Received 14 February 2024, accepted 3 May 2024, date of publication 15 May 2024, date of current version 28 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3401623

SURVEY

A Survey of Vulnerability Detection Techniques by Smart Contract Tools

ZULFIQAR ALI KHAN AND AKBAR SIAMI NAMIN
Computer Science Department, Texas Tech University, Lubbock, TX 79401, USA

Corresponding author: Zulfiqar Ali Khan (zulfi.khan@ttu.edu)

This work was supported in part by the National Science Foundation under Grant 1723765 and Grant 1821560.

ABSTRACT Ethereum Blockchain technology introduced a competitive environment in the financial sector. Consequently, new technologies emerged, such as Smart Contracts (SCs), which preclude code corrections due to their immutable nature. However, the incorrect and faulty uploaded SCs led to uninvited penetrations into SCs' accounts, resulting in considerable customer losses. This SC's drawback requires tools to test the SCs and paves the way for research on vulnerability detection techniques. Our survey paper comprehensively reviews 41 SC tools and presents the vulnerability detection techniques (VDTs) of several previously invented tools by dividing them into general and specific classes. Finally, we also perform a classification of detection techniques to standardize the approaches. Thus, our study will help SC developers and security analysts to streamline the security of SCs and reduce the chances of malicious monetary transfers.

INDEX TERMS Ethereum blockchain, smart contracts, survey, vulnerability detection techniques, classification of smart contracts, vulnerability detection tools, static analysis, dynamic analysis.

I. INTRODUCTION

A Blockchain is a database of blocks containing a list of transactions of fixed or variable sizes. Henceforth, the immense volume of stored information and the mechanisms for creating and handling such gigantic information can invite several anomalies. For instance, Blockchain's reliance on the distributed consensus mechanism can result in a 51% attack [1]. The attack requires that most of the computing power is malicious. Fortunately, in an extensive distributed network, it isn't easily achieved. However, the expansion of Blockchain can lead to scalability problems, causing slow processing of the transactions. Again this problem overwhelms only if the transaction is urgent [2]. But still, Blockchain technology enjoys a less welcoming attitude in several countries because its nascent existence has met with severe attacks like DAO in 2016 and, more recently, the Binance attack in 2022 [3], [4]. These incidents highlight that security problems keep haunting Blockchain investors; hence, Blockchain's programming modules, i.e., smart contracts (SCs), need immediate attention. One way to boost the security of SCs is by developing robust tools to prevent hackers from stealing funds from SC accounts, Brent et al. [5]. Developers with recent exposure to security flaws and research like ours, which focuses on SC vulnerability detection techniques, can be helpful in neutralizing attack vectors in this direction.

A vulnerability [6], [7], [8] refers to a programming practice that can be problematic due to the programmer not anticipating a malicious semantic behavior. In traditional networks of financial institutions, vulnerable code may be present in the repositories [9]. But in EVM there are known cases of the vulnerable patterns causing direct and indirect harm to users' funds. Our work focuses on both direct [10] and indirect [11] losses of cryptocurrency like Ether. Direct loss involves accessing vulnerable code segments by the attacker that can transfer Ether to unknown accounts. On the other hand, indirect loss occurs due to the loss of gas, which acts as fuel for running programs on the EVM and has a cost associated with it. Work in [12] discusses some coding patterns that can cause 'out of gas' exceptions (patterns normally relate to images [13]). Thus, understanding Solidity concepts helps to design secured and economical SCs. Hence, our research does not restrict vulnerability to unsecured access to Ether transfer statements (i.e., unprotected

The associate editor coordinating the review of this manuscript and approving it for publication was Binit Lukose.

Ether withdrawal [14]), costly gas patterns [15], deprecated instructions (useful to identify vulnerabilities in old deployed SCs [16]), mathematical discrepancies [17], misusing SC's program variables [18] and Blockchain variables [19], resulting in miners' violations. But, we include any other coding flaw as minor as ignoring return values or devastating like reentrancy if it highlights a wicked intent.

Our analysis contains over 20 state-of-the-art SC vulnerabilities and 42 detecting tools. We focus on the following three dimensions (a) vulnerability detection techniques and their classification (b) highlighting of Solidity and bytecode-based SCs' exploitable weaknesses (c) workings of various Ethereum SC's testing tools.

A. THEORETICAL FOUNDATION

The motivation is that a better understanding of vulnerability detection techniques would develop (i) more effective tools, (ii) a bug-free Solidity programming language, and (iii) a protective EVM Blockchain. Apart from other research, the work presented by Atzei et al. [20] acted as a backbone for our programming knowledge related to SC vulnerabilities. In our previous survey, we provide a list of vulnerabilities, categorized in the context of domain knowledge of the vulnerable operation of SC [23]. We also show how the adversary could exploit the vulnerabilities to generate attacks such as DAO, Grossman et al. [38] and King of Ether Throne (KOET), Atzei et al. [20]. Comparatively, the following reasons compel us to develop an updated survey:

- *Revision in Solidity Programming Constructs.* The Solidity [39] programming language has undergone major changes. Solidity has introduced new directives like `payable` to indicate that a function can receive Ether or an address is capable of transferring Ether. The '`constructor`' keyword represents the constructor's name instead of SC's. The built-in function names have also been changed, like `revert` replaces `throw`, `selfdestruct` replaces `suicide`, `var` deprecated, and so on.
  Some recent changes are the introduction of a virtual keyword for the overriding and an abstract keyword to indicate that we cannot create the instance of SC because the SC has at least one undefined function. One significant change is the splitting of the fallback function into `receive()`, and `fallback()` functions, where `receive()` only handles monetary transactions, and `fallback()` can handle both data and financial transactions. We replicated the code from previous research (with references) and modified it in conversant with the "solc" compiler version 0.6.1.
- *Covering Additional Vulnerabilities and Code-level Bugs.* Several vulnerabilities and code-level bugs are overlooked in previous surveys, including Atzei et al. [20]. For instance, there is no discussion on Arithmetic Bugs (or Integer Overflow) in the survey, Atzei et al. [20]. Furthermore, the previous surveys do not shed thorough light on gas-related vulnerabilities. Our research addresses this gap by providing detection techniques for integer overflow and gas-related vulnerabilities.
- *Not Adequate Discussions on the Latest SC Tools.* Unlike Atzei et al. [20], we incorporate vulnerability detection techniques developed after Atzei's survey, like the techniques based on fuzzing tools, XML, and invariants; hence our work can provide more up-to-date knowledge for future research. Invariants are always true properties, like the balance invariant, which should not be false for funds withdrawal (i.e., the balance is greater than the withdrawal amount).

Furthermore, recent surveys presented in [21] and [40], Tolmach et al. [41], [42] and Ji et al. [43] are also different from ours, because our survey provides:

- *Thorough Discussion on SC Vulnerability Detection Techniques*. None of the research in [44], [45], Xu et al. [46], Praitheeshan et al. [34], Kushwaha et al. [47] and López Vivar et al. [48] comprehensively discuss SC vulnerability detection techniques and their classification. For the first time, we provide an up-to-date discussion on 41 SC tools within Ethereum Blockchain focusing Solidity programming language and/or EVM bytecode-based SCs. Work in [23] provides a comprehensive discussion about SC vulnerabilities which could be helpful for novice Blockchain developers to understand some of the concepts mentioned in this research. Our work extends the research in [23].

Table 1 provides the section# of our survey, significance, and focus of the research work linked to the mentioned section. We divide Table 1 into five parts discussed below:

Part 1 cites two articles that may arouse interest in Ethereum research. For instance, Atzei et al. [20] discusses several buzzwords like "King of Ether Throne" (i.e., a game), "msg.sender" (i.e., Solidity's keyword representing the caller of SC's any function), "Ponzi Scheme" (i.e., a fraudulent SC promoting investment [49]) and so on. The game offers a lucrative crown to users at the cost of some Ether. However, when the users play the game, they lose their Ether due to mishandled Exception vulnerability. The other work in [21] provides helpful information about the Consensus algorithm that solves the trust problem in Blockchain so that Blockchain doesn't require 3rd party to act as a witness for the transaction. The research compares six consensus algorithms. For instance, the 'Proof of Work' consensus algorithm deals with BitCoin Blockchain. The algorithm requires the computing nodes, called miners, to solve a mathematical puzzle to validate the transactions in a block before the block becomes part of the Blockchain. Finally, the paper focuses on security issues and discusses various vulnerability detection tools like Zeus (Section III-A1) and MadMax (Section III-A16).

In **Part 2**, we discuss two papers. One is [22], which classifies SC vulnerabilities and assigns them IDs. This

TABLE 1. Some vital, novel, and additional contributions related to different sections of our article.

Survey Section	Significance	The Focus of Research Papers
		PART 1
Introduction & Theoretical Foundation, (Section I-A)	Buzzwords, SC & Blockchain Technology	– Survey of attacks on SC, Atzei *et al.* [20]
		– Blockchain Technology & Security, Guo & Yu [21]
		PART 2
Classification of SC Vulnerabilities, (Section II)	SC Vulnerabilities: Classification & Categorization	– Survey: Vulnerability Analysis & Description, Rameder *et al.* [22], Khan & Namin [23]
		PART 3
SC Tools Detecting Vulnerabilities, (Section III)	Latest Work	– MSmart, Fei *et al.* [24], Horstify, Holler *et al.* [25], IR-Fuzz, Liu *et al.* [26], EXGEN, Jin *et al.* [27], TechyTech, Khan [28]
	Information Flow Rules	– Composite Vulnerability Analyzer, Ethainter, Brent *et al.* [5]
	Path Protective Technique	– Cross Contract Static Analysis, Clairvoyance, Xue *et al.* [29]
	Rare Work	– Event Ordering Bugs, Kolluri *et al.* [30]
	Rare Work	– BatchOverFlow Detection, Feng *et al.* [31]
	Description of 13 tools	– Survey: Vulnerabilities and SafeGuards, Pise & Patil [32]
		PART 4
Specific Vulnerability Detection Techniques, (Section V to XIV)	Latest Work	– Survey: SC Analysis Methods, Munir & Taha [33]
	Analysis Types and Methodologies	– Suvey: Vulnerability Analysis Methods, Praitheeshan *et al.* [34]
		PART 5
Future Directions,	Latest Work	– Survey: SC Threat Mitigation, Ivanov *et al.* [35]
	Development	– Reusing, Code Compactness: Brent *et al.* [36]
(Section XVI)	of Tools	– Precise Tool Example, Wang *et al.* [37]

work is the need of the hour because it could provide a standard scheme for referring to the vulnerabilities. However, the authors don't use efficient coding for the subjects. For example, they focused on 54 vulnerabilities and could achieve their coding using 6-bit binary values. Instead, the authors started with 2-Hex digits but later shifted to 3-Hex numbers, which is not scientific. The other work in [23] helps understand the SC vulnerabilities and familiarizes readers with Solidity coding.

In **Part 3**, we introduce five vulnerability detection tools developed in 2023: Exgen (Section III-A24), MSmart (Section III-A14), IR-Fuzz (Section III-A18), Horstify (Section III-A20) and TechyTech (Section III-B11). Additionally, part 3 references some novel tools like Ethainter (Section VII-A6) and Clairvoyance (Section III-A29). Ethainter focuses on information flow rules, and Clairvoyance incorporates path-protective techniques. Information flow partitions information into trusted and untrusted groups and traces their path. Path protective techniques help in the feasibility checking of paths to reduce false positive cases. But the survey in [32] discusses the methods to defend against vulnerabilities, which could be helpful to novice users. Further, we highlight the rare work done in [30] (i.e., detection of event ordering bugs (Section III-B9)), [31] (i.e., summary-based symbolic analysis (Section III-A7), and detection of batch overflow vulnerability (Section VI-B4).

In **Part 4** we include two surveys i.e., [33] (published in 2023), and [34]. Survey [33] focuses on SC properties and provides more than 200 references. The property represents the SC's behavior and helps identify the program's correctness. In addition, the authors classify their analysis methods into four major categories: (a) static, (b) dynamic, (c) formal verification, and (d) machine learning [50]. Finally, the authors organize the analysis methods into more specific ones (like a: static type checking, abstract interpretation, b: Concolic testing, Fuzzing, c: theorem proving, model checking, d: supervised and unsupervised learning) and then discuss the properties associated with them. Survey [34] highlights the uniqueness of attacks. For example, an attacker exploits the `selfdestruct` vulnerability, destroys the SC, and locks the Ether of dependent SCs. However, none of the above three surveys discuss the specific vulnerability detection techniques that we focus on in this article. Part 5 discusses the future research directions (Section XVI), which could influence new researchers in chalking out their research plans.

Our attention has been on the tools, and we think pre-scanning SCs with vulnerability detection tools can reveal code flaws. Several tools are already available for vulnerability detection, but their execution processes are complex, and the tools' research does not classify the detection techniques into a SC-based classification model. We are the first to provide a classification model for SC-based vulnerability detection techniques after going through the original research of the tools. To classify the detection approaches, we first gather techniques from research papers and web articles, verify them through discussion forums

TABLE 2. Categories and percentages: Downloaded and referenced research papers.

Ratios	Tools	Blk. Sec. & Attacks	Testing	Solidity	Vulnerability	Non-Sec. Survey	SWC & DASP	Sec. Survey
R to R	60%	6.6%	10.8%	8.3%	5%	4%	2%	3.3%
D to D	40.9%	5%	8.2%	8.3%	10.6%	3.7%	13.9 %	9.4%

(if there is any ambiguity), and then categorize the techniques into various SC-based vulnerability detection classes.

Furthermore, Ethereum SC technology's ongoing evolution adds new features to the Solidity programming language and Ethereum Blockchain technology, confirming the validity of our research. Consequently, the latest versions of the Solidity programming language do not support many old vulnerability detection tools. Yet, the old vulnerabilities persist because EVM still supports the old SCs. Therefore, it is necessary to highlight the old tools' detection techniques so that the Ethereum Blockchain community can support old SCs and develop new tools to fill the gap related to inefficient and outdated tools.

B. SURVEY METHODOLOGY

Before going into the details of our methodology, we want to state that we try to familiarize ourselves with Truffle[1] and Remix[2] tools to understand how these tools detect SC vulnerabilities.

However, the usage of any tool discussed in (Section III) for analysis of any SC is beyond the scope of this research. We classify the tools into different vulnerability detection classes, listed in (Section IV), using the information retrieved from the tool's associated research article's title, text or specifically the steps involved in the vulnerability detection process (also retrieved from article's text). We identify ten vulnerability detection classes based on software testing, program analysis, and EVM modules.

We started browsing research papers using the search phrase "Ethereum Blockchain Survey". But after reading the article [51] based upon the F* framework, we transformed our unidirectional keyword search into a two-dimensional one because we also started browsing the vulnerability detection tools using their exact titles through the university portal. We emphasize this aspect in Figure 4 using the expression $GoogleSearch + UniversityPortal$. The said article provides information about two buzzwords: reentrancy and Oyente. Similarly, the retrieval of the survey paper published by Atzei et al. [7] provided references to several research papers, which led to the download of 10 additional research papers, including Kalra et al. [40].

Hence, we learned that there are several vulnerabilities apart from reentrancy and several tools in addition to Oyente. Thus our continuous reading of the downloaded articles gave us the sound decision to download which of the remaining 39 articles related to our vulnerability detection tools survey. In short, our strategy was not to download all (i.e., 669) research papers first and then select the 41 tools from them. Instead, we devised the method of download-read-select and repeated the cycle until we completed the download of our remaining 39 tool-based articles.

[1]https://www.trufflesuite.com/
[2]https://remix.ethereum.org

We constructed databases for survey and non-survey papers and moved tool and vulnerability-linked documents from these two databases into a single (or special) 'tool and vulnerability' database. The University portal fortifies the database by targeted search, utilizing the titles of tools' and vulnerabilities' papers. The specialty of the 'tool and vulnerability' database is to keep a count of tools and vulnerabilities and their detection techniques.

We are first downloading the papers and assigning them a category (i.e., a subfolder, represented by a string as mentioned in Table 2) and an id (indicating count of paper). After that, the categorized paper is referenced (if appropriate). Next, we calculate two percentages (i.e., the reference papers percentage and the downloaded papers percentage) for each category. These percentages indicate the nearness factor (defined below) of our categories with our research. If the primary category (defined below) has the highest percentage for both the referenced and downloaded papers, then it indicates that the primary category has the highest nearness factor and hence the correctness of our methodology. We identify the percentages as the ratio of 'Referenced papers of the category to the total Referenced papers' (i.e., R to R) and the ratio of the 'Downloaded papers of the category to the total Downloaded papers' (i.e., D to D) and use the following formulas for their evaluation:

$$R\ to\ R\ of\ a\ Category = \frac{Category\ References\ *\ 100}{Total\ References\ of\ all\ categories}$$

$$D\ to\ D\ of\ a\ Category = \frac{Category\ Downloads\ *\ 100}{Total\ Downloads\ of\ all\ categories}$$

Table 2 categorizes our downloaded research papers. Vulnerability Detection Tools, abbreviated as Tools, is our 'primary category' because our research inclines toward it, as evident from our paper's title. The other categories (along with their abbreviated names, if any) are: Blockchain Security & Attacks (i.e., Blk. Sec. & Attacks), Testing, Solidity, Vulnerability, Non-Security Survey (i.e., Non-Sec. Survey), SWC (Software Weakness Scheme) and & DASP (Decentralized Application Security Project), and Security Survey (i.e., Sec. Survey).

Following are the reasons that forced us to expand our database (i.e., based upon MySQL tool and simply folders) of downloaded research papers once we met the count of the required tool-based research papers (i.e., at least one paper for each tool):

1) We focus on 20 vulnerabilities in our survey. Hence to clearly understand their dangerous behavior, we downloaded scores of research papers that we spread across the vulnerability categories, like, Security Survey, Blockchain Security & Attacks, and SWC and DASP.
2) Finally, the remaining documents, particularly the ones we incorporate in the Testing, Solidity, and Non-Security categories, help us to understand the software engineering and programming-based issues.

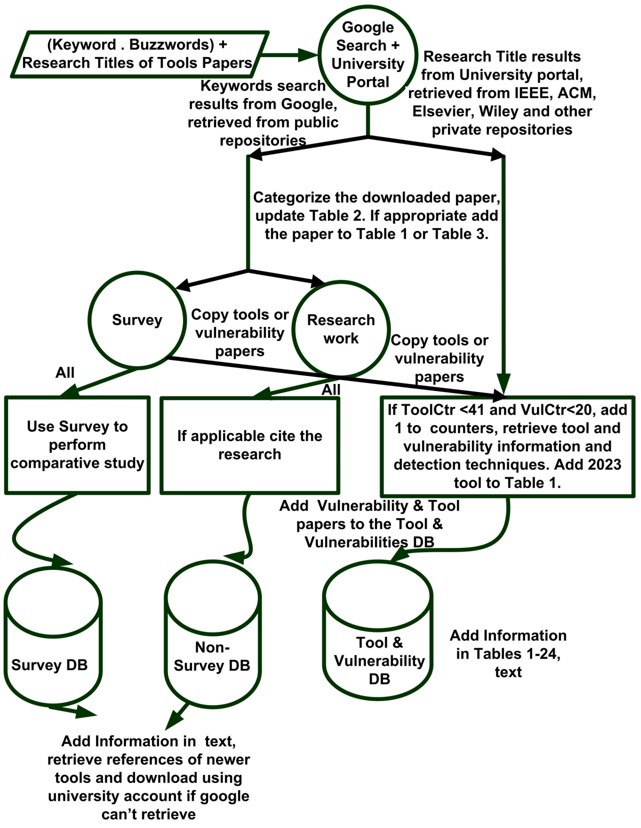

(Keyword . Buzzwords) + Research Titles of Tools Papers
Google Search + University Portal
Keywords search results from Google, retrieved from public repositories
Research Title results from University portal, retrieved from IEEE, ACM, Elsevier, Wiley and other private repositories
Categorize the downloaded paper, update Table 2. If appropriate add the paper to Table 1 or Table 3.
Survey
Copy tools or vulnerability papers
Research work
Copy tools or vulnerability papers
All
All
Use Survey to perform comparative study
If applicable cite the research
If ToolCtr <41 and VulCtr<20, add 1 to counters, retrieve tool and vulnerability information and detection techniques. Add 2023 tool to Table 1.
Add Vulnerability & Tool papers to the Tool & Vulnerabilities DB
Survey DB
Non-Survey DB
Tool & Vulnerability DB
Add Information in Tables 1-24, text
Add Information in text, retrieve references of newer tools and download using university account if google can't retrieve

FIGURE 1. Algorithm: Methodology and criteria for downloading research papers.

Inclusion and Exclusion Policy & Nearness Factor

Exclusion Policy: We do not incorporate non-CS issues in our research. But in the CS domain, we do not focus on non-Ethereum, non-Solidity, non-testing, and non-security-based issues.

Inclusion Policy: We incorporate software engineering-based issues and include research papers and articles providing definitions for some complex Blockchain terms and helping understand mathematical proofs and equations.

Nearness Factor: We define the nearness factor as the proximity of the downloaded and referenced research papers to our primary category as much as possible. This factor means that the primary class should lead other categories regarding the count of downloaded and referenced documents. In our case, this is true, because the 'Tools' category has the highest R to R and D to D ratio as evident from the data shown in Table 2.

C. RESEARCH QUESTIONS

Our research tries to find the answers to the following questions:
1) How do the workings of some vital, novel, and latest SC tools differ?
2) Which vulnerabilities (if any) are detected by the tools retrieved through research Q1 (i.e. RQ1)?
3) What are the vulnerability detection techniques (if any) applied by the selected tools through RQ1?
4) How can we classify the vulnerability detection techniques (if any) of the tools retrieved through RQ1?

D. KEY CONTRIBUTIONS OF THIS SURVEY PAPER

This paper is the first thorough survey on SCs based upon the following research perspectives enumerated below:
1) We collect the vulnerabilities, their associated SWC and DASP IDs, tools detecting them, and the detection technique of each tool in Table 4, 5, 6, 7. This information helps Ethereum researchers, software engineers, and security analysts. They can use the information to develop new tools by varying the techniques discussed in this research, supporting the growth of Solidity, EVM, and SCs.
2) The survey discusses the vulnerability detection techniques corresponding to different tools. We deduce four important facts (a) "reentrancy" is a highly researched threat; our research reports 19 tools currently available for its detection (b) SmartCheck, Tikhomirov et al. related research [52] provides the most significant number of vulnerability detection techniques (c) static analysis and AI-based approaches (or specifically the Solver-based approaches) are leading the domain of vulnerability detection of Ethereum SCs (d) Summarization of vulnerability detection techniques of 41 SC tools in the Tables 4, 5, 6, 7.
3) The survey also provides some insight to academicians, developers, and software engineers to design better SCs and testing tools by identifying vulnerable code.
4) To the best of our knowledge, we are the first to deduce the popularity factor of tools. This factor provides some clues to the highly referenced tools in the surveyed literature.
5) We showed the applicability of the Constructive ECF's immunity approach, discussed in Albert et al. [53] for reentrancy detection.
6) For the first time, we classified the vulnerability detection techniques into general and specific groups. The specific groups help to identify the novelty of tools. General groups utilize information related to software engineering, software security principles, and Ethereum modules, whereas specific groups focus on significant detection chores. This grouping provides a broader view of software engineering approaches to novice users.
7) To the best of our knowledge, we are the first to provide a classification for vulnerability detection techniques

incorporating around 41 tools. This classification can provide a standard mechanism for referring to vulnerability detection techniques.

8) Additionally, we collectively list all the techniques applied by the tools in a Multilevel classification table. The top-level classification dictates the novice software engineers that there are only two major testing approaches (i.e., static and dynamic analysis), and the rest of the approaches reside under them. This classification enables the researchers to know how to test the applications of newer CS domains.
9) We distinguish the EVM opcodes with Solidity statements (e.g., `send`) to better understand the text by using CAPITAL LETTERS for the opcode.
10) We create a nearness factor to enable the researchers to determine if the research has achieved its goals. This factor depends upon the downloaded papers and those referenced from the downloaded documents.

E. STRUCTURE OF THE SURVEY

Section II sheds some light on the advantages of categorizing vulnerabilities into six groups as in our earlier research [23]. Section III discusses the workings of 41 static and dynamic analysis tools. Section IV lists ten vulnerability detection classes, and we discuss their detection techniques from Section V to Section XIV in connection with several tools discussed in Section III. Section XV provides a multilevel description of detection techniques and lists them in a table. Section XVI embeds a thorough discussion on Future Work by dividing it into innovations related to vulnerability detection tools, SC testing, Solidity, and EVM. Section XVII incorporates the conclusion.

