The Journal of Systems and Software 217 (2024) 112160

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Vulnerability detection techniques for smart contracts: A systematic literature review☆

Fernando Richter Vidal *, Naghmeh Ivaki, Nuno Laranjeiro

University of Coimbra, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, Portugal

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.8109651

Keywords:
Blockchain
Smart contract
Smart contract security
Vulnerability detection
Verification tools

ABSTRACT

The number of applications supported by blockchain smart contracts has been greatly increasing in recent years, with smart contracts now being used across several domains, such as the music industry, finance, and retail, to name a few. Despite being used in business-critical contexts, the number of security vulnerabilities in smart contracts has also been increasing, with many of them being exploited and resulting in huge financial and reputation losses. This is despite the enormous effort that is being placed into the research and development of vulnerability detection tools and techniques, which have also greatly increased in number and type in the last few years. Motivated by the recent increase in both vulnerabilities and vulnerability detection techniques, this paper reviews the latest research in smart contract vulnerability detection, emphasizing the techniques being used, the vulnerabilities targeted, and the characteristics of the dataset used for evaluating the technique. We mapped the vulnerabilities against two common vulnerability classification schemes (DASP and SWC) and performed a consolidated analysis. We identified the current research trends and gaps in each technique and highlighted future research opportunities in the field.

1. Introduction

A blockchain system can be generally described as a distributed ledger that uses a peer-to-peer network to enable its participating members to obtain a complete copy of the current state of the ledger (Zhang et al., 2019b). By using a consensus-based verification system, all users agree if the blocks are valid or not, preventing and rejecting attempts of tampering with blocks. This securely substitutes the use of a centralized ledger, like a single database or authority for validation (Yaga et al., 2018). The advances in blockchain technology quickly introduced smart contracts (Swan, 2015), which are programmed applications that are stored and executed on the blockchain in order to automate the validation and execution of transactions according to the conditions for which the contract is programmed (Alharby and van Moorsel, 2017). Ethereum (2020) was the first blockchain to support smart contracts and was specifically designed for that purpose. Ethereum is a distributed state machine in which the "ledger" state is supported by the Ethereum Virtual Machine (EVM), whose purpose is to run code for any decentralized application that is implemented in its network. A very popular language for implementing smart contracts targeting the EVM is Solidity (Chen et al., 2021), a statically typed, object-oriented, and high-level programming language that supports inheritance, libraries, and complex user-defined data types. Solidity's design was influenced by C++, Python, and JavaScript syntax.

Smart contracts are a rising technology due to the potential they offer (Zheng et al., 2021). However, faulty contracts deployed in the immutable blockchain can compromise blockchain security (Praitheeshan et al., 2020). Compared to conventional programs, building secure and fault-free smart contracts is more challenging due to several reasons: (i) smart contract programming languages are still immature (e.g., Solidity does not have a standard library to allow programmers to compare strings, forcing each programmer to implement his own solution) (Chittoda, 2019) and inherently carry many constraints (e.g., developers can only choose from a limited set of functions for their cryptographic operations) (Wang et al., 2020), (ii) smart contract developers are, many times, not really experienced and many vulnerabilities are actually related to bad programming practices (Zou et al., 2019) (e.g., "Erroneous Constructor Name" Hu et al., 2023, "Unchecked Transfer Value" Zhang et al., 2020a), and (iii) existing supporting tools for development and verification of smart contracts do not offer strong mechanisms to detect and avoid many vulnerabilities, with this being aggravated by the fast-paced appearance of new vulnerabilities.

In this paper, **we present a systematic literature review on the state of the art in smart contract vulnerability detection research.**

☆ Editor: Yan Cai.

* Corresponding author.
E-mail addresses: fernandovidal@dei.uc.pt (F.R. Vidal), naghmeh@dei.uc.pt (N. Ivaki), cnl@dei.uc.pt (N. Laranjeiro).

https://doi.org/10.1016/j.jss.2024.112160
Received 3 July 2023; Received in revised form 5 April 2024; Accepted 16 July 2024
Available online 20 July 2024

Besides systematizing and categorizing the different vulnerability detection techniques, we map the vulnerabilities identified by each tool or technique against two common vulnerability classification schemes, namely Decentralized Application Security Project (DASP) (NCC Group, 2019) and Smart Contract Weakness Classification (SWC) (MythX, 2020). Among our observations, we highlight that a significant number of vulnerabilities no longer fit in a popular and fine-grained scheme like SWC, opening space for new schemes that can classify new vulnerabilities. Also, the tools announced detection capabilities are quite heterogeneous and really limited, especially concerning the new vulnerabilities. Moreover, results reported by tools' authors are very difficult to compare due to the heterogeneity described and the specific evaluation settings, opening spaces for benchmarks that evaluate tools' effectiveness in a standard manner.

The rest of this paper is organized as follows. Section 2 describes the methodology. Section 3 presents a comprehensive analysis of the state of the art on vulnerability detection in smart contracts. Section 4 discusses the main findings of our analysis. Section 5 highlights the gaps and limitations identified. Finally, Section 6 concludes this paper.

2. Methodology

In this section, we describe the methodology used to perform this systematic literature review on smart contract vulnerability detection, which is based on well-established guidelines for conducting systematic reviews (Kitchenham, 2004), and is comprised of the following steps:

(1) **Analysis of related surveys**: We start by identifying and analyzing other review studies, including surveys, other systematic reviews, or mapping studies, which somehow cover the topic of vulnerability detection in smart contracts. We aim to highlight the gaps and limitations on this topic in the existing review studies to be covered and discussed in our own systematic review.
(2) **Definition of research questions**: Based on the gaps and limitations identified in the previous step, we define a set of research questions that build the principal objectives of our review.
(3) **Identification of studies**: We identify and choose the main databases (e.g., search engines, online libraries) that are frequently used by researchers to identify relevant peer-reviewed and authentic studies, i.e., the studies that contribute to the topic under study (i.e., security assessment of smart contracts). We then define a general query string to be used to perform the search, followed by the snowballing process (Wohlin, 2014) to complement the identification of studies.
(4) **Study selection**: This step focuses on the definition of the inclusion and exclusion criteria and the quality assessment criteria to be applied to the primary studies found and collected in the previous step. The goal is to include the related works in this study that perfectly comply with our systematic review's research objectives and scope.
(5) **Data extraction and synthesis**: We finally extract the relevant data from the selected primary studies according to the research questions. The extracted data is then synthesized to allow answering the research questions.

The following subsections present each of the above steps in further detail. The final subsection summarizes the general outcome of the reviewing process and the main differences between related surveys and our own.

2.1. Analysis of related surveys

Overall, we found the following surveys related to vulnerability detection and security assessment or verification.

di Angelo and Salzer (2019) discuss 27 tools for analyzing Ethereum smart contracts in terms of purpose, abstraction level, implementation maturity level, methods employed, and security issues detected. The authors observed basically the presence of two groups of tools (i.e., static and dynamic tools) and concluded that static tools' prevalence is higher in this context. Tools were also grouped into two sets, one targeting contracts already deployed in the blockchain and another set with tools to be used with contracts still being developed (i.e., before deployment). 18 out of the 27 tools focused on vulnerability detection rather than correction, and the authors highlight 5 of the tools, namely FSolidM, KEVM, Securify, MAIAN, and Mythril. The authors also emphasize the need for standardized benchmarks for facilitating comparative analysis, the presence of contract datasets with known vulnerabilities (although unbalanced in terms of types of vulnerabilities), along with the need for publicly available tools that are built based on reusable components.

Singh et al. (2019) review research based on formal methods for smart contracts and discuss the characteristics of a total of 31 works. The authors identify the formal methods used in the literature, which include Theorem proving (the most prevalent one in the reviewed works), Model Checking, Formal Modeling, and Symbolic Execution. The work also identifies the main aspects targeted by the works, from which the majority focuses on functionality verification (24%), followed by security (6%) and Privacy (3%). The authors highlight that despite formal methods being quite effective, they have not been fully explored in the context of smart contracts. Also, there is a need for the development of efficient and effective formal verification tools, as the technique can be costly and time-consuming depending on the number of execution states and the complexity of smart contracts.

Liu and Liu (2019) survey state of the art on security verification of smart contracts. The authors separate the discussion into two groups: security assurance and correctness verification. The first group is divided into three categories: environment security (i.e., blockchain system security); vulnerability scanning (i.e., characterized by the authors as the study of known vulnerabilities); and performance impact (i.e., from the perspective of performance impacting the secure execution of smart contracts). The second group is divided into two categories: programming correctness (i.e., involving aspects related to software engineering, such as the definition of standards, new languages, semantic analyses, and the use of development tools) and formal verification (i.e., either program- or behavior-based verification). As main highlights, the authors mention that vulnerability scanning is the most popular topic despite the immaturity of tools (e.g., high false-positive rates and weak preparedness for unknown vulnerabilities). The authors also discuss the increasing use of formal methods as a reliable technique for smart contract verification, as well as the need for tools that can combine such methods with vulnerability analysis. Finally, just a few tools are able to detect environmental vulnerabilities (e.g., malicious miners) and are prepared to handle contracts that require the use of external data.

Chen et al. (2021) present a systematic survey on Ethereum systems security, evaluating both the environment and the smart contracts. The authors considered the following aspects: (i) the vulnerabilities of smart contracts (i.e., 40 vulnerabilities) and their causes (e.g., programming errors, P2P protocol, consensus mechanism, etc.); (ii) the attacks exploiting these vulnerabilities and possible defenses. Due to the blockchain immutability, the authors highlight the consequences for the network when a vulnerability is activated. The study also reveals that authorization and authentication issues are major security problems in Ethereum smart contracts and the challenge of detecting security issues in contracts that have external dependencies. Regarding defense mechanisms, the authors concluded that reactive mechanisms can defend against a few vulnerabilities, while proactive mechanisms may allow defending against a broader set. The vulnerabilities identified in Chen et al. are classified into five groups: Application, Data, Consensus, Network, and Ethereum-specific. The authors map the root

causes of these vulnerabilities into known attacks. According to the authors, Ethereum's "permissionless" and "immutability" properties are problematic because they allow attackers to exploit vulnerabilities at will and, on the other hand, do not allow correction mechanisms to take place. As a contribution, they present defense mechanisms (i.e., proactive and reactive) to address the identified vulnerabilities, with the exception of 14 vulnerabilities (14) signaled as indefensible due to the Ethereum design and implementation.

Wang et al. (2020) present a systematic review of the security of Ethereum smart contracts. Vulnerability detection is one of the several topics discussed. The authors discuss the works in three main groups: (i) static analysis, which is subdivided into symbolic execution and formal verification; (ii) Dynamic analysis; and (iii) Code similarity. As this review touches on several distinct topics, the discussion on vulnerability detection approaches is quite short.

Almakhour et al. survey methods for verification of smart contracts in Almakhour et al. (2020). The methods, which are found in 25 papers, are grouped into two sets: (i) formal verification methods, which are mapped to theorem proving, model checking, and runtime verification, and (ii) vulnerability detection methods, which are mapped to symbolic execution, abstract interpretation, and fuzzing. The authors also discuss some vulnerabilities mentioned in the literature, with a particular focus on 16 vulnerabilities and how the different tools are targeting them. The main gaps identified by the authors include the need for methods that can address complex smart contracts (i.e., the analyzed tools only verify simple smart contracts) and also the need for methods combination for achieving higher detection effectiveness.

In Ghaleb and Pattabiraman (2020a), a set of static analysis tools are analyzed and compared, namely Oyente, Securify, Mythril, SmartCheck, Manticore, and Slither. The methodology for comparison involves injecting eight types of software faults in three ways: Full Code Snippet (i.e., used to inject large blocks of malicious code, such as reentrancy), Code Transformation (i.e., injecting known vulnerability patterns, such as the use of tx.origin), and Weakening Security Mechanism (i.e., to assess program behavior, especially in external calls, such as unhandled exception bugs).

The authors analyze whether the different detection tools can detect the injected faults. Tools are evaluated according to false positive (a known significant challenge in static analyzers) and false negative rates. The conclusion is that many tools do not detect the faults because they should have more elaborate semantic code analysis instead of just analyzing syntax and symbol trees.

Hu et al. (2021) present an extensive analysis regarding the development of smart contracts, which touches on several topics, including vulnerability detection. The authors separate a total of 40 tools into specific-purpose tools (i.e., tools designed specifically to detect a single type of vulnerability) and general-purpose tools (i.e., which aim at detecting more than one type of vulnerability). The techniques identified include symbolic execution, syntax analysis, abstract interpretation, model checking, and fuzzing. The challenges identified in the survey include the frequent occurrence of vulnerabilities, incomplete design paradigms, inefficient vulnerability detection tools, low processing rate, limited contract complexity, and lack of privacy.

Hewa et al. (2021) review smart contract research from multiple perspectives, which include the identification of vulnerabilities and countermeasures, including vulnerability detection tools. The authors review 16 works that aim at detecting vulnerabilities and map them to a relatively small set of known vulnerabilities.

Surucu et al. (2022) present a review of machine learning vulnerability detection tools. The review organizes the identified works into three sets: (i) deep learning models (10 works), classical machine learning models (3 works), and ensemble learning models (3 works). The paper indicates that supervised learning is the most prevalent technique. This implies that it is necessary to have labeled data to create the models, which may be difficult to obtain (information regarding vulnerabilities is generally scarce).

The aforementioned papers hold some limitations and gaps, which we summarize below:

- The survey presented in di Angelo and Salzer (2019) is focused on a particular group of techniques, i.e., techniques based on formal methods. Considering other types of tools would provide a clearer image of the current vulnerability detection landscape. The analysis of the 27 tools is limited to the vulnerabilities for which the tools have been designed to detect, which is reasonable. However, as the number of tools is relatively small, the number and type of vulnerabilities for the comparative analysis are also reduced.
- In Singh et al. (2019), the analysis is focused on formal verification methods, which, as in di Angelo and Salzer (2019), limits the discussion and possible insights to a certain group of techniques.
- The work in Liu and Liu (2019) is a survey that has a relatively broad focus that targets 53 papers, of which 20 focus on security assurance and 33 on correctness verification. The analyzed papers are characterized according to the specific type of method used. However, the work could benefit from further analysis and insights regarding the vulnerabilities targeted and especially have a larger focus on works that do not rely on formal methods (which are the core of the works discussed in the survey, at least in what concerns vulnerability detection).
- The systematic survey presented in Chen et al. (2021) is an in-depth review and systematization of different types of vulnerabilities and is largely focused on associated aspects, such as best practices and principles associated with each vulnerability. However, the analysis of works that refer to vulnerability detection is a secondary aspect of this survey, which ends up reviewing 44 papers on the topic.
- The work presented in Wang et al. (2020) reviews blockchain challenges. While the focus is mostly on security, vulnerability detection is not the primary focus. Consequently, the discussion regarding detection methods is short (i.e., 16 papers are discussed) and not really insightful (as the focus is broader).
- The work in Almakhour et al. (2020) reviews smart contract verification tools, with the focus mostly being on contract correctness and formal verification tools. Again, the focus on vulnerability detection is limited, with only 16 vulnerabilities identified and 23 vulnerability detection works separated into three categories, namely symbolic execution, abstract interpretation, and fuzzing. The need for a broader and also up-to-date review is very clear, with this particular work touching just a portion of the current state of the art.
- In Ghaleb and Pattabiraman (2020a), the authors review six static analyzers and evaluate their detection capabilities. Due to the specificity of the work, a single vulnerability detection technique is considered, which is a fraction of the currently available techniques.
- The survey presented in Hu et al. (2021) reviews the state of the art regarding construction and execution of smart contracts and is a broad but, at the same time, detailed review. On the topic of vulnerability detection, 40 works are reviewed, which is a significant number that, however, falls behind the current state of the art. This is mostly due to the fast-paced development of the area. For example, the work does not discuss machine learning-based approaches for vulnerability detection, which are now gaining momentum. The discussion regarding vulnerabilities detected by the identified tools is generally absent from the work, highlighting the need for systematizing knowledge in this particular topic.
- The survey presented in Hewa et al. (2021) briefly identifies 10 vulnerabilities and 16 vulnerability detection tools. Possibly due to the publication date, the work does not cover a representative set of the state of the art; thus, the associated insights are currently of limited usefulness.

- The work in Surucu et al. (2022) targets vulnerability detection techniques based on machine learning. As such, it provides a limited contribution considering the current variety of techniques available for this purpose. In addition, the discussion on the targeted vulnerabilities is quite short, which emphasizes the need for a more detailed view regarding this topic.

Our work aims to identify and analyze the current state of the art in vulnerability detection for smart contracts, regardless of the specific technique used, and overcome the aforementioned limitations.

2.2. Research questions

Based on the gaps and limitations identified in the analysis of the existing review studies, we defined the following research questions that build the objectives of this systematic review:

- **RQ-1**: Which techniques are being used for detecting vulnerabilities in smart contracts?
- **RQ-2**: Which types of smart contract vulnerabilities are the target of detection by vulnerability detection tools?
- **RQ-3**: What are the characteristics of the datasets used to evaluate the techniques used for smart contract vulnerability detection?

Guided by these research questions, we intend to analyze the different techniques and tools identified in the literature and break each matter into four parts according to the above research questions. In **RQ-1**, we aim to identify, characterize, and categorize the different state-of-the-art techniques (e.g., static analysis, formal verification) and specific sub-techniques; in **RQ-2**, we aim to identify and characterize the types of vulnerabilities detected by each tool, thereby demonstrating the effectiveness of each technique employed; in **RQ-3**, we aim at discussing the characteristics of the dataset used in the evaluation of the proposed techniques and tools, contributing towards guidelines that can help researchers defining new studies;

2.3. Identification of studies

Google Scholar (Google, 2021) was the first data source we selected to search for primary studies. Next, we searched IEEExplorer and the ACM Library to ensure articles not captured by Google Scholar were included in the analysis. To perform the search, we used the following query string:

$$(\text{“}smart\ contract\text{”}\ OR\ \text{“}smartcontracts\text{”})\ AND\ \text{“}vulnerability\ detection\text{”}$$

The query yields a total of 1880 papers, which were passed through the inclusion and exclusion criteria and quality assessment, as described in the next section.

2.4. Study selection and quality assessment

To filter the identified studies, a set of inclusion and exclusion criteria was applied:

- Inclusion Criteria:
  - The work must address smart contract vulnerability detection and, therefore, must specify types of vulnerabilities (e.g., reentrancy Liu et al., 2018).
  - The work must clearly characterize the technique used to perform vulnerability detection so that we can specify its type/subtype while allowing readers to understand its basic mechanics.
- Exclusion Criteria:
  - Works that are published as short papers (e.g., less than four pages) are excluded from the analysis. The same happened with non-peer-reviewed research (i.e., pre-prints, despite being publicly available).
  - Due to the huge number of publications on this topic in recent years, and also serving as a quality assessment measure, we exclude papers published in tier B conferences and lower (we use CORE 2021 Clarivate, 2021, as reference), as well as papers published in JCR Q2 Journals and lower (Research and of Australasia, 2021).

To assess the above criteria, we carefully reviewed each paper, taking into account its title, abstract, and full text. The full-text analysis served initially to understand if vulnerability types are present and if the technique is sufficiently described. The whole process involves manual intervention, which may lead to mistakes in the inclusion or exclusion of papers. We tried to mitigate this threat by having an additional researcher following the process and double-checking for possible mistakes.

2.5. Data extraction and synthesis

Regarding data extraction and synthesis, the most relevant aspect to mention is that the authors tend to use diverse terms that often refer to a single concept (e.g., two terms referring to the same vulnerability detection technique). As such, after collecting the different terms used by the authors, we then individually discussed them and generalized them to more widely accepted terms. The result of this process is described in the following section.

2.6. SLR outcome and other reviews

Table 1 presents the outcome of the systematic literature review process followed in this work in perspective with the related surveys. Starting from the left-hand side, Table 1 identifies a specific survey, its publication year and the period covered in the review, the total number of papers analyzed, and the number of analyzed papers related to smart contact vulnerability detection. The following columns identify if the main focus is on vulnerability detection or not, if the authors reference vulnerability classification schemes (e.g., SWC), which main types of vulnerability techniques are analyzed, if the survey identifies which vulnerabilities the surveyed tools are aiming to detect, and, finally, if smart contract datasets are analyzed.

The main differences between the identified related surveys and our own work, besides the visible difference in the number of papers analyzed, can be summarized as follows. As we can see in Table 1 few reviews specifically target the topic of vulnerability detection. This obviously leads the works to be more superficial on the security aspects analysis, disregarding information such as the identification of vulnerabilities being targeted by the surveyed papers or datasets involved, which is a natural consequence of the broader focus of each work. Another important aspect to mention is that this review analyses the largest set of vulnerability detection techniques among the related reviews (i.e., some focus on a single technique), allowing for a broader and up-to-date view of the various smart contract vulnerability detection techniques and the targeted vulnerabilities.

3. Analysis

In this section, we analyze the techniques and tools identified during our review of state of the art in smart contract vulnerability detection. The identified works are grouped into four categories and explained in the next subsections. The categories are as follows: **formal methods** (Bhargavan et al., 2016) explained in Section 3.1, **code analysis techniques** (Chess and McGraw, 2004) explained in Section 3.2, **software testing techniques** (Singh and Singh, 2012) explained in

Table 1
A perspective of the existing related reviews on smart contract vulnerability detection.

Survey	Publication year	Period	Total papers	Security assessment	Focus on vulnerability detection	Vulnerability classification used	Techniques analyzed	Mapping vulnerabilities to tools	Dataset analysis
di Angelo and Salzer (2019)	2019	2017–2018	29	27	Partial	Custom	Multiple techniques (Formal methods, Code analysis, Software testing)	Y	N
Singh et al. (2019)	2019	2015–2019	35	31	Partial	–	Single technique (Formal methods)	N	N
Liu and Liu (2019)	2019	2015–2019	53	20	Partial	–	Single technique (Formal methods)	N	N
Chen et al. (2021)	2019	2015–2019	72	44	Partial	Custom	Multiple techniques (Formal methods, Software testing)	N	N
Wang et al. (2020)	2020	2010–2020	22	16	Partial	–	Multiple techniques (Formal methods, Code analysis, Software testing)	N	N
Almakhour et al. (2020)	2020	2008–2019	25	23	Partial	Custom	Multiple techniques (Formal methods, Software testing)	Y	N
Ghaleb and Pattabiraman (2020b)	2020	2018–2020	31	6	Y	Custom	Single technique (Code analysis)	Y	N
Hewa et al. (2021)	2021	2014–2020	59	16	Partial	Custom	Single technique (Machine learning)	N	N
Hu et al. (2021)	2021	2008–2020	135	40	Partial	–	Multiple techniques (Formal methods, Code analysis, Software testing)	N	N
Surucu et al. (2022)	2022	2018–2021	16	16	Y	–	Single technique (Machine learning)	Y	N
This work	2024	2016–2023	188	81	Y	DASP, SWC	Multiple techniques (Formal methods, Code analysis, Software testing, Machine learning)	Y	Y

Section 3.3, and **machine learning-based techniques** (Momeni et al., 2019) explained in Section 3.4. Within each of the four general categories of techniques, we find several **specific techniques** (e.g., the category of static analysis includes techniques like taint analysis and abstract interpretation), which also serve to structure the analysis. Thus, each of the following four subsections is further divided into the various specific techniques identified during the analysis of the works. Notice that there are several works (12%, 10 out of 81) that use multiple techniques (e.g., Zeus Kalra et al., 2018 uses both Abstract Interpretation and Model Checking).

