

[152] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, "Modcon: A model-based testing platform for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1601–1605, 2020.

[153] T. Abdellatif and K.-L. Brousmiche, "Formal verification of smart contracts based on users and blockchain behaviors models," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, 2018.

[154] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Smartinspect: solidity smart contract inspector," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 9–18, 2018.

[155] C. Ferreira Torres, H. Jonker, and R. State, "Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 115–128, 2022.

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

[157] C. S. Yashavant, "Secsec: Securing smart ethereum contracts," in *Proceedings of the 17th Innovations in Software Engineering Conference*, pp. 1–4, 2024.

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

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

[160] Y. Ivanova and A. Khritankov, "Regularmutator: a mutation testing tool for solidity smart contracts," *Procedia Computer Science*, vol. 178, pp. 75–83, 2020.

[161] M. Barboni, A. Morichetta, A. Polini, and F. Casoni, "Resumo: a regression strategy and tool for mutation testing of solidity smart contracts," *Software Quality Journal*, vol. 32, no. 1, pp. 225–253, 2024.

[162] M. Barboni, A. Morichetta, and A. Polini, "Sumo: A mutation testing approach and tool for the ethereum blockchain," *Journal of Systems and Software*, vol. 193, p. 111445, 2022.

[163] K. Zhu, X. Wang, Z. Chen, S. Huang, and J. Wu, "Evaluating ethereum reentrancy detection tools via mutation testing," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 545–555, 2023.

[164] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "Musc: A tool for mutation testing of ethereum smart contract," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1198–1201, 2019.

[165] S. Wu, Z. Yu, D. Wang, Y. Zhou, L. Wu, H. Wang, and X. Yuan, "Defiranger: Detecting defi price manipulation attacks," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–15, 2023.

[166] R. Dua, "Ethlint: Linter for solidity code." `https://github.com/duaraghav8/Ethlint`, 2024. Accessed: 2024-06-11.

[167] C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State, "The eye of horus: Spotting and analyzing attacks on ethereum smart contracts," in *International Conference on Financial Cryptography and Data Security*, pp. 33–52, Springer, 2021.

[168] J. Fei, X. Chen, and X. Zhao, "Msmart: Smart contract vulnerability analysis and improved strategies based on smartcheck," *Applied Sciences*, vol. 13, no. 3, p. 1733, 2023.

[169] K. Yan, J. Zhang, X. Liu, W. Diao, and S. Guo, "Bad apples: Understanding the centralized security risks in decentralized ecosystems," in *Proceedings of the ACM Web Conference 2023*, pp. 2274–2283, 2023.

[170] N. Ivanov, H. Guo, and Q. Yan, "Rectifying administrated erc20 tokens," in *Information and Communications Security* (D. Gao, Q. Li, X. Guan, and X. Liao, eds.), (Cham), pp. 22–37, Springer International Publishing, 2021.

[171] Y. Zhang, J. Ma, X. Liu, G. Ye, Q. Jin, J. Ma, and Q. Zhou, "An efficient smart contract vulnerability detector based on semantic contract graphs using approximate graph matching," *IEEE Internet of Things Journal*, vol. 10, no. 24, pp. 21431–21442, 2023.

[172] C. Peng, S. Akca, and A. Rajan, "Sif: A framework for solidity contract instrumentation and analysis," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–473, IEEE, 2019.

[173] W. Dong, T. Zhou, and D. Yan, "Solchecker: A practical static analysis framework for ethereum smart contract," in *2022 International Conference on Networks, Communications and Information Technology (CNCIT)*, pp. 179–186, 2022.

[174] Protofire, "Solhint documentation." `https://protofire.github.io/solhint/`, 2024. Accessed: 2024-06-10.

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

[176] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 1503–1520, 2019.