II. A CLASSIFICATION OF VULNERABILITIES IN SC: ADVANTAGES

Work in [23] discusses a domain knowledge-based categorization of SCs' vulnerabilities. The advantage of this classification is that the category's name gives us information about the cause of vulnerability. In the context of testing SCs for a specific vulnerability, our categorization helps to find a range of tools related to the vulnerability category. This fact implies that one should test SCs with a range of tools instead of one or two to ensure that the SC is not devoid of vulnerabilities. Table 3 lists examples of some of the classes of vulnerabilities identified in SCs in connection with [23]. A transfer of Ether between two SCs, for example, can result in inter-contractual vulnerabilities. Intra-Contractual vulnerabilities like selfdestruct, frozen Ether, and stealing Ether restrict to only a single SC. For instance, if there is a vulnerable `selfdestruct`, it destroys only that particular SC. The other classes are Integer Bugs (causes overflow, underflow, and other mathematical bugs), out-of-gas (which include gas-related vulnerabilities like out-of-gas, wallet griefing), transactional vulnerabilities (like `tx.origin`, transaction order dependence), and deprecated (like var, call stack depth). Interested readers may refer to [23] for detailed descriptions of the vulnerabilities. We did not cover DoS or DDoS vulnerabilities but found that work in [61] extensively covers DDoS-related aspects.

III. TOOLS FOR DETECTING VULNERABILITIES IN SC

We briefly describe the tools' working below and provide a classification of tools instead of vulnerabilities as in [23]. Table 4, 5, 6, 7 list the SC vulnerabilities along with the tools detecting them and the tool's detection technique. This research focuses on seven vulnerability categories retrieved from [23]. But we inducted a new access control category for `selfdestruct` vulnerability in this research as discussed in [62]. Table 4, 5, 6, 7 do not analyze the detection techniques by including the comments from the other research papers about the detection technique's effectiveness. However, Table 4, 5, 6, 7 provide SWC (Smart Contract Weakness Classification) [63] and DASP (Decentralized Application Security project) [64] related details. SWC and DASP are two important vulnerability classifications for SCs. SWC registry discusses 37 vulnerabilities, whereas DASP discusses the top 10 vulnerabilities.

Surveyed literature does not show any comprehensive representation of SCs' vulnerability detection techniques, as provided in the multilayer table in Figure 4. Some of the strategies mentioned by the tools help prevent intrusion. Still, we have included those tools in this research (Section III-A, and Section III-B) to give meat to future researchers so that the esteemed researcher can modify them for detection purposes. The detection techniques focus on several dynamic and static analysis tools. However, static and dynamic analysis tools can borrow each other's capabilities. For example, static analysis tools do not run the SC but retrieve and analyze the code patterns [65]. Still, to reason about the execution of SCs static tools use the symbolic analysis engine as discussed in Nikolić et al. [57]. Similarly, dynamic tools use keyword search, a static analysis strategy, to find the presence of some vulnerable opcodes.

A. STATIC ANALYSIS TOOLS FOR SMART CONTRACTS

Before discussing the vulnerability detection techniques, we provide below a brief discussion about the working and scope of SC tools based upon static analysis:

1) ZEUS, KALRA ET AL. [59], VDT SECTION (VIII-A1)

Zeus focuses on the Solidity-based SCs and a policy specification, a set of syntactical symbols to aid Zeus in avoiding confusion. It detects some of the Inter-Contract (like Unchecked and Failed `send`, reentrancy), Arithmetic (like overflow), and Transactional (like TimeStamp Dependence and Transaction Ordering Dependence) vulnerabilities. Zeus performs formal verification using abstract interpretation and symbolic model checking. Abstract interpretation is a program verification technique in which we define abstract states (like Even or Odd) using concrete values (Even = 2, 4, 6,…& Odd = 1, 3, 5,…) and then use concrete values to define the semantics of the program (like adding two odds

TABLE 3. A classification of SC's vulnerabilities [23].

Type	Vulnerability
Inter-Contract vulnerabilities	reentrancy, Rodler *et al.* [54] Denial of Service [55] Mishandled Exception, Wang *et al.* [37] Gasless `send`, Wang *et al.* [37] `call` transfers all gas (no more a vulnerability, transfers 63/64% gas EIP150) [56]
Intra-Contract vulnerabilities	Frozen Ether, Nikolić *et al.* [57] `selfdestruct` Chang *et al.* [58] Stealing Ether, Brent *et al.* [36]
Integer Bugs:Integer Overflow	Unchecked Maths Tikhomirov *et al.* [52]
Out-of-Gas	Denial of Service, Grech *et al.* [12] Integer Overflow Causing Out-of-Gas, Grech *et al.* [12] Wallet, Griefing Grech *et al.* [12]
Transactional vulnerabilities	Transactional Ordering Dependence (TOD), Kalra *et al.* [59] Timestamp Dependence (TSD), Luu *et al.* [60] `tx.origin`, Tikhomirov *et al.* [52]
Deprecated	call-depth-stack, Luu *et al.* [60] Type Inference (Integer overflow), Tikhomirov *et al.* [52] Type Inference (out-of-gas), Grech *et al.* [12]

values results in an even value) [66]. Symbolic Model Checking performs program verification automatically by using a finite state machine [67]. Zeus executes in a two-prong fashion, i.e., transforms both the Solidity code and the user policy (defined using mathematical assertions) into LLVM (Section V) bit code, IR (i.e., intermediate representation), and then feeds IR into CHC (i.e., Constrained Horn Clauses) engine. Constrained Horn Clauses help in automatic program verification and hence provide a means for supporting symbolic model checking [68].

2) VerX, PERMENEV ET AL. [69]

VerX does not detect vulnerabilities but helps program verification by combining the symbolic execution with reduced temporal properties (i.e., represented using a reduced set of temporal logic commands) and delayed abstraction (term not defined in VerX related research [69]). For testing, VerX inputs several SCs along with a temporal safety property. Temporal logic is time-dependent and uses operators such as 'always' and 'once' for processing inputted SCs. VerX uses temporal operators to extend Solidity code. Working starts by instrumenting the temporal property into SCs and extracting the related Solidity predicates. Finally, VerX transforms the temporal properties into reachability checks to perform reachability verification or provides a counter-example. In case of incomplete data, VerX demands more information. One example of requirement (to transfer funds) and the transformed temporal property in Solidity is given below:

Requirement: Investors cannot claim refunds after more than 10,000 ether is collected.

```
always(claimRefund()==> ! once(sum(deposits)>=
    10000))
```

Where sum(deposits) is a delayed abstraction containing the Solidity code:

```
deposits[p] = deposits[p] + msg.value;
```

Execution of delayed abstraction could indicate that VerX reaches the destination code [70].

3) VeriSolid, MAVRIDOU ET AL. [71]

VeriSolid does not detect vulnerabilities. Instead, it creates a new SC through formally verified safety properties that prevent reentrancy and DoS (Denial of Service) vulnerabilities. Input to the system consists of a graphical specification of SC using a transition system and its solidity code containing variables, actions, and guards, which we call (1). The system also inputs a set of properties for verification, which we call (2). VeriSolid generates the Behavioral Interaction Priority model using (1) and compares the generated model with the translated Computation Tree Logic retrieved from (2). Finally, it compares both outputs; if they are the same, the SC behavior is verifiable. The developers can generate SC code from the verified model.

4) VeriSmart, SO ET AL. [72]

VeriSmart is a tool that helps detect arithmetic vulnerabilities in SCs. The input to VeriSmart is the Solidity SC and the output returned by VeriSmart is the verification result of the assertion. VeriSmart automatically generates assertions for mathematical statements. For example, for the mathematical statement, `a+b`, the tool generates the assertion: `assert(a+b>=a)`. This feature empowers VeriSmart to retrieve the hidden invariants in mathematical statements of SCs. VeriSmart consists of two components: (1) a generator responsible for generating two types of invariants, i.e., loop invariants and transactional invariants. The generator receives back the unproven queries for refinement, (2) the validator is responsible for proving the queries and sends back the unproven to the generator. The validator starts by using the candidate invariant. If the invariant fails, the validator forwards the set of failed invariants to the generator, which refines the unproven invariant by generating a new one.

TABLE 4. Vulnerabilities: classification and detection techniques (Part 1).

Vulnerability Type	Vulnerability	Tool	Technique
Transactional	Transaction Ordering Dependence, SWC(114), DASP(7)	Oyente	Locking Reward Price
		DappGuard	High Gas Activity
		Zeus	Blockstate Variables Performing Write Operations and Ether Transfers
		Securify	`call` Reads Storage Data
	Timestamp Dependence, SWC(116), DASP(8)	Oyente	Using a Symbolic Variable for Path Condition
		Zeus	Ether Transfers Using Blockstate Variables
		DappGaurd	Oyente Engine
		SmartCheck	Detection of Variable `now`
		ContractFuzzer, ILF, SmartScopy	Ether Transfer Depends on TIMESTAMP or `block.timestamp`
	`tx.origin`, SWC(115), DASP(2)	Vandal, SmartCheck	Pattern Matching
	Event Order Bugs	EthRacer	Reordering Two Distinct Invocations Alters Output
Gas Related	Unbounded Mass Operations, SWC(128)	MadMax	Loop Analysis (Non-Resumable Loops)
	Wallet Griefing, SWC(126)	MadMax	Loop Analysis (Transfer Using `send` and Unsized Array within a Loop)
		SmartCheck	Spent Gas Exceeds `block.gaslimit`
		Zeus	Use of `assert` before `throw` (deprecated)
Inter-Contractual	Reentrancy, SWC(107), DASP(1)	DappGuard	Transaction Log File Analysis
		Clairvoyance	Cross SC call-chain
		Zeus, Vandal, Mythrill, Confuzzius, Slither	`call` Analysis &/or `assert`
		Constructive ECF	Simulating a Callback without a callback
		ReGuard	Finite State Machine
		FVF*	State Changes Violate Invariant
		sCompile	Limit Property
		Securify(1), SmartCheck(2)	(1)Semantic Pattern Matching:External Call Followed by Write, (2)Checks-Interactions-Effects
		Sereum	Locking Write Access to Storage Locations
		ITP	Analysis of Failure of Execution
		Vultron	Violation of Balance Invariant
		SmartScopy, ContractFuzzer, sFuzz	Synthesizing an Attacker
	DoS (unexpected `throw`), DASP(5)	SmartCheck	External call and `revert` not within if-condition-body
		VeriSolid	Checking Liveness Property
	Mishandled Exception, SWC(104), DASP(4)	DappGuard	Oyente Engine
		Vultron	Violation of Transaction Invariant
		Zeus	`assert` When Return Value of `send` Causes Inconsistency
		SmartCheck	External `call` not within an if Statement
		Vandal	CALL's Return Value Doesn't Prompt Throwing Exception or Updating Storage
		FVF*	Monadic Behavior & Proof Assistant
		Securify	CALL Doesn't Follow JUMPI
		Oyente	CALL Doesn't Follow ISZERO
		ContractFuzzer, ILF	Offshoot Caller Throws Exception
		SmartScopy	No Influence of CALL's Return Value on Conditional Jumps
		Confuzzius	Missing JUMPI after CALL
	Gasless `send`, SWC(134)	ContractFuzzer	Check: CALL's Return Value, CALL's Input, and GASLIMIT
		DappGuard	Ethereum Name Service
		Vultron	Violation of Transaction Invariant

VeriSmart detects proxyOverflow vulnerability, which is not part of our research.

5) VANDAL, BRENT ET AL. [36], VDT SECTION (VII-A1)
Vandal focuses on the bytecode of SC to create an intermediate relationship using a register transfer language. It detects some of the Inter-Contract (like Unchecked `send`), reentrancy) and Intra-Contract (like Unsecured balance or Stealing Ether, destroyable SC) vulnerabilities by applying AI-based techniques on intermediate representation. Vandal also generates CFG to perform symbolic analysis.

TABLE 5. Vulnerabilities: classification and detection techniques (Part 1-portion-1).

Vulnerability Type	Vulnerability	Tool	Technique
Transactional	Transaction Ordering Dependence, SWC(114), DASP(7)	Oyente	Locking Reward Price
		DappGuard	High Gas Activity
		Zeus	Blockstate Variables Performing Write Operations and Ether Transfers
		Securify	`call` Reads Storage Data
	Timestamp Dependence, SWC(116), DASP(8)	Oyente	Using a Symbolic Variable for Path Condition
		Zeus	Ether Transfers Using Blockstate Variables
		DappGaurd	Oyente Engine
		SmartCheck	Detection of Variable `now`
		ContractFuzzer, ILF, SmartScopy	Ether Transfer Depends on TIMESTAMP or `block.timestamp`
	`tx.origin`, SWC(115), DASP(2)	Vandal, SmartCheck	Pattern Matching
	Event Order Bugs	EthRacer	Reordering Two Distinct Invocations Alters Output
Gas Related	Unbounded Mass Operations, SWC(128)	MadMax	Loop Analysis (Non-Resumable Loops)
	Wallet Griefing, SWC(126)	MadMax	Loop Analysis (Transfer Using `send` and Unsized Array within a Loop)
		SmartCheck	Spent Gas Exceeds `block.gaslimit`
		Zeus	Use of `assert` before `throw` (deprecated)

TABLE 6. Vulnerabilities: classification and detection techniques (Part 1-portion-2).

Vulnerability Type	Vulnerability	Tool	Technique
Inter-Contractual	Reentrancy, SWC(107), DASP(1)	DappGuard	Transaction Log File Analysis
		Clairvoyance	Cross SC call-chain
		Zeus, Vandal, Mythrill, Confuzzius, Slither	`call` Analysis &/or `assert`
		Constructive ECF	Simulating a Callback without a callback
		ReGuard	Finite State Machine
		FVF*	State Changes Violate Invariant
		sCompile	Limit Property
		Securify(1), SmartCheck(2)	(1)Semantic Pattern Matching:External Call Followed by Write, (2)Checks-Interactions-Effects
		Sereum	Locking Write Access to Storage Locations
		ITP	Analysis of Failure of Execution
		Vultron	Violation of Balance Invariant
		SmartScopy, ContractFuzzer, sFuzz	Synthesizing an Attacker
	DoS (unexpected `throw`), DASP(5)	SmartCheck	External call and `revert` not within if-condition-body
		VeriSolid	Checking Liveness Property
	Mishandled Exception, SWC(104), DASP(4)	DappGuard	Oyente Engine
		Vultron	Violation of Transaction Invariant
		Zeus	`assert` When Return Value of `send` Causes Inconsistency
		SmartCheck	External `call` not within an if Statement
		Vandal	CALL's Return Value Doesn't Prompt Throwing Exception or Updating Storage
		FVF*	Monadic Behavior & Proof Assistant
		Securify	CALL Doesn't Follow JUMPI
		Oyente	CALL Doesn't Follow ISZERO
		ContractFuzzer, ILF	Offshoot Caller Throws Exception
		SmartScopy	No Influence of CALL's Return Value on Conditional Jumps
		Confuzzius	Missing JUMPI after CALL
	Gasless `send`, SWC(134)	ContractFuzzer	Check: CALL's Return Value, CALL's Input, and GASLIMIT
		DappGuard	Ethereum Name Service
		Vultron	Violation of Transaction Invariant

6) teEther, [56], VDT SECTION (VII-A4)

Teether detects `selfdestruct` vulnerability, and four others (i.e., Erroneous visibility, Erroneous constructor, Semantic confusion, and Logic Flaw) by using a combination of CFG, symbolic execution, and constraint-solving approaches. First, Teether constructs the CFG of the SC

TABLE 7. Vulnerabilities: classification and detection techniques (Part 2).

Vulnerability Type	Vulnerability	Tool	Technique
Contractual	Frozen Ether	SmartCheck	`payable` Ether Receiving Function but no Ether Transfer
		Maian	Ether Transfer Statements Unreachable and the SC Accepts Ether
		ContractFuzzer	SC Uses DELEGATECALL, Lacks CALL, SUICIDE
		sCompile	Path Analysis
		Confuzzius, Securify, ILF	Lacks Ether Transfer Instruction but Receives Ether
	Malicious Library	SmartCheck	Searches Library Keyword
Access Control	SelfDestructible, SWC(106), DASP(2)	Teether, Maian, sCompile, Mythril	Reachability Analysis &/or Constraints Checking. Maian also checks if the bytecode is reset to '0x'
		Vandal, ILF	UnAuthorized Access
		ITP	Analysis of Failure of Execution
		Ethainter	(1)Executing Function Holding SelfDestruct Destroys SC (2)Use Composite Attacks to Execute SelfDestruct
		Confuzzius	Malicious Account
	Stealing Ether, SWC(105)	Vandal	Amount of CALL non-Zero & Destination Address Suspicious
		Securify	Non-constructor Modifies Storage
		Maian	Amount of Transfer Non-zero & Transfer Address Never Sent Ether to Sender
		Confuzzius	Suspicious Address Related to CALL
		ILF	Transferrable Ownership
Arithmetic	Integer Overflow, SWC(101), DASP(3)	EasyFlow	Executes Transactions for all Possible Integer Values
		Vultron	Violation of Balance Invariant
		OSIRIS	Solver Checks Boundary Requirements and Divisor
		SmartCheck	Arithmetic Operator not inside the Conditional Statement
		Zeus	Presence of Wrap Around Errors
		Confuzzius	Disparity b/w Actual Results and Values Pushed on Stack
	BatchOverflow, DASP(3)	SmartScopy	Synthesizes an Attacker, CALL's Value and Address Controlled by Attacker
		EasyFlow	Checks IF More than One Multiplier Tainted
	Integer Division	SmartCheck	Pattern Matching: Division Operator (/)
Deprecated	Call Stack Depth	Oyente	`send` Fails Regardless of Gas Possessed by SC Detecting Call Stack Size
	Type Inference (Integer Overflow), SWC(111)	SmartCheck	Pattern Matching: Assignment Operator, Left Hand Side is `var` and Right Hand Side is Integer (0..9)
		MadMax	Loop Analysis
	Call, all gas transferred (Table 3)	sCompile	Reachability and Path Analysis
		SmartCheck, Securify	Pattern Matching: `call`, `call` with non-Zero Transfer Amount and Execution is Independent of Sender

using the EVM bytecode. Then, three modules process the CFG. The path generation module detects critical paths and state-changing paths. Critical paths in Teether [56] research mean the paths that can lead to the execution of CALL, CALLCODE, DELEGATECALL, and SELFDESTRUCT instructions. This is different from the critical path (i.e., only money-related paths) discussed in Chang et al. [58]. Next, the constraint generation module uses the Z3 symbolic engine to compute path constraints related to opcodes such as CALL, CALLCODE, DELEGATECALL, and SELFDESTRUCT. Then, the exploit generation module generates the exploits by solving the combined constraints of critical paths and state-changing paths. Finally, Teether validates the exploits.

7) SmartScopy, FENG ET AL. [31], VDT SECTION (VI-A2)

SmartScopy innovates summary-based symbolic evaluation to speed up symbolic execution and focuses on the Application Binary Interface (ABI) and the bytecode of the victim's SC. SmartScopy detects vulnerabilities like reentrancy, timestamp dependence, gasless `send`, and batch overflow by synthesizing adversarial SCs. The tool first develops an adversarial SC using the victim's Application Binary Interface and bytecode. Application Binary Interface retrieves the signatures of the public methods of the victim SC. SmartScopy uses Vandal's decompiler to translate the bytecode of the victim SC into SmartScopy's language for adversarial attacks. The attacker's program consists of small queries. Hence the approach is called summary-based symbolic execution. The attacker SC confirms the summary step by recording side-effects on storage and the global state of Blockchain by running the transactions of public methods of SC on EVM. The summary step also generates summary-based CALL and summary-based SSTORE.

8) SmartCheck, TIKHOMIROV ET AL. [52], VDT SECTION (IX-A1)

Smartcheck analyzes Solidity SCs by generating an XML parse tree as an intermediate representation of solidity SC. The tool then performs pattern matching using Xpath[xpa] queries. Thus the tool takes advantage of using XML documents as a database. XPath is a query-based language, and xpa is an indexing mechanism for reasoning XPath queries [73]. The tool detects the most significant number of vulnerabilities with a complete description of its techniques like reentrancy, `tx.origin`, and so on.

9) SLITHER, FEIST ET AL. [74], VDT SECTION (X-A1)

Slither recovers essential information such as the inheritance graph, control flow graph, and the programming expressions from the Abstract Syntax Tree and converts this information into an intermediate representation. Abstract Syntax Tree helps to represent a program using a tree. Slither's intermediate representation, SlitherIR, is available in static single assignment form, which analyzes read/write variables, data dependency, and unprotected functions. The static single assignment requires a one-time variable assignment and definition before usage. Finally, the built-in code analysis using taint tracking helps in the following ways:

a) the read/write state variables (i.e., same as Java's instance variables) help in reentrancy detection.

b) owner address variables can perform privilege operations.

c) unprotected functions do not use `msg.sender` for comparison.

d) shadowed variables may contain unexpected data.

Slither detects reentrancy vulnerability.

10) SECURIFY, TSANKOV ET AL. [75], VDT (SECTION VII-A2)

Securify inputs bytecode of SC, or Solidity SC and converts SC into bytecode, along with a set of security patterns. Securify performs a data and control flow dependency analysis. The derived dependency predicates formulate characterizations of security patterns using an iterative process. Securify infers the facts about the pattern using the Souffle Datalog solver. Securify uses both compliance and violation patterns. If it detects a violation pattern, then it reports vulnerable instructions. If no pattern matches, then it notifies a warning. However, Securify's pattern-matching approach is different from the syntactical query-based patterns employed by Smartcheck (Section III-A8 and IX-A1). Securify detects vulnerabilities like stealing and frozen Ether, reentrancy, mishandled exception, and Transaction Ordering Dependence. Securify has an updated version Securify2 [76], which detects 37 vulnerabilities.

11) sCompile, CHANG ET AL. [58], VDT SECTION (XI-A2)

sCompile focuses only on money-related paths designated as critical paths. sCompile first converts the inputted bytecode to EVM opcode, which helps to get CFG, and CFG helps detect critical paths that correspond only to the opcode of either a CALL, a DELEGATECALL, or a SELFDESTRUCT statement. Once the tool finds the critical paths, it computes the criticalness (or criticality) of each path, known as the rank of the path. Path ranking helps the Z3 SMT solver to filter out less critical paths. Finally, the visualization component displays the critical paths for the user's inspection. sCompile detects vulnerabilities like reentrancy, be no black hole (i.e., Locked Ether) and unguarded self-destruct.

12) OYENTE, LUU ET AL. [60], VDT SECTION (VI-A1)

Oyente processes the SC's bytecode along with Ethereum's global state to detect vulnerabilities like Transaction Ordering Dependence, TimeStamp Dependence, Mishandled Exception, and Call Depth Stack. First, the tool's CFGBuilder constructs the CFG, which the Explorer module uses to find symbolic traces for each feasible path. The CoreAnalysis module uses Explorer's output to detect vulnerabilities. Finally, the Validator module confirms the results using the Z3 SMT solver's constraint module.

13) OSIRIS, TORRES ET AL. [77], VDT SECTION (X-A2)

OSIRIS applies symbolic analysis to SC's bytecode. The symbolic analysis component creates the CFG and then forwards all the executed instructions to the taint analysis component. The integer error detection component is called only if the instruction is arithmetic. The Taint Analysis Module uses the concept of source and sink and declares a valid bug if the inputted data propagates from an untrusted source (i.e., Taint Source) to a sensitive location (i.e., Taint Sink) [78] using SSTORE, JUMPI, CALL, and RETURN instructions. On the other hand, the Integer Detection Module only scans integer-related operations and employs the Z3 solver if the Taint Analysis Module declares the arguments of integer operations untrusted. Osiris detects arithmetic bugs (overflow/underflow and division by 0), truncation bugs (converting from a larger to smaller data size, e.g., 64-bit data to 16-bit data), and signedness bugs (converting a signed integer to an unsigned integer of the same width and vice versa).