Table 2 presents the *Categories*, the *Techniques* found in the literature within each category, the tools' name, and a reference for each analyzed work/tool. It is interesting to notice that 91% (74 of 81) of the works presented are actually materialized in a tool. The identified categories and techniques are conceptually described in the following paragraphs.

- **Formal Methods**: Methods based on formal proofs or mathematical models of a certain system or part of a system, with the goal of proving its correctness (e.g., mainly functional correctness) (Seligman et al., 2015). The specific techniques in this category are as follows:
  - *Abstract Interpretation*: A technique for approximating the semantics of discrete dynamic systems, generally for computing a set of reachable program states. The idea is to simplify highly complex or undecidable problems to allow verifying specific properties (Cousot, 2021).
  - *Model Checking*: Aims at checking whether a finite-state model of a system satisfies its formal specifications or correctness properties (Clarke et al., 1999).
  - *Symbolic Execution*: Relies on the systematic exploration of possible code execution paths without using concrete inputs. The technique usually abstracts the inputs as symbols and then relies on constraint solvers to build actual instances that would violate certain properties (Baldoni et al., 2019).
  - *Theorem Proving*: A technique that uses formal mathematical methods to prove or disprove the correctness of the program regarding a certain formal specification or property (Schumann, 2001).
- **Code Analysis**: Methods that rely on the inspection of code by various means (e.g., pattern recognition, taint analysis) to discover software defects (Rival, 2016). The specific techniques in this category are the following:
  - *Taint Analysis*: A technique that consists of tracking the propagation of data across the control flow of code to understand if they can reach a possibly vulnerable point in the code (i.e., a sink) (Xu et al., 2018).
  - *Pattern Recognition*: A technique that tries to identify known patterns, which, in this context, represent potential vulnerabilities in code. This technique usually includes simple methods such as text matching using regular expressions (i.e., and not further complex techniques like using neural networks for pattern identification).
- **Software Testing**: Methods that rely on the execution of software with the intention of finding defects (Myers et al., 2012).
  - *Fuzzing*: Refers to the random generation of a large number of inputs that are used during runtime as input to software. The intention is generally to generate errors, crashes, or other types of unexpected behavior (Chen et al., 2018).
  - *Concolic Execution* It is a hybrid solution that combines symbolic execution and fuzzing, offering execution in a complementary manner (Weiss and Schütte, 2019)

Table 2
Tools classification.

Categories	Techniques	Tools	References	Execution mode
Formal methods	Abstract interpretation	HFCCT	Li et al. (2022b)	Static
		MadMax	Grech et al. (2020)	Static
		Securify	Tsankov et al. (2018)	Static
		SoliDetector	Hu et al. (2023)	Static
		Vulpedia	Ye et al. (2022)	Static
		Zeus	Kalra et al. (2018)	Static
	Model checking	–	Crincoli et al. (2022)	Static
		HELMHOLTZ	Nishida Yuki and Saito et al. (2021)	Static
		SmartAce	Wesley Scott and Christakis et al. (2022)	Static
		SmartPulse	Stephens et al. (2021)	Static
		VeriSmart	So et al. (2020)	Static
		VeriSolid	Mavridou et al. (2019)	Static
		Zeus	Kalra et al. (2018)	Static
	Symbolic execution	–	Zhang et al. (2022b)	Static
		DSE	Huang et al. (2022)	Dynamic
		EtherSolve	Pasqua et al. (2023)	Static
		HFCCT	Li et al. (2022b)	Static
		MAR	Wang Zexu and Wen et al. (2021)	Static
		MOPS	Fu et al. (2019)	Static
		Mythril Extension	Yao et al. (2022)	Static
		Osiris	Torres et al. (2018)	Static
		Oyente	Luu et al. (2016)	Static
		Pluto	Ma et al. (2022)	Static
		RA	Chinen et al. (2020)	Static
		SailFish	Bose et al. (2022)	Static
		Siguard	Zhang et al. (2023)	Dynamic
		SmartAce	Wesley Scott and Christakis et al. (2022)	Static
		TransRacer	Ma et al. (2023c)	Dynamic
		sCompile	Chang et al. (2019)	Static
		teEther	Krupp and Rossow (2018)	Static
	Theorem proving	–	Ayoade et al. (2019)	Static
Code analysis	Pattern recognition	SmartCheck	Tikhomirov et al. (2018)	Static
		SmartDagger	Liao et al. (2022)	Static
		Vrust	Cui et al. (2022)	Static
	Taint analysis	EOSIOAnalyzer	Li et al. (2022a)	Static
		EasyFlow	Gao et al. (2019b)	Dynamic
		EthPloit	Zhang et al. (2020a)	Static
		Ethainter	Brent et al. (2020)	Static
		Osiris	Torres et al. (2018)	Static
		Sereum	Rodler et al. (2019)	Dynamic
		Siguard	Zhang et al. (2023)	Dynamic
		Slither	Feist et al. (2019)	Static
		SmartDagger	Liao et al. (2022)	Static
		SmartFast	Li et al. (2022c)	Static
Software testing	Fuzzing	ContractFuzzer	Jiang et al. (2018)	Dynamic
		Echidna	Grieco et al. (2020)	Dynamic
		EthPloit	Zhang et al. (2020a)	Dynamic
		Gas Gauge	Nassirzadeh Behkish and Sun et al. (2023)	Dynamic
		GasFuzzer	Ashraf et al. (2020)	Dynamic
		HFContractFuzzer	Ding et al. (2021)	Dynamic
		IR-Fuzz	Liu et al. (2023b)	Dynamic
		ItyFuzz	Shou et al. (2023)	Dynamic
		Pied-Piper	Ma et al. (2023a)	Dynamic
		RLF	Su et al. (2022)	Dynamic
		ReDefender	Li et al. (2022d)	Dynamic
		Sereum	Rodler et al. (2019)	Dynamic
		SmartAce	Wesley Scott and Christakis et al. (2022)	Dynamic
		SmartFuzzDriverGen	Pani et al. (2023)	Dynamic
		Solanalyser	Akca et al. (2019)	Dynamic
		TechyTech	Khan and Namin (2023)	Dynamic
		WASAI	Chen et al. (2022)	Dynamic
		sFuzz	Nguyen et al. (2020)	Dynamic
		vGas	Ma et al. (2023b)	Dynamic
		xFuzz	Xue et al. (2022)	Dynamic
	Concolic execution	ConFuzzius	Torres et al. (2021)	Dynamic
		EthRacer	Kolluri et al. (2019)	Dynamic
		Etherolic	Ashouri (2020)	Dynamic

(*continued on next page*)

- **Machine Learning**: An area of artificial intelligence that relies on algorithms and models that can learn from or make predictions from data (i.e., vulnerabilities). In this context, most of the techniques are based on supervised learning, which involves using labeled datasets to train models that classify data or predict outcomes for a particular output.

Table 2 (*continued*).

Categories	Techniques	Tools	References	Execution mode
Machine learning	Classical learning	ContractWard	Wang et al. (2021)	Static
		Eth2Vec	Ashizawa et al. (2021)	Static
		MODNN	Zhang et al. (2022c)	Static
		PSCVFINDER	Yu et al. (2023)	Static
		Peculiar	Wu et al. (2021)	Static
		Slicing matrix	Xing et al. (2020)	Static
		SmartDagger	Liao et al. (2022)	Static
		SmartMixModel	Shakya et al. (2022)	Static
		TMLVD	Zhou et al. (2022)	Static
		xFuzz	Xue et al. (2022)	Dynamic
	Deep learning	–	Gupta et al. (2022)	Static
		–	Liu et al. (2021)	Static
		–	Zhuang et al. (2020)	Static
		–	Liu et al. (2023a)	Static
		ASSBert	Sun et al. (2023)	Static
		CodeNet	Hwang et al. (2022)	Static
		ConvMHSA-SCVD	Li et al. (2023)	Static
		DeeSCVHunter	Yu et al. (2021)	Static
		EtherGIS	Zeng et al. (2022)	Static
		GraBit	Zhu et al. (2023)	Static
		MANDO-HGT	Nguyen et al. (2023)	Static
		RLF	Su et al. (2022)	Dynamic
		ReVulDL	Zhang et al. (2022a)	Static
		SCSVM	Yang et al. (2023)	Static
		VSCL	Mi et al. (2021)	Static
	Ensemble learning	Dynamit	Eshghie et al. (2021)	Dynamic

- *Classical Machine Learning*: refers to the traditional form of training models using algorithms (e.g., SVMs or decision trees) that require explicit feature engineering and manual selection of the best-performing models.
- *Deep Learning*: Deep learning is a subclass of machine learning that is based on training multiple-layer networks so that complex patterns and representations are extracted from raw data. In contrast to classical machine learning, the data used for training in deep learning are usually unstructured.
- *Ensemble Learning*: Ensemble learning refers to combining multiple machine learning algorithms. Usually, the objective is to improve performance and increase generalization capabilities by leveraging individual models' diversity and collective knowledge.

The next subsections go through each of the previously mentioned categories of techniques. Within each category, we discuss the following aspects: (i) **we characterize each work according to our research questions** and also other characteristics of interest (e.g., tool name, necessary inputs such as source code, supported languages, the scale of the evaluation, and targeted vulnerabilities); and (ii) **we briefly present one example for each technique**. Further observations are left for the Discussion Section. Notice that the dataset that supports this review is *online* and publicly available at Vidal et al. (2022).

3.1. Formal methods

Formal methods resort to logic and mathematics to check whether the behavior of a system satisfies a given property or sets of properties described by a formal specification (e.g., a model) (Edwards et al., 2002).

Fig. 1 presents an illustrative example. On the left-hand side, a set of specifications and assumptions serve as initial input, which is then formally modeled into a set of verifiable properties (related to security in this context). The contract also serves as input for the generation of a formal model. Then, the formal model will be checked against the properties. The process may result in either passing when the properties are satisfied or failing with an error, which may then be followed by a vulnerability localization process and bug fixing.

Tables 3 and 4 present the works that use formal verification techniques. For each work, we characterize the specific technique

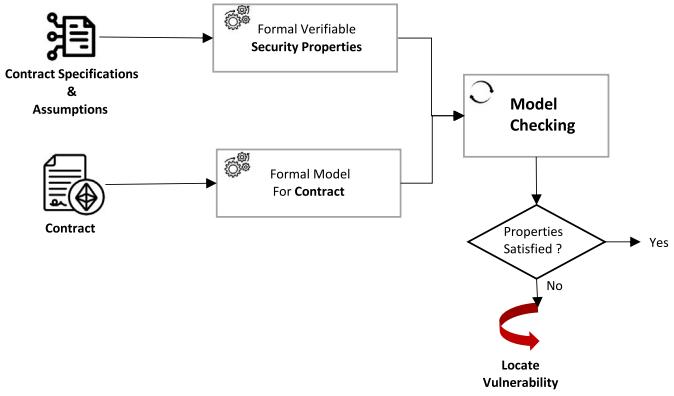

Fig. 1. General overview of model check procedure.
Source: Inspired by Liu and Liu (2019).

(e.g., model checking) used for formal verification (first column), the reference to the work (second column), the tool's name whenever available (third column), whether it uses the source code or bytecode for verification (fourth column), the supported languages (fifth column), size of dataset used for evaluating the performance of the tool (sixth column), and finally, the types of vulnerabilities claimed to be detected by each tool (seventh and eighth columns). To characterize the type of targeted vulnerabilities in each tool, we use DASP (Decentralized Application Security Project) (NCC Group, 2019) and SWC (Smart Contract Weakness Classification) (MythX, 2020) classifications. We selected these classifications for being the most frequently used in the identified works and also because they have two distinct levels of granularity. While DASP is coarse-grained (it classifies 9 + 1 types of issues), SWC is fine-grained and currently has a list of 37 types of vulnerabilities. We also correlate SWC vulnerabilities with DASP vulnerabilities. For instance, DASP-1 was associated with SWC-107 and SWC-104 because all these vulnerabilities refer to the same group (i.e., "Reentrancy"). Among all SWC vulnerabilities, we could not map 10 of them (i.e., SWC-100, SWC-111, SWC-117, SWC-121 SWC-122, SWC-123, SWC-127, SWC-129, SWC-134, and SWC-136) with DASP classifications. Thus, they were classified as DASP-10, which is dedicated to unknown vulnerabilities (e.g., SWC-102) ;

Table 3
Formal methods - Part I.

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
Abstract interpretation	Li et al. (2022b)	HFCCT	Source code	Golang	15	–	–	External Library Calling
						–	–	Field Declarations
						–	–	Golang Grammar Errors
						–	–	Map Structure Iteration
						–	–	Range Query Risks
						–	–	Unhandled Errors
						DASP-1	SWC-104	Cross Channel Chaincode Invocation
						DASP-10	SWC-119	Global Variable
						DASP-10	SWC-124	Read-Write Conflict
						DASP-10	SWC-136	External File Accessing
						DASP-10	SWC-136	System Command Execution
						DASP-10	SWC-136	Web Service
						DASP-5	SWC-113	Concurrency of Program
						DASP-6	SWC-120	Random Number Generation
						DASP-6	SWC-120	System Timestamp
						DASP-9	–	Reified Object Addresses
						DASP-9	–	Unchecked Input Arguments
	Grech et al. (2020)	MadMax	Bytecode	Solidity	91 800	DASP-3	SWC-101	Integer overflows
						DASP-5	SWC-113	Unbound mass operations
						DASP-7	SWC-126	Nonisolated calls (wallet griefing)
	Tsankov et al. (2018)	Securify	Source code Bytecode	Solidity	24 694	–	–	AssemblyUsage
						–	–	CallToDefaultConstructor
						–	–	ConstableStates
						–	–	ERC20Indexed
						–	–	ERC20Interface
						–	–	ERC721Interface
						–	–	ExternalFunctions
						–	–	LockedEther
						–	–	NamingConvention
						–	–	UnhandledException
						DASP-1	SWC-104	CallInLoop
						DASP-1	SWC-104	LowLevelCalls
						DASP-1	SWC-104	UnusedReturn
						DASP-1	SWC-107	DAO
						DASP-1	SWC-107	ReentrancyBenign
						DASP-1	SWC-107	ReentrancyNoETH
						DASP-10	SWC-109	UninitializedLocal
						DASP-10	SWC-109	UninitializedStateVariable
						DASP-10	SWC-109	UninitializedStorage
						DASP-10	SWC-119	ShadowedBuiltin
						DASP-10	SWC-119	ShadowedLocalVariable
						DASP-10	SWC-119	ShadowedStateVariable
						DASP-10	SWC-124	UnrestrictedWrite
						DASP-10	SWC-130	RightToLeftOverride
						DASP-10	SWC-131	UnusedStateVariable
						DASP-10	SWC-132	IncorrectEquality
						DASP-2	SWC-105	UnrestrictedEtherFlow
						DASP-2	SWC-106	UnrestrictedSelfdestruct
						DASP-2	SWC-108	StateVariablesDefaultVisibility
						DASP-2	SWC-112	UnrestrictedDelegateCall
						DASP-2	SWC-115	TxOrigin
						DASP-3	SWC-101	TooManyDigits
						DASP-3	SWC-103	SolcVersion
						DASP-7	SWC-114	TODAmount
						DASP-7	SWC-114	TODReceiver
						DASP-7	SWC-114	TODTransfer
						DASP-8	SWC-116	Timestamp
	Hu et al. (2023)	SoliDetector	Source code	Solidity	634	DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-106	Using suicide
						DASP-2	SWC-108	Usage state variable
						DASP-2	SWC-112	Delegate-call
						DASP-6	SWC-120	Usage of block randomness
						DASP-8	SWC-116	Timestamp dependency
	Ye et al. (2022)	Vulpedia	Source code	Solidity	1777	–	–	Unexpected revert
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-106	Self destruct
						DASP-2	SWC-115	Abuse of tx.origin

(*continued on next page*)

3.1.1. Abstract interpretation

Abstract interpretation is a static analysis technique for analyzing a program's behavior by approximating the possible states that an application can reach at runtime (Hirai, 2017). The technique makes use of abstract objects, based on the program semantics, which allows to simplify the execution but, at the same time, provides information

Table 3 (*continued*).

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
	Kalra et al. (2018)	Zeus	Source code	Solidity, C#, Go, Java.	22 493	DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-105	Unchecked send
						DASP-3	SWC-101	Integer overflow/underflow
						DASP-4	SWC-135	Failed send
						DASP-7	SWC-114	Transaction order dependence
						DASP-8	SWC-116	Block state dependence
						DASP-8	SWC-116	Transaction state dependence
Model checking	Crincoli et al. (2022)	–	Bytecode	Solidity	10 000	DASP-1	SWC-104	Unchecked low-level calls
						DASP-1	SWC-107	Reentrancy
						DASP-2	–	Access control
						DASP-3	SWC-101	Arithmetic
						DASP-5	SWC-113	Denial of Service
						DASP-6	SWC-120	Bad randomness
						DASP-7	–	Front running
						DASP-8	SWC-116	Time manipulation
						DASP-9	–	Short addresses
	Nishida Yuki and Saito et al. (2021)	HELMHOLTZ	Source code	Michelson	10	DASP-10	SWC-117	Signature Verification
	Wesley Scott and Christakis et al. (2022)	SmartAce	Source code	Solidity	256	–	–	Auction
						–	–	Ownable
						–	–	RefundEscrow
	Stephens et al. (2021)	SmartPulse	Source code	Solidity	191	–	–	Locked Funds
						–	–	Push DOS
						–	–	Revert DOS
						–	–	Unprotected Function
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-105	Unchecked Send
						DASP-3	SWC-101	Integer Overflow
						DASP-3	SWC-101	Integer Underflow
						DASP-5	SWC-128	Gas DOS
	So et al. (2020)	VeriSmart	Source code	Solidity	60	DASP-3	SWC-101	Division-by-zeros
						DASP-3	SWC-101	Integer overflow/underflow
	Mavridou et al. (2019)	VeriSolid	Source code Bytecode	Solidity	0	DASP-1	SWC-107	Reentrancy
						DASP-10	SWC-132	Deadlock-freedom
	Kalra et al. (2018)	Zeus	Source code	Solidity, C#, Go, Java.	22 493	DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-105	Unchecked send
						DASP-3	SWC-101	Integer overflow/underflow
						DASP-4	SWC-135	Failed send
						DASP-7	SWC-114	Transaction order dependence
						DASP-8	SWC-116	Block state dependence
						DASP-8	SWC-116	Transaction state dependence

regarding the actual computation. The main difficulty is that the level of abstraction used affects the precision of the analysis, which means that highly abstract models may not capture important aspects of the program's behavior. Thus, reaching the right tradeoff is a quite difficult aspect that must be properly managed so that the outcome is indeed representative of the important aspects of the system (Cousot, 2021).

Securify (Tsankov et al., 2018) is a static vulnerability detection tool that uses the EVM code of the smart contract to infer important semantic information (including control flow and data-flow facts). Securify infers mechanism consists of a set of base facts of the form instr(L, Y, X_1, ..., X_n) where instr is the instruction name, L is the instruction label, Y is the instruction storing the instruction result (if any), and X_i are the arguments. Based on inferred facts, Securify tries to discover security violations (e.g., Ether Liquidity, No Writes After Calls, Restricted Writes, Restricted Transfer, Handled Exception, Transaction Ordering Dependency, and Validated Arguments) or prove compliance with security-relevant instructions. The tool was also compared against Oyente (Luu et al., 2016) and Mythril (ConsenSys, 2021), two smart contract checkers based on symbolic execution.

Based on inferred facts, Securify tries to discover security violations (e.g., Ether Liquidity, No Writes After Calls, Restricted Writes, Restricted Transfer, Handled Exception, Transaction Ordering Dependency, and Validated Arguments) or prove compliance with security-relevant instructions. The tool was also compared against Oyente (Luu et al., 2016) and Mythril (ConsenSys, 2021), two smart contract checkers based on symbolic execution.

Grech et al. (2020) present a static tool, named *MadMax*, focused on analyzing gas depletion. After transforming the source code to the EVM bytecode, it is converted to control flow representation using abstract interpretation. Then, MadMax uses logic-based specifications to perform the analysis, specifically on loops. The tool focuses on detecting gas-focused vulnerabilities such as unbound mass operations (SWC-113), nonisolated calls (wallet griefing) (SWC-126), and integer overflows (SWC-101). The authors used the tool to analyze the whole Ethereum blockchain with about 91,800 contracts at the time, during 10 h. About 4.1% of the contracts were flagged as susceptible to unbounded iteration, 0.12% to wallet griefing, and 1.2% to overflows of loop induction variables. The main limitation of the tool is that it does not support cases involving multi-contract or whole-app attacks (i.e., the interaction between the off-blockchain part and its on-blockchain part). According to the authors, this is a challenging aspect of security analysis tools.

3.1.2. Model checking

Model checking is a technique that operates at a high-level abstraction by verifying whether a finite state model of a system meets a given formal specification or correctness without executing the code (Clarke et al., 1999). Although the technique allows proving that

Table 4
Formal methods - Part II.