[177] N. K. Shah, S. Nandurkar, M. Bilapate, M. A. Maalik, N. Harne, K. Shaik, and A. Kumar, "Smart contract vulnerability detection techniques for hyperledger fabric," in *2023 IEEE 8th International Conference for Convergence in Technology (I2CT)*, pp. 1–7, 2023.

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

[179] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 945–956, IEEE, 2023.

[180] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in ethereum," in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, pp. 110–115, 2020.

[181] W. Li, J. He, G. Zhao, J. Yang, S. Li, R. Lai, P. Li, H. Tang, H. Luo, and Z. Zhou, "Eosioanalyzer: An effective static analysis vulnerability detection framework for eosio smart contracts," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 746–756, 2022.

[182] J. Huang, S. Han, W. You, W. Shi, B. Liang, J. Wu, and Y. Wu, "Hunting vulnerable smart contracts via graph embedding based bytecode matching," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2144–2156, 2021.

[183] Z. Li, W. Guo, Q. Xu, Y. Xu, H. Wang, and M. Xian, "Research on blockchain smart contracts vulnerability and a code audit tool based on matching rules," in *Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies*, pp. 484–489, 2020.

[184] N. Lu, B. Wang, Y. Zhang, W. Shi, and C. Esposito, "Neucheck: A more practical ethereum smart contract security analysis tool," *Software: Practice and Experience*, vol. 51, no. 10, pp. 2065–2084, 2021.

[185] D. Mohanty and D. Anand, "Evaluating the remix solidity static analysis plugin using sb curated dataset," in *2023 7th International Conference On Computing, Communication, Control And Automation (ICCUBEA)*, pp. 1–4, 2023.

[186] H. Chen, X. Zhao, Y. Wang, and Z. Zhen, "Safecheck: Detecting smart contract vulnerabilities based on static program analysis methods," *Security and Privacy*, p. e393.

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

[188] S. Ducasse, H. Rocha, S. Bragagnolo, M. Denker, and C. Francomme, "Smartanvil: Open-source tool suite for smart contract analysis," in *Blockchain and Web 3.0*, pp. 192–222, Routledge, 2019.

[189] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 9–16, 2018.

[190] Z. Liao, Z. Zheng, X. Chen, and Y. Nan, "Smartdagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 752–764, 2022.

[191] M. Ayub, T. Saleem, M. Janjua, and T. Ahmad, "Storage state analysis and extraction of ethereum blockchain smart contracts," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–32, 2023.

[192] T. Hu, B. Li, Z. Pan, and C. Qian, "Detect defects of solidity smart contract based on the knowledge graph," *IEEE Transactions on Reliability*, vol. 73, no. 1, pp. 186–202, 2024.

[193] P. Zhang, F. Xiao, and X. Luo, "Soliditycheck: Quickly detecting smart contract problems through regular expressions," *arXiv preprint arXiv:1911.09425*, 2019.

[194] G. Ramakrishnan, M. Rehan, and G. Sujatha, "Solidity vulnerability scanner," in *2022 International Conference on Data Science, Agents Artificial Intelligence (ICDSAAI)*, vol. 01, pp. 1–5, 2022.

[195] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 716–727, 2022.

[196] A. Ghaleb, "Towards effective static analysis approaches for security vulnerabilities in smart contracts," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–5, 2022.

[197] S. M. Beillahi, E. Keilty, K. Nelaturu, A. Veneris, and F. Long, "Automated auditing of price gouging tod vulnerabilities in smart contracts," in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–6, 2022.

[198] N. Songsom, W. Werapun, J. Suaboot, and N. Rattanavipanon, "The swc-based security analysis tool for smart contract vulnerability detection," in *2022 6th International Conference on Information Technology (InCIT)*, pp. 74–77, 2022.

[199] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "Vultron: Catching vulnerable smart contracts once and for all," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 1–4, 2019.

[200] S. Azzopardi, J. Ellul, and G. J. Pace, "Monitoring smart contracts: Contractlarva and open challenges beyond," in *Runtime Verification* (C. Colombo and M. Leucker, eds.), (Cham), pp. 113–137, Springer International Publishing, 2018.