14) MSmart, FEI ET AL. [24]

MSmart uses SmartCheck's detection framework to convert Solidity SCs into path diagrams using lexical and syntactic analysis. Lexical analysis converts a high-level program into tokens where each token is a sequence of characters. Syntactic analysis (also known as parsing) analyzes the string of tokens if they conform to formal grammar rules. Path diagrams use syntactical analysis to create XML parse tree-like structures that incorporate rules. An example of rules specified in XPATH is discussed in [79]. Path diagrams are then subject to path matching, which helps in vulnerability detection. Path matching uses an expression tree (i.e., XML parse tree) and then adds existing SmartCheck's rules (and new rules formulated by the authors of MSmart). The evaluation of the SC's expression using path matching tells if SC is vulnerable. XML parse tree acts as an intermediate representation for SCs.

15) MAIAN, NIKOLIĆ ET AL. [57], VDT SECTION (XIV-A1)

Maian inputs the bytecode of SC and its state from the Blockchain. The state represents a sequence of invocations, and the authors referred to it as an execution trace.

Maian detects SCs specified as greedy (i.e., locking fund behavior, prodigal (i.e., leaking fund behavior), and suicidal (i.e., self-destructive behavior). The detection process employs inter-procedural symbolic analysis and concrete validation. The interprocedural calls transfer control only to the target within the SC. For implementing symbolic analysis, the authors create a custom EVM to run all traces using symbolic values until the EVM developed by the authors finds a trace violating liveness and safety properties. The custom EVM returns concrete values. The concrete validator uses the values returned by symbolic analysis to determine if the SC is true positive or false positive in exhibiting the violation.

16) MadMax, GRECH ET AL. [12], VDT SECTION (VII-A3)

MadMax uses Vandal's decompiler to transform the EVM bytecode into a structured logic-based intermediate representation. For analysis purposes, MadMax uses the Datalog engine. As a result, MadMax performs both control flow and declarative program analysis. Vandal's decompiler outputs 3-address relations, which helps MadMax (Souffle engine component) generate C++ code for high-level program analysis of loops. MadMax defines strategies for detecting unbounded mass operations, Wallet griefing and integer overflow situations using Datalog rules.

17) KEVM, HILDENBRANDT ET AL. [80]

KEVM framework does not detect vulnerabilities, but its program verifier performs two important tasks (1) verifies the functional correctness of SCs, useful in the context of integer overflow (2) and handles the gas analysis of SCs. The prover performs symbolic analysis by inputting the K-framework's definitions and logical reachability claims. The gas analyzer computes gas bounds during execution and thus can help detect gas-related vulnerabilities.

18) IR-FUZZ, LIU ET AL. [26]

IR-Fuzz's approach prioritizes writing over reading. For instance, a function like `donate()` executes before `withdraw()`. IR-fuzz uses this approach to generate possible SC's function invocation sequences. Later, IR-fuzz extends the invocation sequence using a new set of parameters. Finally, IR-fuzz derives fuzzing input by seed selection and stores the test cases covering a new branch (note, recording helps reveal simple branches). In addition, IR-fuzz proposes a branch distance-based measure to select new test cases that comply with complex branches.

19) INTERACTIVE THEOREM PROVERS (ITPs), [81], VDT SECTION (VIII-A2)

By the term "interactive theorem provers" the author refers to a whole class of general-purpose tools like LEM, Mulligan et al. [82], OCAML [83], and Isabelle/HOL [84], focusing on EVM bytecode for SC's formal verification. For instance, LEM is a mathematical tool for writing, publishing, and managing portable semantic definitions, and the author initially transforms the EVM bytecode into LEM. Then both the community-based test suits and interactive theorem provers like Isabelle/HOL and OCAML test their EVM definitions to prove the desired properties of SCs related to reentrancy attack and self-destruct vulnerability. Isabelle is a proof assistant for proving mathematical formulas using logical calculus, whereas OCAML is a general-purpose functional programming language.

20) HORSTIFY, HOLLER ET AL. [25]

Horstify is a formal proof framework that generates logical proof for program slices (i.e., code related to SCs) and then uses the Datalog solver for verification. This tool can act as additional information for readers focusing on the formal verification of SCs. Horstify uses its analysis for timestamp dependence and discovers program slices by matching security patterns with SC code.

21) GASPER, CHEN ET AL. [85], VDT SECTION (VI-A3)

Gasper performs symbolic execution on Oyente-generated CFG and identifies the code for optimization, Marescotti et al. [86]. During symbolic execution, the tool uses the Z3 solver to handle branch ambiguities and identifies seven gas-costly patterns (i.e., we designate them as Gasper patterns). However, based upon the 0.6.1 solidity compiler, we could not detect high-cost coding patterns related to dead code and opaque predicate as discussed in Gasper-related research. One reason might be the lack of compiler details in Chen et al. [85].

22) FSolidM, [55]

FsolidM is not a vulnerability detection tool. FSolidM uses graphics and text input and incorporates a Solidity parser for syntax checking. In the case of graphical input, FSolidM would generate the Solidity code. FSolidM introduces a set of plugins and design patterns which the developers can add to the SC for implementing locking, maintaining transaction counter, and enforcing timed transitions to safeguard against reentrancy, transaction ordering, and time constraint bugs respectively.

23) FORMAL VERIFICATION USING F* (FVF*) TOOL, BHARGAVAN ET AL. [51], VDT SECTION (VIII-A3)

FVF* tool employs the Solidity code and EVM bytecode to perform formal verification using F*. Solidity* (a subset of F*) produces the verified output of Solidity code, and EVM* (also a subset of F*) produces the verified output of EVM bytecode. The tool compares both the results for Functional Correctness (i.e., unchecked `send`), and Runtime Safety (i.e., reentrancy) but recommends manual proof.

24) ExGen, JIN ET AL. [27]

ExGen is a symbolic analysis tool and can be helpful for additional reading related to Symbolic Execution-based tools, Section VI-A. ExGen detects vulnerabilities like overflow, underflow, reentrancy, `selfdestruct`, and `delegatecall`. In `delegatecall` the called SC has access to the calling SC's state variable. A SC, with no Ether transfer function, uses `delegateCall` to let other SC execute Ether transfer function on its behalf. ExGen first generates an ordered sequence of transactions and transforms them into an attack SC. The next step executes the attack SC with the target SC using symbolic and concrete values retrieved from the public Blockchain. Finally, the symbolic execution retrieves constraints. The solution to the constraints using a solver allows Exgen to access the vulnerable function. Exgen deploys the attack SC on the BlockChain to verify its functionality.

25) EtherTrust, GRISHCHENKO ET AL. [87], VDT SECTION (VII-A5)

EtherTrust applies the reachability property on SC's bytecode to detect a FF responsible for a DAO-like attack and miner-controlled parameters responsible for transaction disorders. EtherTurst converts the SC's bytecode into Horn Clause (Section III-A1) representation and passes it to the SMT solver. Results thus obtained help EtherTrust to reason about the reachability property for verifying single-entrancy and independence of miners. However, the analysis aborts if the SC encases CALLCODE or DELEGATECALL instructions. Single-entrancy is a safety property that rules out vulnerabilities due to unintended callbacks (i.e., unintended invocation from FF) like DAO. Also, single-entrancy and independence of miner is not part of our vulnerability detection techniques.

26) Ethainter, BRENT ET AL. [5], VDT SECTION (VII-A6)

Ethainter uses information flow rules to detect composite vulnerabilities like accessible SELFDESTRUCT (i.e., unguarded SELFDESTRUCT) and tainted SELFDESTRUCT. In tainted one, the SC protects SELFDESTRUCT with a vulnerable guard condition. For example: `EasyToKill()` in Listing 1, line#2, uses `owner` variable

```
1  function initOwner(address _owner) public { owner
      = _owner;}
2  function EasyTokill() public {
3     if (msg.sender == owner)
4     { /* sensitive code like selfdestruct or Ether
          transfer statement */ ... }}
```

LISTING 1. Example of information flow rules, modified from He et al. [88].

initialized in a non-constructor to authenticate the sender. Henceforth attacker gets access to SC's sensitive parts.

Information flow rules deal with programmatic guarding (e.g., guards based on `msg.sender`) to prevent the flow of tainted input.

Ethainter decompiles the SC's bytecode using the Gigahorse, Grech et al. [89] toolchain to obtain a Datalog representation of the information flow model. The Datalog representation is a collection of declarative rules and uses the Souffle Datalog engine to generate C++ code from Datalog input relations. Ethainter uses C++ code to flag vulnerabilities, and Ethainter-kill, a companion tool of Ethainter, kills the vulnerable SC to confirm the vulnerability.

27) DappGuard, COOK ET AL. [90], VDT SECTION (VII-A7)

DappGuard detects diverse VULNERABILITIES in Solidity SCs. Additionally, DappGuard focuses on faulty transaction receipts. DappGuard operates in two modes. The Knowledge Acquisition mode uses receipts to analyze live SCs and declare them malicious based on burst gas usage in a short period. The Active Monitoring and Detection mode uses the Oyente engine to discover vulnerabilities like reentracy, transaction ordering, timestamp dependence and mishandled exceptions. The authors do not provide the source for retrieving transaction receipts.

28) CONSTRUCTIVE ECF (CECF) ANALYSIS, ALBERT ET AL. [53], VDT SECTION (VII-A8)

This work ensures that the behavior of the SC does not change in the presence of callback, motivated by effectively callback-free (ECF) concept, Grossman et al. [38]. Henceforth the callback behavior ensures modularity. The authors create an intermediate representation using the bytecode of SC and applied SMT solver, such as Z3, which queries for modularity checking by dividing the program into segments. SMT solvers generate counter examples of modularity for error detection and fixing callback violations effectively. The danger of callbacks results in stealing Ether and reentrancy vulnerability, and their approach focuses on detecting immunity to reentrancy attacks. However, we discuss how their approach can detect reentrancy.

29) CLAIRVOYANCE, XUE ET AL. [29], VDT SECTION (VI-A4)

Clairvoyance is a reentrancy detection tool and uses Solidity SCs related to the caller and callee SCs [91] to perform static analysis based on a cross-contract call chain (i.e., from one SC to the other SC) for reentrancy detection. The retrieval of cross-contract information reduces false-negative cases by realizing the complete cross-contract call chain. Additionally, Clairvoyance incorporates five path defensive techniques to avoid reentrancy. For instance, the code in Listing 2 assigns the conditional test variable (i.e., reentered) true at line#5 (i.e., locking) and false at line#7 (i.e., unlocking), restraining the reentrancy cycle.

```
1  function receiveDividends () public payable {
2     if (! reEntered ) {
3        ...
4        if ( ActualBalance > 0.01 ether ) {
5           reEntered = true ;
6           ZTHTKN . buyAndSetDivPercentage . value
                ( ActualBalance)( address (0 x0),
              33, "");
7           reEntered = false ; }
8     }
9  }
```

LISTING 2. Highligthing path protective technique, Xue et al. [29].

Clairvoyance detects this locking and unlocking limited to a function definition and does not report reentrancy, outperforming Securify and Slither. The work starts by creating CFG and a call graph for inputted SCs. The next step uses Taint Analysis to identify vulnerable objects, addresses, and call chains. After that, Clairvoyance uses path-defensive techniques for Empirical studies to avoid false-positive cases. The last step is eliminating paths that can't cause reentrancy from SC and output the remaining reachable paths.

30) BRAN, [92]

Bran combines grey box fuzzing capabilities with online static analysis to check the functional correctness of SCs but does not detect any vulnerability. It works directly on the bytecode and seed inputs. Bran first finds no-target-ahead prefixes, termed as look-ahead analysis. Then, the results from the look-ahead exploration feed into a power schedule that determines the fuzzing schedule and guides in locating the target. We provide an example modified from [92] to illustrate the reachability problem associated with grey box fuzzing (i.e., to reach assertion statement, i.e., line#3 in Listing 3 using constant-propagation analysis, i.e., using constants instead of variables) in the following code:

```
1  uint a;
2  ret = 3*a*a + 7*a + 101;
3  assert (ret != 5687) ;
```

LISTING 3. Example of reachability problem analysis, [92].

The `assert` statement is reachable when `a` has a value 42 in line#2, [93]. The grey box fuzzer's probability of generating 42 out of 2^{256} values is minimal in the above code. On the other hand, the power schedule component of Bran assigns more energy (i.e., more iterations) to inputs to allow them to bombard at the target location to generate 42.

B. SMART CONTRACT DYNAMIC ANALYSIS TOOLS

Now we will provide a brief description of dynamic Analysis SC tools:

1) VULTRON, WANG ET AL. [37], VDT SECTION (XII-A1)

Authors use Truffle suite to test Vultron's prototype version to identify vulnerabilities like reentrancy, exception disorder, integer overflow/underflow, and gasless send. Vultron detects irregular transactions by observing changes in the balance invariant. In other words, Vultron requires that the difference between the SC's account balance (i.e., `bal`) and the sum of all participants' bookkeeping balances e.g., `balances` array (Listing# 4, line#4), should be constant, before and after each transaction. Note, Solidity SCs' have no built-in variable `bal` [94]. The tool either inputs bookkeeping information or retrieves it using taint analysis.

2) SEREUM, RODLER ET AL. [54], VDT SECTION (XIII-A1)

Sereum works at the bytecode level and detects different variations of reentrancy attacks. Sereum adds the Dynamic Taint Analysis module and Attack detector to the go-Ethereum

```
1  contract TestBal {
2  mapping (address => uint) public balances;
3  function bookKeepArr(address to) {
4     balances[to] += msg.value;
5  }
6  :
```

LISTING 4. Example of array for balance bookkeeping, [37].

client. Dynamic Taint analysis monitors the data at execution time, Alashjaee et al. [95]. Sereum detects state updates that cause inconsistency by implementing a set of locks. The updating violation helps Sereum to reason about reentrancy.

3) ReGuard, LIU ET AL. [10], VDT SECTION (V-A5)

Reguard detects reentrancy bugs. The tool employs a Fuzzing Engine to generate random transactions after the SC Transformer creates a C++ SC using the inputted source code or bytecode of the SC. Finally, Reguard forwards the SC's execution of the transactions, mainly function calls and access to the storage to the Core Detector for reentrancy detection.

4) EasyFlow, GAO ET AL. [99], VDT SECTION (XIV-A2)

EasyFlow detects overflow in deployed SC by employing taint analysis. EasyFlow's input data consists of Ethereum runtime bytecode, transaction data, Solidity source code of SCs, and a JSON-format file containing state information. The detection algorithm analyzes transactions and mathematical instructions by executing them using different data combinations.

EasyFlow consists of four components: (i) Extended Go-Ethereum traces tainted data, (ii) Log analyzer analyzes the Go-Ethereum logs, like overflow detection logs, and sending various overflow transactions to the transaction constructor, (iii) Transaction constructor constructs the new transactions using input data so that Extended Go-Ethereum executes these transactions to determine potential overflows, (iv) Report generator creates a brief report. EasyFlow divides overflow into: manifested overflow, protected overflow, potential overflow triggered, and potential overflow not triggered. However, authors in [99] do not provide examples for differentiating them.

5) CONFUZZIUS, TORRES ET AL. [96], VDT SECTION (V-A1)

Confuzzius tests SC using its source code and Blockchain state. Source code generates the application binary interface (ABI) and bytecode. ABI allows EVM to understand the code. Confuzzius relies on an evolutionary fuzzing engine and genetic programming to generate an initial population based on SC's ABI. The genetic algorithm helps the fuzzer to generate uncommon input [100]. The evolutionary engine uses the input to execute the SC in the EVM environment. Both genetic algorithm and evolutionary fuzzing are fields of AI (artificial intelligence) and mimic Darwin's theory of Evolution. The genetic algorithm is a component of the evolutionary fuzzer. The genetic algorithm is a continuous process of enhancing the population of existing individual solutions

TABLE 8. The vulnerabilities detected by fuzzing tools developed for SCs.

Vulnerability	Confuzzius [96]	ContractFuzzer [97]	EthRacer [30]	ILF [88]	ReGuard [10]	sFuzz [98]
Reentrancy	✓	✓			✓	✓
Mishandled/Unhandled	✓	✓		✓		
Gasless	✓					✓
Unprotected SELFDESTRUCT	✓			✓		
Leaking Ether	✓			✓		
Frozen Ether	✓	✓		✓		
Integer Overflow	✓					
Timestamp Dependence	✓			✓		
Event Ordering Bugs			✓			

by selecting unique solutions in the current population to act as parents and produce children for the next generation.

Symbolic taint analysis, data dependency analysis, and constraint solver process the generated results for feeding them back to the evolutionary fuzzer. This process continues for several generations or the time allocated for processing elapses. Confuzzius is a hybrid fuzzer. When the traditional fuzzer cannot cover branches, Confuzzius switches to symbolic execution to solve the complex branch condition and then reverts to fuzzing. Confuzzius detects vulnerabilities like assertion failure, block dependency, integer overflow, reentrancy, transaction order dependency, unhandled exception, leaking Ether, locking ether, unprotected `selfdestruct`, and unsafe `delegatecall`.

6) sFuzz, NGUYEN ET AL. [98], VDT SECTION (V-A6)

sFuzz inputs the bytecode and the ABI of the SC and detects several vulnerabilities but we focus on reentrancy, gasless `send`, and freezing Ether. The tool has three components: the runner performs overall management of test cases by generating externally owned accounts, commands to analyze the ABI and the options for the other two components listed below: (i) liboracles monitors the execution of test cases for the detection of vulnerabilities by creating events containing read-only values for the program's stack and memory data structures, and (ii) libfuzzer generates the test cases by using the feedback guided Adaptive Fuzzing algorithm.

sFuzz's technique is adaptive because sFuzz changes the objective function to maximize the count of vulnerabilities detected in the tested program [101]. The algorithm uses quantitative measures for covering the just missed branches, as in the case of the following branch:

```
1  if( responseHash == keccak256 ( _response ) &&
2  msg. value == 100 finney ) {
3     msg.sender.send (this.balance); }
```

Due to condition#2 in the above quiz game SC, retrieved from Nguyen et al. [98], sFuzz fails to detect the gasless `send` vulnerability in Line#3 when sFuzz uses the traditional fuzzing techniques. On the other hand, condition#1 represents the correct answer to a quiz question, but the && operator requires sFuzz to pass both conditional tests to evaluate the branch. sFuzz solves the problem of testing the vulnerable branch by using quantitative measures. sFuzz keeps calculating a numerical value for `msg.value` such that `msg.value` is closer to `100` in an iterative manner and ultimately uses `100` as the value for `msg.value` in 140th iteration, which avoids checking 2^{256} values because `msg.value` has a size of 32 bytes.

7) IMITATION-BASED LEARNING FUZZER (ILF), HE ET AL. [88], VDT SECTION (V-A4)

Imitation-based Learning (ILF) Fuzzer empowers by incorporating symbolic analysis to generate effective inputs using the CFG of SCs (training set). The inputs help in code coverage and analyzing the security of a SC. For instance, in the following code, i.e., Listing 5, the assignment in Line#3 is trustable due to the presence of a guard in Line#2. ILF transform the secured functions into transactions (identified by function signature, address of the initiator, and Ether) and then serve as training sequences for the Neural Networks (NNs). The trained NN then helps improve the coverage of the Fuzzer during testing a SC because it can generate inputs for unseen SCs. ILF detects vulnerabilities like locking or freezing Ether, leaking or stealing Ether, suicidal, block dependency, unhandled exception, and controlled `delegateCall`.

```
1  function setOwner (address newOwner) public {
2     require (msg.sender == owner);
3     owner = newOwner;}
```

LISTING 5. Shows a trustable assignment modified from He et al. [88].

8) ContractFuzzer, JIANG ET AL. [97], VDT SECTION (V-A2)

ContractFuzzer's input consists of SC's bytecode, ABI, and the constructor arguments. It first performs static analysis on the ABI and the bytecode. The output focuses on function signatures, the data types of each argument, and the subset of SCs for interaction. The next step generates valid fuzzing inputs and their bombardment on random function invocations of a SC under test. After that, the tool detects vulnerabilities like gasless send, reentrancy, exception disorder, timestamp dependence, blocknumber dependence, dangerous delegate call, freezing Ether by applying specific oracles (i.e., modules to connect SC with external systems) to generate random invocations of SC under test. ContractFuzzer collets the invocation output as execution logs and analyzes it for vulnerability detection.

9) EthRacer, KOLLURI ET AL. [30], VDT SECTION (V-A3)

EthRacer focuses on partial-order reduction techniques to flag event ordering bugs (discussed below) using happens-before relations. According to the happens-before relationship, if one event should happen before the other, the result must reflect that, even if the events happen randomly [102]. For instance, in the case of banking transactions, the happens-before relationship is essential if an overdraft causes account cancellation [103], [104]. Ethracer either inputs SC's byte code or its Ethereum address. It runs the bytecode directly by combining symbolic analysis with fuzzing events on EVM. First, EthRacer uses symbolic execution to detect public functions termed as events. Next, EthRacer analyzes the output of the events using dynamic symbolic analysis. The dynamic analysis returns SMT formulae for each event, which helps determine concrete values. EthRacer determines concrete values using two approaches. The first one applies SMT solver, and the output obtained by executing SCs under test on Blockchain, and helps in pruning optimizations. The 2nd one uses fuzzing. If two different execution traces (i.e., the bytecode of a path generated by a set of invocations) of the concretely tested events (e.g., function invocations) generate different outputs, EthRacer declares the SC vulnerable to event-order bugs.

10) MYTHRIL [105], VDT SECTION (XI-A1)

Mythril performs symbolic analysis using Laser-Ethereum [106]) (a symbolic virtual machine), Concolic analysis using symbolic and concrete values respectively, and control flow checking on EVM bytecode to detect vulnerabilities (like reentrancy, mishandled exception, selfdestruct, integer overflow or underflow, and transaction order dependence). Laser executes different types of calls associated with symbolic and concrete data, e.g., Laser runs Ethereum SCs and supports inter-SC calls.

11) TechyTech [28]

The tool uses the accounting system of truffle suite. When a SC contains an unprotected `selfdestruct`, then the execution of SC results in destroying the SC as well as transferring its remaining Ether to a designated Truffle account. TechyTech then uses assertions to confirm this terminating transfer. Similarly, TechyTech also discovers reentrancy and `tx.origin` vulnerabilities [107] using an involuntary transfer-based technique. TechyTech develops a categorization string to distinguish between `tx.origin` and reentrancy vulnerabilities. TechyTech applies inbound but no outbound transfer approach for detecting locked Ether vulnerability [108].

IV. VULNERABILITY DETECTION CLASSES

In the following sections, we classify and review the vulnerability detection techniques by different tools. We group the detection techniques into General and Specific approaches. In addition to the above grouping, we provide two tables for each general class. The vulnerability table discusses the vulnerabilities detected by each tool in the general class. The general information table discusses the common attributes of the tools like citations, publication year, analysis type ('S' for static and 'D' for dynamic), and so on. Below we discuss the need for this classification:

a: GENERAL AND SPECIFIC CLASSIFICATION

SC tools can focus on traditional and domain-specific approaches. SC tools borrow traditional methods from other vulnerability detection tools developed in Computer Science. These tools use various common software engineering techniques such as symbolic analysis, formal verification, model-checking, fuzzing, etc. The above techniques are conversant with computer science tools working for non-BlockChain programs and SCs alike, and we refer to them as general approaches in our research. The general approaches are standard analysis methods to start processing or facilitate processing to determine if the attacker secures its Ether transfers [33]. Hence they need Blockchain environment or BlockChain-specific approaches to detect malicious Ether transfer issues.

These specific approaches come into play when SC tools face Blockchain-specific challenges like data storage and validation (use of consensus mechanisms), VMExceptions (use of try-catch block), compatibility with Solidity versions (use of bytecode [109]), locking reward price (a vulnerability), etc. Henceforth to handle these specific challenges, particularly handling of vulnerabilities, the tools' research discusses specific approaches that detect vulnerabilities. The specific approaches innovate new tools in two ways: Either the tools improvise the existing approaches (e.g., `call` analysis used by Mythril and Vandal for reentrancy detection with minor variations) or the tools create new ones (e.g., limit property used by sCompile for reentrancy detection). Tools start their processing using general approaches like symbolic analysis, taint analysis, or SMT solvers but towards the end of their detection techniques, authors devise specific techniques like Checks-Interact-Effect, limit property, locking write access, balance invariant, and so on for vulnerability detection and to make their research innovative.

Below, we provide the names of our general classes, and in the next section, we discuss the specific vulnerability detection techniques beginning with the fuzzing tools.

b: GENERAL CLASSES OF VULNERABILITY DETECTION

As discussed above, general approaches depend upon already conversant software engineering-based approaches. However, we derive the names of some general classes from the tools' development process. For instance, several tools embed Datalog Engine or Ethereum Virtual Machine to infer their results, and henceforth we formulated classes like AI and Customization-based Approaches to group such tools.