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
Symbolic execution	Zhang et al. (2022b)	–	Source code Bytecode	Solidity	546	– DASP-2	– SWC-106	Ether leakage Suicide defects
	Huang et al. (2022)	DSE	Byte Code	Solidity	21 016	DASP-3	SWC-101	Integer Overflow
	Pasqua et al. (2023)	EtherSolve	Byte Code	Solidity	1000	DASP-1	SWC-107	reentrancy
	Li et al. (2022b)	HFCCT	Source code	Golang	15	– – – – – – DASP-1 DASP-10 DASP-10 DASP-10 DASP-10 DASP-10 DASP-5 DASP-6 DASP-6 DASP-9 DASP-9	– – – – – – SWC-104 SWC-119 SWC-124 SWC-136 SWC-136 SWC-136 SWC-113 SWC-120 SWC-120 – –	External Library Calling Field Declarations Golang Grammar Errors Map Structure Iteration Range Query Risks Unhandled Errors Cross Channel Chaincode Invocation Global Variable Read-Write Conflict External File Accessing System Command Execution Web Service Concurrency of Program Random Number Generation System Timestamp Reified Object Addresses Unchecked Input Arguments
	Wang Zexu and Wen et al. (2021)	MAR	Source code	Solidity	42	DASP-1	SWC-107	Reentrancy
	Fu et al. (2019)	MOPS	Source code	Solidity	1000	– – DASP-1 DASP-1 DASP-2 DASP-2 DASP-3 DASP-7	– – SWC-104 SWC-107 SWC-106 SWC-112 SWC-101 SWC-114	Mishandled exception Predictable variable dependency Unchecked return values Reentrancy Suicide Delegate call abuse Integer overflow TOD
	Yao et al. (2022)	Mythril Extension	Source code Bytecode	Solidity	113 472	DASP-1 DASP-1 DASP-2 DASP-3 DASP-7 DASP-8	SWC-104 SWC-107 SWC-115 SWC-101 SWC-114 SWC-116	Unchecked Call Return Value Reentrancy Authorization through tx.origin Integer Overflow and Underflow Transaction Order Dependence Block Values as a Proxy for Time
	Torres et al. (2018)	Osiris	Bytecode	Solidity	1 200 000	DASP-3	SWC-101	Arithmetic bugs, Truncation bugs and Signedness bugs
	Luu et al. (2016)	Oyente	Source code Bytecode	Solidity	19 366	– DASP-1 DASP-7 DASP-8	– SWC-107 SWC-114 SWC-116	Mishandled Exceptions Reentrancy Transaction-Ordering Dependence Timestamp dependence
	Ma et al. (2022)	Pluto	Source code	Solidity	39 443	DASP-1 DASP-3 DASP-8	SWC-107 SWC-101 SWC-116	Reentrancy Integer Overflow and Underflow Timestamp Dependency
	Chinen et al. (2020)	RA	Bytecode	Solidity	5	DASP-1	SWC-107	Reentrancy vulnerabilities
	Bose et al. (2022)	SailFish	Source code	Solidity	89 853	DASP-1 DASP-7	SWC-107 SWC-114	Reentrancy Transaction order dependence (TOD)
	Zhang et al. (2023)	Siguard	Source code Bytecode	Solidity	1000	DASP-10 DASP-10	SWC-117 SWC-122	Stateless Signature Verification Unseparated Signing Domain
	Wesley Scott and Christakis et al. (2022)	SmartAce	Source code	Solidity	256	– – –	– – –	Auction Ownable RefundEscrow
	Ma et al. (2023c)	TransRacer	Source Code	Solidity	50	DASP-7	SWC-114	transaction-ordering-dependent
	Chang et al. (2019)	sCompile	Source code	Solidity	36 099	– – DASP-2 DASP-7 DASP-9	– – SWC-106 SWC-126 –	Be no black hole Respect the limit Guard suicide Gassless send Avoid non-existing address

(*continued on next page*)

Table 4 (*continued*).

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
	Krupp and Rossow (2018)	teEther	Bytecode	Solidity	38 757	–	–	Code injection
						–	–	Direct value transfer
						–	–	Vulnerable state
						DASP-10	SWC-124	Storage modification
Theorem proving	Ayoade et al. (2019)	–	Bytecode	Solidity, Vyper	155 175	DASP-3	SWC-101	Integer overflow and integer underflow

certain properties of interest are fulfilled (e.g., the correctness of consensus protocols (Singh et al., 2019), identify block/transaction state dependence by eliminating infeasible paths (Kalra et al., 2018)), its application in complex settings is quite difficult due to the state explosion of the models, which frequently leads to cases where models are designed to represent a simplification of the real system. The assumptions used as the basis for the model simplification may be harmful to fulfilling the goal of proving that certain properties are indeed reached (Almakhour et al., 2020).

Kalra et al. (2018) present *Zeus*, which is a tool that combines abstract interpretation and symbolic model checking and makes use of Constrained Horn Clauses to verify contracts for safety properties. The tool takes as input the source code and a specification defined by the user and generates an eXtensible Access Control Markup Language (XACML) model (Sinnema and Wilde, 2013). The analysis is performed on an intermediate representation (i.e., the source code is transformed into a low-level intermediate representation such as LLVM bitcode (Lattner and Adve, 2004)) using the verification engine that leverages the constrained horn clauses. The tool showed to be able to detect 94.6% of vulnerable contracts correctly, with zero false negative rate and a low false positive rate.

Mavridou et al. (2019) present a model-based tool, namely *VeriSolid*, that applies formal verification at a high level of abstraction by checking whether contract behavior satisfies properties required by the developer. It notifies the developer in case of a violation, describing the execution sequence that led to it, thereby helping in the identification and correction of design errors that caused the incorrect behavior. It can detect two vulnerabilities: deadlock-freedom (SWC-132) and reentrancy (SWC-107). The tool warns the developer when a smart contract property is not satisfied prior to building it. However, the tool lacks resources to find vulnerabilities in more complex systems because, in these cases, often there is a need to validate the interaction between several smart contracts.

3.1.3. Symbolic execution

Symbolic execution is a method that systematically explores many possible execution paths without necessarily requiring concrete inputs. Instead of using actual input values, the algorithm evaluates the program using symbolic inputs, and a solver finds concrete values for those inputs that lead to errors (Baldoni et al., 2019).

Bose et al. present the SailFish vulnerability detection tool in Bose et al. (2022). The tool was built based on symbolic evaluation to detect reentrancy and transaction order dependency vulnerabilities. It operates in two phases. In the first phase, the algorithm uses some filters to reduce the number of relevant instructions (to the vulnerability). In the next stage, SailFish increases the precision of the search engine, by performing a refined value-summary analysis (i.e., an algorithm created by the owners), but constrains the scope of storage variables. According to the authors, the SAILFISH outperforms five smart contract analyzers (Securify, Mythril, Oyente, Sereum, and Vandal) regarding performance and precision.

Luu et al. (2016) present a symbolic execution tool, namely *Oyente*, that takes as input the bytecode of a contract to be analyzed and the current Ethereum global state and then outputs "problematic" symbolic paths to the users. It detects 4 types of vulnerabilities: transaction-ordering dependency (SWC-116), timestamp dependency (SWC-116), mishandled exceptions (Other), and reentrancy (SWC-107). According to the experimental results presented in Luu et al. (2016), from 19 366 existing Ethereum contracts, the tool flagged 8833 of them as vulnerable.

3.1.4. Theorem proving

Theorem proving is a method that relies on a mathematical proof to demonstrate the validity of a statement, i.e., the theorem. In a computer context, a Theorem prover is essentially a program that can provide automated support for verifying a finite state machine, a Markov model, a set of logical predicates, or a pseudo-code abstraction from the software without executing the code (Schumann, 2001) While the method allows for a formal demonstration that a certain statement is true, the complexity of applying the method (which is aggravated with the increasing complexity of the smart contracts) and limited automation is a major drawback (Hirai, 2017).

Ayoade et al. (2019) presents a framework that can detect vulnerable bytecode instructions and modify the Ethereum bytecode by inserting guard code to mitigate attacker exploits. The authors use a theorem prover to establish input–output equivalence between the original and modified code. The evaluation demonstrates that the system can enforce policies that protect against integer overflow and underflow vulnerabilities in real Ethereum contract bytecode, and overhead is measured in terms of instruction counts. This work is limited to identifying arithmetic vulnerabilities by searching for the occurrence of ADD and SUB instructions.

3.2. Code analysis

Code analysis is a method for inspecting code, mostly without running it, aiming to detect bugs, bad practices, or vulnerabilities. The method usually involves the identification of patterns, usually by either matching code blocks or some form of abstract representation of the code with known vulnerability patterns (Chess and McGraw, 2004). Fig. 2 presents a common code analysis technique, namely taint analysis. As we can see in Fig. 2, the main idea is to track the propagation of a possible tainted value (x in the figure) and each variable that derives from it (y1 and y2 in the figure) to a point in the code where the value is used in an insecure manner (i.e., a sensitive sink).

Table 5 overviews the code analysis tools identified and their characteristics. We use the same organization as previously in Table 3.

3.2.1. Pattern recognition

Pattern recognition is an analysis method to identify patterns in data. In our case, the patterns refer to keywords or excerpts in the source code. For instance, a possible occurrence of an integer overflow in a smart contract can be quickly identified by examining how an integer value is being incremented (e.g., by using the ++ operator). At the same time, such identification may be difficult to make if the new integer value is the result of some complex calculation (Broberg et al., 2004). Another example may be the identification of the use of insecure deprecated functions (e.g., sha3 should be replaced with keccak256, as discussed in SWC-111 (SWC Registry, 2018).

Tikhomirov et al. (2018) present a static analysis tool, namely *SmartCheck*, that translates Solidity source code into an intermediate

Table 5
Code analysis.

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
Pattern recognition	Tikhomirov et al. (2018)	SmartCheck	Source code	Solidity, Vyper	4600	–	–	Byte array
						–	–	Costly loop
						–	–	Locked money
						–	–	Malicious libraries
						–	–	Redundant fallback function
						–	–	Send instead of transfer
						–	–	Style guide violation
						–	–	Token API violation
						–	–	Transfer forwards all gas
						–	–	Unchecked math
						–	–	Unsafe type inference
						–	–	private modifier
						DASP-1	SWC-104	Unchecked external call
						DASP-1	SWC-107	Reentrancy
						DASP-10	SWC-102	Compiler version not fixed
						DASP-10	SWC-132	Balance equality
						DASP-2	SWC-108	Visibility level
						DASP-2	SWC-115	Tx.origin
						DASP-3	SWC-101	Integer division
						DASP-5	SWC-128	DoS by external contract
						DASP-8	SWC-116	Timestamp dependence
	Liao et al. (2022)	SmartDagger	Bytecode	Solidity	844	–	–	Taint for owner
						DASP-1	SWC-107	Reentrancy
						DASP-5	SWC-113	Dos Attack
						DASP-8	SWC-116	Timestamp manipulation
	Cui et al. (2022)	Vrust	Source code	Rust, C, and C++	1000	–	–	Account Confusion
						–	–	Cross Program Invocation
						–	–	Missing Owner Check
						DASP-10	SWC-122	Missing Signer Check
						DASP-3	SWC-101	Missing Key Check, Overflow/Underflow
						DASP-3	SWC-101	Numerical Precision Error
						DASP-6	SWC-120	Bump Seeds
	Li et al. (2022a)	EOSIOAnalyzer	Bytecode	C++	3963	–	–	Fake EOS Transfer
						–	–	Forged Transfer Notification
						DASP-10	SWC-117	Block Information Dependency
	Gao et al. (2019b)	EasyFlow	Source code	Solidity	0	DASP-3	SWC-101	Overflow detector
	Zhang et al. (2020a)	EthPloit	Source code	Solidity	45 308	–	–	Unsolvable constraints
						DASP-10	SWC-136	Exposed secret
						DASP-2	SWC-105	Unchecked transfer value
						DASP-2	SWC-105	Vulnerable access control
						DASP-8	SWC-116	Blockchain Effects Time Dependency
	Brent et al. (2020)	Ethainter	Source code	Solidity	100 000	–	–	Multiple Transaction
	Torres et al. (2018)	Osiris	Bytecode	Solidity	1 200 000	DASP-3	SWC-101	Arithmetic bugs, Truncation bugs and Signedness bugs
	Rodler et al. (2019)	Sereum	Bytecode	Solidity	93 942	DASP-1	SWC-107	Reentrancy
	Zhang et al. (2023)	Siguard	Source code Bytecode	Solidity	1000	DASP-10	SWC-117	Stateless Signature Verification
						DASP-10	SWC-122	Unseparated Signing Domain
	Feist et al. (2019)	Slither	Source code	Solidity, Vyper	1000	–	–	Arbitrary sending of ether
						–	–	Locked ether
						DASP-1	SWC-107	Reentrancy
						DASP-10	SWC-119	Shadowing
						DASP-10	SWC-131	Uninitialized variables
						DASP-2	SWC-106	Suicidal contracts
	Liao et al. (2022)	SmartDagger	Bytecode	Solidity	844	–	–	Taint for owner
						DASP-1	SWC-107	Reentrancy
						DASP-5	SWC-113	Dos Attack
						DASP-8	SWC-116	Timestamp manipulation

(*continued on next page*)

representation (i.e., XML-based) generated by ANTLR (ANTLR, 2019) parser. The vulnerabilities are checked by finding XPath (W3C, 2011) patterns. The authors evaluated the tool with 4600 verified contracts downloaded from Etherscan (2021) and concluded that 99.9% of the contracts have issues, with 63.2% having critical vulnerabilities. The most prevalent issue is Implicit visibility level, followed by DoS by external contract, Timestamp dependency, and reentrancy. The authors mentioned the opportunity to improve the grammar database by making patterns more precise and adding support for other languages.

Zhang et al. (2019a) present a static code analysis tool, namely *SolidityCheck*, that uses regular expressions to define the characteristics of problematic statements and then uses regular expression matching and program instrumentation to prevent or detect problems. It can locate 17 types of vulnerabilities in the source code of smart contracts. The authors evaluated the tool with 1363 smart contracts written in Solidity from EtherScan, showing benefits when compared with eight other analyzers (Remix Remix team, 2021, Mythx ConsenSys, 2019, Oyente Luu et al., 2016, Solhint Protofire, 2024, Securify Tsankov et al.,

Table 5 (*continued*).

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
						–	–	Exception-State
						–	–	abiencoderv2-array
						–	–	array-by-reference
						–	–	arrays
						–	–	assembly
						–	–	callstack
						–	–	complex-function
						–	–	constant-function-state
						–	–	controlled-array-length
						–	–	delete-dynamic
						–	–	erc20-approve
						–	–	erc20-indexed
						–	–	erc20-throw
						–	–	function-init-state
						–	–	function-problem
						–	–	length-manipulation
						–	–	modulo
						–	–	msgvalue-equals-zero,
						–	–	mul-var-len-arguments
						–	–	multiple-calls-in-transaction
						–	–	naming-convention
						–	–	overpowered-role
						–	–	public-mappings-nested
						–	–	redundant-fallback
						–	–	shift-parameter-mixup
Taint analysis						–	–	solidity-dos-with-throw
						–	–	taint-pass-project
						–	–	unprotected-upgrade
						–	–	upgrade-050
						DASP-1	SWC-104	low-level-calls
						DASP-1	SWC-104	unchecked-lowlevel
						DASP-1	SWC-107	reentrancy-eth
						DASP-1	SWC-107	reentrancy-limited-events
						DASP-1	SWC-107	reentrancy-limited-gas
						DASP-1	SWC-107	reentrancy-limited-gas-no-eth
						DASP-1	SWC-107	reentrancy-no-eth
						DASP-1	SWC-107	repeat-call
	Li et al. (2022c)	SmartFast	Source code	Solidity	13 687	DASP-10	SWC-102	solc-version
						DASP-10	SWC-109	uninitialized-local
						DASP-10	SWC-109	uninitialized-state
						DASP-10	SWC-109	uninitialized-storage
						DASP-10	SWC-110	assert-state-change
						DASP-10	SWC-111	deprecated-standards
						DASP-10	SWC-119	names-reused
						DASP-10	SWC-119	shadowing -state
						DASP-10	SWC-119	shadowing-builtin
						DASP-10	SWC-119	similar-names
						DASP-10	SWC-124	storage-array
						DASP-10	SWC-124	writeto-arbitrarystorage
						DASP-10	SWC-129	typographical-error
						DASP-10	SWC-130	rtlo
						DASP-10	SWC-134	hardcoded
						DASP-2	SWC-105	unchecked-send
						DASP-2	SWC-106	suicidal
						DASP-2	SWC-112	controlled-delegatecall
						DASP-2	SWC-112	parity-multisig-bug
						DASP-2	SWC-115	tx-origin
						DASP-2	SWC-118	incorrect-constructor
						DASP-2	SWC-118	multiple-constructors
						DASP-2	SWC-118	reused-constructor
						DASP-3	SWC-101	divide-before-multiply
						DASP-3	SWC-101	division
						DASP-3	SWC-101	integer-overflow
						DASP-3	SWC-101	integer-underflow
						DASP-3	SWC-101	signedness
						DASP-3	SWC-101	truncation
						DASP-3	SWC-103	pragma
						DASP-3	SWC-103	weak-prng
						DASP-4	SWC-135	unimplemented-functions
						DASP-4	SWC-135	unused-return
						DASP-5	SWC-113	arbitrary-send
						DASP-5	SWC-113	continue-in-loop
						DASP-5	SWC-128	costly-loop
						DASP-7	SWC-114	TOD-ether
						DASP-7	SWC-114	TOD-receiver
						DASP-9	–	missing-input-validation

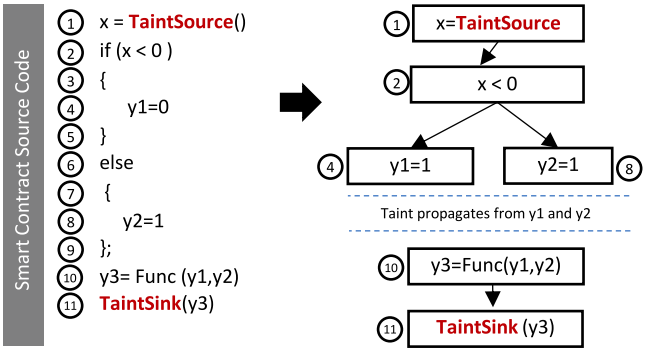

Fig. 2. Taint analysis adapted from Song et al. (2015).

2018, SmartCheck Tikhomirov et al., 2018, ContractFuzzer Jiang et al., 2018, Osiris Torres et al., 2018) in terms of detection quality and efficiency. As a disadvantage, SolidityCheck does not provide much useful feedback (i.e., problematic statements) to developers, which results in a user experience that could be improved.

3.2.2. Taint analysis

Taint analysis aims at identifying and tracking the flow of data within a program with the main goal of determining if a certain unanticipated input (possibly malicious and named 'tainted' in this context) can affect program execution in a harmful manner. By tracking the tainted data, it is possible to identify points in the code where data validation or sanitization is required to avoid security issues (Xu et al., 2018). This technique can be carried out statically (Ghaleb et al., 2022) or dynamically (Ji et al., 2021), i.e., respectively, without or with running the code. In the case of static taint analysis, the main challenges include modeling all possible execution paths in the code, inter-contract analysis is challenging, and keeping low false positive rates is also a known issue frequently associated with such static techniques. Regarding dynamic taint analysis, the quality of the workload used may influence the overall understanding of the different code executions and, hence, determine the quality of the outcome (e.g., a smart contract vulnerability may not be activated because the workload does not have sufficient coverage). Also, if external systems are involved, a dynamic technique may struggle to understand the exact impact on the program's behavior.

Torres et al. (2018) present a tool, namely *Osiris*, that combines taint analysis with symbolic execution to find integer bugs, including arithmetic, truncation, and signedness bugs in the EVM bytecode of smart contracts. The authors evaluated the tool's performance on a large experimental dataset containing the bytecode of more than 1.2 million smart contracts from the public Ethereum blockchain. They were able to detect integer vulnerabilities in 42,108 contracts. Osiris does not support tracking taint across multiple contracts (inter-contract analysis) and across different method invocations (trace analysis).

Feist et al. (2019) present *Slither*, a static code analysis tool that uses taint analysis. The tool compiles the Solidity smart contract source code into a Solidity *Abstract Syntax Tree* (AST) to extract the contract's inheritance graph, the control flow graph (CFG), and the list of expressions. Then, it transforms the code of the contract to SlithIR, its internal representation language, that uses static single assessment (SSA) to facilitate the computation of several code analyses. It also includes a Graph for Code understanding and Assisted code review. The authors evaluated the tool in the top 1,000 most used smart contracts (i.e., contracts that had the largest number of transactions at the time) to find that it outperformed three other popular tools (Solhint Protofire, 2024, SmartCheck Tikhomirov et al., 2018, Securify Tsankov et al., 2018) in terms of performance, robustness (i.e., whether the tool completed the execution), and effectiveness (i.e., considering the number of reported false positives).

3.3. Software testing

Software testing techniques require that the software is executed to verify certain functional or non-functional properties. The intention is to discover defects (i.e., software faults or vulnerabilities). Fig. 3 presents the fuzzing technique, which we found to be a quite common testing method in the context of smart contracts. The main idea behind this technique is that information about the smart contract interface is passed on to the fuzzer, which then uses a specific mechanism (e.g., random, malicious, dictionary-based) to generate a workload that is then sent to the running system as input. The system is monitored throughout the executions to identify errors, failures, or any anomalous behavior (Singh and Singh, 2012). Table 6 presents the software testing works and their corresponding characteristics, identified in the context of smart contracts.

3.3.1. Fuzzing

Fuzzing is a dynamic vulnerability detection technique that is based on generating/managing irregular test data input into a target program to find a vulnerable state during program execution (Chen et al., 2018). This method may be quite easy to use (the point of view is external), but at the same time may depend on the quality of the workload to achieve good code coverage. As a typical black-box method, the lack of information about the internal structures of the code is an obstacle to more accurate vulnerability detection. Also, certain types of bugs (i.e., the more complex ones, such as reentrancy) may be difficult to activate using a fuzzer.

Jiang et al. (2018) present a fuzz testing tool, namely *ContractFuzzer*, that generates fuzzing inputs based on the ABI specifications (i.e., Application Binary Interface) of smart contracts, defines new test oracles for different types of vulnerabilities, instruments the EVM to log smart contract runtime behavior, and then analyzes these logs to detect security vulnerabilities. The authors compared the tool with Oyente with 6991 fuzzed smart contracts extracted from Etherscan, showing ContractFuzzer can detect a higher number of vulnerabilities, flagging a total of more than 459 vulnerabilities.

Ashraf et al. (2020) present a fuzz testing tool, namely *GasFuzzer*, that introduces a gas-greedy strategy that seeks to select the transactions with the highest gas consumption (to be used in a mutation process). Then, GasFuzzer reduces the gas of those transactions (i.e., the authors name it gas-leveling) and begins fuzzing the smart contract (i.e., the execution considers mutated and non-mutated transactions together). The tool mainly aims at gas-oriented Mishandled exceptions vulnerabilities.

3.3.2. Concolic execution

Concolic execution is a technique that combines both concrete (real) and symbolic (representative) execution of a program. First, the program behavior is observed with real values, and then the program is executed symbolically, which allows for exploring different paths without specific inputs. With the progress of symbolic execution, constraints regarding program variables are collected. Constraint solvers are then used to understand if certain paths are feasible and to generate new inputs that satisfy such constraints (so that potential vulnerabilities are uncovered) (Kolluri et al., 2019).