[201] T. Cook, A. Latham, and J. H. Lee, "Dappguard: Active monitoring and defense for solidity smart contracts," *Retrieved July*, vol. 18, p. 2018, 2017.

[202] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–28, 2017.

[203] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi, "Evm: From offline detection to online reinforcement for ethereum virtual machine," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 554–558, 2019.

[204] X. Zhang, W. Sun, Z. Xu, H. Cheng, C. Cai, H. Cui, and Q. Li, "Evm-shield: In-contract state access control for fast vulnerability detection and prevention," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 2517–2532, 2024.

[205] J. Zhang, J. Gao, Y. Li, Z. Chen, Z. Guan, and Z. Chen, "Xscope: Hunting for cross-chain bridge attacks," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–4, 2022.

[206] C. Ferreira Torres, M. Baden, R. Norvill, B. B. Fiz Pontiveros, H. Jonker, and S. Mauw, "Ægis: Shielding vulnerable smart contracts against attacks," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pp. 584–597, 2020.

[207] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung, "Contractguard: Defend ethereum smart contracts with embedded intrusion detection," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 314–328, 2020.

[208] B. Nassirzadeh, H. Sun, S. Banescu, and V. Ganesh, "Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities," in *Mathematical Research for Blockchain Economy* (P. Pardalos, I. Kotsireas, Y. Guo, and W. Knottenbelt, eds.), (Cham), pp. 143–167, Springer International Publishing, 2023.

[209] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, *et al.*, "Soda: A generic online detection framework for smart contracts.," in *NDSS*, 2020.

[210] L. Stegeman, "Solitor: runtime verification of smart contracts on the ethereum network," Master's thesis, University of Twente, 2018.

[211] X. Wu, X. Du, Q. Yang, A. Liu, N. Wang, and W. Wang, "Taintguard: Preventing implicit privilege leakage in smart contract based on taint tracking at abstract syntax tree level," *Journal of Systems Architecture*, vol. 141, p. 102925, 2023.

[212] Q.-B. Nguyen, A.-Q. Nguyen, V.-H. Nguyen, T. Nguyen-Le, and K. Nguyen-An, "Detect abnormal behaviours in ethereum smart contracts using attack vectors," in *Future Data and Security Engineering: 6th International Conference, FDSE 2019, Nha Trang City, Vietnam, November 27–29, 2019, Proceedings 6*, pp. 485–505, Springer, 2019.

[213] T. Yin, C. Zhang, Y. Ni, Y. Wu, T. Wong, X. Luo, Z. Li, and Y. Guo, "An empirical study on implicit constraints in smart contract static analysis," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pp. 31–32, 2022.

[214] Y. Li, "Finding concurrency exploits on smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 144–146, 2019.

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

[216] Q. Han, L. Wang, H. Zhang, L. Shi, and D. Wang, "Ethchecker: a context-guided fuzzing for smart contracts," *The Journal of Supercomputing*, pp. 1–27, 2024.

[217] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 454–469, 2020.

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

[219] L. Jin, Y. Cao, Y. Chen, D. Zhang, and S. Campanoni, "Exgen: Cross-platform, automated exploit generation for smart contract vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 650–664, 2023.

[220] Z. Wang, W. Dai, K.-K. R. Choo, H. Jin, and D. Zou, "Fsfc: An input filter-based secure framework for smart contract," *Journal of Network and Computer Applications*, vol. 154, p. 102530, 2020.

[221] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1433–1448, 2021.

[222] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 442–446, 2017.

[223] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 1591–1607, USENIX Association, Aug. 2019.

[224] Y. Fang, C. Wang, Z. Sun, and H. Cheng, "Jyane: Detecting reentrancy vulnerabilities based on path profiling method," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 274–282, 2021.