Below we list the various general classes and then discuss the specific derived approaches by the vulnerability detection tools:

i) Fuzzing-based Approaches (Section V), ii) Symbolic Execution Approaches (Section VI), iii) AI-based Approaches (Section VII), iv) Formal Verification approaches (Section VIII), v) Query Language-based Approaches (Section IX), vi) Variable Analysis-based Approaches (Section X), vii) Control Flow Graph-based Approaches (Section XI), viii) Invariant-based Checking Approaches (Section XII), ix) Execution State Validation Approaches (Section XIII), and x) Customization-based Approaches (Section XIV).

V. FUZZING-BASED APPROACHES

We discuss fuzzing as the technique to provide input to test a SC. The input can be random or targeted to feed the function arguments so that a conditional test (or a wrongful code within the function) exposes SC vulnerabilities. SC fuzzers benefit from the previous work related to American Fuzzy Lop (AFL) [110], and LibFuzzer [111]. However, AFL fuzzer employs a genetic algorithm technique, whereas LibFuzzer is a LLVM [112] utility and communicates with the library under test. Thus, LibFuzzer tests the libraries and their APIs instead of individual programs [113]. LLVM project stood for the low-level virtual machine and started developing compiler and toolchain technologies.

Table 8 lists the vulnerabilities detected by Fuzzing tools. Table 9 lists features of Fuzzing tools while thorough details of detection techniques are discussed below:

A. GENERAL TECHNIQUES OF VULNERABILITY DETECTION BY FUZZING TOOLS

1) FUZZING TECHNIQUE OF CONFUZZIUS (SECTION III-B5), TORRES ET AL. [96]

Confuzzius-related research focuses on hybrid fuzzing (Section III-B5), a combination of symbolic analysis and traditional fuzzing techniques. Additionally, Confuzzius uses a genetic algorithm (Section III-B5) and hence acts like an Evolutionary fuzzer. But Evolutionary fuzzing has limited access to deeper code [96], such as complex branch conditions. However, hybrid fuzzing aided by a constraint solver generates inputs to satisfy complex branches. After sending inputs to SC under test, Confuzzius analyzes the execution traces (Section III-B9) to know which opcodes the SC executes before termination. If the SC terminates with the opcodes like REVERT or INVALID, the tool considers the input incorrect and does not use it for testing purposes.

2) FUZZING TECHNIQUE OF CONTRACTFUZZER (SECTION III-B8), JIANG ET AL. [97]

ContractFuzzer interacts with each function of a given SC during testing by generating *k* random inputs for each function's arguments according to the input domain and data type. First, ContractFuzzer uses the inputs for generating calls to the SCs' functions during fuzzing. Then ContractFuzzer merges all generated calls into a pool of calls to separate the calls to a specific function. Then, ContractFuzzer randomly selects a call to a function from the pool to perform fuzzing. In addition to the randomly generated inputs, ContractFuzzer also generates specialized inputs (e.g. for address-typed arguments) according to the static analysis requirements.

3) FUZZING TECHNIQUE OF ETHRACER (SECTION III-B9), KOLLURI ET AL. [30]

EthRacer combines fuzzing with symbolic analysis. Fuzzing starts with randomly chosen, finite, concrete event traces and bombards events' input statements to note the outputs. Next, the fuzzer repeats the process by forming combinations by permuting event pairs (e.g., code containing call/return statement), triples, and quadruples. Then the fuzzing process matches the outputs related to the event trace to decide the bugs. However, each combination member should not violate the happens-before (Section III-B9) relationship and ensure that the matching pairs like call/return execute atomically.

4) FUZZING TECHNIQUE OF IMITATION LEARNING FUZZER (ILF) (SECTION III-B7), HE ET AL. [88]

ILF exploits the standard functionality of SCs to train a NN on thousands of programs. Moreover, symbolic execution on the history of transactions retrieves features like current function coverage, opcodes encased in the function body, and function name to help in training. The NN's output facilitates the automatic generation of test inputs to improve coverage. The generated transactions by the ILF fuzzer, comply with the signatures of functions belonging to the SC under test.

5) FUZZING TECHNIQUE OF REGUARD (SECTION III-B3), LIU ET AL. [10]

Reguard's research extends the fuzzing engine of fuzzers like AFL and LibFuzzer for SCs. First, Reguard uses Fuzzing to generate random but diverse function calls related to C++-transformed SCs. Then it records malicious execution traces of the SCs at runtime and analyzes their functional invocations, returned values, storage, Blockchain variables, and branching conditions using Finite State Machine.

6) FUZZING TECHNIQUE OF SFUZZ (SECTION III-B6), NGUYEN ET AL. [98]

First, the authors generate an initial test suite and use it as **'seeds'** to create new test cases. The initial test suite consists of **'n'** sequences of function invocations with appropriate random values for parameters based upon the parameter type and size, where n is the number of functions in the SC under test. sFuzz executes the seeds and then evaluates their fitness according to the unique branch coverage criteria. sFuzz then determines the uncovered branches of the SC and finds the seeds closest to the uncovered branch using a distance function. Then sFuzz modifies all the existing seeds using a genetic algorithm to generate more seeds but discards identical test cases. sFuzz repeats the process until a pre-defined

TABLE 9. **Fuzzing tools: General information at a glance.**

Tool Name	Citation	Publication Year	Age	Popularity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
Confuzzius	2	2021	1	2	√	10	D
ContractFuzzer	294	2018	4	73	√	7	D
EthRacer	70	2018	4	17.5	X	1	S
ILF	82	2019	3	27.3	X	6	S
Reguard	158	2018	4	39.5	X	1	D
sFuzz	74	2020	2	37	√	3	S

time. Finally, sFuzz executes the test cases and monitors the vulnerability detection process.

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY FUZZING TOOLS

1) REENTRANCY DETECTION BY FUZZING

1) **Confuzzius** (Section III-B5 and V-A1): Confuzzius declares a SC susceptible to reentrancy if the SC reads a variable from the storage before the transfer using CALL and updates the same variable but after the CALL. Thus there is a violation of Checks-Effects-Interactions (Section IX-A1) pattern.
2) **ContractFuzzer** (Section III-B8 and V-A2): The ContractFuzzer creates an Attacker agent SC, which interacts with each function of the SC under reentrancy test by sending Ether using CALL. The fuzzing engine determines the value of Ether sent. The interaction might result in the called function sending a bounty (i.e., Ether) back to the attacker using a CALL. The attacker's FF receives the reward and calls the bounty-sending function of the testing SC again. This process repeats until all the Ether of the testing SC depletes. Thus ContractFuzzer detects reentrancy vulnerability in the testing SC by triggering a reentrancy Attack cycle and confirms it by retrieving information from the interaction like the opcode of the transfer statement and gas spent by the testing SC.
3) **sFuzz** (Section III-B6 and V-A6): sFuzz creates a reentrant attacker, which calls back the function that transfers Ether to the attacker, invoking the attacker's FF.
4) **ReGuard** (Section III-B3 and V-A5): Reguard uses a combination of a Fuzzing Engine and a Finite State Machine (with five states) for detecting reentrancy vulnerability. START is the first state of these five states, ERROR is the final state, and a SC lands in the error state if the SC transfers Ether to the same address repeatedly using `call`.

2) MISHANDLED/UNHANDLED EXCEPTION DETECTION BY FUZZING

1) **Confuzzius** (Section III-B5 and V-A1): Solidity supports `send`, `transfer` and `call` but EVM contains only CALL opcode. For this reason, Confuzzius, focuses on CALL opcode in the execution trace (Section III-B9) to detect unhandled exceptions. In responsible coding, the SC checks the top of the stack for value '1' (i.e., an exception occurred) and then performs a JUMPI instruction for handling this bug. But if the JUMPI is missing after CALL in the execution trace, then Confuzzius reports the presence of unhandled exception vulnerability.
2) **ContractFuzzer** (Section III-B8 and V-A2): ContractFuzzer, detects Mishandled Exception (i.e., exception disorder) in the context of call chain [114], when the offshoot caller throws an exception but the root caller does not throw an exception. However, Vultron-related research Wang et al. [37] indicates that ConractFuzzer's strategy may produce false-positive results if all SCs in the call chain properly handle the exceptions.
3) **Imitation Learning-based Fuzzer (ILF)** (Section III-B7 and V-A4): ILF detects unhandled exception if the root call during fuzzing does not `revert()`, but the offshoot call reverts.

3) GASLESS `send` DETECTION BY FUZZING

1) **ContractFuzzer** (Section III-B8 and V-A2): Solidity uses `call`, `send`, and `transfer` for Ether delivery. EVM uses CALL for Ether transfer. If the CALL uses 0 gas argument as discussed in Darby et al. [115] and shown below:

```
address.call.value().gas()() //Solidity
    call: zero gas argument
```

then it represents Solidity's `send`. ContractFuzzer uses the above information combined with two more pieces of evidence as in Cook et al. [90] for the detection of **"Gasless send"** like CALL returns an error code of ErrOutOfGas, Jiang et al. [97], and the GASLIMIT is 2300.
2) **sFuzz** (Section III-B6 and V-A6): sFuzz checks the bytecode of CALL such that the **"value"** is greater than zero and **"gas"** cost is 2300.

4) UNPROTECTED SELFDESTRUCT DETECTION BY FUZZING

1) **Confuzzius** (Section III-B5 and V-A1): Confuzzius detects `selfdestruct` vulnerability if the attacker employs a malicious account (i.e., not previously used or not known to the victim). For example, if the runtime bytecode contains SELFDESTRUCT opcode for a transaction involving a malicious account, then

Confuzzius alerts the presence of SELFDESTRUCT vulnerability.

2) **ILF** (Section III-B7 and V-A4): A SC is vulnerable to `selfdestruct` if ILF finds that an attacker can kill the SC (i.e., an unknown user can execute SELFDESTRUCT).

5) LEAKING ETHER DETECTION BY FUZZING

1) **Confuzzius** (Section III-B5 and V-A1): Confuzzius detects leaking Ether (or stealing Ether) vulnerability if it finds any record of the CALL instruction (in the runtime bytecode of SC), which transfers Ether to a suspicious address (i.e., the address from which the SC never received Ether and the address has never been used as a parameter in a function).
2) **ILF** (Section III-B7 and V-A4): A SC is susceptible to leaking Ether vulnerability if SC ownership is transferrable to untrusted users.

6) FROZEN (OR LOCKING) ETHER DETECTION BY FUZZING TOOLS

1) **Confuzzius** (Section III-B5 and V-A1): Confuzzius checks the runtime bytecode for the non-presence of instructions like CREATE, CALL, DELEGATECALL, or SELFDESTRUCT and the presence of a receiving Ether transaction having greater than 0 value such that it ends with a STOP instruction.
2) **ILF** (Section III-B7 and V-A4): ILF detects freezing Ether vulnerability if the SC under test does not contain opcodes like CREATE, CALL, DELEGATECALL, and SELFDESTRUCT because these opcodes allow the SC to transfer ether to other SCs. Instead, the SC receives Ether.
3) **ContractFuzzer** (Section III-B8 and V-A2): ContractFuzzer determines the freezing Ether vulnerability if the SC uses DELEGATECALL, but simultaneously, the SC is devoid of instructions like CALL, and SUICIDE (deprecated), which shift the Ether outside the SC.

7) INTEGER OVERFLOW DETECTION BY FUZZING

1) **Confuzzius** (Section III-B5 and V-A1): Confuzzius analyzes the bytecode of ADD, MUL, and SUB instructions and compares the results of these mathematical instructions with the results generated by authors. If there is a disparity, then Confuzzius declares the existence of overflow vulnerability.

8) DETECTION OF TIMESTAMP DEPENDENCE VULNERABILITY BY FUZZING

1) **ContractFuzzer** (Section III-B8 and V-A2): ContractFuzzer scans the TIMESTAMP opcode in conjunction with the CALL opcode such that CALL has zero gas argument (i.e., EVM CALL behaving like Solidity's `send` or `transfer` method), and non-zero value argument [116]. But Feng et al. [31] argues that the ContractFuzzer's timestamp assertion may cause a false alarm because of the use of 'fuzzing', which is a random operation.
2) **ILF** (Section III-B7 and V-A4): ILF detects timestamp dependence vulnerability if the SC's Ether transfer depends upon Blockchain's state variables (i.e., on the output of opcodes like TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, and COINBASE) as shown in the following code modified from [100]:

```
uint constant FEE_AMOUNT =1;
address public bank = msg.sender;
var random = uint (sha3(block.timestamp)
    )%2;
if(random == 0){
   bank.transfer(FEE_AMOUNT);
}
```

LISTING 6. A TimeStamp dependence violation pattern.

A malicious miner can influence Blockchain's state variables.

9) DETECTION OF EVENT ORDERING BUGS BY FUZZING

1) **EthRacer** (Section III-B9 and V-A3): EthRacer generates outputs for each event separately and performs fuzzing using a random trace of k concrete events with distinct entry points. EthRacer assigns ids to call-return pairs and checks that they execute atomically, a condition known as linearizability.

VI. SYMBOLIC EXECUTION APPROACHES

Symbolic execution is traversing a program and determining inputs to execute different parts. For this purpose, the symbolic engine generates expressions using symbolic variables and then uses solvers to create inputs to evaluate those expressions. This merger allows the symbolic engine to reason about branches, functions, and other control structures. However, symbolic execution suffers from path explosion problems due to increased program size or unbounded loops [117].

Table 10 lists the vulnerabilities detected by Symbolic Execution tools. Table 11 highlights the features of Symbolic Execution tools, while thorough details of detection techniques are listed below:

A. GENERAL TECHNIQUES OF VULNERABILITY DETECTION BY SYMBOLIC EXECUTION

1) SYMBOLIC EXECUTION TECHNIQUE OF OYENTE (SECTION III-A12), LUU ET AL. [60]

Initially, Oyente uses the Z3 SMT solver as a constraint solver to query the branch condition. Then, the Explorer component (Section III-A12) jumps to the appropriate address based on Z3's output. If there is no result, then the Explorer traverses both branches. At the end of the exploration phase, Z3 uses the set of symbolic traces output by Explorer to eliminate infeasible paths. Finally, Z3 confirms the vulnerabilities generated by CoreAnalysis component (Section III-A12).

TABLE 10. The vulnerabilities detected by symbolic execution tools developed for SCs.

Vulnerability	SmartScopy [31]	Gasper [85]	Clairvoyance [29]	Oyente [60]
Reentrancy	✓		✓	
Costly Gas Pattern		✓		
Unchecked send	✓			✓
BatchOverflow	✓			
Timestamp Dependence	✓			✓
Call Stack Depth				✓
Transaction Ordering Dependence				✓

TABLE 11. Symbolic execution tools: General information at a glance.

Tool Name	Citation	Publication Year	Age	Publicity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
SmartScopy	33	2019	3	11	√	4	S
Gasper	308	2017	5	61.6	X	1	S
Clairvoyance	10	2020	2	5	√	1	S
Oyente	1615	2016	6	269.1	√	4	S

2) SYMBOLIC EXECUTION TECHNIQUE OF SMARTSCOPY (SECTION III-A7), FENG ET AL. [31]

SmartScopy utilizes summary-based symbolic evaluation to solve space explosion problems related to symbolic execution. The summary-based technique incorporates early pruning to search for vulnerability patterns like 'CALL after SSTORE' in SCs. If the SC lacks the pattern, then the SC is subject to vulnerability analysis. Again SmartScopy uses such patterns for developing attack synthesizers. Path pruning techniques remove the already explored path of the program from the focus of the symbolic analyzer to avoid path explosion problem [118]. Furthermore, SmartScopy symbolically executes each public function only once instead of several times using different symbolic arguments.

3) SYMBOLIC EXECUTION TECHNIQUE OF GASPER (SECTION III-A21), CHEN ET AL. [85]

Gasper performs symbolic execution of SC and declares the CFG blocks not executed by symbolic execution as Dead Code Pattern, Chen et al. [85]. Similarly, an opaque predicate always evaluates to true or false. Hence its evaluation wastes gas. To detect expensive operations in the loop, Gasper analyzes the CFG for JUMP and JUMPI opcodes and collects information related to the entry and exit blocks and the number of blocks within the loop. Gasper then passes this information to the Dijkstra algorithm to detect expensive loop operations.

4) SYMBOLIC EXECUTION TECHNIQUE OF CLAIRVOYANCE (SECTION III-A29), HE ET AL. [88]

Clairvoyance innovates a static analysis-based reentrancy detection technique by applying cross-contract and cross-function analysis on Solidity SCs. The advantage is tracing tainted parts related to the interprocedural call chain. The authors first construct a CFG exhibiting a call chain between the SC under test and a remote SC. Then, use the extended CFG, i.e., XCFG, to perform taint analysis to detect paths containing reentrancy bugs. XCFG represents a cross-contract call graph merged with CFGs of SC's functions. Finally, the detected paths are subject to symbolic computation using path protection techniques to eliminate non-reentrant traces. Path protection techniques are countermeasures to avoid reentrancy attacks discussed by He et al. [88].

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY SYMBOLIC EXECUTION TOOLS

1) REENTRANCY DETECTION BY SYMBOLIC EXECUTION

1) **Clairvoyance** (Section III-A29 and VI-A4): Clairvoyance detects reentrancy vulnerability by finding cross-contract CALL chains, which loop to callback the current caller.
2) **SmartScopy** (Section III-A7 and VI-A2): SmartScopy confirms the reentrancy by synthesizing an attacker SC. The attacker acts as a recipient of a CALL transaction. Thus the attacker receives sufficient gas and generates successive CALL invocations that do not modify the storage (i.e., the balance variable) in each CALL, i.e., line#4 in listing [51] below never executes. This makes the condition in line#2 always true.

```
1  function Withdraw(uint amount) {
2   if(balances[msg.sender] >= amount) {
3    msg.sender.send(amount);
4    balances[msg.sender] -= amount;}}
```

2) COSTLY GAS PATTERN (I.E., DEAD CODE AND OPAQUE PREDICATE) DETECTION BY SYMBOLIC EXECUTION

1) **Gasper** (Section III-A21 and VI-A3): For dead code, Gasper uses symbolic execution to record addresses of executed blocks. Next, Gasper records the addresses of CFG blocks related to the dead code. Finally, Gasper warns about CFG blocks which symbolic execution

does not execute as dead code patterns. For opaque predicate detection, Gasper records the branch condition during the symbolic execution of the conditional jump statement. Any branch related to a conditional jump that never executes is vulnerable to the opaque predicate.

3) UNCHECKED `send` OR MISHANDLED EXCEPTION DETECTION BY SYMBOLIC EXECUTION

1) **SmartScopy** (Section III-A7 and VI-A2): Suppose a SC does not use CALL's return value in any conditional jump statement. In that case, the SC is vulnerable to unchecked `send` because the SC does not check CALL's return value.
2) **Oyente** (Section III-A12 and VI-A1): Oyente detects (UnChecked `send` or Mishandled Exception) by checking that each CALL instruction in the SC follows the (EVM opcode) ISZERO. Otherwise, the code is vulnerable to mishandled exception, Atzei et al. [20]. When an exception occurs, SC pushes zero on the stack, and ISZERO checks if the top of the stack is zero [119].

4) BATCHOVERFLOW DETECTION BY SYMBOLIC EXECUTION

1) **SmartScopy** (Section III-A7 and VI-A2): BatchOverflow vulnerability occurs when the multiplication operation overflows. SmartScopy detects this vulnerability using the automatic synthesis of an attacker program, which disables the checks and overflows one of the multiplication operands. Lastly, the attacker manipulates the transfer's address to become the transaction's beneficiary.

5) TIMESTAMP DEPENDENCE DETECTION BY SYMBOLIC EXECUTION

1) **SmartScopy** (Section III-A7 and VI-A2): SmartScopy declares the Ether transfer vulnerable to TIMESTAMP if Ether transfer statements like CALL (i.e., Solidity `call` and `send`) depend upon `block.timestamp` (i.e., Solidity's representation of TIMESTAMP [120]).
2) **Oyente**, (Section III-A12 and VI-A1): Oyente detects the timestamp threat by the presence of a symbolic variable (as the tool uses Symbolic Execution) and assumes that the path condition for the timestamp threat must employ the symbolic variable and hence becomes vulnerable to timestamp dependency. However, Atzei et al. [20] have argued that this approach may result in false-positive detection as the SC can check the path condition using `block.timestamp` without employing the symbolic variable.

6) TRANSACTION ORDERING DEPENDENCE DETECTION BY SYMBOLIC EXECUTION

1) **Oyente** (Section III-A12 and VI-A1): Oyente discusses two techniques for transaction ordering dependence. Methods can detect transaction ordering problems in the marketplace context and reward for solving a puzzle. The first technique detects transaction Ordering vulnerability if a SC has pairs of Ether flowing paths showing the disparity in the transaction order. However, according to new rules Luu et al. [60], Oyente uses the guard condition 'g', which if fails, the transaction fails. 'g' states that the 'reward == R' where R is the current value of the reward. Participating SC locks the original reward amount in R. If the miner disorders the transaction, then the locked price of R will not equal the original value of the reward.

7) CALL STACK DEPTH DETECTION BY SYMBOLIC EXECUTION

1) **Oyente** (Section III-A12 and VI-A1): Oyente detects call stack depth threats by checking if the SC tries to exceed the call stack beyond its maximum size, a condition that would result in remote call failure irrespective of how much gas a SC uses.

VII. AI-BASED APPROACHES

This research focuses only on programming-based approaches. Hence machine learning [121] is not part of our domain. On the other hand, we incorporate tools that use rule-based, logic-based, or solver-based techniques for vulnerability detection.

Table 12 indicates features of AI-based tools, while thorough details of detection techniques are listed below. Table 13 lists the vulnerabilities detected by AI-based tools.

A. GENERAL TECHNIQUES OF VULNERABILITY DETECTION BY AI

1) LOGIC RELATION-BASED TECHNIQUE OF VANDAL (SECTION III-A5), BRENT ET AL. [36]

Logic relations ease vulnerability detection techniques as they require few lines of code, for instance, using Souffle. Logic relations focus on rules. For example, authors use boolean operators to specify the security analysis-related rules in Vandal's development research, Brent et al. [36]. Thus, the logic-based approach is compact compared to the traditional methods based on functional and OOP languages. An example is given below: Authors incorporate

```
1  uncheckedCall(u) :-
2   callResult($_$, u),
3   !checkedCallThrows(u),
4   !checkedCallStateUpdate(u)
```

LISTING 7. Vandal's boolean logic-based security analysis, Brent et al. [36].

logic relation using a static analysis library, Brent et al. [36] that processes CFG, EVM opcodes, control, and data flow constructs. For instance, Listing 7 shows the use of logic relations to implement unchecked `send` detection procedure (designated as `uncheckedCall`, line#1) discussed in (section VII-B). The procedure claims SC is vulnerable if the SC does not check the return value `u` of a transfer

TABLE 12. AI-based tools: General information at a glance.

Tool Name	Citation	Publication Year	Age	Publicity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
Vandal	173	2018	4	43.25	X	5	S
Securify	517	2018	4	129.2	√	5	S
MadMax	253	2018	4	63.2	X	3	S
TeEther	196	2018	4	49	X	5	S
EtherTrust	30	2018	4	7.5	X	2	S
Ethainter	38	2020	2	19	X	1	S
DappGuard	22	2017	5	4.4	√	4	S
Constructive ECF	11	2020	2	5.5	X	1	S

(i.e. CALL which is part of EVM bytecode but is used to detect problems in Solidity `send` and `throw` which is deprecated), line#2, i.e., if the return value tells about vulnerable `throw` (deprecated), line#3. Line#4 tells no action is taken if the vulnerable code alters EVM storage. Deprecated `throw` returns an exception whereas `send` returns false.

2) STRATIFIED DATALOG-BASED TECHNIQUE OF SECURIFY (SECTION III-A10), TSANKOV ET AL. [75]

First, Securify applies Stratified Datalog to the SC to create a dependency graph and then analyzes the dependence graph using Datalog solvers (i.e., Securify constructs a dependence graph using symbolic information and Datalog engine for vulnerable patterns). The benefit is the creation of inference rules to make logical reasoning about the SC's storage and memory utilization. Next, Souffle Datalog solvers compute a fixed point using all facts, which is the minimal model for the SC. The fixed point is a Datalog engine for declaring rules and relations and specifying queries [122]. Finally, Securify analyzes the fixed point to get information about vulnerable instructions and warnings.