Ashouri (2020) presents *Etherolic*, a tool that uses dynamic taint tracking and concolic testing to analyze the bytecode level of smart contracts running on the EVM. It can identify 10 security issues: integer overflow (SWC-101) and underflow (SWC-101), bad randomness (SWC-120), reentrancy (SWC-107), locked ether (Other), mishandled exceptions (Other), denial of service (SWC-113), short address attack (DASP-9), race condition (SWC-114), shadow memory (SWC-119). The authors evaluated the tool with their own synthetic benchmark, namely DummyContracts, which contains 128 buggy contracts written in Solidity 1.3.5, with the aforementioned security issues, resulting in the identification of 204 actual security violations.

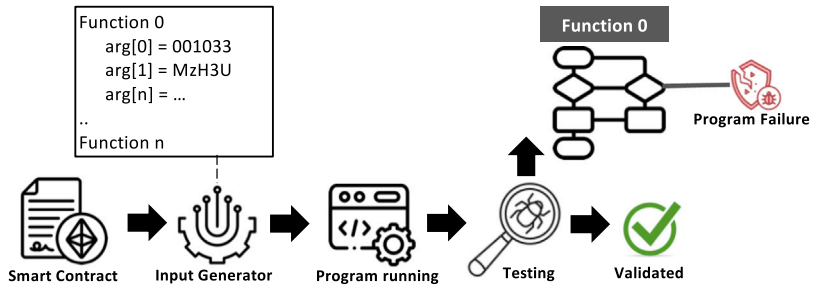

Fig. 3. Fuzzing testing approaches inspired from Feng et al. (2019b).

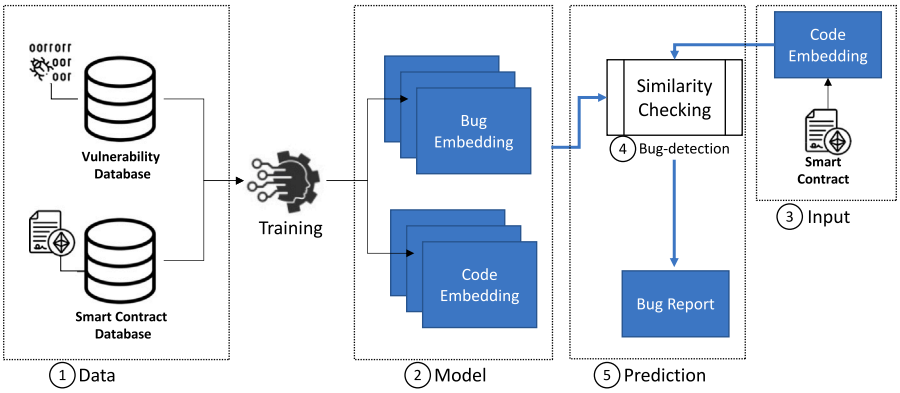

Fig. 4. Machine Learning techniques adapted from Gao et al. (2019a).

Kolluri et al. (2019) present *EthRacer* that analyzes EVM bytecode of smart contracts and checks whether changing the ordering of input events (function invocations) of a contract results in differing outputs, e.g., to identify Event-Ordering bugs. It works by constructing inputs and systematically trying all possible function orderings until a budgeted timeout is reached. The approach combines symbolic execution of contract events with fast randomized fuzzing of event sequences. The authors compared their tool against Oyente, concluding that it detects more vulnerabilities in a set of 5000 contracts for which source code was available and in 5000 contracts randomly selected from the Ethereum blockchain, where only the bytecode was available. Overall, it detected 789 contracts with synchronization bugs and 47 contracts with linearizability bugs.

3.4. Machine learning

Recently, machine learning techniques have started to be applied for vulnerability detection in smart contracts (Gao et al., 2019a). Fig. 4 presents a general view of the solution implemented by SmartEmbed (Gao et al., 2019a). On the left-hand side of the figure, two databases are used for training the model: vulnerability dataset (i.e., cataloged vulnerabilities) and smart contract dataset (i.e., a set of programs that have known faults). On the right-hand side of the figure, the smart contract to be analyzed is given as input, and the algorithm starts to check (i.e., based on the prior computed model from the left-hand side of the figure) if there are any vulnerabilities in the code. Table 7 presents the works that rely on machine learning for vulnerability detection.

3.4.1. Classical learning

Classical machine learning refers to methods that use data (labeled or unlabeled) to learn data-related aspects (e.g., patterns, relations, representations) with the goal of creating a model that is able to make decisions or predictions over new data, without the need for explicitly programming such decisions (Blum and Mitchell, 1998). In the context of smart contracts, their application to vulnerability detection is quite recent, but the typical problems generally apply. In particular, balancing false positive and false negative rates and having sufficiently balanced data for training the models, which in this context may be difficult because the occurrence of vulnerabilities is scarce (when compared with actual code without known vulnerabilities) (França et al., 2023).

Wang et al. (2021) present an automated vulnerability detection tool for Ethereum smart contracts, namely *ContractWard*. To build the dataset for training the models, the authors collected a large number of contracts, transformed them into operation codes (opcodes), simplified them, extracted 1619 features from simplified operation codes of smart contracts, labeled them, and then employed five machine learning algorithms and two sampling algorithms to build the models. The authors demonstrate the effectiveness and efficiency of the tool against 49,502 real-world smart contracts running on Ethereum. The results show that the predictive Micro-F1 and Macro-F1 (Takahashi et al., 2022) of the tool are over 96% and the average detection time is 4 s on each smart contract when it uses XGBoost for training the models and SMOTETomek for balancing the training sets.

SmartDagger (Liao et al., 2022) was developed for cross-contract analysis. To accomplish this, the authors implemented semantic recovery via neural machine translation, which detects connections between two or more contracts at the EVM bytecode level. The authors noted that cross-contract analysis is an emerging field and expressed the lack of related works, at the time, available for comparison purposes.

3.4.2. Deep learning

Deep learning is a subfield of machine learning, where methods are based on artificial neural networks with multiple layers. The corresponding models are able to capture complex aspects from the data but require large amounts of data to be trained (Janiesch et al., 2021). This is an issue in the context of smart contract vulnerabilities, where such information is relatively scarce (França et al., 2023).

Zhuang et al. (2020) present a graph neural networks (GNNs) approach for smart contract vulnerability detection. The authors used a technique to represent the program in a graph that represents both the

Table 6
Software testing tools.

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
Concolic execution	Torres et al. (2021)	ConFuzzius	Source code	Solidity	21 128	–	–	Leaking Ether
						–	–	Locking Ether
						–	–	Unhandled Exception
						DASP-1	SWC-107	Reentrancy
						DASP-10	SWC-110	Assertion Failure
						DASP-2	SWC-106	Unprotected Selfdestruct
						DASP-2	SWC-112	Unsafe Delegatecall
						DASP-3	SWC-101	Integer Overflows
						DASP-6	SWC-120	Block Dependency
						DASP-7	SWC-114	Transaction Order Dependency
	Kolluri et al. (2019)	EthRacer	Bytecode	Solidity	5000	DASP-8	SWC-116	Event-ordering (EO) bugs
	Ashouri (2020)	Etherolic	Bytecode	Solidity	140	–	–	Mishandled exceptions
						DASP-1	SWC-107	Locked Ether
						DASP-10	SWC-119	Shadow memory
						DASP-3	SWC-101	Bad randomness
						DASP-3	SWC-101	Integer overflow and underflow
						DASP-5	SWC-113	Denial of Service
						DASP-6	SWC-120	Reentrancy
						DASP-7	SWC-114	Race condition
						DASP-9	–	Short Address Attack
Fuzzing	Jiang et al. (2018)	ContractFuzzer	Bytecode	Solidity	6991	–	–	Freezing Ether
						–	–	Mishandled exceptions
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-112	Dangerous DelegateCall
						DASP-6	SWC-120	Block number dependency
						DASP-7	SWC-126	Gassless send
						DASP-8	SWC-116	Timestamp dependency
	Grieco et al. (2020)	Echidna	Fuzzing	Solidity, Vyper	415	–	–	Addresses cannot be set to incorrect values
						–	–	Invalid Input Data
						DASP-3	SWC-101	Integer overflows
						DASP-7	SWC-114	Transaction-Ordering Dependence
	Zhang et al. (2020a)	EthPloit	Source code	Solidity	45 308	–	–	Unsolvable constraints
						DASP-10	SWC-136	Exposed secret
						DASP-2	SWC-105	Unchecked transfer value
						DASP-2	SWC-105	Vulnerable access control
						DASP-8	SWC-116	Blockchain Effects Time Dependency
	Nassirzadeh Behkish and Sun et al. (2023)	Gas Gauge	Source code	Solidity	997	DASP-5	SWC-128	gas limit DoS on a contract via unbounded operations
	Ashraf et al. (2020)	GasFuzzer	Bytecode	Solidity	3170	–	–	Freezing Ether
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-112	Dangerous DelegateCall
						DASP-6	SWC-120	Block No. Dependency
						DASP-7	SWC-126	Gasless Send
						DASP-8	SWC-116	Timestamp Dependency
	Ding et al. (2021)	HFContractFuzzer	Source code	Golang	5	–	–	Logic loopholes
						DASP-3	SWC-101	Integer overflow
	Liu et al. (2023b)	IR-Fuzz	Source Code	Solidity	12 000	DASP-1	SWC-104	UNCHECKED EXTERNAL CALL
						DASP-1	SWC-107	Reentrancy
						DASP-10	SWC-132	DANGEROUS ETHER STRICT EQUALITY
						DASP-3	SWC-101	INTEGER OVERFLOW
						DASP-7	SWC-114	Block Number Dependency
						DASP-8	SWC-116	DANGEROUS DELEGATECALL
	Shou et al. (2023)	ItyFuzz	Bytecode	Solidity	150 000	–	–	Price Manipulation
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-115	Access Control
						DASP-5	SWC-113	DoS
	Ma et al. (2023a)	Pied-Piper	Source code Bytecode	Solidity	13 484	–	–	Arbitrarily Transfer Threat
						–	–	Destroy Tokens
						–	–	Disable Transferring
						–	–	Freeze Account
						–	–	Generate Tokens After ICO
	Su et al. (2022)	RLF	Bytecode	Solidity	1291	–	–	Ether Frezzing
						–	–	Ether Leaking
						–	–	Unhandled Exception
						DASP-2	SWC-106	Suicidal Contract
						DASP-2	SWC-112	Dangerous Delegatecall
						DASP-8	SWC-116	Block State Dependency

(*continued on next page*)

Table 6 (*continued*).

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
	Li et al. (2022d)	ReDefender	Source code	Solidity	4776	DASP-1	SWC-107	Reentrancy
	Rodler et al. (2019)	Sereum	Bytecode	Solidity	93 942	DASP-1	SWC-107	Reentrancy
	Wesley Scott and Christakis et al. (2022)	SmartAce	Source code	Solidity	256	–	–	Auction
						–	–	Ownable
						–	–	RefundEscrow
	Pani et al. (2023)	SmartFuzzDriverGen	Source code	Golang	19	–	–	Buffer-overflow
						–	–	Buffer-overwrite
						–	–	Buffer-underflow
						DASP-3	SWC-101	Divide-by-zero
	Akca et al. (2019)	Solanalyser	Source code Bytecode	Solidity	12 866	DASP-1	SWC-104	Unchecked send
						DASP-1	SWC-107	Repetitive call
						DASP-10	SWC-129	Out of gas
						DASP-2	SWC-115	TxOrigin
						DASP-3	SWC-101	Division by zero
						DASP-3	SWC-101	Overflow/Underflow
						DASP-8	SWC-116	Timestamp dependency
	Khan and Namin (2023)	TechyTech	Source code	Solidity	0	DASP-2	SWC-106	selfdestruct
	Chen et al. (2022)	WASAI	Source code	C++	1091	–	–	Fake EOS
						–	–	Fake Notification
						–	–	Missing Authorization Verification
						–	–	Rollback
						DASP-6	SWC-120	Blockinfo Dependency
	Nguyen et al. (2020)	sFuzz	Source code Bytecode	Solidity	4112	–	–	Freezing Ether
						–	–	Mishandled exceptions
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-112	Unchecked delegatecall function
						DASP-3	SWC-101	Integer overflow and underflow
						DASP-6	SWC-120	Block number dependency
						DASP-7	SWC-126	Gasless send
						DASP-8	SWC-116	Timestamp dependency
	Ma et al. (2023b)	vGas	Byte Code	Solidity	736	–	–	unknown out-of-gas
	Xue et al. (2022)	xFuzz	Source code	Solidity	7391	DASP-1	SWC-107	reentrancy
						DASP-2	SWC-112	delegatecall
						DASP-2	SWC-115	tx-origin

syntactic and semantic structures of a smart contract function. Nodes in the graph represent critical function invocations or variables, while edges capture their temporal execution traces. To improve the performance, they proposed an elimination phase consisting of criteria to remove unwanted nodes and edges. The experiment was performed using 300,000 smart contracts, and according to the authors, the solution consistently outperforms state-of-the-art methods in detecting different types of vulnerabilities related to reentrancy, timestamp dependencies, and infinite loop vulnerabilities.

Zhang et al. (2022a) presents ReVulDL, a tool focused on detecting the Reentrancy (SWC-107) vulnerability. The tool operates in two phases: vulnerability detection and vulnerability localization. In the first phase, the tool utilizes a graph-based pre-training model to learn relationships in the contract paths to identify reentrancy vulnerabilities. If the first step generates a positive alarm, the localization phase is activated. This latter phase relies on interpretable machine learning to pinpoint suspicious statements in the code and determine whether the detected alarm indicates a reentrancy vulnerability or not.

3.4.3. Ensemble learning

Ensemble learning is a technique that uses a combination of multiple machine learning algorithms as a strategy to achieve greater results. It has the potential to combine the strengths of the individual methods (Surucu et al., 2022). For instance, a certain method may perform better in detecting a specific group of vulnerabilities, while another may be better with other groups. Despite this, the ensemble must be properly tuned so that the best tools for a certain scenario are favored in the decision, which is a problem that is hard to generalize (Zhou, 2012). .

Eshghie et al. presents the tool *Dynamit* in Eshghie et al. (2021). Unlike other tools, which attempt to find vulnerabilities before the smart contract is deployed, Dynamit analyzes the content of transactions when the program is already running. The authors limited the project's scope, proposing to detect only reentrancy vulnerabilities. The architecture is divided into parts: (1) the monitor, which reads transactions in the blockchain, and (2) the detector, which classifies the content as benign or malicious. The experiment was performed by using 25 contracts, generating 105 transactions, in which the tool was able to achieve 94% accuracy.

4. Findings and discussion

In this section, we discuss the main findings identified during our analysis of the state of the art by going through the research questions presented earlier in Section 2.2.

4.1. RQ-1: Which techniques are being used for detecting vulnerabilities in smart contracts?

Techniques identified in our literature review are generally classified into two groups regarding Execution Mode (as indicated in the last column of Table 2): static or dynamic. Our analysis reveals that most tools fall into the static group (61 out of 93). In addition to this general classification, it is important to know the distribution of the tools across the identified techniques. Fig. 5 shows the distribution of the identified works with respect to the four categories identified in Section 3 and their corresponding techniques. The top pie chart (a) details the prevalence of the different categories, whereas charts (b) to (e) show the prevalence of each specific technique within each category. As shown in the top pie chart, most of the existing works fit in the Formal Methods category, which is then followed by Machine Learning and then Software Testing. Code Analysis is the minority class.

Table 7
Machine learning techniques.

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
Classical learning	Wang et al. (2021)	ContractWard	Bytecode	Solidity	49 502	–	–	Callstack Depth Attack
						DASP-1	SWC-107	Reentrancy
						DASP-3	SWC-101	Integer Overflow
						DASP-3	SWC-101	Integer Underflow
						DASP-7	SWC-114	TOD
						DASP-8	SWC-116	Timestamp dependency
	Ashizawa et al. (2021)	Eth2Vec	Bytecode	Solidity	5000	–	–	ERC-20 Transfer
						–	–	Gas Consumption
						–	–	Implicit Visibility
						DASP-1	SWC-107	Reentrancy
						DASP-3	SWC-101	Integer Overflow
						DASP-3	SWC-101	Integer Underflow
						DASP-8	SWC-116	Time Dependency
	Zhang et al. (2022c)	MODNN	Source code Bytecode	Solidity	18 000	–	–	CallDepth
						–	–	CheckEffects
						–	–	InlineAssembly
						DASP-1	SWC-104	LowlevelCalls
						DASP-1	SWC-107	Reentrancy
						DASP-10	SWC-110	AssertFail
						DASP-2	SWC-115	TxOrigin
						DASP-3	SWC-101	Overflow
						DASP-3	SWC-101	Underflow
						DASP-6	SWC-120	BlockTimestamp
						DASP-6	SWC-120	TimeDep
						DASP-7	SWC-114	TOD
	Yu et al. (2023)	PSCVFINDER	Source code	Solidity	200 000	DASP-1	SWC-107	reentrancy
						DASP-8	SWC-116	timestamp dependency
	Wu et al. (2021)	Peculiar	Source code	Solidity	20 000	DASP-1	SWC-107	Reentrancy
	Xing et al. (2020)	Slicing matrix	Bytecode	Solidity	19 145	–	–	is_greedy
						DASP-3	SWC-101	has_flows
						DASP-9	–	has_short_address
	Liao et al. (2022)	SmartDagger	Bytecode	Solidity	844	–	–	Taint for owner
						DASP-1	SWC-107	Reentrancy
						DASP-5	SWC-113	Dos Attack
						DASP-8	SWC-116	Timestamp manipulation
	Shakya et al. (2022)	SmartMixModel	Source code Bytecode	Solidity	70 000	–	–	Costly loop
						–	–	Extra gas consumption
						–	–	Locked Money
						–	–	Send instead of transfer
						–	–	Unsafe arrays length manipulation
						DASP-10	SWC-132	Checking for strict balance equality
						DASP-2	SWC-115	Using tx.origin for authorization
						DASP-4	SWC-135	Redundant Fallback function
						DASP-5	SWC-128	ETH transfer inside the loop
						DASP-9	–	Hardcoded Address
	Zhou et al. (2022)	TMLVD	Source code	Solidity	10 567	–	–	Exception State
						–	–	Use Predicable Variable
						DASP-1	SWC-107	Reentrancy
						DASP-3	SWC-101	Integer Overflow
						DASP-3	SWC-101	Integer Underflow
	Xue et al. (2022)	xFuzz	Source code	Solidity	7391	DASP-1	SWC-107	reentrancy
						DASP-2	SWC-112	delegatecall
						DASP-2	SWC-115	tx-origin

(*continued on next page*)

Among all techniques, Fuzzing is the most popular one found in 20 works, followed by Symbolic Execution, which is used in 17 papers, and Deep Learning with 15 cases. Other significant cases include Taint Analysis and Classical Learning, which were found in 10 papers. In contrast, Theorem Proving is the least common technique with 1 case.

Fig. 6 presents the prevalence of the different techniques across the years. It is clear that the number of works has been steadily increasing until the time of writing (December 2023). Notice that, very recently, the prevalence of machine learning techniques has been increasing. There is some fluctuation in the static analysis category, formal verification, and software testing techniques in which they have experienced no increase in numbers overall.

Several tools actually use a combination of two or more techniques. For instance, Osiris (Torres et al., 2018) performs symbolic execution to detect known vulnerabilities, but due to the limitation of symbolic execution to analyze vulnerabilities across multiple contracts, it uses the taint analysis to provide a more effective solution. The main reason behind combining approaches is to complement one technique with another by covering their limitations. For instance, static analysis techniques are able to detect many types of vulnerabilities, but they usually produce a high number of false alarms. Several tools have combined static with runtime techniques (e.g., ethPloit (Zhang et al., 2020a) and Sereum (Rodler et al., 2019) combined Taint analysis with Fuzzing). Fuzzing is the technique most combined with others.

Some of the tools identified are also frequently analyzed in other works, particularly in terms of their effectiveness in detecting defects, often to facilitate comparisons with alternative approaches. We highlight the following top 3 tools in this context:

Table 7 (*continued*).

Technique	Reference	Tool name	Input	Supported language(s)	Data size	DASP	SWC	Vulnerability name
Deep learning	Gupta et al. (2022)	–	Source code	Solidity	7000	–	–	ether lost transferred
						–	–	immutable bugs
						DASP-1	SWC-107	Reentrancy bugs
						DASP-2	SWC-115	TX.origin
						DASP-3	SWC-101	integer overflow and underflow
						DASP-3	SWC-101	zero division risk
						DASP-6	SWC-120	generating randomness
						DASP-7	SWC-114	transaction ordering
						DASP-8	SWC-116	time-stamp expansion
	Liu et al. (2021)	–	Source code	Solidity	40 000	–	–	infinite loop
						DASP-1	SWC-107	Reentrancy
						DASP-8	SWC-116	timestamp dependence
	Zhuang et al. (2020)	–	Source code	Solidity	300 000	DASP-1	SWC-107	Reentrancy
						DASP-10	SWC-129	infinite loop vulnerabilities
						DASP-8	SWC-116	timestamp dependence
	Liu et al. (2023a)	–	Source code	Solidity	40 932	DASP-1	SWC-107	Reentrancy
						DASP-10	SWC-134	CALL WITH HARDCODE GAS AMOUNT
						DASP-2	SWC-112	CODE INJECTION
						DASP-8	SWC-116	TIMESTAMP DEPENDENCY
	Sun et al. (2023)	ASSBert	Source code	Solidity	20 829	–	–	CallDepth
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-115	TxOrgin
						DASP-3	SWC-101	Flow
						DASP-6	SWC-120	Timestamp
						DASP-7	SWC-114	TOD
	Hwang et al. (2022)	CodeNet	Source code	Solidity	13 522	DASP-1	SWC-104	Unchecked LL Calls
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-115	Tx. Origin
						DASP-8	SWC-116	Timestamp Dependency
	Li et al. (2023)	ConvMHSA-SCVD	Source code	Solidity	5340	–	–	Infinite loop
						DASP-1	SWC-107	Reentrancy
						DASP-8	SWC-116	Timestamp dependency
	Yu et al. (2021)	DeeSCVHunter	Source code	Solidity	40 932	DASP-1	SWC-107	Reentrancy
						DASP-8	SWC-116	Time dependence
	Zeng et al. (2022)	EtherGIS	Source code	Solidity	10 000	DASP-1	SWC-107	reentrancy
						DASP-2	SWC-106	self destruct
						DASP-2	SWC-112	delegate call
						DASP-2	SWC-115	Tx. Orig
						DASP-7	SWC-114	TOD
						DASP-8	SWC-116	Timestamp
	Zhu et al. (2023)	GraBit	Source code	Solidity	47 398	DASP-1	SWC-107	reentrancy
	Nguyen et al. (2023)	MANDO-HGT	Source code Bytecode	Solidity	55 000	DASP-1	SWC-104	Unchecked Low Level Calls
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-106	Access Control
						DASP-3	SWC-101	Arithmetic
						DASP-5	SWC-113	Denial of Service
						DASP-7	SWC-114	Front Running
						DASP-8	SWC-116	TimeManipulation
	Su et al. (2022)	RLF	Bytecode	Solidity	1291	–	–	Ether Frezzing
						–	–	Ether Leaking
						–	–	Unhandled Exception
						DASP-2	SWC-106	Suicidal Contract
						DASP-2	SWC-112	Dangerous Delegatecall
						DASP-8	SWC-116	Block State Dependency
	Zhang et al. (2022a)	ReVulDL	Source code	Solidity	47 398	DASP-1	SWC-107	Reentrancy
	Yang et al. (2023)	SCSVM	Source code	Solidity	47 587	DASP-1	SWC-104	Unchecked LL Calls
						DASP-1	SWC-107	Reentrancy
						DASP-2	SWC-115	Tx.origin,
						DASP-7	SWC-114	Timestamp Dependency
	Mi et al. (2021)	VSCL	Bytecode	Solidity	205 848	DASP-1	SWC-107	reentrancy vulnerability
Ensemble learning	Eshghie et al. (2021)	Dynamit	Bytecode	Solidity, Vyper	105	DASP-1	SWC-107	Reentrancy

(1) **Oyente** (Luu et al., 2016), which is based on symbolic execution and is the most frequently compared tool, namely in Zeus (Kalra et al., 2018), VeriSmart (So et al., 2020), ContractFuzzer (Jiang et al., 2018), Sereum (Rodler et al., 2019), sFuzz (Nguyen et al., 2020), EthRacer (Kolluri et al., 2019), Solanalyser (Akca et al., 2019), SmartDagger (Liao et al., 2022), SmartCheck (Tikhomirov et al., 2018), Peculiar (Wu et al., 2021), Pluto (Ma et al., 2022), Clairvoyance (Clarivate, 2021) and RA (Chinen et al., 2020).