[225] W. Zhang, S. Banescu, L. Pasos, S. Stewart, and V. Ganesh, "Mpro: Combining static and symbolic analysis for scalable testing of smart contract," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 456–462, 2019.

[226] D. Prechtel, T. Groß, and T. Müller, "Evaluating spread of 'gasless send' in ethereum smart contracts," in *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–6, 2019.

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

[228] F. Ma, Z. Xu, M. Ren, Z. Yin, Y. Chen, L. Qiao, B. Gu, H. Li, Y. Jiang, and J. Sun, "Pluto: Exposing vulnerabilities in inter-contract scenarios," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4380–4396, 2022.

[229] Crytic, "Rattle: An evm binary static analysis framework." `https://github.com/crytic/rattle`, 2024. GitHub repository.

[230] R. Yu, J. Shu, D. Yan, and X. Jia, "Redetect: Reentrancy vulnerability detection in smart contracts with high accuracy," in *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*, pp. 412–419, 2021.

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

[232] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 161–178, 2022.

[233] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[234] Z. Yang, H. Liu, Y. Li, H. Zheng, L. Wang, and B. Chen, "Seraph: enabling cross-platform security analysis for evm and wasm smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pp. 21–24, 2020.

[235] J. Wu, Y. Wang, R. Wang, J. Chen, and Z. Zheng, "Can neural networks help smart contract testing? an empirical study," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, pp. 79–89, 2023.

[236] Y. Fang, D. Wu, X. Yi, S. Wang, Y. Chen, M. Chen, Y. Liu, and L. Jiang, "Beyond "protected" and "private": An empirical security analysis of custom function modifiers in smart contracts," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1157–1168, 2023.

[237] S. Hwang and S. Ryu, "Gap between theory and practice: An empirical study of security patches in solidity," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 542–553, 2020.

[238] Y. Yao, H. Li, X. Yang, and Y. Le, "An improved vulnerability detection system of smart contracts based on symbolic execution," in *2022 IEEE International Conference on Big Data (Big Data)*, pp. 3225–3234, 2022.

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

[240] C. Ma, W. Song, and J. Huang, "Transracer: Function dependence-guided transaction race detection for smart contracts," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 947–959, 2023.

[241] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1661–1677, 2020.

[242] K. Weiss and J. Schütte, "Annotary: A concolic execution system for developing secure smart contracts," in *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pp. 747–766, Springer, 2019.

[243] N. Veloso, "Conkas: Concolic analysis for ethereum smart contracts." `https://github.com/nveloso/conkas`, 2024. Accessed: 2024-06-11.

[244] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2189–2207, 2022.

[245] J. Frank, C. Aschermann, and T. Holz, "{ETHBMC}: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2757–2774, 2020.

[246] E. Albert, J. Correas, P. Gordillo, G. Román-D´iez, and A. Rubio, "Gasol: Gas analysis and optimization for ethereum smart contracts," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 118–125, Springer, 2020.

[247] P. Li, S. Li, M. Ding, J. Yu, H. Zhang, X. Zhou, and J. Li, "A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis," in *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, pp. 366–374, 2022.

[248] Y. Liu and L. Cai, "Honeytoken-detector: A symbolic execution-based honeypot token detection tool," in *2023 26th ACIS International Winter Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD-Winter)*, pp. 134–139, IEEE, 2023.

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

[250] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Rocsu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 912–915, 2018.

[251] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Maian: Automatic detection of greedy, prodigal, and suicidal contracts," 2018. Accessed: 2024-06-14.

[252] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1186–1189, 2019.

[253] M. Fu, L. Wu, Z. Hong, F. Zhu, H. Sun, and W. Feng, "A critical-path-coverage-based vulnerability detection method for smart contracts," *IEEE Access*, vol. 7, pp. 147327–147344, 2019.

[254] ConsenSys, "Mythril: Security analysis tool for ethereum smart contracts," 2018. Accessed: 2024-06-14.