3) DATALOG RULE-BASED TECHNIQUE OF MADMAX (SECTION III-A16), GRECH ET AL. [12]

MadMax defines Datalog rules for vulnerability analysis and compiles them using Souffle. The Datalog program focuses on relational instructions with the help of specialized data structures. MadMax implements analysis using a layered code where the first layer focuses on loops and induction variables (i.e., variables whose value depends on the loop iteration variable's value). The next layer focuses on memory, dynamic data structures, nested arrays, and SCs whose storage increases on reentrance, even though MadMax does not discover reentrancy vulnerability. The last layer deals with gas-related vulnerabilities.

4) CONSTRAINT SOLVER-BASED TECHNIQUE OF TEETHER (SECTION III-A6), [56]

Using constraint solver Z3, the TeEther infers vulnerable transactions related to critical instructions (or functions) like CALL, CALLCODE, DELEGATECALL, and SELFDESTRUCT. But, Teether does not focus on user-defined functions. Instead, it uses an iterative constraint-solving approach by finding the critical paths and the related path constraints leading to said built-in functions (e.g., path constraints like conditional tests to access `selfdestruct`). Then uses, the constraint solver to obtain a satisfying assignment that can generate tangible values to invoke the critical instructions of SC under test. Finally, TeEther uses the satisfying assignment to create transactions for the attacker to exploit the SC.

5) Z3's SPACER ENGINE BASED-TECHNIQUE OF ETHERTRUST (SECTION III-A25), GRISHCHENKO ET AL. [87]

EtherTrust's static analysis supports the reachability property. For this purpose, EtherTrust converts the bytecode into an internal Horn Clause (Section III-A1) representation supported by Z3's SPACER. Then, SPACER finds a counter-example and constructs safe invariants by extending the reachability concept, Kostyukov et al. [123]. However, if Z3 can't satisfy the query, then EtherTrust assumes the query as satisfiable based upon soundness theory, Grishchenko et al. [87]. When writing the research, Grishchenko et al. [87], EtherTrust does not support CREATE, CALLCODE, or DELEGATECALL instructions.

6) DATALOG AND REPEATED TRANSACTIONS-BASED TECHNIQUE OF ETHAINTER (SECTION III-A26), BRENT ET AL. [5]

Ethainter converts the Datalog representation into C++ code for vulnerability detection. In addition, the authors designed rules in abstractions to model the attacker, owner, and guards. For instance, `ReachableByAttacker` models the parts of the code executable by an Attacker. The attacker would taint the executable parts, even variables meant for guarding. Hence Ethainter invokes `AttackerModelInfoflow` for taint analysis and initiates recursion between `ReachableByAttacker` and `AttackerModelI- nfoflow` for catching composite violations.

TABLE 13. The vulnerabilities detected by AI tools: Vandal [36], Securify [75], MadMax [12], TeEther [56], sCompile [58], Ethainter [5], DappGuard [90], ECF [53], developed for SCs.

Vulnerability	Vandal	Securify	MadMax	TeEther	Ethainter	DappGuard	ECF
Reentrancy	✓	✓			✓		
Wallet Griefing			✓				
UnChecked `send`	✓	✓				✓	
UnSecured Balance	✓	✓					
Unbounded Mass Operation			✓				
Unprotected SELFDESTRUCT	✓			✓	✓		✓
Leaking Ether	✓	✓					
Frozen Ether		✓			✓		
Integer Overflow			✓				
Transaction Ordering Dependence		✓				✓	
Transaction Origin	✓						

7) OYENTE ENGINE AND RULES ENGINE BASED-TECHNIQUE OF DappGuard (SECTION III-A27), COOK ET AL. [90]

DappGuard applies a multivariant approach for vulnerability detection. For instance, to detect vulnerabilities like transaction ordering dependence, exception disorder, and timestamp dependence, DappGuard's validator runs Oyente on the SC under test by applying constraints on arguments' data and views the anomalies reported by transactions. But for other vulnerabilities, DappGuard uses the rules engine to determine the validity of transactions related to the SC under test. Rules Engine stores Blockchain's transaction history, report anomalies, and updates its security indicators based on its judgments.

8) SMT SOLVER BASED-TECHNIQUE OF CONSTRUCTIVE ECF (SECTION III-A28), ALBERT ET AL. [53]

Constructive ECF detects reentrancy vulnerability using the effectively callback-free (ECF) notion. However, according to ECF, reentrant functions can't be callback-free. Thus, a program exhibiting reentrancy can't remove a reentrant function like `withdraw()` function because due to the removal of `withdraw()`, the program cannot handle the callback. For instance, the listing 9 uses `withdraw(..)` to transfer Ether using `call` which results in a callback. The authors used SMT solver to prove that removing `withdraw()` would violate commutative property.

B. *SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY AI TOOLS*

1) REENTRANCY DETECTION BY AI

1) **Constructive ECF** (Section III-A28 and VII-A8): Constructive ECF detects reentrancy by checking if it can simulate an execution with callback using a code without callbacks.
2) **DappGuard** (Section III-A27 and VII-A7): The reentrancy attack triggers an abnormally high gas consumption, and DappGuard uses this information to identify reentrancy. DappGuard applies the technique of *transaction log file*. The amount of gas used in the transaction can act as a determining factor for the reentrancy attack.
3) **Securify** (Section III-A10 and VII-A2): Securify skims for external calls, followed by writes to storage. Even though this pattern-matching skill can simplify automated testing, related work in Rodler et al. [54] labels this approach as too conservative.
4) **Vandal** (Section III-A5 and VII-A1): The authors of Vandal detect reentrancy by critically analyzing the CALL code to see if it's recursive. The recursive block does not apply mutex, and the CALL's gas argument is substantial. Note that mutex-free code permits the attacker to perform unnecessary state changes. Solidity implements mutex using locks as discussed in [124]. Line#5 in the listing [125] below initiates a `call` transfer. The attacker's FF generates a reentrant call but the lock being true, line#2, prevents reentrancy.

```
1 function withdraw() external {
2  require(!lock);
3  lock = true;
4  uint256 amnt = balances[msg.sender];
5  require(msg.sender.call.value(amnt)());
6  balances[msg.sender] = 0;
7  lock = false;
8 }
```

Note in the above code the initial value of the lock is false so that we can bypass require check.

2) UNCHECKED `send` (OR MISHANDLED EXCEPTION) DETECTION BY AI

1) **Securify** (Section III-A10 and VII-A2): Securify detects mishandled exception if CALL does not follow a JUMPI, which should depend on the return value of CALL, Tsankov et al. [75]. Note there is no goto in Solidity [126] but Solidity depends on EVM JUMP and JUMPI for translation of its loops [127].
2) **Vandal** (Section III-A5 and VII-A1): Vandal declares the SC vulnerable to "UnChecked `send`" if the SC takes no tangible action like throwing an exception or updating persistent storage based on the return value of `CALL`.

3) SELFDESTRUCT DETECTION BY AI

1) **Ethainter** (Section III-A26 and VII-A6): Ethainter discusses two variants of SELFDESTRUCT: accessible SELFDESTRUCT, in which SELFDESTRUCT command lies in a public function, and execution of the public function destroys the SC. Ethainter confirms the vulnerability by executing the public function using the companion tool, Ethainter-Kill. Another variant is "tainted selfdestruct" which furnishes composite attacks because the attacker has to invoke multiple transactions to get the required permissions to execute the SELFDESTRUCT command. Still, the code is vulnerable if the code initializes the owner field outside the `constructor`. Again Ethainter confirms SELFDESTRUCT vulnerability by killing the SC using Ethainter-kill.
2) **TeEther** (Section III-A6 and VII-A4): TeEther detects SELFDESTRUCT vulnerability if the SELFDESTRUCT opcode is reachable and the topmost stack argument is vulnerable to external control. Thus the attacker modifies the top of the stack by its address.
3) **Vandal** (Section III-A5 and VII-A1): Vandal discovers selfdestruct vulnerability if the developer exposes SELFDESTRUCT instruction to an arbitrary caller without proper authentication.

4) UNSECURED BALANCE (OR STEALING ETHER) DETECTION BY AI

1) **Vandal** (Section III-A5 and VII-A1): Vandal uses the term "unsecured balance" and provides the example of Rubixi Ponzi SC, Atzei et al. [20], to refer to the stealing of Ether vulnerability. Vandal highlights that Unsecure Balance vulnerability can cause an arbitrary caller to manipulate the Ether transfer address of CALL and henceforth transfer Ether to any undesired address. Due to this transfer threat, Vandal checks the CALL's destination address to verify whether the attacker can manipulate the address (possible if the victim's SC initializes the Ether transfer address in a non-constructor method) and if the accompanied amount of CALL is non-zero.
2) **Securify** (Section III-A10 and VII-A2): Tsankov et al. [75] equip **Securify** with violation and compliance patterns [44]. For stealing Ether threat, Securify scans the code of initialization function other than a `constructor` and alarms a violation pattern. The violation pattern may indulge in free writing (i.e., permissionless writing) to a storage variable.

5) FREEZING ETHER DETECTION BY AI

1) **Securify** (Section III-A10 and VII-A2): Securify discovers the vulnerabilities related to freezing Ether by detecting the two facts, i.e., users can credit the amount in the SC's account, and at the same time, Ether transfer instructions are missing from the SC. For transfer, Securify checks that all the CALL instructions transfer 0 ether. For a deposit instruction, Securify verifies a successful path through which the SC can receive Ether, i.e., the path terminates with STOP, and the amount that SC could receive is greater than 0.

6) UNBOUNDED MASS OPERATIONS OR DoS (DUE TO OUT-OF-GAS) DETECTION BY AI

1) **MadMax** (Section III-A16 and VII-A3): MadMax detects gas vulnerabilities resulting in the "out-of-gas" problem, which is directly related to Denial of Service. Due to a lack of gas, some transactions may not run to completion. MadMax detects Denial of Service in the context of the unbounded mass operation (i.e., operation on an enormously large array) carried out within a loop. According to unbounded mass operation vulnerability the loop is running through array size but not resumable [23] or the loop is not checking the amount of gas at each iteration. A resumable loop keeps track of how far it has run. Hence, we can restart the loop from the point of failure instead of redoing the whole operation again.

7) WALLET GRIEFING (SECTION VIII-B3) DETECTION BY AI

1) **MadMax** (Section III-A16 and VII-A3): MadMax detects Wallet Griefing vulnerability (Section VIII-B3) by detecting `for` loop containing both `send` and `throw` (deprecated) or `revert` (instead of `throw`) and a dynamic array (i.e., a runtime value determines the loop's number of iterations) that loads the addresses of the recipient for transfer using `send`. Research, Grech et al. [12], declares Wallet Griefing, a severe problem because `revert` can occur in the middle of the loop [23].

8) INTEGER OVERFLOW DETECTION BY AI

1) **MadMax** (Section III-A16 and VII-A3): MadMax detects integer overflow in connection with loops if the loops have a dynamically bounded termination condition. Dynamically bounded means that the number of iterations of the loop is determined at runtime. Example in [12] shows that MadMax uses this strategy for `var`, which is a deprecated vulnerability.

9) TRANSACTION ORDERING DEPENDENCE DETECTION BY AI

1) **DappGuard)** (Section III-A27 and VII-A7): DappGuard detects transaction ordering when the user tries to broadcast high gas activity after placing the bet, which baits the miner from influencing the transaction's outcome immediately before the SC's execution. Higher gas means that the bet-related transaction might have more than the average value.
2) **Securify** (Section III-A10 and VII-A2): Securify detects transaction ordering in the context of selling initially low-priced tokens, helping startups earn high bonuses when the token prices are low. However, these

TABLE 14. The vulnerabilities detected by formal verification tools developed for SCs.

Vulnerability	FVF* [51]	Zeus [59]	ITPs [81]
Reentrancy	✓	✓	✓
Failed send (or Wallet Griefing)		✓	
Unchecked send	✓	✓	
Integer (Overflow/Underflow)		✓	
selfdestruct		✓	✓
Timestamp Dependence		✓	
Transaction Ordering Dependence		✓	

transactions may be subject to miners' exploitation. Securify detects manipulations if the amount of CALL depends upon reading storage data.

10) TRANSACTION ORIGIN DETECTION BY AI

1) **Vandal** (Section III-A5 and VII-A1): Vandal detects `tx.origin` vulnerability at the bytecode level if the SC uses the bytecode of `tx.origin` (i.e., ORIGIN) for altering state variables or as a conditional test. However, security gurus discourage the comparison of `tx.origin` with the owner (i.e., initiator of the transaction) but not with `msg.sender`, Tikhomirov et al. [52].

VIII. FORMAL VERIFICATION APPROACHES

Formal verification is a process of proving mathematically that a system obeys its design [128]. Formal verification tests a program using an entire set of inputs instead of one or two values. One way to achieve this is to consider a class of values, such as positive or negative.

Table 14 lists the vulnerabilities detected by formal verification tools. Table 15 lists features of Formal Verification-based tools, while thorough details of detection techniques are discussed below:

A. GENERAL TECHNIQUES OF VULNERABILITY DETECTION BY FORMAL VERIFICATION

To implement formal verification, Zeus focuses both on abstraction and the entire data domain. Tools also relied on **Invariants**, which are always true properties. Our research speaks about vulnerability detection using formal verification in connection with tools like Zeus, interactive theorem provers, and FVF* as discussed below:

1) FORMAL VERIFICATION BY ZEUS (SECTION III-A1), [59]

To implement Formal Verification, Zeus aims to verify and validate SC's correctness based on syntax and adhering to business logic. Moreover, Zeus focuses on automatic formal auditing instead of manual work. Auditing means compliance with the specification. Zeus inputs a policy file for compliance checking. Zeus achieves its goal by utilizing abstract interpretation, formal model checking, and constrained horn clauses. Zeus applies abstract interpretation on a Solidity SC converted into an LLVM bit code-based interpretation to compute loops and public functions. Abstract interpretation verifies a program's semantics using conceptual ideas (like positive or negative values) instead of fixed values [129]. Model-checking implies verifying and validating the SC's behavior, which requires specifications supplied using a policy file. Zeus checks only policy specifications related to SC's security using SeaHorn [130] tool. Horn clauses use recursion and bounded iteration techniques. In addition, Zeus puts assert statements before vulnerable statements for testing purposes.

2) FORMAL VERIFICATION USING EVM DEFINITION BY INTERACTIVE THEOREM PROVERS (ITPs) (SECTION III-A19), [81]

The author focused on interactive theorem provers like LEM, OCAML, and Isabelle/HOL. For example, Isabelle is a proof assistant for proving mathematical formulas using logical calculus, whereas OCAML is a general-purpose functional programming language. In collaboration with his team, the author translated the EVM bytecode in LEM and used the translation to verify SCs. LEM is a mathematical tool for writing, publishing, and managing portable semantic definitions. Moreover, LEM's EVM description supports its translation into other interactive theorem provers like OCAML, which the author used for testing against the VM test suite. Next, the author translated the LEM definition into Isabelle/HOL, tested the translation on the VM test suite, and got the same results as with OCAML. This approach helped in the formal verification of SCs. In addition, the author uses an invariant (Section VIII-A) and shows that the failure of the invariant is a sign of vulnerability, such as reentrancy and SELFDESTRUCT. The author has discussed that SELFDESTRUCT execution does not immediately remove the SC from the Blockchain or the removal of SC does not occur during execution.

3) FORMAL VERIFICATION BY FVF* (SECTION III-A23), BHARGAVAN ET AL. [51]

Authors performed formal verification of Ethereum SCs using F*. They developed a Solidity language-based approach incorporating F*, which uses two tools (1) Solidity*, which converts Solidity programs into F* (2) EVM*, which converts EVM bytecode into F*. Authors then used the Solidity* and EVM* tools to perform three types of formal verification: (1) Given a Solidity program, authors translated

TABLE 15. Formal verification tools: General information at a glance.

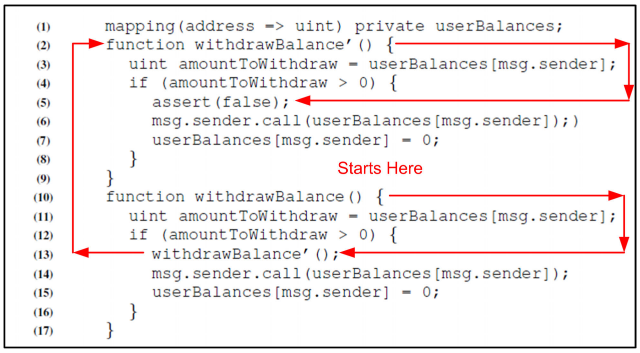
Tool Name	Citation	Publication Year	Age	Publicity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
Zeus	459	2018	4	114.7	√	6	S
ITPs	271	2017	5	54.2	X	2	S
FVF*	563	2016	6	93.8	√	2	S

the program into F* using Solidity* and used the Solidity source code to verify SC invariants (Section VIII-A) and runtime safety. (2) Given EVM bytecode author decompiled the bytecode into Solidity using EVM* to verify the amount of gas consumed by calls. (3) Given a Solidity program and its EVM bytecode, authors verified their equivalence by converting them into F*.

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY FORMAL VERIFICATION TOOLS

1) REENTRANCY DETECTION BY FORMAL VERIFICATION

1) **FVF*** (Section III-A23 and VIII-A3): FVF* detects reentrancy if the SC modifies storage variable after `send`. However, the approach discussed in FVF* related research, Bhargavan et al. [51], uses `send` instead of `call`. `send` transfers only 2300 gas to the caller and doesn't cause reentrancy.
2) **Interactive Theorem Provers (ITPs)** (Section III-A19 and VIII-A2): Interactive Theorem Provers use the *failure of execution* as an indication of reentrancy. The reentrant SC fails due to the violation of invariant property. The invariant property does not hold because the reentrant message-calls result in changes in the account state, i.e., SC's balance and storage.
3) **Zeus** (Section III-A1 and VIII-A1): We will now discuss an example of reentrancy detection using Zeus, which clarifies its working. Figure 2, line#10, shows the function, `withdrawBalance()`, retrieved from [59]. Red lines in Figure 2 highlight the steps of execution. Zeus clones the `withdrawBalance()` and names it as `withdrawBalance()'`, line#2. Zeus then performs static analysis, inserts an `assert` statement in the cloned function at line#5, and invokes the cloned function at line#13. Note both these insertions (i.e., cloned function and `assert(..)`) are before `call`-transfers and hence detect reentrancy vulnerability. The call-chain consists of the SC verifier invoking the `withdrawBalance()`, which in turn invokes the cloned function and halts due to `assert`. `assert` violations help the verifier to detect policy violations. However, Zeus modifies the SC under test. Subsequently, this technique is not scalable and unsuitable for cross-function reentrancy testing.



```
(1)      mapping(address => uint) private userBalances;
(2)      function withdrawBalance'() {
(3)        uint amountToWithdraw = userBalances[msg.sender];
(4)        if (amountToWithdraw > 0) {
(5)          assert(false);
(6)          msg.sender.call(userBalances[msg.sender]););
(7)          userBalances[msg.sender] = 0;
(8)        }
(9)      }
(10)     function withdrawBalance() {
(11)       uint amountToWithdraw = userBalances[msg.sender];
(12)       if (amountToWithdraw > 0) {
(13)         withdrawBalance'();
(14)         msg.sender.call(userBalances[msg.sender]);
(15)         userBalances[msg.sender] = 0;
(16)       }
(17)     }
```

FIGURE 2. Reentrancy detection by zeus, modified from [9].

2) MISHANDLED EXCEPTION (OR UNCHECKED `send` DETECTION BY FORMAL VERIFICATION

1) **FVF*** (Section III-A23 and VIII-A3): For determining unchecked send vulnerability, the authors merged the concept of exception with monadic behavior, i.e., related to chaining operations [131], i.e., computation depends upon the result of a computation that came before it [132]. A boolean variable implements the monadic behavior; for instance, throws an exception if the boolean variable is false. The built-in interactive proof assistant then verifies the translations.
2) **Zeus** (Section III-A1 and VIII-A1): Zeus maintains that "unchecked `send`" threat arises due to failure of `send` in the destination SC and leads to inconsistency if the caller does not handle failed `send` invocations. Therefore, Zeus initializes a global variable (i.e., `gv`) to true if the send fails and inserts an assert statement before any subsequent write to the global state to confirm that unchecked `send` should not cause inconsistency in the SC.

3) FAILED `send` DETECTION BY FORMAL VERIFICATION

1) **Zeus** (Section III-A1 and VIII-A1): In connection with failed `send`, the authors of Zeus discuss a similar SC in [59] and grouped failed `send` with Wallet Griefing. In Wallet Griefing attacker acts as an investor but provides a callback function that does not handle sub-call [133]. Zeus tries to avoid invoking of `throw` (deprecated) because it can facilitate the unlawful activity of ransom, as discussed in [134]. `throw` can cause reverting of all transactions due to the failure of one. Thus Zeus automatically puts an assertion before `throw` to detect the failed send vulnerability.

4) `selfdestruct` VULNERABILITY DETECTION BY FORMAL VERIFICATION

1) **Interactive Theorem Provers (ITPs)** (Section III-A19 and VIII-A2): Interactive Theorem

TABLE 16. Query language-based tool: General information at a glance.

Tool Name	Citation	Publication Year	Age	Popularity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
SmartCheck	354	2018	4	88.5	X	21	S

Provers detect SELFDESTRUCT if the invariant fails, i.e., the state changes occur due to the execution of SELFDESTRUCT. State changes occur because SELFDESTRUCT's execution destroys SC and hence clears the state variables; hence the invariant fails. Thus the SC is vulnerable.

5) INTEGER OVERFLOW/UNDERFLOW DETECTION BY FORMAL VERIFICATION

1) **Zeus** (Section III-A1 and VIII-A1): Zeus remains consistent with Solidity's semantic, [135], for detecting overflow and underflow, i.e., Solidity wraps around the maximum value of an integer data type to zero on increment and acts conversely for decrement operation (i.e., if we subtract 1 from a binary number with all zeros, all digits will become 1). Zeus marks the SCs unsafe (unnecessary in some cases) by detecting wrap-around errors as discussed above. Perhaps, Zeus performs overflow detection by checking the boundary values.

6) `timestamp` DEPENDENCE DETECTION BY FORMAL VERIFICATION

1) **Zeus** (Section III-A1 and VIII-A1): Zeus applies broader path constraints, taint analysis, and symbolic model checking to detect timestamp dependence bugs. Thus detection incorporates several block state variables such as COINBASE, NUMBER, DIFFICULTY, and GASLIMIT in addition to the TIMESTAMP to generate alarms if the `send` or `call` path uses said variables. Symbolic model checking (Section III-A1) helps to eliminate infeasible paths.

7) TRANSACTION ORDERING DEPENDENCE DETECTION BY FORMAL VERIFICATION

1) **Zeus** (Section III-A1 and VIII-A1): Zeus tool applies taint analysis on global variables. Zeus signals global variables (i.e., predefined variables [136]) vulnerable to Transaction Ordering if the tainted variables perform write-operations and then influence `send` or `call` statements.

IX. QUERY LANGUAGE APPROACHES

One example of implementing a Query-based approach is transforming the Solidity SC into XML representation and then using XPATH-based query language to detect vulnerabilities as discussed in (Section III-A8) in connection with SmartCheck.

Table 16 highlights features of SmartCheck, while thorough details of detection techniques are listed in Table 17, which lists some vulnerabilities detected by SmartCheck. SmartCheck detects several other vulnerabilities, and we would refer the interested reader to the research in [52] to get more knowledge about SmartCheck's vulnerability detection techniques.

A. GENERAL TECHNIQUE OF VULNERABILITY DETECTION BY QUERY LANGUAGE

1) XPATH BASED-TECHNIQUE BY SMARTCHECK (SECTION III-A8), TIKHOMIROV ET AL. [52]

SmartCheck first develops an XML parse tree from the Solidity code and then uses the XPATH expressions to traverse the tree. XPATH expressions are similar to path expressions. Smartcheck uses XPATH[xpa] queries to detect vulnerability patterns. The advantage of XML documents is that we can store vulnerability descriptions in the document, and the XPATH[xpa] query can retrieve that information. However, XPATH is a simple mechanism for retrieving data from XML documents and subsequently can't support complex vulnerability patterns and may output false-positive results for catching Checks-Effects-Interactions violations in connection with reentrancy vulnerability. Hence for complex XML documents, XPATH benefits from the xpa tool to track statistical information related to XML documents, such as calculating the number of function calls which can help in reentrancy detection.