(2) **Securify** (Tsankov et al., 2018) is a tool based on Abstract Interpretation that has been compared with the following works identified in this review, including Peculiar (Wu et al., 2021),

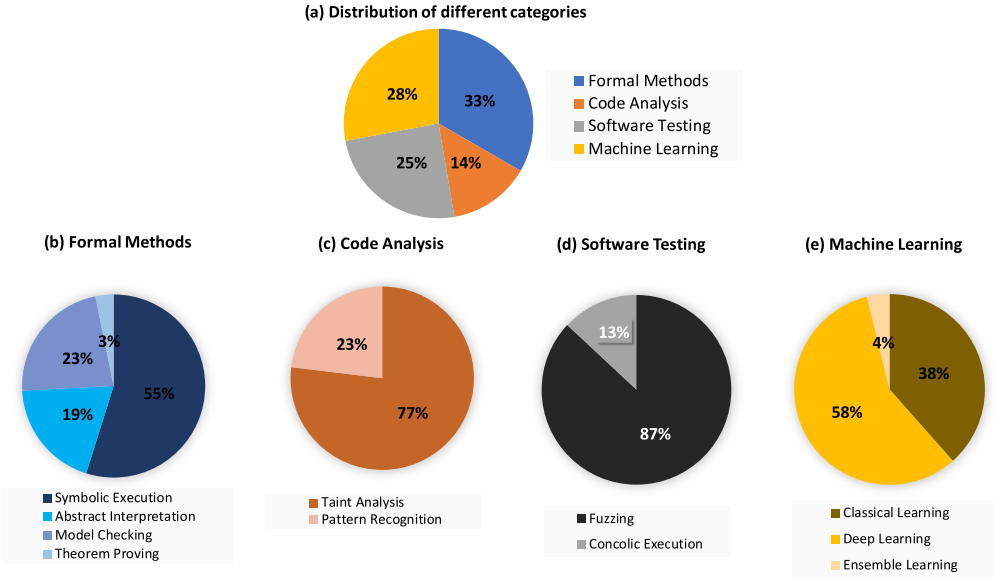

Fig. 5. Distribution of identified works.

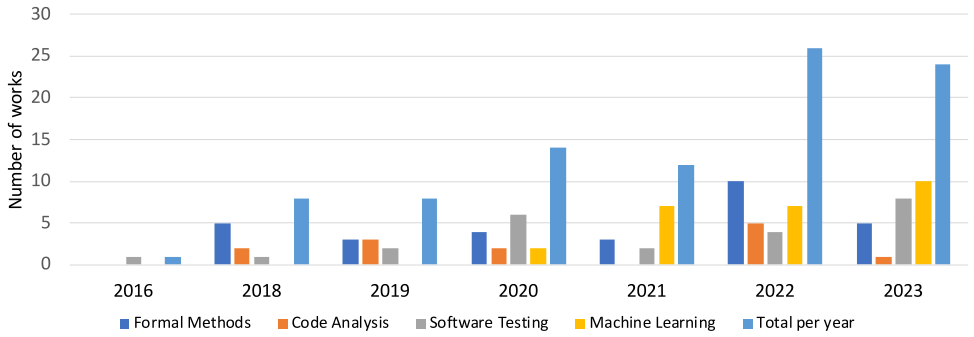

Fig. 6. Distribution of the different techniques across the years.

Pluto (Ma et al., 2022), Clairvoyance (Clarivate, 2021), Sereum (Rodler et al., 2019), Slither (Feist et al., 2019), SmartCheck (Tikhomirov et al., 2018), Solanalyser (Akca et al., 2019) and RA (Chinen et al., 2020).

(3) **ContractFuzzer** is a tool based on fuzzing, whose effectiveness has been analyzed in GasFuzzer (Ashraf et al., 2020), Gas Gauge (Nassirzadeh Behkish and Sun et al., 2023) and sFuzz (Nguyen et al., 2020).

In the following paragraphs, we discuss the general effectiveness and limitations of existing techniques for detecting vulnerabilities in smart contracts. We go through the main categories, i.e., formal methods, code analysis, software testing, and machine learning, and through each technique identified.

Formal methods group several techniques used in across multiple works. In what concerns the **Abstract Interpretation** technique, when smart contracts are converted from source code to byte code some instructions end up simplified. For instance, calls to outside contracts are translated simply to jump instructions. By using an approximation approach, the abstract interpretation technique when applied to the simplified instruction on EVM code, can produce unexpected behavior (Grech et al., 2020).

Abstract interpretation is effective for speeding up code analysis, as MadMax has shown with an analysis over the whole blockchain environment (i.e., about 91.8K contracts) in around 10 h. However, it may end up producing false positives as a side effect. Some tools have implemented extra mechanisms to mitigate the problem. For instance, Securify (Tsankov et al., 2018) defined security properties that validate whether the detected vulnerability is a false report or not. Another example is SoliDetector (Hu et al., 2023), which combines abstract interpretation with a customized technique based on a knowledge graph in order to identify which syntactic and logical relationships are captured. Despite these actions improving the effectiveness of the tools, the solution becomes more complex as each new vulnerability requires validation through an additional custom mechanism. Another approach to improve the effectiveness of abstract interpretation tools involves modifying existing techniques. For example, Vulpedia (Ye et al., 2022) implements the "Signature abstraction" approach, which computes signatures based on an abstraction, such as approximation. As a result, the authors claim that the technique can detect unknown vulnerabilities, but they note that the overhead (i.e., the opposite of the benefit from abstract interpretation) is reported as a negative aspect.

Regarding **Model Checking**, Kalra et al. (2018) found a limitation in checking fairness properties due to manually defined mathematical expressions that, in practice, can be represented in different formats, resulting in the same outcome. This issue is aggravated in smart contracts because the syntax is constantly evolving, while efforts to enhance language security are ongoing (Kalra et al., 2018). Moreover, in So et al. (2020), the authors identified an issue with the model checking technique concerning the comparison of array contents and simple variables. Additionally, the authors noted that Verismart's performance highly depends on the SMT Solver (Z3) analyzer. In the case of SmartPulse (Stephens et al., 2021), the tool implements a model-checking

approach in which the authors established an adapted checking process specification that includes the attack model. In addition, the tool provides a mechanism for the developer to customize the input model, e.g., attack simulations through external calls.

Despite the above limitations, overall the technique is well suited to operate over the design phase of smart contracts, i.e., to detect vulnerabilities during the development phase before the contract is deployed (Mavridou et al., 2019).

Symbolic execution techniques are known to consume a large amount of computer resources when attempting to explore all program paths. However, in the blockchain context, this limitation has a lower impact because smart contracts are typically smaller than traditional programs, owing to the more restrictive environment of the ledger (e.g., due to gas fees, block size limitations, etc.) (Li et al., 2022b). In certain context, symbolic execution can outperform other techniques; for instance, HFCCT (Li et al., 2022b) achieves analysis in 68 s compared to 13 min required by the fuzzing tool HFContractFuzzer to analyze a single vulnerability.

In the blockchain context, the transformation from EVM code to a structure like Control Flow Graph (CFG) diminishes the quality of the code representation. This is primarily because there is no opcode to directly represent certain keywords. For example, the "return" from functions is implemented by pushing the return address onto the stack and then executing a jump. Consequently, this conversion adversely affects the operation of symbolic execution techniques, as the code must be represented in the CFG format before beginning the simulation using symbols. To address this challenge, several tools (Wesley Scott and Christakis et al., 2022; Feist et al., 2019; Liao et al., 2022) prefer alternative code representations, such as AST. However, this approach may also introduce limitations, as AST often requires source code instead of just bytecode (Pasqua et al., 2023). The DSE tool (Huang et al., 2022) implements symbolic execution using an approach based on parallelism, with the goal of speeding up performance. The technique was only applied to arithmetic vulnerabilities, leaving space for using the technique with other vulnerabilities.

Regarding **Theorem Proving**, besides the complexity of the technique, Ayoade et al. (2019) demonstrated that it can be applied for detecting arithmetic vulnerabilities in bytecode, while also offering the foundations for allowing transforming vulnerable code (e.g., through rewriting) into secure code.

The ***Code Analysis*** category includes **Pattern Recognition**, which is the simplest technique for vulnerability detection. Generally, it struggles to match the effectiveness of other techniques and is limited to a well-defined scope. However, it can be utilized as a strategy for addressing simplistic bugs because it consumes fewer resources (Tikhomirov et al., 2018). In smart contracts, vulnerabilities primarily manifest as logical bugs (e.g., reentrancy), while simplistic errors (e.g., use of a deprecated function) are typically detected by the language syntax or compilers. This suggests that pattern-based tools should offer more than simply identifying predefined patterns (Cui et al., 2022).

Taint Analysis can be an option to simplify smart contract verification by enabling the tracking of taint propagation during transaction execution without the necessity of setting up a blockchain environment, as in a blockchain environment, executing a transaction requires specific setup parameters (such as account address, balance, and global variable values), which can be time-consuming. This is the case for static taint analysis, but may not be the case of dynamic taint analysis, which involves running the contracts. This latter case may bring in more complexity to the setup, but at the same time the outcome may benefit from the additional runtime information available. Ethainter (Brent et al., 2020) is an example of a tool that implements static taint analysis by checking information flow in smart contracts. The technique can be useful to emulate attacks without running the code through an escalation of tainted information, which can simulate multiple transactions that may lead to severe violations. The tool Easyflow (Gao et al., 2019b) is a case that implements dynamic taint analysis with overflow checking using taint transactions with special input data and message values (Gao et al., 2019b), making use of the available runtime information.

The ***Software Testing*** category includes the **Fuzzing** technique, which is one of the most prevalent techniques in vulnerability detection tools. Nonetheless, it has its limitations. An example is the case of inter-contract vulnerabilities, which encompass exploitable bugs (e.g., reentrancy, delegatecall, and tx.origin) that arise in the presence of more than two interacting contracts. Fuzzing, by itself, struggles to adequately address this issue due to the larger search space involved compared to that of a single contract. This is one case that calls for solutions that, for instance, combine fuzzing with other techniques (e.g., Machine Learning) (Xue et al., 2022). Some Fuzzers, such as ContractFuzzer (Jiang et al., 2018), use an approach to reduce the complexity of workload generation. By considering ABI as the target to generate fuzzing inputs, the ContractFuzzer reduces the number of types, e.g., "address payable" and "contract" are unified with the "address" type; facilitating the input generation process and increasing the compatibility with languages working over the Ethereum blockchain.

In **Concolic Execution**, the integration of symbolic execution with fuzzing techniques yields meaningful inputs. For instance, when the fuzzer execution becomes stuck in generating a valid input, the tool can switch to the symbolic execution engine, thereby generating a symbolic input and allowing the execution to continue. However, this can lead to a performance decline (Torres et al., 2021). Additionally, the state/path explosion problem, akin to that encountered in symbolic execution, is also present in the concolic technique, particularly for larger-scale code (Ashouri, 2020).

The challenge in the ***Machine Learning*** category, is generally focused after the code transformation process (i.e., from text to image). Once the semantics and context of the contracts are specific (e.g., address account, balance, state variables, transaction fees, etc.), new approaches to decode the input image must be developed (Hwang et al., 2022).

In Wu et al. (2021), the authors highlight the complexity of using the popular data flow graph due to its deep and non-optimized hierarchy. Therefore, they propose simplifying the method by building a reduced critical graph. For example, call.value is seen as a critical node in what concerns the detection of the reentrancy vulnerability. This critical node is then related to other existing critical nodes, and unnecessary information is discarded from the graph. The adaptation of the technique was tested with reentrancy, and according to the authors, it reached 92% accuracy.

Regarding tools that employ **Combined Techniques**, performance can be impacted since each technique relies on different analyzers. The challenge arises from modifying a set of different techniques to work cohesively within a unified approach (Kalra et al., 2018). As an example, SmartAce (Wesley Scott and Christakis et al., 2022) combines three techniques: model checking (i.e., based on the Seahorn component), fuzzing (i.e., based on the libFuzzer component), and symbolic execution (i.e., based on the Klee component). Considering the Bad Randomness vulnerability (Vidal et al., 2023), in which a non-deterministic feature is required, SeaHorn will compute the randomness using symbolic values while libFuzzer will compute randomness based on seeds. For instance, in certain operations such as testing interval (i.e., address between x and y), computing x and z can consume more resources by seed than computing them as a symbolic value approach.

Moreover, the execution behavior can be affected by wrong parameters which may have different, and sometimes (in practice) unpredictable, impacts on each of the different techniques used. It also can affect the results, e.g., for example, a tool can produce more false negatives, if a "timeout setting" is defined as extremely short (So et al., 2020).

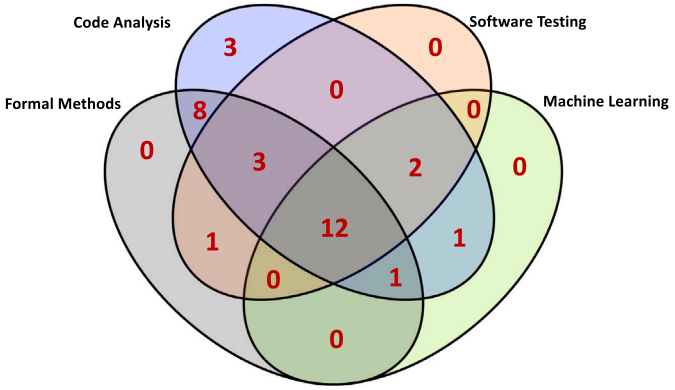

Fig. 7. Overlap of techniques' detection capabilities using SWC as reference.

4.2. RQ-2 which types of smart contract vulnerabilities are the target of detection by vulnerability detection tools?

Regarding which vulnerabilities are being targeted by researchers, we found that all groups of techniques were able to touch all DASP vulnerabilities, however, this is due to the fact that DASP is a coarse-grain classification. Concerning SWC, the different works show diverse capabilities. Fig. 7 presents the number of different vulnerabilities detected by the techniques of each category. We emphasize the fact that 3 vulnerabilities are detected by static analysis tools and no other technique (SWC-111, SWC-118, SWC-102). Some of these are detected only by static analysis because they violate development standards or practices, which are quite easy to detect using static means and much more difficult to detect if the technique is runtime-based. For instance, SWC-111 refers to the use of deprecated solidity functions, which can be easily captured by textual analysis of the code. On the opposite side, 12 vulnerabilities are being targeted by all types of techniques (SWC-106, SWC-107, SWC-101, SWC-115, SWC-120, SWC-113, SWC-132, SWC-112, SWC-114, SWC-104, SWC-116, SWC-128). These 12 vulnerabilities belong to two different groups. On the one hand, we find vulnerabilities that are very well-known and linked to the blockchain context, such as "Reentrancy" or "Bad Randomness". On the other hand, we find vulnerabilities that are quite generic and, in addition to the blockchain domain, they are also well-known in several other domains and programming languages, such as the different types of "Overflow" vulnerabilities. We must emphasize that the SWC classification naturally limits this analysis, so, for instance, the machine learning-based techniques may actually focus on vulnerabilities that no other technique targets. However, currently, there is no universal vulnerability classification that can allow us to make such an analysis. This is an open area of research, allowing this kind of comparison among many others (e.g., comparing tools' effectiveness).

Fig. 8 presents a different view of the vulnerabilities that are detected by the identified techniques. It shows the number of works for each category of techniques that could detect each DASP vulnerability. As we can see in Fig. 8, DASP-1 (Reentrancy) is detected by most of the tools and techniques, followed by DASP-2 and DASP-3. DASP-1 refers to reentrancy vulnerabilities. After the DAO attack (Siegel, 2016), where millions of dollars were lost, vulnerability detection tools have largely focused on detecting it. DASP-2 refers to access control issues, which again have been related to high-impact attacks, such as the famous attack against the Parity Wallet library contract, in which 150,000 Ether was improperly retrieved (Palladino, 2019). DASP-3 refers to arithmetic operations that have also been significant sources of security issues in smart contracts (So et al., 2020).

Fig. 9 shows how many works currently target each SWC vulnerability. Again, we can see in the figure that the majority of works and tools detect known vulnerabilities, such as "Reentrancy" (SWC-107) and "Overflow" (SWC-101). We can also note that programmers' lack of experience in blockchain technology has led tools to detect bad programming practices (e.g., SWC-116, SWC-114, SWC-120).

We found 26% (21 of 81) of papers that present specialized solutions for a single vulnerability. They are listed in Table 8. The majority of these solutions (10 papers) focus on Reentrancy (SWC-107), followed by Arithmetic Bugs (SWC-101) (4 papers). It would be interesting to analyze and compare the effectiveness of these highly specialized solutions against tools that can detect multiple vulnerabilities. In contrast, only 8% (7 of 81) of the tools propose detecting more than 10 types of vulnerabilities. These include SmartFast (Li et al., 2022c) (76), Securify (Tsankov et al., 2018) (37), SmartCheck (Tikhomirov et al., 2018) (21), HFCCT (Li et al., 2022b) (17), MODNN (Zhang et al., 2022c) (12), SmartMixModel (Shakya et al., 2022) (10), ConFuzzius (Torres et al., 2021) (10). Overall, we have observed a tendency for tools to focus on detecting a limited number of vulnerabilities.

4.3. RQ-3 what are the characteristics of the datasets used to evaluate the techniques used for smart contract vulnerability detection?

The third question focuses on datasets used to evaluate the proposed techniques. This information includes the number of contracts, how they were prepared and organized for the evaluation, and the type of contracts used (e.g., vulnerable, non-vulnerable).

We observed that the Ethereum blockchain was the main blockchain platform targeted for collecting smart contracts and analysis of faults (i.e., despite the existence of some works using other blockchain platforms (e.g., Hyperledger (Ding et al., 2021), Tezos (Nishida Yuki and Saito et al., 2021)), they are few when compared with Ethereum). The contracts are collected from sources like Etherscan (a block explorer, search, API, and analytics platform for Ethereum (Etherscan, 2021)), Truffle Suite (a development environment, testing framework, and asset pipeline for blockchains using the Ethereum Virtual Machine (Group, 2021)), or from known deployed contracts on the Ethereum blockchain. One tool, Etherolic (Ashouri, 2020), uses a customized contract suite built by the authors.

Fig. 10 presents an analysis of the data size used by the works We can observe that the number of contracts used by most tools was very similar (i.e., around 45 096), except for tools like Osiris, which performed a huge experiment with 1.2 million contracts. Several tools have based their experiment on SmartBugs's dataset (Ferreira et al., 2020) such as ReVulDl (Zhang et al., 2022a), Pluto (Ma et al., 2022), CodeNet (Hwang et al., 2022), Solidetector (Hu et al., 2023), to name a few. In addition, the works usually made their data publicly available (i.e., ContractWard (Wang et al., 2021)), which are then used by other tools for their experiments (i.e., DeeSCVHunter (Yu et al., 2021)). The authors usually prefer to use an existing dataset because downloading source code directly from Blockchain platforms and selecting and maintaining them is a time-consuming process.

It is worthwhile mentioning that, although not really proposing a method for vulnerability detection, there are several works that discuss or provide methods and tools for the automatic collection of smart contracts or for building datasets of smart contracts. BugGetter (Zhang et al., 2020c) is a crawler program used to extend the Jiuzhou dataset (Zhang et al., 2020b), a dataset of bugs. It automatically receives updates when a new bug emerges so the new bug can be added to the dataset for constructing buggy smart contracts. It contains 49 bug types from the Solidity version 0.4.26 to 0.6.2. SmartBugs (Durieux et al., 2020) is an extensible execution framework tool that provides datasets of smart contracts to facilitate the integration and comparison between multiple analysis tools and the analysis of Ethereum smart contracts. SolidiFI (Solidity Fault Injector) (Ghaleb and Pattabiraman, 2020b) is a bug injection tool to create vulnerable contracts. It can inject nine types of bugs, including Reentrancy,

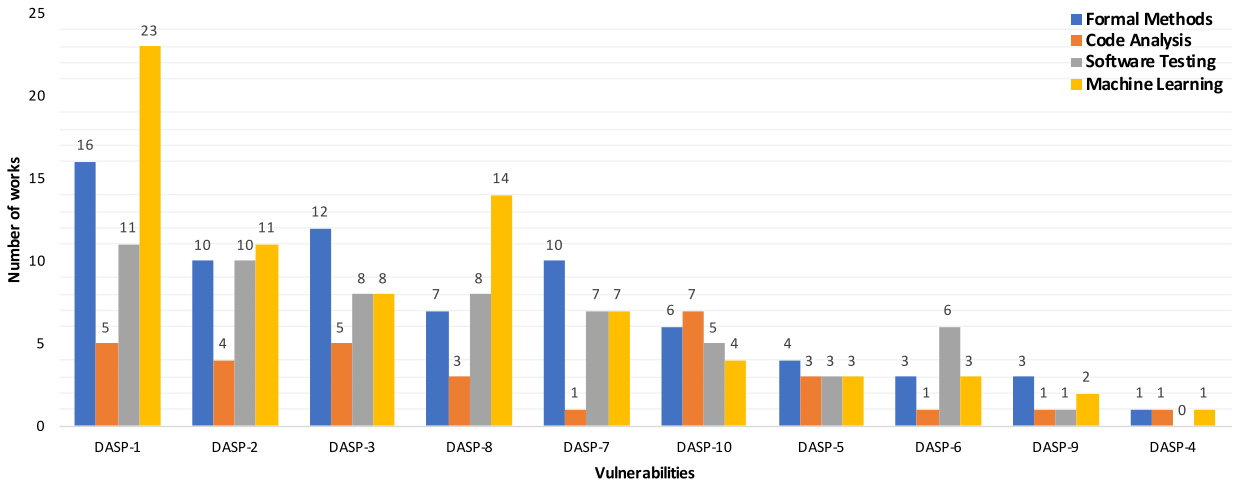

Fig. 8. Number of works in each category that detect each DASP vulnerability.