[255] B. Mueller, "Awesome mythx smart contract security tools," 2020. Accessed: 2024-06-14.

[256] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in nft smart contracts," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 373–384, 2023.

[257] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th annual computer security applications conference*, pp. 664–676, 2018.

[258] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269, 2016.

[259] P. Keoleian, "pakala: A tool for packaging and publishing python libraries," 2024. GitHub repository.

[260] P. Zheng, Z. Zheng, and X. Luo, "Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 740–751, 2022.

[261] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura, "Ra: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis," in *2020 IEEE International Conference on Blockchain (Blockchain)*, pp. 327–336, 2020.

[262] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "scompile: Critical path identification and analysis for smart contracts," in *Formal Methods and Software Engineering* (Y. Ait-Ameur and S. Qin, eds.), (Cham), pp. 286–304, Springer International Publishing, 2019.

[263] S.-W. Lin, P. Tolmach, Y. Liu, and Y. Li, "Solsee: a source-level symbolic execution engine for solidity," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1687–1691, 2022.

[264] Y. Feng, E. Torlak, and R. Bod´ik, "Summary-based symbolic evaluation for smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1141–1152, 2020.

[265] Y. Smaragdakis, N. Grech, S. Lagouvardos, K. Triantafyllou, and I. Tsatiris, "Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[266] B. Jiang, Y. Chen, D. Wang, I. Ashraf, and W. Chan, "Wana: Symbolic execution of wasm bytecode for extensible smart contract vulnerability detection," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 926–937, 2021.

[267] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1029–1040, 2020.

[268] J. Xu, J. Shen, and S. Tan, "Doschecker: An efficient and dedicated tool for detecting dos vulnerability in smart contracts," in *2023 International Conference on Data Security and Privacy Protection (DSPP)*, pp. 22–31, 2023.

[269] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "Easyflow: Keep ethereum away from overflow," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 23–26, 2019.

[270] P. Qian, J. He, L. Lu, S. Wu, Z. Lu, L. Wu, Y. Zhou, and Q. He, "Demystifying random number in ethereum smart contract: Taxonomy, vulnerability identification, and attack detection," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3793–3810, 2023.

[271] A. Ali, Z. U. Abideen, and K. Ullah, "Sescon: Secure ethereum smart contracts by vulnerable patterns' detection," *Security and Communication Networks*, vol. 2021, no. 1, p. 2897565, 2021.

[272] J. Zhang, Y. Li, J. Gao, Z. Guan, and Z. Chen, "Siguard: Detecting signature-related vulnerabilities in smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 31–35, IEEE, 2023.

[273] N. F. Samreen and M. H. Alalfi, "Smartscan: An approach to detect denial of service vulnerability in ethereum smart contracts," in *2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 17–26, 2021.

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

[275] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse engineering ethereum's opaque smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 1371–1385, USENIX Association, Aug. 2018.

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

[277] Y. Liu and Y. Li, "Invcon: A dynamic invariant detector for ethereum smart contracts," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–4, 2022.

[278] G. A. Pierro, "Smart-graph: Graphical representations for smart contract on the ethereum blockchain," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 708–714, 2021.

[279] R. Revere, "Solgraph: Visualizing control flow of a solidity contract," 2018. GitHub repository.

[280] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15, 2019.

Vita

Niosha Hejazi

B.Sc. in Industrial Engineering, Sharif University of Technology, Tehran, Iran
2017–2022

Publications:
Niosha Hejazi, Arash Habibi Lashkari, **"A Comprehensive Survey of Smart Contracts Vulnerability Detection Tools: Techniques and Methodologies"**, published in Journal of Network and Computer Applications (Elsevier), Volume 237, 2025, Article 104142.

Niosha Hejazi, Arash Habibi Lashkari, **"Smart contract vulnerability detection using graph neural networks with transformer-based encoders"**, *Manuscript in preparation*.