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY QUERY LANGUAGE TOOLS

1) REENTRANCY DETECTION BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): SmartCheck tests reentrancy depending upon three actions: Checks-Effects-Interactions. Values of these patterns can be, for instance: (1) *Checks:* This means that the invariant (Section VIII-A) must always be true. For instance, in the case of Ether transfer, the withdrawal amount should always be less than the balance, i.e., `amount < balance`, (2) *Effects:* After the invariant test and before the transfer, we must implement the effect. In the case of transfer, the effect is that the balance would become zero or the balance would reduce by withdrawal amount, i.e., the effect would be: `balance = 0`, or `balance -= amount`. (3) *Interactions:*. Finally, we would do the transfer using `call` so that the transfer should not fail if the receiving FF is costly.

```
msg.sender.call.value(balances[msg.
    sender])().} //Solidity version 5
    syntax
```

Listing 8 presents the violation pattern, i.e., CHECK-INTERACTION-EFFECT. The attacker implements

TABLE 17. The vulnerabilities detected by query language tool (i.e., SmartCheck [52]), CFG-based tool (i.e., Mythril [137], sCompile [58]), Invariant Violation Tool (i.e., Vultron [37], VeriSmart [72]), and runtime monitoring and validation tool (i.e., Sereum [54]) developed for SCs.

Vulnerability	SmartCheck	Mythril	Vultron	Sereum	sCompile	VeriSmart
Reentrancy	✓	✓	✓	✓	✓	
Costly Loop (or Wallet Griefing)	✓					
Unchecked send	✓		✓			
Integer (Overflow/Underflow)			✓			✓
Gasless send			✓			
selfdestruct		✓			✓	
Timestamp Dependence	✓					
DoS (unexpected throw, deprecated)	✓					
Malicious Library	✓					
Frozen Ether	✓				✓	
tx.origin	✓					
Type Inference	✓					

the DAO attack by calling `withdrawMoney()`, line#1, and then repeats the call using its fallback function. The sequence of invocations would be:
• `withdrawMoney` (i.e. a user-defined unprotected Ether transfer function in the sender SC)
• Fallback Function (i.e., a Solidity-based Ether receiver function in an attacker SC)
• `withdrawMoney` (i.e., an invocation to the sender's Ether transfer function in the attacker's fallback function, which initiated the transfer)

```
1  function withdrawMoney() {
2      uint amountToWithdraw = userBalances
          [msg.sender];
3      // CHECK
4      if (amountToWithdraw > 0) {
5          // INTERACTION
6          msg.sender.call(amountToWithdraw
              );
7          // EFFECT
8          userBalances[msg.sender] -=
               amountToWithdraw;
9      }
10 }
```

LISTING 8. Checks-effects-interaction violation pattern, [138].

The above pattern repeats until an out-of-gas situation arises. CHECK-EFFECT-INTERACTION pattern can mitigate the reentrancy problem as discussed in [139] in the context of mitigation and prevention of reentrancy attacks.

2) DENIAL OF SERVICE DUE TO UNEXPECTED `throw` (DEPRECATED), DETECTION BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): If a conditional statement depends on the external function such as `call` or `transfer`, then it may cause DoS because the receiver may `throw` (deprecated) or `revert` as discussed in [23]. For example, SmartCheck uses the pattern: if-statement followed by external `call` in the condition, and the body contains `revert`. Similarly, the if-pattern discussed above can exist within a `for` loop.

3) UNCHECKED EXTERNAL CALL OR MISHANDLED EXCEPTION DETECTION BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): SmartCheck detects the SC vulnerable for an unchecked external `call` invocation if the search pattern detects `call`, `delegatecall` or `send` without an `if`-statement.

4) DETECTION OF LIBRARY VULNERABILITY BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): Libraries help in code reduction, but this vulnerability occurred when the Parity SC used a library for transferring Ether [23]. But when the attacker killed Library SC, there was no way out to release the funds of Parity SC. SmartCheck detects the library keyword.

5) DETECTION OF LOCKED ETHER OR FREEZING ETHER BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1):
A SC can cause locking of funds. SmartCheck detects the presence of `payable` directive in such SCs to indicate the locked Ether vulnerability but the converse statements like `transfer`, `send`, and `call.value()()`, which withdraw Ether from the SC, are non-existent.

6) DETECTION OF INTEGER DIVISION BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): SmartCheck detects the division operator in a binary operation based upon numerical values.

7) DETECTION OF DENIAL OF SERVICE DUE TO OUT-OF-GAS (OR WALLET GRIEFING (SECTION VIII-B3)) BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): SmartCheck detects this vulnerability in the context of a `for` or

`while` loop accessing all elements of an extensively large array, and the body of the loop invokes a function [23]. Array traversal can trigger an out-of-gas exception if the gas spent in the traversal exceeds the block GASLIMIT.

8) DETECTION OF TIMESTAMP DEPENDENCE VULNERABILITY BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): SmartCheck detects timestamp by detecting Blockchain environment variable `now` (deprecated). But, again, this can lead to false-negative cases because SC can use variables containing the value of `Blockchain.timestamp`, which can easily fool the tool.

9) DETECTION OF tx.origin VULNERABILITY BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): `tx.origin` represents the originator of the transaction, which is always the initial caller. Hence, if the SC uses the condition: `tx.origin == owner` for authentication, SmartCheck declares the SC vulnerable. The above condition is always true because `owner` and `tx.origin` point to the same address. SmartCheck searches the pattern `tx.origin`.

10) DETECTION OF TYPE INFERENCE VULNERABILITY (USE OF DEPRECATED **var**) BY QUERY LANGUAGE

1) **SmartCheck** (Section III-A8 and IX-A1): SmartCheck uses the assignment statement pattern with `var` (deprecated) on the left of the assignment operator and an integer on the right.

TABLE 18. The vulnerabilities detected by variable analysis tools developed for SCs.

Vulnerability	Slither [74]	Osiris [77]
Reentrancy	✓	
Integer Overflow/Underflow		✓

X. VARIABLE ANALYSIS APPROACHES

Taint Analysis and Static Single Assignment help in program analysis. Taint analysis checks the program variables for boundary values. Similarly, a static single assignment analyzes the variables in intermediate representation for a single assignment and definition before use. The assignment simplifies control flow and data flow analysis [140] such that every variable is allotted a value only once and is defined before its use.

Table 18 provides a brief list of vulnerabilities detected by variable analysis tools. Table 19 incorporates features of variable analysis tools, while thorough details of detection techniques are listed below:

A. GENERAL TECHNIQUES OF VULNERABILITY DETECTION BY ANALYSING VARIABLES

1) STATIC SINGLE ASSIGNMENT BASED TECHNIQUE BY SLITHER (SECTION III-A9), FEIST ET AL. [74]

Slither retrieves important information such as the inheritance graph, control flow graph, and the programming expressions from the Abstract Syntax Tree and converts this information into an intermediate representation. Next, it performs a single static assessment to analyze read/write variables, data dependency, and unprotected functions. Static single assessment requires a one-time variable assignment and definition before usage. Finally, the analyzed code helps in detecting reentrancy, owner authorization, unprotected functions, and shadowed variables (SWC-19).

2) TAINT ANALYSIS BY OSIRIS (SECTION III-A13), TORRES ET AL. [77]

External data can be malicious. SCs receive external data through external functions, which store it in a space known as `msg.data` or calldata. CALLDATACOPY copies this data into memory, and CALLDATALOAD copies calldata into the stack. In addition to stack and memory, calldata interacts with instructions like SSTORE, JUMPI, CALL, and RETURN. Hence, OSIRIS's taint component designates the storage location of calldata as "Sources" and the above instructions as "Sinks" because the instructions access storage (e.g. SSTORE), send Ether (e.g., CALL), and manipulate the execution path (JUMPI). However, Taint Analysis only focuses on the executed instructions related to stack memory and storage. It identifies the operand of the instruction, and if the operand influences any other data, then the impacted data also becomes tainted.

In contrast, another component of OSIRIS, i.e., integer error detection, checks the integer bugs. For instance, the integer error detection component checks integer overflow if the symbolic analysis executes ADD or MUL. OSIRIS also performs width conversion checks if the symbolic analysis executes AND or SIGNEXTEND instruction. Width conversion check is a method for overflow detection such that an arithmetic operation creates a value beyond the range represented using a given number of digits [141]. An example of SIGNEXTEND instruction is given in connection with -28:

Binary value = 00011100
1's complement = 11100011
2's complement = 11100100
SIGNEXTEND = 1111111111100100.

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY VARIABLE ANALYSIS TOOLS

1) DETECTION OF REENTRANCY VULNERABILITY BY VARIABLE ANALYSIS

1) **Slither** (Section III-A9 and X-A1): Slither detects reentrancy if the state variable changes after external call.

TABLE 19. Variable analysis tools: General information at a glance.

Tool Name	Citation	Publication Year	Age	Popularity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
Slither	159	2019	3	53	√	76	S
OSIRIS	126	2018	4	31.5	X	3	S

2) DETECTION OF INTEGER BUGS BY VARIABLE ANALYSIS

1) **OSIRIS** (Section III-A13 and X-A2): To detect overflow/underflow, and division bugs, Osiris uses a solver and instructs it to check the boundary requirements such that the operation's result should not exceed the resultant's maximum possible value. This requirement helps detect overflow related to ADD, MUL, and underflow related to SUB instructions. The solver discards DIV and SDIV if the divisor is zero. Similarly, the solver renders the truncation operation vulnerable if the destination operand's size is smaller than the source operand. However, Osiris detects signedness bugs using the Molnar approach, Molnar et al. [142].

XI. CONTROL FLOW GRAPH (CFG)-BASED APPROACH

We found several static analysis tools incorporating CFG for analyzing the bytecode of SCs, but Mythril uses CFG specifically for vulnerability detection. Table 17 lists the vulnerabilities detected by CFG tools. We can't find the citations of Mythril's research-related article [143], hence we can't evaluate its popularity factor. However, Table 20 highlights features of CFG tools, while thorough details of vulnerability detection techniques are listed below:

A. GENERAL TECHNIQUE OF VULNERABILITY DETECTION BY TOOLS ANALYZING CFG

1) CFG-BASED TECHNIQUE OF MYTHRIL (SECTION III-B10), [105]

Mythril uses LASER Ethereum to create a CFG that helps retrieve program states to prove specific properties to detect reentrancy and other vulnerabilities. LASER Ethereum, Mueller et al. [144], is a symbolic interpreter that works on SC's bytecode and returns the SC's state, such as account information, program counter, and local EVM client's stack information.

2) CFG-BASED TECHNIQUE OF SCOMPILE (SECTION III-A11), CHANG ET AL. [58]

sCompile uses a combination of CFG, symbolic execution, and SMT solver-based techniques for vulnerability detection. sCompile first uses CFG to list program paths. These program paths then help to identify monetary paths (or critical paths, Section III-A11). sCompile ranks the critical paths by their criticalness score. Finally, sCompile uses symbolic analysis to filter out the less critical path. sCompile's symbolic execution engine utilizes the capabilities of Z3 SMT solver.

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY ANALYSING CFG

1) REENTRANCY DETECTION BY CFG

1) **Mythril** (Section III-B10 and XI-A1): Mythril detects message calls other than invoking `send` and `transfer` functions because they consume gas amounting to only 2300 Wei. Thus, Mythril analyzes the CFG of external calls to untrusted addresses and generates a reentrancy warning if state changes occur after the CALL invocations.
2) **sCompile** (Section III-A11 and XI-A2): sCompile detects reentrancy by applying a combined approach of analyzing critical paths and respecting the limitation property. The limit property uses a limit variable with an assigned value. Each time the critical path transfers Ether out of the SC, sCompile decrements the limit variable with the transferred amount; when the limit value becomes negative, sCompile generates a violation message.

2) `selfdestruct` DETECTION BY CFG

1) **Mythril** (Section III-B10 and XI-A1): In case of self-destruct vulnerability, Myhril generates alerts if the SELFDESTRUCT opcode is reachable and if the argument of SELFDESTRUCT is writeable.
2) **sCompile** (Section III-A11 and XI-A2): sCompile starts after finding out if SELFDESTRUCT is reachable or not. After that sCompile checks the constraints like the address associated with SELFDESTRUCT using Etherscan. If the address does not exist, sCompile declares it a violation. sCompile also checks other restrictions like block number, date, and owner's address associated with the conditional test statement of SELFDESTRUCT. If no constraints are associated with SELFDESTRUCT, then SELFDESTRUCT is vulnerable.

3) FREEZING ETHER DETECTION BY CFG

1) **sCompile** (Section III-A11 and XI-A2): sCompile, develops a CFG which identifies a node as money-related if its basic block contains the opcode for instructions like CALL, CREATE, DELEGATECALL, or SELFDESTRUCT. One differentiating factor is that sCompile generates a warning message labeling a SC as a "black hole" if sCompile encounters a freezing Ether threat. By "black hole" property, sCompile means that the SC has no path to send the Ether out, but it has one such path, which allows the transfer of Ether into the

TABLE 20. Control flow graph tool: General information at a glance ('*' means information not available).

Tool Name	Citation	Publication Year	Age	Popularity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
sCompile	84	2019	4	21	√	3	S
Mythril	*	*	*	*	√	11	D

SC. For the later one, sCompile checks the `payable` directive.

XII. CHECKING INVARIANT VIOLATION APPROACH

An essential technique for vulnerability detection is to define a set of invariants that should always be true. In computer science, an invariant is a property that should hold to start/stop the execution of some parts of a program (e.g., a loop invariant) [91]. Hence, the invariant violation to initiate a sensitive operation (like Ether transfer) indicates a vulnerability [145]. However, VeriSmart, Section III-A4, extends this approach by detecting hidden invariants. But we could not find sufficient details for specific vulnerability detection techniques. Table 17 lists the vulnerabilities detected by Vultron. Table 21 lists Vultron's features, while thorough details of Vultron's invariant-based detection techniques are listed below:

A. GENERAL TECHNIQUE OF VULNERABILITY DETECTION BY CHECKING INVARIANT VIOLATION

1) INVARIANT-BASED TECHNIQUE OF VULTRON (SECTION III-B1), WANG ET AL. [37]

One common purpose of SCs is sending and receiving Ether; henceforth, SCs share some code in declaring data structures and variables. Thus the commonality of goals and variables can merge to define standard invariants for SC that do not change, and if they change, we can trigger a vulnerability alarm. For example, research related to Vultron, Wang et al. [37], exploits the common invariant notion by creating a balance invariant that requires that the SC's balance before and after a transaction and the sum of all participants' bookkeeping balances remain constant. However, a bookkeeping balance is an intra-contract property (i.e., declared inside the SC using local variables).

2) HIDDEN INVARIANT-BASED TECHNIQUE OF VERISMART (SECTION III-A4), SO, ET AL. [72]

VeriSmart generates the hidden invariant related to the mathematical statements and proves that the invariant holds; otherwise, reprocesses the unproved invariant for refinement.

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUE BY DETECTING INVARIANT VIOLATION

1) REENTRANCY DETECTION BY INVARIANT VIOLATION

1) **Vultron** (Section III-B1 and XII-A1): Vultron uses a balance invariant technique due to which the difference of SC's balance and the sum of all participant's bookkeeping balances should remain constant. Still, if the transaction does not update the bookkeeping balance but the SC's balance changes, it indicates a reentrancy problem because of the difference between the two (i.e. balance invariant) is not constant. Consider the example in Table 22, modified from Wang et al. [37], initially Victim's balance is 10 and the Attacker's balance is zero, henceforth the difference is 10. Next, the attacker donates, increasing Victim's balance to 11 and the Attacker's bookkeeping balance by one, but the difference is still 10. However, the next transaction in step 3 violates the balance invariant because the Victim's balance decreases to 10, but the bookkeeping balance does not change, and hence the difference, i.e., 9, is a violation of invariant property.

2) MISHANDLED EXCEPTION DETECTION BY INVARIANT VIOLATION

1) **Vultron** (Section III-B1 and XII-A1): Vultron detects mishandled exception using transaction invariant concept. Let's suppose the sender SC has no bookkeeping variable, hence Vultron inserts a ghost variable for keeping SC's balance information locally. When the SC performs a transaction, the receiver generates an exception. Despite the exception, the sender updates the value of the ghost variable instead of handling the exception. Thus, the receiver's balance will not change, but the sender's balance (i.e., the ghost variable) will decrease, violating the transaction invariant.

3) INTEGER OVERFLOW DETECTION BY INVARIANT VIOLATION

1) **Vultron** (Section III-B1 and XII-A1): Vultron uses the concept of balance invariant to detect integer overflow as Vultron does with reentrancy detection. Due to overflow/underflow, the balance invariant may violates, which helps in the detection of overflow/underflow. Consider the following line of code:

```
balances[msg.sender] -= amount;
```

if balances[msg.sender] is 1 and the amount is 2 but after the code above executes balances[msg.sender] becomes $2^{256} - 1$, which violates the balance invariant.

4) GASLESS `send` DETECTION BY INVARIANT VIOLATION

1) **Vultron** (Section III-B1 and XII-A1): Vultron proposes transaction invariant to detect gasless `send` as discussed in (Section XII). If the sender SC does not handle the out-of-gas exception due to a failed transfer, the SC balance would decrease as discussed in (Section XII-B2). Still, the receiver won't

TABLE 21. Checking invariant violation tool: General information at a glance.

Tool Name	Citation	Publication Year	Age	Popularity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
Vultron	39	2019	3	13	√	4	D
VeriSmart	92	2019	3	30	X	1	S

TABLE 22. Reentrancy example modified from Wang et al. [37].

Victim's Balance	Attacker's Balance (i.e., the sum of all Bookkeeping balances)	Difference
1) 10	0	10
2) 11	1 (execution of the donate function)	10
3) 10	1 (line#9 does not execute)	9 (violation)

receive any Ether. Hence the authors use a ghost variable (Section XII-B2) in conjunction with transaction invariant.

XIII. RUNTIME MONITORING AND VALIDATION APPROACHES

We found only Sereum employing Runtime Monitoring and Validation approach by using storage data as an execution state. However, based upon Wikipedia's definition [146], we can call Sereum's approach just runtime verification. Table 17 provides list of vulnerabilities detected by Sereum. Table 23 embeds common features of Sereum, while thorough details of detection techniques are listed below:

A. GENERAL TECHNIQUE OF VULNERABILITY DETECTION BY VALIDATING PROGRAM EXECUTION STATE

1) RUNTIME MONITORING AND VALIDATION BASED-TECHNIQUE OF SEREUM (SECTION III-B2), RODLER ET AL. [54]

Sereum uses taint analysis and locking of write access to the storage for reentrancy detection. Sereum waits for tainted information to reach the taint sink from the taint source and then invokes the reentrancy detection module. The detection module interprets the SSTORE's write operation and checks if the address has write- permissions or not. If the write fails, Sereum alarms reentrancy attack and aborts execution.

However, a thorough study of the Sereum-related research highlights the fact that Sereum focuses on the detection of inconsistent states, which is not necessary for reentrancy detection. Because reentrancy means re-entrance, it doesn't need to cause state inconsistency. For instance, the code in Listing 9 causes state inconsistency (i.e., balance becomes negative) because the program does not deduct the transferred amount with the balance, i.e., line#17 never executes. However, the code shown in the tutorial in [147] can also cause reentrancy where the attacker can re-enter the SC and transfer the unguarded funds using the transfer statement shown in line#7 of the victim SC, even though there is no state inconsistency issue.

```
1  //SPDX-License-Identifier: UNLICENSED
2  pragma solidity ^0.8.10;
3  contract ModifiedMalicous{
4    ModifiedBank mb;
5    constructor(address payable addressOfBank,
         uint amount) public{
6      mb = ModifiedBank(addressOfBank);
7      mb.withdraw(amount);
8    }
9    fallback() external payable{
10     mb.withdraw(msg.value);}}
11 contract ModifiedBank{
12   mapping(address=>uint)bal;
13   //more
14   function withdraw(uint _amount) payable
         public returns(bytes memory message) {
15     if(bal[msg.sender]>= _amount){
16       (bool success, bytes memory
             returnsMessage) = msg.sender.call{
             value: msg.value, gas: 5000}("");
17       bal[msg.sender]-=_amount;
18       /*more*/}}
19   fallback() external payable{}}
```

LISTING 9. Example of a reentrancy program modified from [51].

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY VALIDATING PROGRAM EXECUTION

1) REENTRANCY DETECTION BY VALIDATING PROGRAM EXECUTION STATE

1) **Sereum** (Section III-B2 and XIII-A1): The tool locks the write access to storage addresses to influence control flow decisions. When a SC attempts to write to a locked address after the external call, Sereum bangs due to a reentrancy attack and aborts execution. Sereum implements this logic because of its dynamic analysis capabilities, particularly runtime analysis. Replication of the runtime environment requires connecting with the public Ethereum network, re-executing transactions, and skipping blocks requiring high execution time, causing Denial of Service.

XIV. CUSTOMIZATION-BASED APPROACHES

By Customization, we mean altering the operating environments like Ethereum Mainnet and Ethereum Clients. Tools use Ethereum modules for this purpose. Ethereum Mainnet is the public Ethereum Blockchain where the actual transactions occur. Ethereum Client is the downloadable software that acts

TABLE 23. Runtime monitoring and validation tool: General information at a glance.

Tool Name	Citation	Publication Year	Age	Popularity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
Sereum	132	2018	4	33	√	1	D

as a node for Ethereum Blockchain. Table 24 provides a brief description of Ethereum modules-based tools, while thorough details of detection techniques are listed below:

A. GENERAL TECHNIQUE OF VULNERABILITY DETECTION BY DEVELOPING CUSTOMIZATION MODULES

1) CUSTOMIZATION-BASED MODULE FOR CONCRETE VALIDATION BY MAIAN (SECTION III-A15), NIKOLIĆ ET AL. [57]

A private fork of Ethereum works under the concrete validation component of Maian. It comes into play when the symbolic analysis component discharges its output. The output consists of concrete values, and the private fork of Ethereum uses concrete values to validate the SC's security property.

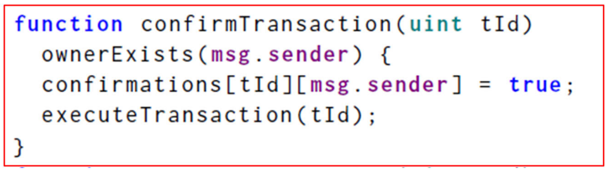
2) CUSTOMIZATION-BASED MODULE FOR TAINT TRACKING BY EASYFLOW (SECTION III-B4), GAO ET AL. [99]

Initially Taint Analysis component divides the SCs into safe, overflow, and potential overflow. SCs declared as safe by taint analysis are not subject to further processing. Otherwise, EasyFlow declares an arithmetic operation based upon arguments as potential overflow and marks the arguments as taints even if the result of the operation does not overflow. Easyflow tests potential overflows by re-executing them using a series of invocations. On the other hand, the SCs recognized as overflow by taint analysis or SCs incorporating the SafeMath library are compared with protection patterns to identify them as protected or manifested overflows. The authors developed an extended go-Ethereum module for tracking taints during function executions.

B. SPECIFIC VULNERABILITY DETECTION TECHNIQUES BY DEVELOPING CUSTOMIZATION MODULES

1) GREEDY (OR LOCKING FUND) DETECTION BY ETHEREUM MODULES

1) **Maian** (Section III-A15 and XIV-A1): In connection with Frozen Ether, the novelty of Maian is that it labels the Frozen Ether SCs as "Greedy". Maian-related research shows that sometimes it is difficult to detect true positive cases in a greedy SC situation. One reason is that the release of Ether may require a confirmation from the majority of owners, which in turn requires the majority of owners, for example, to execute the confirmation granting function as in Figure 3. Otherwise, Maian uses a traditional approach, i.e., if the Ether transfer instructions like CALL are unreachable but the SC accepts Ether, then the SC is greedy.



```
function confirmTransaction(uint tId)
  ownerExists(msg.sender) {
  confirmations[tId][msg.sender] = true;
  executeTransaction(tId);
}
```

FIGURE 3. ConfirmTransaction(..), Majority of owners confirm transaction, [57].