Table 8
Specialized tools - Single vulnerability.

Tool	Vulnerability name	DASP	SWC	Reference
Ethainter	Multiple Transaction	–	–	Brent et al. (2020)
vGas	unknown out-of-gas	–	–	Ma et al. (2023b)
EasyFlow	Overflow detector	DASP-3	SWC-101	Gao et al. (2019b)
DSE	Integer Overflow	DASP-3	SWC-101	Huang et al. (2022)
–	Integer overflow and integer underflow	DASP-3	SWC-101	Ayoade et al. (2019)
Osiris	Arithmetic bugs, Truncation bugs and Signedness bugs	DASP-3	SWC-101	Torres et al. (2018)
TechyTech	selfdestruct	DASP-2	SWC-106	Khan and Namin (2023)
VSCL	reentrancy vulnerability	DASP-1	SWC-107	Mi et al. (2021)
Dynamit	Reentrancy	DASP-1	SWC-107	Eshghie et al. (2021)
EtherSolve	reentrancy	DASP-1	SWC-107	Pasqua et al. (2023)
MAR	Reentrancy	DASP-1	SWC-107	Wang Zexu and Wen et al. (2021)
ReDefender	Reentrancy	DASP-1	SWC-107	Li et al. (2022d)
Sereum	Reentrancy	DASP-1	SWC-107	Rodler et al. (2019)
RA	Reentrancy vulnerabilities	DASP-1	SWC-107	Chinen et al. (2020)
Peculiar	Reentrancy	DASP-1	SWC-107	Wu et al. (2021)
ReVulDL	Reentrancy	DASP-1	SWC-107	Zhang et al. (2022a)
GraBit	Reentrancy	DASP-1	SWC-107	Zhu et al. (2023)
TransRacer	transaction-ordering-dependent	DASP-7	SWC-114	Ma et al. (2023c)
EthRacer	Event-ordering (EO) bugs	DASP-8	SWC-116	Kolluri et al. (2019)
HELMHOLTZ	Signature Verification	DASP-10	SWC-117	Nishida Yuki and Saito et al. (2021)
Gas Gauge	gas limit DoS on a contract via unbounded operations	DASP-5	SWC-128	Nassirzadeh Behkish and Sun et al. (2023)

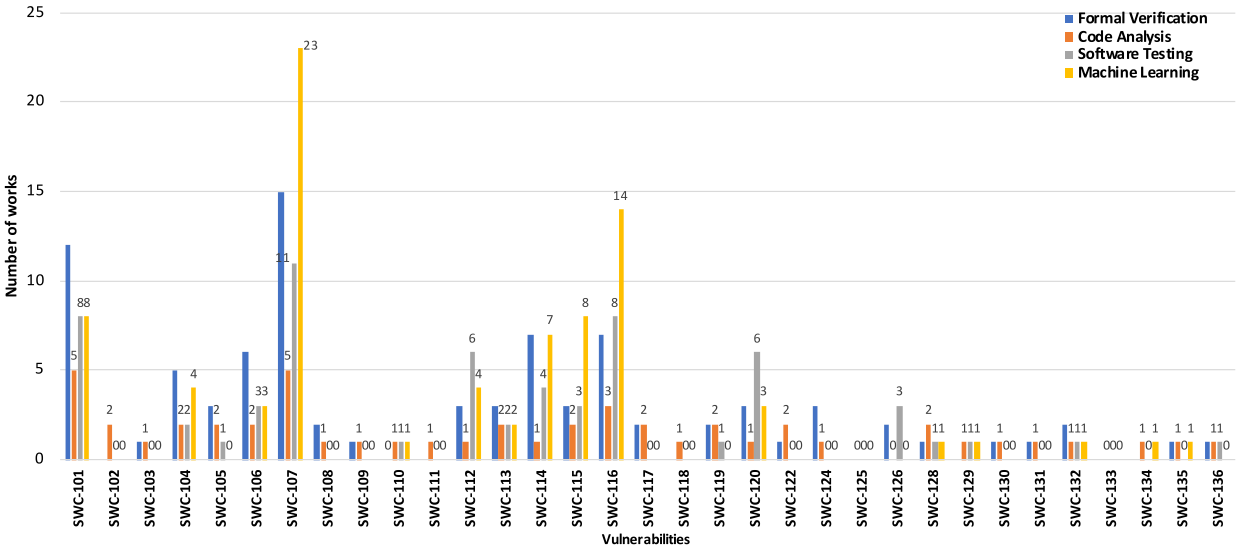

Fig. 9. Number of tools/works in each category that detect each SWC vulnerability.

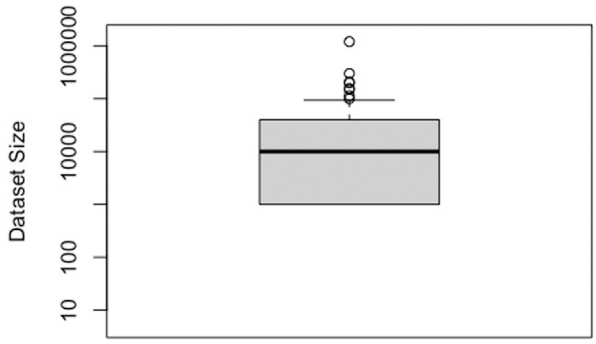

Fig. 10. Average size of datasets used by the works/tools previously presented.

Transaction ordering dependency, DoS, Timestamp dependency, Mishandled exceptions, Integer overflow and underflow, Use of tx.origin, and Unchecked send function. With this tool, the researchers can create their own dataset by modifying original contracts from any dataset.

Building on the work presented in di Angelo et al., di Angelo et al. present Smartbugs 2.0, which, besides allowing the evaluation of different vulnerability detection tools, identifies the following important characteristics of this kind of platforms: (i) support for compiler version identification; (ii) support for complex source code format; (iii) source code mappings support; (iv) extensibility regarding new tools. Currently, Smartbugs 2.0 supports more than 20 vulnerability detection tools, and its dataset comprises 295 879 contracts. The framework could benefit from a fine identification of the techniques used by each supported vulnerability detection tool.

Overall, regarding the datasets used in the state of the art, we observe high diversity and, most of all, the absence of guidelines for creating meaningful sets of contracts that could help researchers in various ways. Namely, in assisting in the evaluation and comparison of the effectiveness of vulnerability detection tools.

Ye et al. (2022) report the need for datasets labeled as non-detectable to be used by vulnerability detection tools. The authors aimed at finding contracts with which tools had difficulties, because working on them could reduce false alarms. To achieve this, they manually collected such information from the Slither, Oyente, and Securify experiments. This difficulty shows that existing datasets could be organized according to the detection capabilities of current tools, i.e., based on the number of tools capable of detecting such vulnerability.

Openzeppelin (2015) was used by SmartACE (Wesley Scott and Christakis et al., 2022) to validate the vulnerability detection. It has a public license available on GitHub (Openzeppelin, 2015) and has a community evaluation through comments and star ratings. It has backward compatibility, reusable solidity components, and implementations of the standards ERC20 and ERC721. There are frequent audits on the contracts, reporting the problems according to low, medium, and high severity. Thus, it can be a good source for experiments that need reliable contracts for fault injection campaigns. However, by focusing on providing reuse libraries, OpenZeppelin does not provide complete systems, unlike what GitHub does (Openzeppelin, 2024).

Due to the lack of an adequate vulnerability classification scheme, the Verismart tool (So et al., 2020) mapped 60 faulty smart contracts (i.e., arithmetic flaws) with CVE IDs. As a result, the authors presented a standardized analysis regarding arithmetic flaws. This approach (i.e., mapping the smart contract vulnerability to existing faulty classifications) works only for vulnerabilities that are both present in conventional and smart contract programs. However, this methodology may not be suitable for tools that report more specific blockchain failures (e.g., gas depletion, short address, etc.).

Nishida Yuki and Saito et al. (2021) presents an approach and analysis based on the number of instructions instead of a number of contracts or functions. The authors extracted instructions from each contract included in their dataset with the goal of achieving a more comprehensive analysis. The main idea is that having a number of instructions facilitates understanding the real contract's complexity, contributing to measuring the impact of faults and more fairness in the analysis of the results.

SmartPulse (Stephens et al., 2021) aims to prove whether a given contract can meet liveness properties or not. For example, users cannot withdraw more money than their credit, or funds cannot be transferred to inaccessible accounts. The authors identify the need for datasets labeled with liveness properties. The manual effort employed by the authors resulted in only 191 contracts.

5. Gaps and limitations

Several gaps and limitations were made clear during the analysis of the works identified in this review. The first gap concerns the **lack of a good and up-to-date vulnerability classification scheme or taxonomy**. The classification schemes that exist for smart contract vulnerabilities are limited, and despite being in open repositories, as is the case of SWC, no relevant activity is currently observed. Indeed, we identified a significant number of vulnerabilities that could not be classified by SWC or even DASP (e.g., Costly Loop, Ether Leakage). Also, a scheme such as DASP is largely insufficient due to the lack of detail associated with each of its broad categories. Table 9 identifies the vulnerabilities found in this work, which are currently left unclassified by SWC. Moreover, due to the lack of a standard scheme, researchers use names that are difficult to interpret (e.g., Gain/Lose (Chapman et al., 2019)) or dubious (e.g., Unknown Libraries (Lu et al., 2019)). Also, the lack of a standard classification makes any comparison between techniques very difficult, including comparisons over key aspects such as the tools' detection effectiveness. As an example of the extent of the issue, we observed that several vulnerabilities, such as "unhandled exception", "exception stat", or "mishandled exceptions" are exactly the same vulnerability (Grishchenko et al., 2018; Tsankov et al., 2018; Argañaraz et al., 2020; Zhang et al., 2019a; Jiang et al., 2018; Ashouri, 2020; Nguyen et al., 2020; Luu et al., 2016; Wang et al., 2019; Chapman et al., 2019; Momeni et al., 2019). Obviously, the reverse occurs when the same name is used to represent different vulnerabilities. For example, "Transaction Order Dependency (TOD)" is the name used in Liao et al. (2019) and in Bose et al. (2022), which refers respectively to the case where a transfer amount is dependent on transaction order and to another case where the transfer recipient may erroneously vary depending on the transaction order.

The second gap concerns the **limited scope and effectiveness of existing tools**. Overall, many of the works (e.g., Anastasia and Laszka (2018), Ayoade et al. (2019), Gao et al. (2019a), Zhang et al. (2019a), Wang et al. (2021), Ashraf et al. (2020), Jiang et al. (2018) and Kolluri et al. (2019)) highlight the need for extending and improving tools to detect more vulnerabilities with higher precision and with low false alarms. In general, most of the tools are still in the early stages and tend to detect a relatively small number of vulnerability types, and their effectiveness is known to have a large space for improvement (Dia et al., 2021; Zhang et al., 2022c). It is also worthwhile mentioning that some works indicate the need for a complementary technique to address more types of vulnerabilities (Ghaleb and Pattabiraman, 2020b; di Angelo and Salzer, 2019). The most obvious example is perhaps the combination of static and dynamic approaches so that detection is more effective, as is the case of Sereum (Rodler et al., 2019) and EthPloit (Zhang et al., 2020a), which perform Taint Analysis and Fuzzing respectively. There are also other types of combination, such as Zeus (Kalra et al., 2018), which combines Model Checking and Abstract Interpretation, and xFuzz (Xue et al., 2022), which combines Fuzzing and Classical Learning, to name a few.

There is one particular aspect regarding inter-contract interaction that is mentioned in some works (e.g., Akca et al. (2019), Grech et al. (2020), Torres et al. (2018), Chinen et al. (2020), Mavridou et al.

Table 9
Unclassified vulnerabilities.

Vulnerability	Reference	Vulnerability	Reference
Transfer forwards all gas	Tikhomirov et al. (2018)	erc20-throw	Li et al. (2022c)
Revert DOS	Stephens et al. (2021)	ether lost transferred	Gupta et al. (2022)
External Library Calling	Li et al. (2022b)	shift-parameter-mixup	Li et al. (2022c)
erc20-approve	Li et al. (2022c)	AssemblyUsage	Tsankov et al. (2018)
Freezing Ether	Ashraf et al. (2020)	Mishandled exceptions	Jiang et al. (2018)
Unhandled Exception	Torres et al. (2021)	Locked money	Tikhomirov et al. (2018)
Unhandled Exception	Su et al. (2022)	abiencoderv2-array	Li et al. (2022c)
Addresses cannot be set to incorrect values	Grieco et al. (2020)	Locked Money	Shakya et al. (2022)
Multiple Transaction	Brent et al. (2020)	Missing Owner Check	Cui et al. (2022)
ConstableStates	Tsankov et al. (2018)	arrays	Li et al. (2022c)
Range Query Risks	Li et al. (2022b)	Infinite loop	Li et al. (2023)
Unhandled Errors	Li et al. (2022b)	Use Predicable Variable	Zhou et al. (2022)
complex-function	Li et al. (2022c)	Destroy Tokens	Ma et al. (2023a)
Invalid Input Data	Grieco et al. (2020)	Buffer-overflow	Pani et al. (2023)
Redundant fallback function	Tikhomirov et al. (2018)	Length-manipulation	Li et al. (2022c)
LockedEther	Tsankov et al. (2018)	Buffer-underflow	Pani et al. (2023)
infinite loop	Liu et al. (2021)	Costly loop	Tikhomirov et al. (2018)
Ownable	Wesley Scott and Christakis et al. (2022)	Locked Funds	Stephens et al. (2021)
unprotected-upgrade	Li et al. (2022c)	Leaking Ether	Torres et al. (2021)
multiple-calls-in-transaction	Li et al. (2022c)	Extra gas consumption	Shakya et al. (2022)
Logic loopholes	Ding et al. (2021)	unknown out-of-gas	Ma et al. (2023b)
Arbitrarily Transfer Threat	Ma et al. (2023a)	Direct value transfer	Krupp and Rossow (2018)
Fake EOS	Chen et al. (2022)	Fake Notification	Chen et al. (2022)
NamingConvention	Tsankov et al. (2018)	Rollback	Chen et al. (2022)
CallDepth	Sun et al. (2023)	controlled-array-length	Li et al. (2022c)
assembly	Li et al. (2022c)	overpowered-role	Li et al. (2022c)
Token API violation	Tikhomirov et al. (2018)	Predictable variable dependency	Fu et al. (2019)
Golang Grammar Errors	Li et al. (2022b)	Unchecked math	Tikhomirov et al. (2018)
Mishandled Exceptions	Luu et al. (2016)	ExternalFunctions	Tsankov et al. (2018)
Fake EOS Transfer	Li et al. (2022a)	Exception State	Zhou et al. (2022)
Code injection	Krupp and Rossow (2018)	Unsolvable constraints	Zhang et al. (2020a)
immutable bugs	Gupta et al. (2022)	Callstack Depth Attack	Wang et al. (2021)
taint-pass-project	Li et al. (2022c)	Malicious libraries	Tikhomirov et al. (2018)
function-problem	Li et al. (2022c)	Freeze Account	Ma et al. (2023a)
Freezing Ether	Nguyen et al. (2020)	Mishandled exceptions	Ashouri (2020)
RefundEscrow	Wesley Scott and Christakis et al. (2022)	is_greedy	Xing et al. (2020)
constant-function-state	Li et al. (2022c)	Ether leakage	Zhang et al. (2022b)
Mishandled exception	Fu et al. (2019)	array-by-reference	Li et al. (2022c)
Vulnerable state	Krupp and Rossow (2018)	redundant-fallback	Li et al. (2022c)
Costly loop	Shakya et al. (2022)	UnhandledException	Tsankov et al. (2018)
Style guide violation	Tikhomirov et al. (2018)	erc20-indexed	Li et al. (2022c)
Arbitrary sending of ether	Feist et al. (2019)	Disable Transferring	Ma et al. (2023a)
ERC721Interface	Tsankov et al. (2018)	InlineAssembly	Zhang et al. (2022c)
Byte array	Tikhomirov et al. (2018)	Forged Transfer Notification	Li et al. (2022a)
Field Declarations	Li et al. (2022b)	CallToDefaultConstructor	Tsankov et al. (2018)
Buffer-overwrite	Pani et al. (2023)	msgvalue-equals-zero	Li et al. (2022c)
Unsafe arrays length manipulation	Shakya et al. (2022)	Taint for owner	Liao et al. (2022)
Gas Consumption	Ashizawa et al. (2021)	Ether Leaking	Su et al. (2022)
Implicit Visibility	Ashizawa et al. (2021)	Be no black hole	Chang et al. (2019)
Mishandled exceptions	Nguyen et al. (2020)	Send instead of transfer	Tikhomirov et al. (2018)
Send instead of transfer	Shakya et al. (2022)	callstack	Li et al. (2022c)
ERC20Indexed	Tsankov et al. (2018)	Push DOS	Stephens et al. (2021)
Cross Program Invocation	Cui et al. (2022)	CheckEffects	Zhang et al. (2022c)
Unsafe type inference	Tikhomirov et al. (2018)	Locking Ether	Torres et al. (2021)
solidity-dos-with-throw	Li et al. (2022c)	CallDepth	Zhang et al. (2022c)
Auction	Wesley Scott and Christakis et al. (2022)	Respect the limit	Chang et al. (2019)
Ether Frezzing	Su et al. (2022)	delete-dynamic	Li et al. (2022c)
Price Manipulation	Shou et al. (2023)	ERC-20 Transfer	Ashizawa et al. (2021)
Unexpected revert	Ye et al. (2022)	Freezing Ether	Jiang et al. (2018)
Map Structure Iteration	Li et al. (2022b)	ERC20Interface	Tsankov et al. (2018)
naming-convention	Li et al. (2022c)	Account Confusion	Cui et al. (2022)
public-mappings-nested	Li et al. (2022c)	Locked ether	Feist et al. (2019)
upgrade-050	Li et al. (2022c)	Unprotected Function	Stephens et al. (2021)
private modifier	Tikhomirov et al. (2018)	Exception-State	Li et al. (2022c)
Missing Authorization Verification	Chen et al. (2022)	function-init-state	Li et al. (2022c)
Generate Tokens After ICO	Ma et al. (2023a)	modulo	Li et al. (2022c)
		mul-var-len-arguments	Li et al. (2022c)

(2019) and Feng et al. (2019a)) that is a prominent problem, at least in Ethereum, which is the need to further focus on vulnerabilities that involve interaction between multiple contracts and transactions, which can be quite complex to detect, due to the challenges of dealing with the state changes between the contracts at runtime.

In Ethereum, when two contracts interact, the caller always initializes a new stack, thus opening a new context different from the contract being executed. Traditional tools cannot correctly trace semantic information in this kind of inter-contract scenario, making it very challenging to identify which paths in the program flow are truly reachable (Ma et al., 2022). Another challenge is related to the rapid evolution of the languages being used to write smart contracts, which ends up in the interaction of code originally written in different versions. When a contract calls another one written in a more inferior

version, there is a risk of incompatibility of Ethereum virtual machine operation codes, which results in exceptions being triggered during execution (Khan et al., 2021). If not properly handled, gas may be lost, and the transaction will not conclude. Pluto (Ma et al., 2022; Zhang et al., 2022b) are two works proposing methods to address the inter-contract issue. Pluto generates multiple Control Flow Graphs (CFGs) for each involved contract in inter-contract execution, connecting them through identified connection points like 'CALL', 'DELEGATECALL', 'CALLCODE', and 'STATICCALL' opcodes. Using individual CFGs and mapped connection points, it forms a unified Inter-contract Control Flow Graph (ICFG). Vulnerability searches are conducted at these points using an SMT solver, limited to detecting three vulnerabilities (integer overflow, timestamp dependency, and reentrancy). This indicates potential for further exploration in the field. In Zhang et al. (2022b), the authors propose a solution that addresses some of the symbolic execution method limitations. Their modified method, symbolic execution inter-path, starts with a code pre-analysis (data-analysis dependency) using LASER Ethereum (Mueller, 2018) to identify external contracts needed for execution. Pruning discards non-critical paths, drastically reducing program execution costs (Vidal et al., 2021). The tool then performs a symbolic execution analysis on the remaining paths, searching for potential contract vulnerability issues.

The third gap observed is related to the **heterogeneity of the experimental settings (e.g., datasets with different characteristics and sizes) and measurement metrics and methods** used to evaluate the proposed techniques and tools. It is difficult and unfair to compare tools based on their reported evaluation results. For instance, the results presented in Feist et al. (2019) show Smartcheck (Tikhomirov et al., 2018) with a higher false-positive rate (73.6%) than Slither (Feist et al., 2019) (10%). In contrast, the results presented in Ghaleb and Pattabiraman (2020b) show that 90% of Slither's alarms are false alarms, while Smartcheck reports no false alarms in the authors' experiments. The main reason for this divergence is that the datasets used for evaluation are different (Ren et al., 2021). Thus, there is a need to specify and quantify properties that allow defining what a representative dataset for this purpose would be and also to specify standard sets of metrics for this goal.

Due to the need to make contracts more secure, current programming languages have been undergoing significant syntax and/or semantic changes (Kaleem et al., 2020; Coblenz, 2019). As an example, Solidity introduced changes starting from version 0.5, such as "the implementation of C99-style scoping rules for function local variables" and "changes in the argument passing in functions call(), .delegatecall(), staticcall(), keccak256(), sha256(), and ripemd160()", just to name a few (Solidity, 2023). In this regard, we did not find information about such changes in the datasets identified in this review (e.g., Zhang et al. (2020c,b), Durieux et al. (2020) and di Angelo et al. (2023)). We believe that future work in dataset definition should provide information about the breaking changes in each contract version contained in the dataset. Some operations could benefit from such information. For instance, a vulnerability injector cannot inject a vulnerability in a contract that has language-level protection mechanisms for that specific vulnerability (i.e., injecting it would result in a contract that would be impossible to write and, as such, unrealistic).

Another opportunity to improve datasets is to organize them according to the type of functions written in the contract. This can be important for dynamic techniques (e.g., fuzzing) to allow them to know for which functions they should or should not generate calls (or their relative importance for testing). For example, the GapFuzzer tool (Ashraf and Chant, 2022) classifies contracts from the SmarBug dataset according to the type of function: pure/view, non-pure/view, non-pure/view with input, and payable functions. Pure and view functions are, respectively, functions that do not interact with the blockchain state and functions that are typically used to read data from the contract without making changes. Thus, these may be less interesting candidates for fuzzers to test.

Current datasets should also hold meta-information regarding the individual complexity of the set of contracts. Also, as blockchain applications evolve, so does the complexity of the contracts, which are often composed of more than one physical file (di Angelo et al., 2023). A representative dataset should include a mix of contracts in which high-complexity contracts should also be present. The recent huge increase in new smart contract vulnerability detection tools is quite visible even with a quick search on Scholar databases. To the best of our knowledge, no online services provide updated and curated datasets or repositories organized by characteristics relevant for evaluating and comparing tools (e.g., identified vulnerabilities, compiler version, blockchain platform). This makes assembling datasets a laborious process that requires search, inspection, and manual filtering, which can easily lead to errors and affect assessment results.

The fourth gap is related to the **scarce support for smart contract programming languages**. Indeed, we observed that just a few tools, including (Kalra et al., 2018; Ayoade et al., 2019; Argañaraz et al., 2020; Tikhomirov et al., 2018; Chen et al., 2020; Grieco et al., 2020; Feist et al., 2019), support different smart contract languages beyond of Solidity. These tools are actually the exception. Generally, tools focus on a single language, and very frequently, that language is Solidity. On the one hand, this reflects the importance of Solidity in this field, and on the other hand, the absence of different and good quality vulnerability detection tools may discourage developers from using languages other than Solidity.

6. Conclusion

In this paper, we presented a systematic literature review on the state of the art regarding techniques for detecting security vulnerabilities in smart contracts. In addition to the systematization and categorization of the different vulnerability detection techniques, we mapped the vulnerabilities identified in the literature against two common vulnerability classification schemes, namely DASP and SWC. Among our observations, we highlight that a significant number of vulnerabilities no longer fit in a popular and fine-grained scheme like SWC, opening space for new schemes or taxonomies that can classify new vulnerabilities. Also, the tools announced detection capabilities are quite heterogeneous and very limited, especially concerning the new vulnerabilities. Moreover, results reported by tools' authors are very difficult to compare due to the heterogeneity described and include the specific evaluation settings, opening space for benchmarks that allow evaluating tools' effectiveness in a standard manner.

CRediT authorship contribution statement

Fernando Richter Vidal: Writing – review & editing, Writing – original draft, Methodology, Investigation. **Naghmeh Ivaki:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Conceptualization. **Nuno Laranjeiro:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Conceptualization.

Declaration of competing interest

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

Data availability

Data available at https://doi.org/10.5281/zenodo.8109651.

Acknowledgments

This work is supported by the Foundation for Science and Technology (FCT), Portugal, I.P./MCTES through national funds (PIDDAC),

within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020; and by Project "Agenda Mobilizadora Sines Nexus". ref. No. 7113, supported by the Recovery and Resilience Plan (PRR), Portugal and by the European Funds Next Generation EU, following Notice No. 02/C05-i01/2022, Component 5 - Capitalization and Business Innovation - Mobilizing Agendas for Business Innovation and by FCT, Portugal Grant Nr. 2023.03131.BD.

References

Akca, Sefa, Rajan, Ajitha, Peng, Chao, 2019. SolAnalyser: A framework for analysing and testing smart contracts. In: 2019 26th Asia-Pacific Software Engineering Conference. APSEC, IEEE, Putrajaya, Malaysia, ISBN: 978-1-7281-4648-5, pp. 482–489. http://dx.doi.org/10.1109/APSEC48747.2019.00071, URL https://ieeexplore.ieee.org/document/8945725/.

Alharby, Maher, van Moorsel, Aad, 2017. Blockchain based smart contracts : A systematic mapping study. In: Computer Science & Information Technology. CS & IT, Academy & Industry Research Collaboration Center (AIRCC), Dubai, ISBN: 9781921987700, pp. 125–140. http://dx.doi.org/10.5121/csit.2017.71011, URL http://airccj.org/CSCP/vol7/csit77211.pdf.

Almakhour, Mouhamad, Sliman, Layth, Samhat, Abed Ellatif, Mellouk, Abdelhamid, 2020. Verification of smart contracts: A survey. Pervasive Mob. Comput. (ISSN: 1574-1192) 67, 101227. http://dx.doi.org/10.1016/j.pmcj.2020.101227, URL https://www.sciencedirect.com/science/article/pii/S1574119220300821.

Anastasia, Mavridou, Laszka, Aron, 2018. Designing secure ethereum smart contracts: A finite state machine based approach. In: Sarah, Meiklejohn, Sako, Kazue (Eds.), Financial Cryptography and Data Security. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-662-58387-6, pp. 523–540, URL https://www.springerprofessional.de/en/designing-secure-ethereum-smart-contracts-a-finite-state-machine/17118720.

ANTLR, 2019. ANTLR. URL https://github.com/antlr/antlr4.

Argañaraz, Mauro C, Berón, Mario M, Pereira, Maria J Varanda, Henriques, Pedro Rangel, 2020. Detection of vulnerabilities in smart contracts specifications in ethereum platforms. In: 9th Symposium on Languages, Applications and Technologies. SLATE 2020, In: OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Barcelos, Portugal, ISBN: 978-3-95977-165-8, p. 16. http://dx.doi.org/10.4230/OASIcs.SLATE.2020.0.

Ashizawa, Nami, Yanai, Naoto, Cruz, Jason Paul, Okamura, Shingo, 2021. Eth2Vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In: Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure. ACM, New York, NY, USA, ISBN: 9781450384001, pp. 47–59. http://dx.doi.org/10.1145/3457337.3457841.

Ashouri, Mohammadreza, 2020. Etherolic. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, ISBN: 9781450368667, pp. 353–356. http://dx.doi.org/10.1145/3341105.3374226, URL https://dl.acm.org/doi/10.1145/3341105.3374226.

Ashraf, Imran, Chant, W.K., 2022. An empirical study on the effects of entry function pairs in fuzzing smart contracts. In: 2022 IEEE 46th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, ISBN: 978-1-6654-8810-5, pp. 1716–1721. http://dx.doi.org/10.1109/COMPSAC54236.2022.00273.

Ashraf, I., Ma, X., Jiang, B., Chan, W.K., 2020. GasFuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. IEEE Access 8, 99552–99564. http://dx.doi.org/10.1109/ACCESS.2020.2995183.

Ayoade, Gbadebo, Bauman, Erick, Khan, Latifur, Hamlen, Kevin, 2019. Smart contract defense through bytecode rewriting. In: 2019 IEEE International Conference on Blockchain (Blockchain). IEEE, Atlanta, GA, USA, ISBN: 978-1-7281-4693-5, pp. 384–389. http://dx.doi.org/10.1109/Blockchain.2019.00059, URL https://ieeexplore.ieee.org/document/8946210/.

Baldoni, Roberto, Coppa, Emilio, D'elia, Daniele Cono, Demetrescu, Camil, Finocchi, Irene, 2019. A survey of symbolic execution techniques. ACM Comput. Surv. (ISSN: 0360-0300) 51 (3), 1–39. http://dx.doi.org/10.1145/3182657, URL https://dl.acm.org/doi/10.1145/3182657.

Bhargavan, Karthikeyan, Delignat-Lavaud, Antoine, Fournet, Cédric, Gollamudi, Anitha, Gonthier, Georges, Kobeissi, Nadim, Kulatova, Natalia, Rastogi, Aseem, Sibut-Pinote, Thomas, Swamy, Nikhil, Zanella-Béguelin, Santiago, 2016. Formal verification of smart contracts. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. ACM, New York, NY, USA, ISBN: 9781450345743, pp. 91–96. http://dx.doi.org/10.1145/2993600.2993611, URL https://dl.acm.org/doi/10.1145/2993600.2993611.

Blum, Avrim, Mitchell, Tom, 1998. Combining Labeled and Unlabeled Data with Co-Training. ACM, New York, NY, USA, ISBN: 1581130570, pp. 92–100. http://dx.doi.org/10.1145/279943.279962.

Bose, Priyanka, Das, Dipanjan, Chen, Yanju, Feng, Yu, Kruegel, Christopher, Vigna, Giovanni, 2022. SAILFISH: Vetting smart contract state-inconsistency bugs in seconds. In: 2022 IEEE Symposium on Security and Privacy. SP, IEEE, San Francisco, CA, USA, ISBN: 978-1-6654-1316-9, pp. 161–178. http://dx.doi.org/10.1109/SP46214.2022.9833721, URL https://ieeexplore.ieee.org/document/9833721/.

Brent, Lexi, Grech, Neville, Lagouvardos, Sifis, Scholz, Bernhard, Smaragdakis, Yannis, 2020. Ethainter: A smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. In: PLDI 2020, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450376136, pp. 454–469. http://dx.doi.org/10.1145/3385412.3385990.

Broberg, Niklas, Farre, Andreas, Svenningsson, Josef, 2004. Regular Expression Patterns. ACM, New York, NY, USA, ISBN: 1581139055, pp. 67–78. http://dx.doi.org/10.1145/1016850.1016863.

Chang, Jialiang, Gao, Bo, Xiao, Hao, Sun, Jun, Cai, Yan, Yang, Zijiang, 2019. sCompile: Critical path identification and analysis for smart contracts. In: Ait-Ameur, Yamine, Qin, Shengchao (Eds.), Formal Methods and Software Engineering. Springer International Publishing, Cham, ISBN: 978-3-030-32409-4, pp. 286–304.

Chapman, Patrick, Xu, Dianxiang, Deng, Lin, Xiong, Yin, 2019. Deviant: A mutation testing tool for solidity smart contracts. In: 2019 IEEE International Conference on Blockchain (Blockchain). IEEE, Atlanta, GA, USA, ISBN: 978-1-7281-4693-5, pp. 319–324. http://dx.doi.org/10.1109/Blockchain.2019.00050, URL https://ieeexplore.ieee.org/document/8946219/.

Chen, Ting, Cao, Rong, Li, Ting, Luo, Xiapu, Gu, Guofei, Zhang, Yufei, Liao, Zhou, Zhu, Hang, Chen, Gang, He, Zheyuan, Tang, Yuxing, Lin, Xiaodong, Zhang, Xiaosong, 2020. SODA: A generic online detection framework for smart contracts. In: Proceedings 2020 Network and Distributed System Security Symposium. Internet Society, Reston, VA, ISBN: 1-891562-61-4, http://dx.doi.org/10.14722/ndss.2020.24449, URL https://www.ndss-symposium.org/wp-content/uploads/2020/02/24449.pdf.

Chen, Chen, Cui, Baojiang, Ma, Jinxin, Wu, Runpu, Guo, Jianchao, Liu, Wenqian, 2018. A systematic review of fuzzing techniques. Comput. Secur. (ISSN: 01674048) 75, 118–137. http://dx.doi.org/10.1016/j.cose.2018.02.002, URL https://linkinghub.elsevier.com/retrieve/pii/S0167404818300658.

Chen, Huashan, Pendleton, Marcus, Njilla, Laurent, Xu, Shouhuai, 2021. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. ACM Comput. Surv. (ISSN: 0360-0300) 53 (3), 1–43. http://dx.doi.org/10.1145/3391195, URL https://dl.acm.org/doi/10.1145/3391195.

Chen, Weimin, Sun, Zihan, Wang, Haoyu, Luo, Xiapu, Cai, Haipeng, Wu, Lei, 2022. WASAI: uncovering vulnerabilities in Wasm smart contracts. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, ISBN: 9781450393799, pp. 703–715. http://dx.doi.org/10.1145/3533767.3534218.

Chess, B., McGraw, G., 2004. Static analysis for security. IEEE Secur. Priv. Mag. (ISSN: 1540-7993) 2 (6), 76–79. http://dx.doi.org/10.1109/MSP.2004.111, URL http://ieeexplore.ieee.org/document/1366126/.

Chinen, Yuchiro, Yanai, Naoto, Cruz, Jason Paul, Okamura, Shingo, 2020. RA: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis. In: 2020 IEEE International Conference on Blockchain (Blockchain). IEEE, Rhodes, Greece, ISBN: 978-0-7381-0495-9, pp. 327–336. http://dx.doi.org/10.1109/Blockchain50366.2020.00048, URL https://ieeexplore.ieee.org/document/9284679/.

Chittoda, Jitendra, 2019. Mastering Blockchain Programming with Solidity, Vol. 1, Packt Publishing, ISBN: 9781839218262.

Clarivate, 2021. Journal Citation Reports (JCR). URL http://jcr.clarivate.com.

Clarke, Edmund M., Clarke, Jr., Edmund M., Grumberg, Orna, 1999. Model Checking. Mit Press, ISBN: 0262032708, p. 330.

Coblenz, Michael, 2019. The obsidian smart contract language. URL https://obsidian.readthedocs.io/en/latest/.

ConsenSys, 2019. Security analysis tool for evm bytecode. supports smart contracts built for ethereum, quorum, vechain,roostock, tron and other evm-compatible blockchains. Mythx. URL https://mythx.io/.

ConsenSys, 2021. Mythril. URL https://github.com/ConsenSys/mythril.

Cousot, Patrick, 2021. Principles of Abstract Interpretation. The MIT Press, New York, ISBN: 9780262361521, URL https://mitpress.mit.edu/9780262361521/principles-of-abstract-interpretation/.

Crincoli, Giuseppe, Iadarola, Giacomo, La Rocca, Piera Elena, Martinelli, Fabio, Mercaldo, Francesco, Santone, Antonella, 2022. Vulnerable smart contract detection by means of model checking. In: Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure. ACM, New York, NY, USA, ISBN: 9781450391757, pp. 3–10. http://dx.doi.org/10.1145/3494106.3528672.

Cui, Siwei, Zhao, Gang, Gao, Yifei, Tavu, Tien, Huang, Jeff, 2022. VRust. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York, NY, USA, ISBN: 9781450394505, pp. 639–652. http://dx.doi.org/10.1145/3548606.3560552, URL https://dl.acm.org/doi/10.1145/3548606.3560552.

di Angelo, Monika, Durieux, Thomas, Ferreira, João F., Salzer, Gernot, 2023. SmartBugs 2.0: An execution framework for weakness detection in ethereum smart contracts.

di Angelo, Monika, Salzer, Gernot, 2019. A survey of tools for analyzing ethereum smart contracts. In: 2019 IEEE International Conference on Decentralized Applications and Infrastructures. DAPPCON, IEEE, Newark, CA, USA, ISBN: 978-1-7281-1264-0, pp. 69–78. http://dx.doi.org/10.1109/DAPPCON.2019.00018, URL https://ieeexplore.ieee.org/document/8782988/.

Dia, Bruno, Ivaki, Naghmeh, Laranjeiro, Nuno, 2021. An empirical evaluation of the effectiveness of smart contract verification tools. In: 2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing. PRDC, IEEE, Perth, Australia, ISBN: 978-1-6654-2476-9, pp. 17–26. http://dx.doi.org/10.1109/PRDC53464.2021.00013, URL https://ieeexplore.ieee.org/document/9667768/.

Ding, Mengjie, Li, Peiru, Li, Shanshan, Zhang, He, 2021. HFContractFuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection. In: Evaluation and Assessment in Software Engineering. ACM, New York, NY, USA, ISBN: 9781450390538, pp. 321–328. http://dx.doi.org/10.1145/3463274.3463351.

Durieux, Thomas, Ferreira, João F, Abreu, Rui, Cruz, Pedro, 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450371216, pp. 530–541. http://dx.doi.org/10.1145/3377811.3380364.

Edwards, Stephen, Lavagno, Luciano, Lee, Edward A, Sangiovanni-Vincentelli, Alberto, 2002. Design of embedded systems: Formal models, validation, and synthesis. In: De Micheli, Giovanni, Ernst, Rolf, Wolf, Wayne (Eds.), Readings in Hardware/Software Co-Design. In: Systems on Silicon, Morgan Kaufmann, San Francisco, pp. 86–107. http://dx.doi.org/10.1016/B978-155860702-6/50009-0, URL https://www.sciencedirect.com/science/article/pii/B9781558607026500090.

Eshghie, Mojtaba, Artho, Cyrille, Gurov, Dilian, 2021. Dynamic vulnerability detection on smart contracts using machine learning. In: Evaluation and Assessment in Software Engineering. ACM, New York, NY, USA, ISBN: 9781450390538, pp. 305–312. http://dx.doi.org/10.1145/3463274.3463348, URL https://dl.acm.org/doi/10.1145/3463274.3463348.

Ethereum, 2020. Ethereum Virtual Machine (EVM). https://ethereum.org.

Etherscan, 2021. About etherscan. https://etherscan.io/aboutus.

Feist, Josselin, Grieco, Gustavo, Groce, Alex, 2019. Slither: A static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain. WETSEB, In: WETSEB '19, IEEE, Montreal, QC, Canada, ISBN: 978-1-7281-2257-1, pp. 8–15. http://dx.doi.org/10.1109/WETSEB.2019.00008, URL https://doi.org/10.1109/WETSEB.2019.00008 https://ieeexplore.ieee.org/document/8823898/.

Feng, Yu, Torlak, Emina, Bodik, Rastislav, 2019a. Precise attack synthesis for smart contracts. URL http://arxiv.org/abs/1902.06067.

Feng, Xiaotao, Wang, Qin, Zhu, Xiaogang, Wen, Sheng, 2019b. Bug searching in smart contract. URL http://arxiv.org/abs/1905.00799.

Ferreira, João F., Cruz, Pedro, Durieux, Thomas, Abreu, Rui, 2020. SmartBugs. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, ISBN: 9781450367684, pp. 1349–1352. http://dx.doi.org/10.1145/3324884.3415298, URL https://dl.acm.org/doi/10.1145/3324884.3415298.

França, Horácio L., Teixeira, César, Laranjeiro, Nuno, 2023. An Empirical Analysis of Rebalancing Methods for Security Issue Report Identification. IEEE, Singapore.

Fu, Menglin, Wu, Lifa, Hong, Zheng, Zhu, Feng, Sun, He, Feng, Wenbo, 2019. A critical-path-coverage-based vulnerability detection method for smart contracts. IEEE Access (ISSN: 2169-3536) 7, 147327–147344. http://dx.doi.org/10.1109/ACCESS.2019.2947146.

Gao, Zhipeng, Jayasundara, Vinoj, Jiang, LingXiao, Xia, Xin, Lo, David, Grundy, John, 2019a. SmartEmbed: A tool for clone and bug detection in smart contracts through structural code embedding. In: 2019 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, Cleveland, OH, USA, ISBN: 978-1-7281-3094-1, pp. 394–397. http://dx.doi.org/10.1109/ICSME.2019.00067, URL https://ieeexplore.ieee.org/document/8919164/.

Gao, Jianbo, Liu, Han, Liu, Chao, Li, Qingshan, Guan, Zhi, Chen, Zhong, 2019b. EASYFLOW: Keep ethereum away from overflow. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings. ICSE-Companion, IEEE, Montreal, QC, Canada, ISBN: 978-1-7281-1764-5, pp. 23–26. http://dx.doi.org/10.1109/ICSE-Companion.2019.00029, URL https://ieeexplore.ieee.org/document/8802775/.

Ghaleb, Asem, Pattabiraman, Karthik, 2020a. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, ISBN: 9781450380089, pp. 415–427. http://dx.doi.org/10.1145/3395363.3397385, URL https://dl.acm.org/doi/10.1145/3395363.3397385.

Ghaleb, Asem, Pattabiraman, Karthik, 2020b. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2020, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450380089, pp. 415–427. http://dx.doi.org/10.1145/3395363.3397385.

Ghaleb, Asem, Rubin, Julia, Pattabiraman, Karthik, 2022. ETainter: Detecting gas-related vulnerabilities in smart contracts. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2022, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450393799, pp. 728–739. http://dx.doi.org/10.1145/3533767.3534378.

Google, 2021. Google scholar. https://scholar.google.com.

Grech, Neville, Kong, Michael, Jurisevic, Anton, Brent, Lexi, Scholz, Bernhard, Smaragdakis, Yannis, 2020. MadMax: Analyzing the out-of-gas world of smart contracts. Commun. ACM (ISSN: 0001-0782) 63 (10), 87–95. http://dx.doi.org/10.1145/3416262.

Grieco, Gustavo, Song, Will, Cygan, Artur, Feist, Josselin, Groce, Alex, 2020. Echidna: Effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2020, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450380089, pp. 557–560. http://dx.doi.org/10.1145/3395363.3404366.

Grishchenko, Ilya, Maffei, Matteo, Schneidewind, Clara, 2018. A semantic framework for the security analysis of ethereum smart contracts. In: Bauer, Lujo, Küsters, Ralf (Eds.), Principles of Security and Trust. Vol. 10804, Springer International Publishing, Uppsala, Sweden, pp. 243–269. http://dx.doi.org/10.1007/978-3-319-89722-6_10, ISBN 978-3-319-89721-9 978-3-319-89722-6, URL http://link.springer.com/10.1007/978-3-319-89722-6_10.

Group, Truffle Blockchain, 2021. Sweet tools for smart contracts. https://www.trufflesuite.com.

Gupta, Rajesh, Patel, Mohil Maheshkumar, Shukla, Arpit, Tanwar, Sudeep, 2022. Deep learning-based malicious smart contract detection scheme for internet of things environment. Comput. Electr. Eng. (ISSN: 00457906) 97, 107583. http://dx.doi.org/10.1016/j.compeleceng.2021.107583.

Hewa, Tharaka Mawanane, Hu, Yining, Liyanage, Madhusanka, Kanhare, Salil S., Ylianttila, Mika, 2021. Survey on blockchain-based smart contracts: Technical aspects and future research. IEEE Access (ISSN: 2169-3536) 9, 87643–87662. http://dx.doi.org/10.1109/ACCESS.2021.3068178.

Hirai, Yoichi, 2017. Defining the ethereum virtual machine for interactive theorem provers. pp. 520–535. http://dx.doi.org/10.1007/978-3-319-70278-0_33.

Hu, Tianyuan, Li, Bixin, Pan, Zhenyu, Qian, Chen, 2023. Detect defects of solidity smart contract based on the knowledge graph. IEEE Trans. Reliab. (ISSN: 0018-9529) 1–17. http://dx.doi.org/10.1109/TR.2023.3233999, URL https://ieeexplore.ieee.org/document/10025570/.