2) SELFDESTRUCT (OR SUICIDAL) DETECTION BY ETHEREUM MODULES

1) **Maian** (Section III-A15 and XIV-A1): Due to fund transfer capability, Maian, detects SELFDESTRUCT by building the transaction trace such that the last transaction is reachable to SELFDESTRUCT opcode.

3) PRODIGAL (OR LEAKING ETHER) DETECTION BY ETHEREUM MODULES

1) **Maian** (Section III-A15 and XIV-A1): Maian detects prodigal SCs if the CALL is reachable, the transfer amount is non-zero, and the recipient is an arbitrary account. However, Maian also checks success if the process ends with STOP or RETURN, which is unnecessary because the return value of the transaction determines the success.

4) INTEGER OVERFLOW DETECTION BY ETHEREUM MODULES

1) **EasyFlow** (Section III-B4 and XIV-A2): EasyFlow checks the bytecode associated with arithmetic instructions like ADD, SUB, MUL, DIV, and EXP to analyze the values and protection patterns like conditional statements and the SafeMath library. Then, when EasyFlow suspects a check for potential overflow, EasyFlow creates a transaction consisting of the original `msg.value` and an integer value between 0 to MAX_UINT256 as input data. Finally, these transactions are re-executed for all the possible combinations of integer values to test the presence of overflow.

XV. MULTI-LAYER CLASSIFICATION OF TOOLS' TECHNIQUES

We provide four classes of SC tools in Figure 4 and divide each class into three parts: (1) Higher Level Techniques, which specifies the top-level software engineering techniques used by a SC's tool to initiate the vulnerability detection process (2) Implementation: supporting modules and tools, specifies the user-defined modules and/or built-in tools to provide the input values and/or the environment necessary

TABLE 24. Ethereum modules-based tools: General information at a glance.

Tool Name	Citation	Publication Year	Age	Popularity Factor	Additional Inputs	Detection Cnt	Analysis Type (S/D)
Maian	430	2018	4	107.5	√	3	S
EasyFlow	27	2019	3	9	√	1	D

TABLE 25. The vulnerabilities detected by ethereum module-based tools developed for SCs.

Vulnerability	Maian [57]	EasyFlow [99]
Frozen Ether	✓	
selfdestruct	✓	
Leaking Ether	✓	
Integer (Overflow/Underflow)		✓

to execute the newly developed tool and/or to perform the reachability analysis, (3) Tool(Vulnerability Detected), specifies the tool name and the vulnerabilities detected by the developed tool.

Class C1 focuses on the CFG-based seven vulnerability detection tools incorporating static analysis. As a first step, all the tools perform symbolic execution apart from Securify (Section III-A10), which uses static single assignment. After that, Securify follows a pure AI-based approach incorporating Datalog analysis which uses the Souffle engine to reason about the vulnerabilities by computing fixed-point (Section III-A10). On the other hand, tools like Oyente (Section III-A12), sCompile (Section III-A11), teEther (Section III-A6), Gasper(Section III-A6) utilize SMT solvers like Z3 to solve integer or boolean-related constraints. But tools like Osiris (Section III-A6) and Clairvoyance (Section III-A6) subject the SC to Taint Analysis (Section III-A13) and/or path-protective technique (i.e., PPT, Section I-A) before using solvers.

Class C2 presents eight SC tools incorporating static analysis based on intermediate representations. However authors of constructive ECF (Section III-A28) do not elaborate on its intermediate representation technique and focus only on Z3 solver for vulnerability detection. Vandal (Section III-A5) and MadMax (Section III-A16) apply AI-based processing to their intermediate representations using Datalog analysis. AI analysis helps MadMax to generate vulnerability detection results, but Vandal further applies the Souffle engine to highlight vulnerabilities in C++. Similarly, other tools also process the intermediate representation. For instance, SmartScopy (Section III-A7) retrieves the intermediate representation through Vandal's engine and processes it using Summary-based Symbolic evaluation (Section III-A7), a faster version of symbolic execution. SmartScopy generates an attacker to help with reentrancy and batch overflow detection. For reentrancy, the attacker generates a callback. For batch overflow, the attacker generates a huge value which, when multiplied with the SC's balance (for example), causes the result to overflow. SmartScopy uses SMT solver to create concrete values.

On the other hand, EtherTrust(Section III-A25) and Zeus (Section III-A1) use Horn Clause representation (Section III-A1) in two different ways. EtherTrust uses the representation as an Intermediate representation, which is subject to reachability analysis. Zeus performs Horn Clause verification on the output obtained by formal verification of LLVM's intermediate representation. SmartCheck (Section III-A8) starts its processing using an XML parse tree as an intermediate representation for Solidity SC. The parse tree then helps in vulnerability detection using XPATH[xpa] queries. Slither (Section III-A9) starts with its intermediate representation, i.e., SlitherIR, which is subject to a static single assignment approach (Section III-A9). However, the authors do not provide much detail about SlitherIR.

Class C3 incorporates four SC tools applying various static analysis approaches. For instance, FVF* (Section III-A23) converts SC and its EVM representations (i.e., Solidity and bytecode versions of SC) into F* and then checks their equivalence by applying formal verification. ITP (Section III-A19) also performs formal verification by first generating EVM definition in LEM (Section III-A19) and extracting it using OCAML (Section III-A19) and Isabelle/HOL (Section III-A19). Then ITP confirms the OCAML implementation using a VM Test [81], holding a database of EVM implementations. On the other hand, Ethainter (Section III-A26) uses the information flow rules (Section III-A26) to focus on guard conditions to sanitize information. The Datalog engine implements the Information flow rules. Taint Analysis (Section III-A13) tracks data input during SC's execution. If taint propagates through storage, then a sizable number of SC's are flagged vulnerable. In this situation, exploits override a guard by overwriting the owner variable in the storage, which allows the Ethainterkill to destroy the vulnerable SCs. But DappGuard (Section III-A27) adopts knowledge acquisition for implementing AI. Rules Engine uses the knowledge for vulnerability detection. DappGuard also applies the Oyente engine for vulnerability detection, which requires symbolic execution.

Class C4 discusses eleven SC tools based on dynamic analysis. We will first examine the multi-level schemes for six of them based on intermediate representation. For instance, Confuzzius (Section III-B5) and sFuzz (Section III-B6) embed evolutionary fuzzing through a Genetic algorithm (Section III-B5), but Confuzzius can switch to hybrid fuzzing if the fuzzer cannot solve the branching problem. sFuzz uses a feedback-guided adaptive fuzzing approach to handle the missed branches and generates new test cases by applying the existing ones. sFuzz repeats the process until the termination condition occurs.

On the other hand, ILF (Section III-B7) incorporates probabilistic and learning-based fuzzing. Probabilistic fuzzing assigns probabilities to integer arguments and functions. Learning-based fuzzing first uses symbolic analysis to generate inputs for learning the fuzzer using a Neural Network. The learned fuzzer then uses inputs to test the unseen program. Regaurd's (Section III-B3) testing involves randomized fuzzing which injects random data (i.e., random transactions) into the SC. Regaurd records execution traces and feeds the traces into reentrancy automata for reentrancy detection. EtherRacer (Section III-B9) applies dynamic symbolic execution after randomized fuzzing to generate outputs of each event separately. Another significant approach is the partial order reduction which helps EtherRacer to prevent state explosion problems [148]. However, EtherRacer uses partial order reduction to check if reordering a set of functions yields the same output. Linearizability optimizer in EtherRacer finds the matching `call`/`return` pairs in the called functions. On the other hand, the happens-before optimizer catches event-ordering bugs. Finally, we discuss contractFuzzer (Section III-B8), which uses fuzzing to produce random data according to function arguments. Next ContractFuzzer uses the generated inputs to bombard the function invocations arbitrarily. Instrumentation [45] is a technique to record the opcode execution by providing an interpreter-based function for the opcode in the implementation.

Now we will discuss CFG-based dynamic analysis tools related to SCs. For instance, Mythril (Section III-B10) performs symbolic and concrete analysis to test the SCs. Afterward, Mythril uses the Laser Ethereum to represent the execution of SC in propositional logic as a combination of states and formulas. Mythril then uses the Z3 solver with the Python script [105] to check if the proposition logic is satisfactory. If an exception occurs, it indicates errors like dividing by zero, asserting violations, writing to storage, etc. Another tool, i.e., Maian III-A15), uses a custom EVM to facilitate symbolic execution. Once the symbolic execution flags a vulnerability, Maian uses the concrete value returned by the symbolic analysis component to validate the result for true positive detection.

Now we discuss dynamic analysis tools incorporating miscellaneous techniques. For example, EasyFlow (Section III-B4) uses taint analysis (Section III-A13) to monitor overflow vulnerability in mathematical statements and transactions. EasyFlow checks that both operands are not tainted, for instance, in a multiplication step. Also, EasyFlow analyzes the multiplication result such that the result doesn't overflow. EasyFlow declares a potential overflow in a statement if the result doesn't overflow, but both operands are tainted. The go-Ethereum component also helps EasyFlow

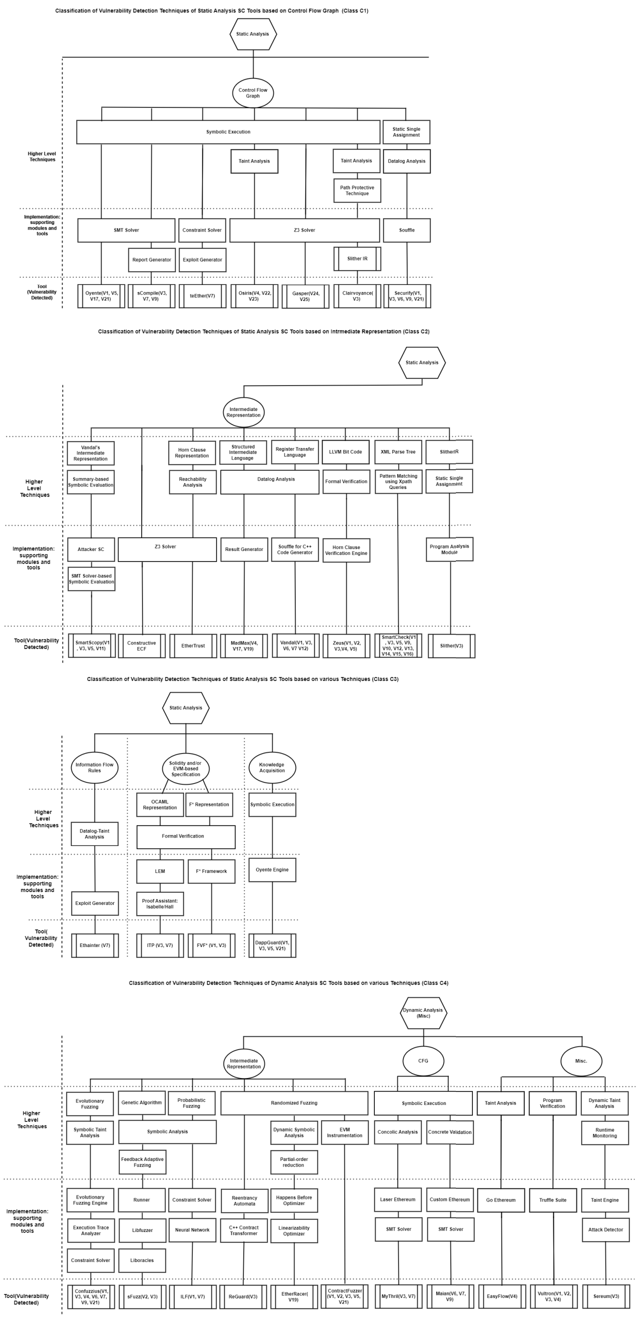


FIGURE 4. Classification of SC's vulnerability detection techniques.

in tracing the tainted data [99]. Vultron (Section III-B1) focuses on program verification. But instead of using a set of properties, Vultron uses the concept of invariants. The advantage is the translation of invariants into test oracles (i.e., a mechanism for analysis of test outputs, [149]) and assertions in the truffle environment. For instance, each SC uses a balance variable (i.e. `bal` as per [37], section III-B1), which one can extend to create a balance invariant to test SC's code. Sereum starts taint tracking (Section III-A13) by allocating shadow memory (i.e., a protected memory area, [150])

TABLE 26. Vulnerability names and codes.

Vulnerability Names	Codes	Vulnerability Names	Codes
Unchecked send	V1	Failed send	V2
Reentrancy	V3	Arithmetic overflow/underflow (Arithmetic Bugs)	V4
Timestamp dependence	V5	Unsecured balance	V6
Selfdestruct	V7	Batchoverflow	V8
Locked Ether	V9	Division operator	V10
Multiplication operand	V11	Tx.origin	V12
Deprecated var	V13	DoS: unexpected throw	V14
DoS: out-of-gas	V15	Library keyword	V16
Call Stack Depth	V17	Wallet griefing	V18
Event order bugs	V19	Unbounded mass operation	V20,
Transaction ordering dependence	V21	Truncation bugs	V22
Signedness bugs	V23	Dead Code	V24
Opaque predicate	V25		

to store storage variables that can influence control flow decisions (i.e., due to instructions like JUMPI). Once the execution of the SC ends, Sereum forwards the addresses of the recorded storage variables to the attack detection unit to identify the vulnerabilities.

XVI. FUTURE RESEARCH DIRECTIONS

This paper presents a survey that reviewed 41 smart contracts' (SCs') tools, explicitly focusing on 30 vulnerability detection tools (VDTs). The thorough discussion of vulnerability detection techniques of 30 tools makes this work the first comprehensive survey on SC Vulnerability detection tools. The future work on vulnerability detection tools can be expanded in the following three directions:

A. DEVELOPMENT OF ADDITIONAL VULNERABILITY DETECTION TOOLS

- *Reusing Modules and Code Compactness.* Tool developers are focusing on incorporating EVM bytecode (as in the case of Vultron) or modifying EVM clients (like Geth as in the case of EasyFlow by Gao et al. [99], and Sereum by Rodler et al. [54]). Tool developers can also incorporate hevm (i.e., symbolic analysis) [151] which supports version 8 of Solidity for unit testing and debugging and has an additional symbolic execution module. Other approaches needing research attention are dynamic taint analysis (DTA) (again as in Sereum, Rodler et al. [54]) or Summary based Symbolic evaluation for the generation of attacker programs as in the case of SmartScopy, Feng et al. [31]). Creating lightweight tools would require less operational cost, speeding up the analysis process. In the context of reusing modules, Oyente is an excellent success for academia.
- *Formal Verification Tools.* Formal verification tools resort to correct proof of the programs (i.e., SCs). Apart from FVF*, tools like KEVM and ITPs demonstrated their formalism only on one or two SCs, which requires more justification in the future for better understanding.
- *Testing of Bug Detection Tools.* Solidifi [152] is a unique tool that performs analysis of static analysis tools. However, it is currently available for only six static analysis tools. But one can extend this idea to dynamic analysis tools.
- *Expanding the scope of Development Tools.* Solidity, Javascript, and Python are spearheading the Ethereum tools' development. All these languages benefit from the availability of Ethereum libraries. A recent addition is 'Web3j for Java', which is very useful because java programming is an essential second or third-year course in the CS undergraduate programs. Still, the C language, a driving force in developing several programming languages, is devoid of the Ethereum library [153]. Hence future research can focus on developing a C/C++ library for Ethereum SCs. Apart from this, we focussed on 20 vulnerabilities, but zak100 et al. [154] provides links to several new SC vulnerabilities. Future research can develop tools to explore these vulnerabilities.

B. CHALLENGES WITH TESTING OF SMART CONTRACT

- *Speeds up the Testing Process.* Due to the increase in the number of deployed SCs, the testing of SCs is becoming more challenging. Thus, one can expect that future research will try to improve the performance of testing activities or handle the issue using parallel testing of SCs.
- *Enhancing the Testing Domain.* Tools can also focus on Viper (another language to code SCs) to compare Viper's performance with Solidity. Recently launched Remix IDE supports both Viper and Solidity. This development is an indication of increasing attention to viper-based SCs by EVM. Dynamic analysis approaches execute the programs. Hence their results are trustworthy. However, both static and dynamic analysis approaches can apply techniques to confirm the results as done in [107] with the help of assertions. Machine learning approaches can also be adopted using the training techniques as applied in several current research.
- *Empirical Analysis and Comparison of the performance of Vulnerability Detection and Testing Tools.* Testing tools are necessary for the survival of SC, but the independent research shows only four endeavors as in: [155], Parizi et al. [156], [157], [158] to compare these tools.

There is a lot of room for extension of this research, particularly involving new tools like Fuzzing tools.

- *Database of Malicious SCs for controlled experimentation.* It is worth doing quantitative analysis [157], and the obtained results can be a source of future actions. A signature-based profiling technique can help to capture malicious SCs and their features/patterns.
- *Mitigation Approaches.* Work in [159] recommends using SafeMath library to mitigate overflow/underflow and divide by zero errors. However, the use of `transfer` instead of `call` to mitigate reentrancy is no longer recommended; instead, developers should protect `call` transfers properly.

C. ENHANCEMENT OF SOLIDITY, EVM AND THREAT DETECTION TECHNIQUES

- *Unpredictable Functionality of Remote SCs:* Threats like Denial of Service and "Gasless send" are a source of loss of Ether because FF of remote SC throws or incorporates state changes. DappGuard-related research, Cook et al. [90], provides some solutions for this problem, but DappGuard research does not offer any practical implementation. Future research can handle this problem.
- *Handling SC with Vulnerable FF:* Users can't send Ether to a SC whose FF function reverts. Work in [160] discusses a solution for this problem using `selfdestruct` opcode. However, the above work recommends a pre-commit scheme that later submits the details. Future research can perform comprehensive research on the implementation of pre-commit schemes.
- *Enhancing the Scope of ByteCode-Related Tools:* SmartCheck is a Solidity-based tool that detects the most significant number of threats. However, extending the capabilities of bytecode-related tools is still open to research. This type of tester would fill the demands of a broader community because EVM uses bytecode, so the tool will not require translation for executing SCs.
- *Event Ordering Bugs:* Surveyed literature reports only one research related to EthRacer, Kolluri et al. [30] for detecting event ordering (EO) bugs (Section III-B9) at the time of writing of this paper. EthRacer uses EO bugs for reentrancy detection. Future research can use this concept to detect vulnerabilities like miners' manipulation in transaction ordering dependence bugs. Thus EO bugs need more research to highlight how this concept can be utilized to detect other vulnerabilities and bugs.
- *Batch Overflow Vulnerability:* This vulnerability occurs when the multiplication operation overflows. SmartScopy detects this vulnerability using the automatic synthesis of an attacker program. Future research can opt for direct detection of this bug (i.e., without using an attacker), which would be faster.
- *Automation of Minning Process:* This has become a neglected area. That's why the Blockchain concept cannot fully penetrate the E-commerce realm (i.e., the retail stores). Credit and debit cards dominate this sector. One solution is to design the SC, which can automate the process of mining [161], [162]. Future research can improve on this.
- *Enhancement to ZKP for 3rd party privacy in E-commerce:* Currently, E-commerce systems are using ZKP for maintaining 3rd party privacy. However, this concept has not yet been applied to Ethereum SC and hence traders hesitate to use public Blockchains because they leak out transaction information. Thus ZKP implementation for Ethereum can be a domain for future research.
- *Follow-up on Gasper Patterns:* Work in Chen et al. [85] detects some software patterns which consume a high amount of gas. However, the above research does not specify the Solidity compiler version which creates confusion about this work. Our research identifies them as "Gasper Patterns" (Section III-A21). Thus, validation of Gasper patterns using the latest Solidity compiler can pave the way for highlighting the current advancements of Solidity.

XVII. CONCLUSION

Cryptocurrency hacks have become commonplace in the last couple of years. Some significant losses exploit bugs in SCs. Automated vulnerability detection tools can help to prevent such situations. Otherwise, unabated hacks can endanger the growth of SCs despite their versatility. Hence, SCs' progress hook up to the tools to combat security threats. Briefly, SCs' strength depends upon three crucial players: Solidity, the developer, and the EVM. Developers must test [163] the SC through testing tools before launching them on the Blockchain. Solidity is continuously in the renovation phase, which is a good sign. And EVM can respond by keeping track of the erroneous SCs so that attackers cannot succeed in victimization again. Apart from the above said domains, the role of academia is also vital in providing positive feedback and highlighting vulnerable code, which can be helpful for developers, software engineers, and security analysts, as we have done in this survey paper. Due to the importance of tools, all aspects of our survey focus on tools. Along with data collection, the research title and our research questions also focus on vulnerability detection tools.

We answer our research question 1, Section (I-C), in Section (III) by discussing the working of thirty static analysis tools and ten dynamic analysis tools. Similarly, for research question 2, Section (I-C), we provide the list of vulnerabilities detected by each tool in figure 4 in connection with our multi-level detection techniques. For question 3, we provide a thorough discussion on vulnerability detection techniques of our selected tools in Section (V) to Section XIV. Our last question relates to the classification of tools' vulnerability detection techniques, and we refer the readers to figure 4 and its related Section (XV). Our future work will focus on developing dynamic analysis-based tools.

REFERENCES

[1] Z. Voell. (Sep. 2021). *Ethereum Classic Hit By 3rd 51% Attack in a Month.* [Online]. Available: https://www.coindesk.com/markets/2020/08/29/ethereum-classic-hit-by-third-51-attack-in-a-month/

[2] (Feb. 2021). *Blockchain Risks Every Cio Should Know*. [Online]. Available: https://101blockchains.com/blockchain-risks/

[3] A. Berwick and T. Wilson. (2022). *How Crypto Giant Binance Became a Hub for Hackers, Fraudsters and Drug Traffickers*. [Online]. Available: https://www.reuters.com/investigates/special-report/fintech-crypto-binance-dirtymoney/

[4] J. Mirza. (2022). *Pakistan to Investigate Crypto Exchange Binance in an Alleged $100 Million Scam*. [Online]. Available: https://cryptoslate.com/pakistan-to-investigate-crypto-exchange-binance-in-an-alleged-100-million-scam/

[5] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 454–469.

[6] Z. A. Khan and A. S. Namin, "The applications of blockchains in addressing the integration and security of IoT systems: A survey," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2021, pp. 2421–2426.

[7] N. I. Haque, M. Ngouen, M. A. Rahman, S. Uluagac, and L. Njilla, "SHATTER: Control and defense-aware attack analytics for activity-driven smart home systems," in *Proc. 53rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2023, pp. 1–13.

[8] X. Yi, J. Wu, G. Li, A. K. Bashir, J. Li, and A. A. AlZubi, "Recurrent semantic learning-driven fast binary vulnerability detection in healthcare cyber physical systems," *IEEE Trans. Netw. Sci. Eng.*, vol. 10, no. 5, pp. 2537–2550, Aug. 2023.

[9] D. Setó-Rey, J. I. Santos-Martín, and C. López-Nozal, "Vulnerability of package dependency networks," *IEEE Trans. Netw. Sci. Eng.*, vol. 10, no. 6, pp. 3396–3408, Aug. 2023.

[10] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: Finding reentrancy bugs in smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng., Companion (ICSE-Companion)*, May 2018, pp. 65–68.

[11] B. Gao, S. Shen, L. Shi, J. Li, J. Sun, and L. Bu, "Verification assisted gas reduction for smart contracts," in *Proc. 28th Asia–Pacific Softw. Eng. Conf. (APSEC)*, 2021, pp. 264–274.

[12] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in Ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–27, Oct. 2018.

[13] Z. A. Khan, "Cycle detection in signature images," *Int. J. Sci. Eng. Res.*, pp. 1327–1332, Jan. 2013.

[14] D. S. Connelly. (Jan. 2020). *Smart Contract Vulnerabilities on the Ethereum Blockchain: A Current Perspective*. [Online]. Available: https://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=6512&context=open_access_etds

[15] J. Li, Z. Zhao, Z. Su, and W. Meng, "Gas-expensive patterns detection to optimize smart contracts," *Appl. Soft Comput.*, vol. 145, Sep. 2023, Art. no. 110542.

[16] *Deprecated/Historical*. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/attacks/deprecated/

[17] *8 Most Common Smart Contract Vulnerabilities*. [Online]. Available: https://losslessdefi.medium.com/8-most-common-smart-contract-vulnerabilities-1ed7dc03080

[18] Z. Oualid. (Jul. 2022). *Smart Contract Shadowing State Variable Vulnerability*. [Online]. Available: https://www.getsecureworld.com/blog/smart-contract-shadowing-state-variables-vulnerability/

[19] D. Han, Q. Li, L. Zhang, and T. Xu, "A smart contract vulnerability detection model based on syntactic and semantic fusion learning," *Wireless Commun. Mobile Comput.*, vol. 2023, pp. 1–12, Feb. 2023.

[20] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Proc. Int. Conf. Princ. Secur. Trust*. Cham, Switzerland: Springer, 2017, pp. 164–186.

[21] H. Guo and X. Yu, "A survey on blockchain technology and its security," *Blockchain: Res. Appl.*, vol. 3, no. 2, Jun. 2022, Art. no. 100067.

[22] H. Rameder, M. di Angelo, and G. Salzer, "Review of automated vulnerability analysis of smart contracts on Ethereum," *Frontiers Blockchain*, vol. 5, Mar. 2022.

[23] Z. A. Khan and A. S. Namin, "Ethereum smart contracts: Vulnerabilities and their classifications," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2020, pp. 1–10.

[24] J. Fei, X. Chen, and X. Zhao, "MSmart: Smart contract vulnerability analysis and improved strategies based on smartcheck," *Appl. Sci.*, vol. 13, no. 3, p. 1733, Jan. 2023.

[25] S. Holler, S. Biewer, and C. Schneidewind, "HoRStify: Sound security analysis of smart contracts," in *Proc. IEEE 36th Comput. Secur. Found. Symp. (CSF)*, Jul. 2023.

[26] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, and X. Zhang, "Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting," Tech. Rep., 2023.

[27] L. Jin, Y. Cao, Y. Chen, D. Zhang, and S. Campanoni, "ExGen: Cross-platform, automated exploit generation for smart contract vulnerabilities," *IEEE Trans. Depend. Secure Comput.*, vol. 20, no. 1, pp. 650–664, Jan. 2023.

[28] Z. A. Khan and A. S. Namin, "Dynamic analysis for detection of self-destructive smart contracts," in *Proc. IEEE 47th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jun. 2023, pp. 1093–1100.

[29] 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 *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Dec. 2020, pp. 1029–1040.

[30] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2019, pp. 363–373.

[31] Y. Feng, E. Torlak, and R. Bodík, "Precise attack synthesis for smart contracts," 2019, *arXiv:1902.06067*.

[32] R. Pise and S. Patil, "A survey on smart contract vulnerabilities and safeguards in blockchain," *Int. J. Intell. Syst. Appl. Eng.*, vol. 10, no. 3, pp. 1–16, Dec. 2022.

[33] S. Munir and W. Taha, "Pre-deployment analysis of smart contracts—A survey," Tech. Rep., 2023.

[34] P. Praitheeshan, L. Pan, J. Yu, J. K. Liu, and R. Doss, "Security analysis methods on Ethereum smart contract vulnerabilities: A survey," 2019, *arXiv:1908.08605*.

[35] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, and X. Luo, "Security threat mitigation for smart contracts: A survey," Tech. Rep., 2023.

[36] 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," 2018, *arXiv:1809.03981*.

[37] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "VULTRON: Catching vulnerable smart contracts once and for all," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., New Ideas Emerg. Results (ICSE-NIER)*, May 2019, pp. 1–4.

[38] 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," 2018, *arXiv:1801.04032*.

[39] J. M. Potter. (May 2018). *The Problem With Solidity*. [Online]. Available: https://medium.com/@XBY_Today/the-problem-with-solidity-be7e6c277a58

[40] C. Lal and D. Marijan, "Blockchain testing: Challenges, techniques, and research directions," 2021, *arXiv:2103.10074*.

[41] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 1–38, Sep. 2022.

[42] H. Rameder, "Systematic review of Ethereum smart contract security vulnerabilities, analysis methods and tools," M.S. thesis, Technische Universität Wien, Vienna, Austria, 2021.

[43] S. Ji, D. Kim, and H. Im, "Evaluating countermeasures for verifying the integrity of Ethereum smart contract applications," *IEEE Access*, vol. 9, pp. 90029–90042, 2021.

[44] S. Rouhani and R. Deters, "Security, performance, and applications of smart contracts: A systematic survey," *IEEE Access*, vol. 7, pp. 50759–50779, 2019.

[45] M. di Angelo and G. Salzer, "A survey of tools for analyzing Ethereum smart contracts," in *Proc. IEEE Int. Conf. Decentralized Appl. Infrastruct. (DAPPCON)*, Apr. 2019, pp. 69–78.

[46] J. Xu, F. Dang, X. Ding, and M. Zhou, "A survey on vulnerability detection tools of smart contract bytecode," in *Proc. IEEE 3rd Int. Conf. Inf. Syst. Comput. Aided Educ. (ICISCAE)*, Sep. 2020, pp. 94–98.

[47] 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.

[48] A. L. Vivar, A. T. Castedo, A. L. S. Orozco, and L. J. G. Villalba, "An analysis of smart contracts security threats alongside existing solutions," *Entropy*, vol. 22, no. 2, p. 203, Feb. 2020.

[49] A. Roan. (Feb. 2020). *Ethereum Smart Contract Ponzi Schemes*. [Online]. Available: https://medium.com/blockcentric/ethereum-smart-contract-ponzi-schemes-9e43015b56f8

[50] 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.

[51] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proc. ACM Workshop Program. Lang. Anal. Secur.*, 2016, pp. 91–96.

[52] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, Gothenburg, Sweden, May 2018, pp. 9–16.

[53] E. Albert, S. Grossman, N. Rinetzky, C. Rodríguez-Núñez, A. Rubio, and M. Sagiv, "Taming callbacks for smart contract modularity," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–30, Nov. 2020.

[54] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," 2018, *arXiv:1812.05934*.

[55] A. Mavridou and A. Laszka, "Tool demonstration: Fsolidm for designing secure Ethereum smart contracts," in *Proc. Int. Conf. Princ. Secur. Trust*. Cham, Switzerland: Springer, 2018, pp. 270–277.

[56] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to automatically exploit smart contracts," in *Proc. 27th USENIX Secur. Symp.*, Aug. 2018, pp. 1317–1333.

[57] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 653–663.

[58] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," 2018, *arXiv:1808.00624*.

[59] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[60] S. Badruddoja, R. Dantu, Y. He, K. Upadhayay, and M. Thompson, "Making smart contracts smarter," in *Proc. IEEE Int. Conf. Blockchain Cryptocurrency (ICBC)*, May 2021, pp. 1–3.

[61] Z. A. Khan and A. S. Namin, "A survey of DDOS attack detection techniques for IoT systems using BlockChain technology," *Electronics*, vol. 11, no. 23, p. 3892, Nov. 2022.

[62] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "SmartBugs: A framework to analyze solidity smart contracts," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2020, pp. 1349–1352.

[63] M. di Angelo and G. Salzer, "Consolidation of ground truth sets for weakness detection in smart contracts," Tech. Rep., 2023.

[64] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, and M. Ye, "Dappscan: Building large-scale datasets for smart contract weaknesses in dapp projects," Tech. Rep., 2023.

[65] A. Ding, G. Li, X. Yi, and Y. Liu, "IVSign: Interpretable vulnerability signature via code embedding and static analysis," in *Proc. 53rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun. 2023, pp. 25–31.

[66] A. Solar-Lezama. (2022). *Introduction to Program Synthesis*. [Online]. Available: https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture19.htm

[67] (2022). *Model Checking*. [Online]. Available: https://en.wikipedia.org/wiki/Model_checking

[68] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Fields of Logic and Computation II*, 2015, pp. 24–51.

[69] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "VerX: Safety verification of smart contracts," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1661–1677.

[70] G. Li and I. Ghosh, "Lazy symbolic execution through abstraction and sub-space search," in *Hardware and Software: Verification and Testing*. Cham, Switzerland: Springer, 2013, pp. 295–310.

[71] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," 2019, *arXiv:1901.01292*.

[72] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VeriSmart: A highly precise safety verifier for Ethereum smart contracts," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1678–1694.

[73] M. Krátký and R. Baca, "A comparison of element-based and path-based approaches to indexing xml data," *Databases, Texts*, vol. 176, p. 103, Jan. 2006.

[74] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2019, pp. 8–15.

[75] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 67–82.

[76] P. Tsankov, C. N. Yannis Sachinoglou, and Iczc. (2020). *Securify 2.0*. [Online]. Available: https://github.com/eth-sri/securify2

[77] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 664–676.

[78] (Aug. 2021). *Taint Analysis (Taint Checking)*. [Online]. Available: https://pvs-studio.com/en/blog/terms/6496/

[79] J. Ryan. (2003). *Xpath Rules*. [Online]. Available: https://www.developer.com/java/xpath-rules/

[80] 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 *Proc. IEEE 31st Comput. Secur. Found. Symp. (CSF)*, Jul. 2018, pp. 204–217.

[81] Y. Hirai, "Defining the Ethereum virtual machine for interactive theorem provers," in *Proc. Financial Cryptogr. Data Secur. FC Int. Workshops, WAHC, BITCOIN, VOTING, WTSC, TA, Sliema, Malta*, vol. 10323. Cham, Switzerland: Springer, 2017, pp. 520–535.

[82] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: Reusable engineering of real-world semantics," *SIGPLAN Not.*, vol. 49, no. 9, pp. 175–188, 2014.

[83] Y. Minsky, "OCaml for the masses: Why the next language you learn should be functional," *Queue*, vol. 9, no. 9, pp. 40–49, Sep. 2011.

[84] A. Yushkovskiy and S. Tripakis, "Comparison of two theorem provers: Isabelle/HOL and Coq," 2018, *arXiv:1808.09701*.

[85] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2017, pp. 442–446.

[86] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina, "Computing exact worst-case gas consumption for smart contracts," in *Proc. Int. Symp. Leveraging Appl. Formal Methods ISoLA Leveraging Appl. Formal Methods, Verification and Validation. Ind. Pract.* Nicosia, Cyprus: Springer, 2018, pp. 450–455.

[87] I. Grishchenko, M. Maffei, and C. Schneidewind, "Ethertrust: Sound static analysis of Ethereum bytecode," Tech. Rep., 2018.

[88] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019.

[89] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 1176–1186.

[90] T. Cook, A. Latham, and J. H. Le, "Dappguard: Active monitoring and defense for solidity smart contracts," Tech. Rep., 2017.

[91] (Feb. 2020). *Invariant (Mathematics)*. [Online]. Available: https://en.wikipedia.org/wiki/Invariant_(mathematics)

[92] V. Wüstholz and M. Christakis, "Targeted greybox fuzzing with static lookahead analysis," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 789–800.

[93] Snoo20972 and Teraflop. (2021). *Fuzzing: Why T14 and T19 Unreachable and T19 Reachable*. [Online]. Available: https://www.reddit.com/r/AskComputerScience/comments/o7fiif/fuzzing_why_t14_and_t19_unreachable_and_t19/

[94] *Has 'Bal' Got Deprecated*, Snoo20972, Adrewmc, and Flygoing, 2023.

[95] A. M. Alashjaee, S. Duraibi, and J. Song, "Dynamic taint analysis tools: A review," *Int. J. Comput. Sci. Secur.*, vol. 13, no. 6, pp. 231–243, Dec. 2019.

[96] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Sep. 2021, pp. 103–119.

[97] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*. New York, NY, USA: ACM, Sep. 2018, pp. 259–269.

[98] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, pp. 778–788, Aug. 2020.

[99] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "EASYFLOW: Keep Ethereum away from overflow," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion (ICSE-Companion)*, May 2019, pp. 23–26.

[100] B. L. Dalmazo, J. A. Marques, L. R. Costa, M. S. Bonfim, R. N. Carvalho, A. S. da Silva, S. Fernandes, J. L. Bordim, E. Alchieri, A. Schaeffer-Filho, L. P. Gaspary, and W. Cordeiro, "A systematic review on distributed denial of service attack defense mechanisms in programmable networks," *Int. J. Netw. Manage.*, vol. 31, no. 6, Nov. 2021, Art. no. e2163.

[101] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program learning," 2018, *arXiv:1807.05620*.

[102] (Nov. 2021). *Happened-Before*. [Online]. Available: https://en.wikipedia.org/wiki/Happened-before

[103] Snoo20972 and east_lisp_junk. (2022). *What Is, "Partial Order Reduction' As Discussed In: Exploiting the Laws of Order in Smart Contracts*. [Online]. Available: https://www.reddit.com/r/AskComputerScience/comments/s0ah2n/what_is_partial_order_reduction_as_discussed_in/

[104] K. Payne and D. Foreman. (2021). *Understanding Checking Account Overdraft Protection and Fees*. [Online]. Available: https://www.forbes.com/advisor/banking/understanding-checking-account-overdraft-protection-and-fees/

[105] B. Mueller, "Smashing Ethereum smart contracts for fun and real profit," in *Proc. 9th Annu. HITB Secur. Conf.*, 2018.

[106] H. Hackeerrrr. (2020). *Information Regarding Laser and Ethereum*. [Online]. Available: https://ethereum.stackexchange.com/questions/79938/information-regarding-laser-and-ethereum

[107] Z. A. Khan and A. S. Namin, "Involuntary transfer: A vulnerability pattern in smart contracts," *IEEE Access*, vol. 12, pp. 62459–62479, 2024.

[108] Z. A. Khan and A. S. Namin, "Dynamic analysis for the detection of locked ether smart contracts," in *Proc. IEEE Int. Conf. Big Data (BigData)*, Dec. 2023, pp. 2466–2472.

[109] S. Chadokar, L. Peltonen, and A. Dossa. (2018). *How Smart Contract Versioning Work With Compiler*. [Online]. Available: https://ethereum.stackexchange.com/questions/51331/how-smart-contract-versioning-work-with-compiler

[110] (2021). *American Fuzzy Lop (AFL)*. [Online]. Available: https://en.wikipedia.org/wiki/American_fuzzy_lop_

[111] K. Frankowicz and K. Rytarowski. (Sep. 2020). *An Introduction To LLVM Libfuzzer*. [Online]. Available: https://www.moritz.systems/blog/an-introduction-to-llvm-libfuzzer/

[112] (2019). *LLVM*. [Online]. Available: https://en.wikipedia.org/wiki/LLVM

[113] K. Frankowicz and K. Rytarowski, "An introduction to LLVM libfuzzer," Tech. Rep., Sep. 2020.

[114] R. H. Paul Razvan Berg. (2019). *Who is Msg. Sender in a Nested Call Chain*. [Online]. Available: https://ethereum.stackexchange.com/questions/73103/who-is-msg-sender-in-a-nested-call-chain

[115] A. Darby, T. Hess, X. L. B9lab, B. Bellaj, and Medvedev1088. (Feb. 2018). *Send Vs Call Differences and When to Use and When Not to Use*. [Online]. Available: https://ethereum.stackexchange.com/questions/6470/send-vs-call-differences-and-when-to-use-and-when-not-to-use

[116] P. R. Berg and R. Hitchens. (2020). *Who is Msg.Sender in a Nested Call Chain*. [Online]. Available: https://immunebytes.com/time-dependency-in-smart-contracts/

[117] *Symbolic Execution*. [Online]. Available: https://en.wikipedia.org/wiki/Symbolic_execution

[118] S. Cha, M. Lee, S. Lee, and H. Oh, "SYMTUNER: Maximizing the power of symbolic execution by adaptively tuning external parameters," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 2068–2079.

[119] zak100, haxerl, and PrashanthDev. (Feb. 2023). *What is the Purpose of Iszero*. [Online]. Available: https://ethereum.stackexchange.com/questions/71333/what-is-the-purpose-of-iszero

[120] Snoo20972 and Oriaj_13. (Feb. 2023). *Difference Between Timestamp and Block.timestamp*. [Online]. Available: https://www.reddit.com/r/ethdev/comments/10sohbp/difference_betwee_timestamp_and_blocktimestamp/

[121] S. Gopali, Z. A. Khan, B. Chhetri, B. Karki, and A. S. Namin, "Vulnerability detection in smart contracts using deep learning," in *Proc. IEEE 46th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jun. 2022, pp. 1249–1255.

[122] (Feb. 2023). *Fixedpoints*. [Online]. Available: https://ericpony.github.io/z3py-tutorial/fixpoint-examples.htm

[123] Y. Kostyukov, D. Mordvinov, and G. Fedyukovich, "Beyond the elementary representations of program invariants over algebraic data types," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, Jun. 2021, pp. 451–465.

[124] W. Shahda. (2019). *Protect Your Solidity Smart Contracts From Reentrancy Attacks*. [Online]. Available: https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21

[125] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," Tech. Rep., 2016.

[126] D. Quint. (Apr. 2017). *Control Structures-Why is Switch and Goto Not in Solidity*. [Online]. Available: https://ethereum.stackexchange.com/questions/13952/control-structures-why-is-switch-and-goto-not-in-solidity

[127] N. Rob and A. Munagala. (Nov. 2017). *How Are Ethereum Bytecode Jumps and Jumpdests Resolved*. [Online]. Available: https://stackoverflow.com/questions/47520515/how-are-ethereum-bytecode-jumps-and-jumpdests-resolved

[128] *Formal Verification*. [Online]. Available: https://csrc.nist.gov/glossary/term/formal_verification

[129] M. Rosendahl. *Abstract Interpretation*. [Online]. Available: http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/10.ABSTRACT-INTERPRETATION.html

[130] (2018). *Seahorn Founders*. [Online]. Available: https://seahorn.github.io/

[131] Community Wiki. (2015). *What is a Monad*. [Online]. Available: https://stackoverflow.com/questions/44965/what-is-a-monad

[132] (May 2022). *What's a Monad*. [Online]. Available: https://gist.github.com/Daenyth/e5014ca0e1593f310fbd8d540b24a844

[133] Kadenzipfel Kaden. (Mar. 2023). *Insufficient Gas Griefing*. [Online]. Available: https://github.com/kadenzipfel/smart-contract-vulnerabilities/blob/master/vulnerabilities/insufficient-gas-griefing.md

[134] S. Marx. (Aug. 2019). *Provable Fair Ransom*. [Online]. Available: https://medium.com/consensys-diligence/provable-fair-ransom-1547ba894d0

[135] J. Jiao, S. Kan, S. Lin, D. Sanán, Y. Liu, and J. Sun, "Executable operational semantics of solidity," 2018, *arXiv:1804.01295*.

[136] sanketnagare. *Solidity Global Variables*. [Online]. Available: https://www.geeksforgeeks.org/solidity-global-variables/

[137] R. Bernstein. (Sep. 2018). *Mythril Detection Capabilities*. [Online]. Available: https://github.com/ConsenSys/mythril/wiki/Mythril-Detection-Capabilities

[138] zak100 and hroussille. (2015). *Detection of Same Funtion Reentrancy Vulnerability*. [Online]. Available: https://ethereum.stackexchange.com/questions/116919/detection-of-same-function-reentrancy-vulnerability

[139] H. Hon and B. Liu. (2022). *Forensics of Attacks and Exploits in Defi*. [Online]. Available: https://content-hub-static.crypto.com/wp-content/uploads/2022/02/Forensics_of_Attacks_and_Exploits_in_DeFi.pdf

[140] *Dataflow Analysis, Iterative Data-flow Analysis and Static-Single-Assignment*. [Online]. Available: httphttp://www.cs.uccs.edu/http://www.cs.uccs.edu/~qyi/UTSA-classes/cs5363/slides/dataflowAnalysis.pdfqyi/UTSA-classes/cs5363/slides/dataflowAnalysis.pdf

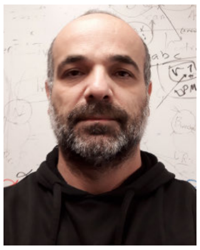
[141] (Feb. 2023). *Integer Overflow*. [Online]. Available: https://en.wikipedia.org/wiki/Integer_overflow

[142] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," in *Proc. 18th Conf. USENIX Secur. Symp.*, 2009, pp. 67–82.

[143] B. Mueller. (2018). *Smashing Ethereum Smart Contracts for Fun and Real Profit*. [Online]. Available: https://github.com/b-mueller/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf

[144] B. Mueller, JoranHonig, S. Bgrara, S. Pogodin, and J. Asplund. (2018). *Laser Ethereum*. [Online]. Available: https://github.com/b-mueller/laser-ethereum
[145] A. Chauhan. (Feb. 2022). *How to Find Vulnerability in Smart Contracts Unexpected Ether*. [Online]. Available: https://betterprogramming.pub/how-to-find-vulnerability-in-smart-contracts-unexpected-ether-89f157ce2888
[146] Wikipedia. (Dec. 2021). *Runtime Verification*. [Online]. Available: https://en.wikipedia.org/wiki/Runtime_verification
[147] D. Wesley. (Oct. 2017). *Reentrancy Attack, on a Smart Contract*. [Online]. Available: https://medium.com/@JusDev1988/reentrancy-attack-on-a-smart-contract-677eae1300f2
[148] B. Cirisci, C. Enea, A. Farzan, and S. O. Mutluergil, "A pragmatic approach to stateful partial order reduction," in *Proc. Int. Conf. Verification, Model Checking Abstract Interpretation*, 2022.
[149] M. V. H. Prakash, A. A. Rao, and P. R. Raju, "Improving state invariant test Oracle strategy using mutation analysis," *Int. J. Comput. Trends Technol.*, vol. 61, no. 2, pp. 11–116, Jul. 2018.
[150] *What is Shadow Memory? Complete Guide*. [Online]. Available: https://www.velocenetwork.com/tech/what-is-shadow-memory/
[151] cleanunicorn. (2020). *Hevm*. [Online]. Available: https://github.com/dapphub/dapptools/tree/master/src/hevm
[152] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 415–427.
[153] zak100 and kbw. (2021). *Developing Contraol Flow Graph for Solidity Smart Contracts in C/c++ Language*. [Online]. Available: http://www.cplusplus.com/forum/general/278355/
[154] Zak100, Abcoathup, and Marganaraz. (2021). *How to Create a Tool for Detecting Evm Vulnerabilities*. [Online]. Available: https://ethereum.stackexchange.com/questions/116919/detection-of-samefunction-reentrancy-vulnerability
[155] A. Dika and M. Nowostawski, "Security vulnerabilities in Ethereum smart contracts," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Phys. Social Comput. (CPSCom) IEEE Smart Data (SmartData)*, Jul. 2018, pp. 955–962.
[156] R. M. Parizi, A. Dehghantanha, K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," 2018, *arXiv:1809.02702*.
[157] R. Fontein, "Comparison of static analysis tooling for smart contracts on the EVM," in *Proc. 28th Twente Student Conf. (IT)*, 2018.
[158] D. Pérez and B. Livshits, "Smart contract vulnerabilities: Does anyone care?" 2019, *arXiv:1902.06710*.
[159] C. DeCusatis, B. Gormanly, J. Iacino, R. Percelay, A. Pingue, and J. Valdez, "Cybersecurity test bed for smart contracts," *Cryptography*, vol. 7, no. 1, p. 15, Mar. 2023.
[160] K. Zipfel. (Dec. 2019). *The Encyclopedia of Smart Contract Attacks and Vulnerabilities*. [Online]. Available: https://betterprogramming.pub/the-encyclopedia-of-smart-contract-attacks-vulnerabilities-dfc1129fdaac
[161] S. Zhang and J.-H. Lee, "Smart contract-based secure model for miner registration and block validation," *IEEE Access*, vol. 7, pp. 132087–132094, 2019.
[162] E. Chain. *The First Self-mining Smart Contract*. [Online]. Available: https://www.minereum.com/
[163] P. Hartel and R. Schumi, "Mutation testing of smart contracts at scale," in *Proc. Int. Conf. Tests Proofs*, vol. 12165, 2020, pp. 23–42.

ZULFIQAR ALI KHAN is currently pursuing the Ph.D. degree with the Computer Science Department, Texas Tech University. His research interests include Ethereum smart contracts and the Internet of Things. However, he is proficient in several programming languages like Java, Python, Solidity, and Javascript. In addition, he has more than 20 years of teaching experience in the leading universities of Pakistan, including Sir Syed University of Engineering and Technology, where he is also an Assistant Professor on study leave.



AKBAR SIAMI NAMIN received the Ph.D. degree in computer science from Western University, London, Canada, in August 2008. He is currently an Associate Professor in computer science with Texas Tech University. He has coauthored over 80 research articles published in premier journals and venues. His research interests and expertise include software engineering, testing, program analysis, software and cyber security and malware analysis, and machine and deep learning. His research on cyber security research and education is funded by the National Science Foundation.