Hu, Bin, Zhang, Zongyang, Liu, Jianwei, Liu, Yizhong, Yin, Jiayuan, Lu, Rongxing, Lin, Xiaodong, 2021. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. Patterns (ISSN: 26663899) 2 (2), 100179. http://dx.doi.org/10.1016/j.patter.2020.100179.

Huang, Jianjun, Jiang, Jiasheng, You, Wei, Liang, Bin, 2022. Precise dynamic symbolic execution for nonuniform data access in smart contracts. IEEE Trans. Comput. (ISSN: 0018-9340) 71 (7), 1551–1563. http://dx.doi.org/10.1109/TC.2021.3092639.

Hwang, Seon-Jin, Choi, Seok-Hwan, Shin, Jinmyeong, Choi, Yoon-Ho, 2022. CodeNet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection. IEEE Access (ISSN: 2169-3536) 10, 32595–32607. http://dx.doi.org/10.1109/ACCESS.2022.3162065.

Janiesch, Christian, Zschech, Patrick, Heinrich, Kai, 2021. Machine learning and deep learning. Electron. Mark. (ISSN: 1019-6781) 31, 685–695. http://dx.doi.org/10.1007/s12525-021-00475-2.

Ji, Songyan, Dong, Jian, Qiu, Junfu, Gu, Bowen, Wang, Ye, Wang, Tongqi, 2021. Increasing fuzz testing coverage for smart contracts with dynamic taint analysis. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security. QRS, pp. 243–247. http://dx.doi.org/10.1109/QRS54544.2021.00035.

Jiang, Bo, Liu, Ye, Chan, W.K., 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. In: ASE 2018, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450359375, pp. 259–269. http://dx.doi.org/10.1145/3238147.3238177.

Kaleem, Mudabbir, Mavridou, Anastasia, Laszka, Aron, 2020. Vyper: A security comparison with solidity based on common vulnerabilities. In: 2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services. BRAINS, IEEE, ISBN: 978-1-7281-7091-6, pp. 107–111. http://dx.doi.org/10.1109/BRAINS49436.2020.9223278.

Kalra, Sukrit, Goel, Seep, Dhawan, Mohan, Sharma, Subodh, 2018. ZEUS: Analyzing safety of smart contracts. In: Proceedings 2018 Network and Distributed System Security Symposium. Internet Society, Reston, VA, ISBN: 1-891562-49-5, pp. 1–15. http://dx.doi.org/10.14722/ndss.2018.23082, 2018-2002, URL https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf.

Khan, Sajjad, Amin, Muhammad Bilal, Azar, Ahmad Taher, Aslam, Sheraz, 2021. Towards interoperable blockchains: A survey on the role of smart contracts in blockchain interoperability. IEEE Access (ISSN: 2169-3536) 9, 116672–116691. http://dx.doi.org/10.1109/ACCESS.2021.3106384.

Khan, Zulfiqar Ali, Namin, Akbar Siami, 2023. Dynamic analysis for detection of self-destructive smart contracts. In: 2023 IEEE 47th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, ISBN: 979-8-3503-2697-0, pp. 1093–1100. http://dx.doi.org/10.1109/COMPSAC57700.2023.00165.

Kitchenham, Barbara, 2004. Procedures for Performing Systematic Reviews. Technical Report, Keele University, Department of Computer Science, Keele University, UK.

Kolluri, Aashish, Nikolic, Ivica, Sergey, Ilya, Hobor, Aquinas, Saxena, Prateek, 2019. Exploiting the laws of order in smart contracts. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450362245, pp. 363–373. http://dx.doi.org/10.1145/3293882.3330560.

Krupp, Johannes, Rossow, Christian, 2018. TEETHER: Gnawing at ethereum to automatically exploit smart contracts. In: Proceedings of the 27th USENIX Conference on Security Symposium. SEC '18, USENIX Association, USA, ISBN: 9781931971461, pp. 1317–1333.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, IEEE, San Jose, CA, USA, ISBN: 0-7695-2102-9, pp. 75–86. http://dx.doi.org/10.1109/CGO.2004.1281665, URL http://ieeexplore.ieee.org/document/1281665/.

Li, Wenyuan, He, Jiahao, Zhao, Gansen, Yang, Jinji, Li, Shuangyin, Lai, Ruilin, Li, Ping, Tang, Hua, Luo, Haoyu, Zhou, Ziheng, 2022a. EOSIOAnalyzer: An effective static analysis vulnerability detection framework for EOSIO smart contracts. In: 2022 IEEE 46th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, Los Alamitos, CA, USA, ISBN: 978-1-6654-8810-5, pp. 746–756. http://dx.doi.org/10.1109/COMPSAC54236.2022.00124, URL https://ieeexplore.ieee.org/document/9842620/.

Li, Peiru, Li, Shanshan, Ding, Mengjie, Yu, Jiapeng, Zhang, He, Zhou, Xin, Li, Jingyue, 2022b. A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis. In: The International Conference on Evaluation and Assessment in Software Engineering 2022. ACM, New York, NY, USA, ISBN: 9781450396134, pp. 366–374. http://dx.doi.org/10.1145/3530019.3531342.

Li, Zhaoxuan, Lu, Siqi, Zhang, Rui, Xue, Rui, Ma, Wenqiu, Liang, Rujin, Zhao, Ziming, Gao, Sheng, 2022c. SmartFast: an accurate and robust formal analysis tool for Ethereum smart contracts. Empir. Softw. Eng. (ISSN: 1382-3256) 27 (7), 197. http://dx.doi.org/10.1007/s10664-022-10218-2.

Li, Bixin, Pan, Zhenyu, Hu, Tianyuan, 2022d. ReDefender: Detecting reentrancy vulnerabilities in smart contracts automatically. IEEE Trans. Reliab. (ISSN: 0018-9529) 71 (2), 984–999. http://dx.doi.org/10.1109/TR.2022.3161634.

Li, Mengliang, Ren, Xiaoxue, Fu, Han, Li, Zhuo, Sun, Jianling, 2023. ConvMHSA-SCVD: Enhancing smart contract vulnerability detection through a knowledge-driven and data-driven framework. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering. ISSRE, IEEE, ISBN: 979-8-3503-1594-3, pp. 578–589. http://dx.doi.org/10.1109/ISSRE59848.2023.00025.

Liao, Jian-Wei, Tsai, Tsung-Ta, He, Chia-Kang, Tien, Chin-Wei, 2019. SoliAudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security. IOTSMS, IEEE, Granada, Spain, ISBN: 978-1-7281-2949-5, pp. 458–465. http://dx.doi.org/10.1109/IOTSMS48152.2019.8939256, URL https://ieeexplore.ieee.org/document/8939256/.

Liao, Zeqin, Zheng, Zibin, Chen, Xiao, Nan, Yuhong, 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, ISBN: 9781450393799, pp. 752–764. http://dx.doi.org/10.1145/3533767.3534222.

Liu, Zhenpeng, Jiang, Mingxiao, Zhang, Shengcong, Zhang, Jialiang, Liu, Yi, 2023a. A smart contract vulnerability detection mechanism based on deep learning and expert rules. IEEE Access (ISSN: 2169-3536) 11, 77990–77999. http://dx.doi.org/10.1109/ACCESS.2023.3298048.

Liu, J., Liu, Z., 2019. A survey on security verification of blockchain smart contracts. IEEE Access (ISSN: 2169-3536) 7, 77894–77904. http://dx.doi.org/10.1109/ACCESS.2019.2921624.

Liu, Chao, Liu, Han, Cao, Zhao, Chen, Zhong, Chen, Bangdao, Roscoe, Bill, 2018. ReGuard: Finding reentrancy bugs in smart contracts. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ACM, New York, NY, USA, ISBN: 9781450356633, pp. 65–68. http://dx.doi.org/10.1145/3183440.3183495, URL https://dl.acm.org/doi/10.1145/3183440.3183495.

Liu, Zhenguang, Qian, Peng, Wang, Xiaoyang, Zhuang, Yuan, Qiu, Lin, Wang, Xun, 2021. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans. Knowl. Data Eng. (ISSN: 1041-4347) 35 (2), 1. http://dx.doi.org/10.1109/TKDE.2021.3095196, URL https://ieeexplore.ieee.org/document/9477066/.

Liu, Zhenguang, Qian, Peng, Yang, Jiaxu, Liu, Lingfeng, Xu, Xiaojun, He, Qinming, Zhang, Xiaosong, 2023b. Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. IEEE Trans. Inf. Forensics Secur. (ISSN: 1556-6013) 18, 1237–1251. http://dx.doi.org/10.1109/TIFS.2023.3237370.

Lu, Ning, Wang, Bin, Zhang, Yongxin, Shi, Wenbo, Esposito, Christian, 2019. NeuCheck: A more practical Ethereum smart contract security analysis tool. Softw. - Pract. Exp. n/a (n/a), 1–20. http://dx.doi.org/10.1002/spe.2745, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2745.

Luu, Loi, Chu, Duc-Hiep, Olickel, Hrishi, Saxena, Prateek, Hobor, Aquinas, 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450341394, pp. 254–269. http://dx.doi.org/10.1145/2976749.2978309.

Ma, Fuchen, Ren, Meng, Ouyang, Lerong, Chen, Yuanliang, Zhu, Juan, Chen, Ting, Zheng, Yingli, Dai, Xiao, Jiang, Yu, Sun, Jiaguang, 2023a. Pied-piper: Revealing the backdoor threats in ethereum ERC token contracts. ACM Trans. Softw. Eng. Methodol. (ISSN: 1049-331X) 32 (3), 1–24. http://dx.doi.org/10.1145/3560264.

Ma, Fuchen, Ren, Meng, Ying, Fu, Sun, Wanting, Song, Houbing, Shi, Heyuan, Jiang, Yu, Li, Huizhong, 2023b. V-Gas: Generating high gas consumption inputs to avoid out-of-gas vulnerability. ACM Trans. Internet Technol. (ISSN: 1533-5399) 23 (3), 1–22. http://dx.doi.org/10.1145/3511900.

Ma, Chenyang, Song, Wei, Huang, Jeff, 2023c. TransRacer: Function dependence-guided transaction race detection for smart contracts. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, New York, NY, USA, ISBN: 9798400703270, pp. 947–959. http://dx.doi.org/10.1145/3611643.3616281.

Ma, Fuchen, Xu, Zhenyang, Ren, Meng, Yin, Zijing, Chen, Yuanliang, Qiao, Lei, Gu, Bin, Li, Huizhong, Jiang, Yu, Sun, Jiaguang, 2022. Pluto: Exposing vulnerabilities in inter-contract scenarios. IEEE Trans. Softw. Eng. (ISSN: 0098-5589) 48 (11), 4380–4396. http://dx.doi.org/10.1109/TSE.2021.3117966, URL https://ieeexplore.ieee.org/document/9562567/.

Mavridou, Anastasia, Laszka, Aron, Stachtiari, Emmanouela, Dubey, Abhishek, 2019. VeriSolid: Correct-by-design smart contracts for ethereum. In: Goldberg, Ian, Moore, Tyler (Eds.), Financial Cryptography and Data Security. Springer International Publishing, Cham, ISBN: 978-3-030-32101-7, pp. 446–465.

Mi, Feng, Wang, Zhuoyi, Zhao, Chen, Guo, Jinghui, Ahmed, Fawaz, Khan, Latifur, 2021. VSCL: Automating vulnerability detection in smart contracts with deep learning. In: 2021 IEEE International Conference on Blockchain and Cryptocurrency. ICBC, IEEE, Sydney, Australia, ISBN: 978-1-6654-3578-9, pp. 1–9. http://dx.doi.org/10.1109/ICBC51069.2021.9461050, URL https://ieeexplore.ieee.org/document/9461050/.

Momeni, Pouyan, Wang, Yu, Samavi, Reza, 2019. Machine learning model for smart contracts security analysis. In: 2019 17th International Conference on Privacy, Security and Trust. PST, IEEE, Fredericton, NB, Canada, ISBN: 978-1-7281-3265-5, pp. 1–6. http://dx.doi.org/10.1109/PST47121.2019.8949045, URL https://ieeexplore.ieee.org/document/8949045/.

Mueller, Bernhard, 2018. LASER-ethereum. URL https://github.com/muellerberndt/laser-ethereum.

Myers, Glenford J., Badgett, Tom, Sandler, Corey (Eds.), 2012. The Art of Software Testing. Wiley, ISBN: 9781118031964, http://dx.doi.org/10.1002/9781119202486, URL https://onlinelibrary.wiley.com/doi/book/10.1002/9781119202486.

MythX, 2020. SWC. URL https://swcregistry.io/.

Nassirzadeh Behkish and Sun, Huaiying, Sebastian, Banescu, Vijay, Ganesh, 2023. Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities. In: Pardalos Panos and Kotsireas, Ilias, Yike, Guo, William, Knottenbelt (Eds.), Mathematical Research for Blockchain Economy. Springer International Publishing, Cham, ISBN: 978-3-031-18679-0, pp. 143–167.

NCC Group, 2019. DASP. URL https://dasp.co/.

Nguyen, Hoang H., Nguyen, Nhat-Minh, Xie, Chunyao, Ahmadi, Zahra, Kudendo, Daniel, Doan, Thanh-Nam, Jiang, Lingxiao, 2023. MANDO-HGT: Heterogeneous graph transformers for smart contract vulnerability detection. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories. MSR, IEEE, ISBN: 979-8-3503-1184-6, pp. 334–346. http://dx.doi.org/10.1109/MSR59073.2023.00052.

Nguyen, Tai D, Pham, Long H, Sun, Jun, Lin, Yun, Minh, Quang Tran, 2020. SFuzz: An efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450371216, pp. 778–788. http://dx.doi.org/10.1145/3377811.3380334.

Nishida Yuki and Saito, Hiromasa, Ran, Chen, Akira, Kawata, Jun, Furuse, Kohei, Suenaga, Atsushi, Igarashi, 2021. Helmholtz: A verifier for tezos smart contracts based on refinement types. In: Friso, Groote Jan, Larsen, Kim Guldstrand (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham, ISBN: 978-3-030-72013-1, pp. 262–280.

Openzeppelin, 2015. GitHub openzeppelin. URL https://github.com/OpenZeppelin/.

Openzeppelin, 2015. OpenZeppelin. URL https://www.openzeppelin.com/.

Openzeppelin, 2024. GitHub dapps. URL https://github.com/topics/dapps.

Palladino, Santiago, 2019. The Parity Wallet Hack Explained. OpenZeppelin, URL https://loess.ru/pdf/2017-07-20_14.33.01 https_blog.zeppelin.solutions_on-the-parity-wallet-multisig-hack-405a8c1.pdf.

Pani, Siddhasagar, Nallagonda, Harshita Vani, Vigneswaran, Medicherla, Raveendra Kumar, M, Rajan, 2023. SmartFuzzDriverGen: Smart contract fuzzing automation for golang. In: 16th Innovations in Software Engineering Conference. ACM, New York, NY, USA, ISBN: 9798400700644, pp. 1–11. http://dx.doi.org/10.1145/3578527.3578538.

Pasqua, Michele, Benini, Andrea, Contro, Filippo, Crosara, Marco, Dalla Preda, Mila, Ceccato, Mariano, 2023. Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode. J. Syst. Softw. (ISSN: 01641212) 200, 111653. http://dx.doi.org/10.1016/j.jss.2023.111653.

Praitheeshan, Purathani, Pan, Lei, Yu, Jiangshan, Liu, Joseph, Doss, Robin, 2020. Security analysis methods on ethereum smart contract vulnerabilities: A survey.

Protofire, 2024. Solhint. URL https://protofire.github.io/solhint/.

Remix team, 2021. Remix. URL https://github.com/DeNetPRO/remix-verify-tool.

Ren, Meng, Yin, Zijing, Ma, Fuchen, Xu, Zhenyang, Jiang, Yu, Sun, Chengnian, Li, Huizhong, Cai, Yan, 2021. Empirical evaluation of smart contract testing: what is the best choice? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, ISBN: 9781450384599, pp. 566–579. http://dx.doi.org/10.1145/3460319.3464837, URL https://dl.acm.org/doi/10.1145/3460319.3464837.

Research, The Computing, of Australasia, Education Association, 2021. CORE conference ranking. URL http://portal.core.edu.au/conf-ranks/.

Rival, Xavier (Ed.), 2016. Static analysis. In: Lecture Notes in Computer Science, Vol. 9837, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-662-53412-0, http://dx.doi.org/10.1007/978-3-662-53413-7, URL http://link.springer.com/10.1007/978-3-662-53413-7.

Rodler, Michael, Li, Wenting, Karame, Ghassan O., Davi, Lucas, 2019. Sereum: Protecting existing smart contracts against re-entrancy attacks. In: Proceedings 2019 Network and Distributed System Security Symposium. Internet Society, Reston, VA, ISBN: 1-891562-55-X, http://dx.doi.org/10.14722/ndss.2019.23413, URL https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_09-3_Rodler_paper.pdf.

Schumann, Johann M., 2001. Automated Theorem Proving in Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-642-08759-2, p. 228. http://dx.doi.org/10.1007/978-3-662-22646-9, URL http://link.springer.com/10.1007/978-3-322-90100-2 http://link.springer.com/10.1007/978-3-662-22646-9.

Seligman, Erik, Schubert, Tom, Achutha Kiran Kumar, M.V., 2015. Formal Verification, first ed. Elsevier, ISBN: 9780128007273, http://dx.doi.org/10.1016/C2013-0-18672-2, URL https://linkinghub.elsevier.com/retrieve/pii/C20130186722.

Shakya, Supriya, Mukherjee, Arnab, Halder, Raju, Maiti, Abyayananda, Chaturvedi, Amrita, 2022. SmartMixModel: Machine learning-based vulnerability detection of solidity smart contracts. In: 2022 IEEE International Conference on Blockchain (Blockchain). IEEE, Espoo, Finland, ISBN: 978-1-6654-6104-7, pp. 37–44. http://dx.doi.org/10.1109/Blockchain55522.2022.00016, URL https://ieeexplore.ieee.org/document/9881798/.

Shou, Chaofan, Tan, Shangyin, Sen, Koushik, 2023. ItyFuzz: Snapshot-based fuzzer for smart contract. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, ISBN: 9798400702211, pp. 322–333. http://dx.doi.org/10.1145/3597926.3598059.

Siegel, David, 2016. Understanding the DAO attack. URL https://www.coindesk.com/understanding-dao-hack-journalists.

Singh, Amritraj, Parizi, Reza, Zhang, Qi, Choo, Kim-Kwang Raymond, Dehghantanha, Ali, 2019. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. Comput. Secur. (ISSN: 01674048) 88, 101654. http://dx.doi.org/10.1016/j.cose.2019.101654.

Singh, Sanjay Kumar, Singh, Amarjeet, 2012. Software Testing. Vandana Publications, India, ISBN: 978-81-941110-6-1.

Sinnema, R., Wilde, E., 2013. XACML. URL https://datatracker.ietf.org/doc/html/rfc7061.

So, Sunbeom, Lee, Myungho, Park, Jisu, Lee, Heejo, Oh, Hakjoo, 2020. VERISMART: A highly precise safety verifier for Ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, San Francisco, CA, USA, ISBN: 978-1-7281-3497-0, pp. 1678–1694. http://dx.doi.org/10.1109/SP40000.2020.00032, URL https://ieeexplore.ieee.org/document/9152689/.

Solidity, 2023. Solidity documentation - Breaking changes 0.5.0. URL https://docs.soliditylang.org/en/latest/050-breaking-changes.html.

Song, Daniel, Zhao, Jisheng, Burke, Michael, Sbîrlea, Dragoş, Wallach, Dan, Sarkar, Vivek, 2015. Finding Tizen security bugs through whole-system static analysis. Comput. Sci. 1–15, URL https://arxiv.org/pdf/1504.05967.pdf http://arxiv.org/abs/1504.05967.

Stephens, Jon, Ferles, Kostas, Mariano, Benjamin, Lahiri, Shuvendu, Dillig, Isil, 2021. SmartPulse: Automated checking of temporal properties in smart contracts. In: 2021 IEEE Symposium on Security and Privacy. SP, IEEE, San Francisco, CA, USA, ISBN: 978-1-7281-8934-5, pp. 555–571. http://dx.doi.org/10.1109/SP40001.2021.00085, URL https://ieeexplore.ieee.org/document/9519387/.

Su, Jianzhong, Dai, Hong-Ning, Zhao, Lingjun, Zheng, Zibin, Luo, Xiapu, 2022. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, ISBN: 9781450394758, pp. 1–12. http://dx.doi.org/10.1145/3551349.3560429.

Sun, Xiaobing, Tu, Liangqiong, Zhang, Jiale, Cai, Jie, Li, Bin, Wang, Yu, 2023. ASSBert: Active and semi-supervised bert for smart contract vulnerability detection. J. Inf. Secur. Appl. (ISSN: 22142126) 73, 103423. http://dx.doi.org/10.1016/j.jisa.2023.103423.

Surucu, Onur, Yeprem, Uygar, Wilkinson, Connor, Hilal, Waleed, Gadsden, Stephen Andrew, Yawney, John, Alsadi, Naseem, Giuliano, Alessandro, 2022. A survey on ethereum smart contract vulnerability detection using machine learning. In: Blowers, Misty, Hall, Russell D., Dasari, Venkateswara R. (Eds.), Disruptive Technologies in Information Sciences VI. SPIE, Orlando, Florida, United States, ISBN: 9781510651104, p. 12. http://dx.doi.org/10.1117/12.2618899, URL https://www.spiedigitallibrary.org/conference-proceedings-of-spie/12117/2618899/A-survey-on-ethereum-smart-contract-vulnerability-detection-using-machine/10.1117/12.2618899.full.

Swan, Melanie, 2015. Blockchain - Blueprint for a New Economy, first ed. O'Reilly Media, Inc., Sebastopol, CA, EUA, ISBN: 9781491920497.

SWC Registry, 2018. SWC-111. URL https://swcregistry.io/docs/SWC-111/.

Takahashi, Kanae, Yamamoto, Kouji, Kuchiba, Aya, Koyama, Tatsuki, 2022. Confidence interval for micro-averaged F1 and macro-averaged F1 scores. Appl. Intell. (ISSN: 0924-669X) 52 (5), 4961–4972. http://dx.doi.org/10.1007/s10489-021-02635-5, URL https://link.springer.com/10.1007/s10489-021-02635-5.

Tikhomirov, Sergei, Voskresenskaya, Ekaterina, Ivanitskiy, Ivan, Takhaviev, Ramil, Marchenko, Evgeny, Alexandrov, Yaroslav, 2018. SmartCheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. ACM, New York, NY, USA, ISBN: 9781450357265, pp. 9–16. http://dx.doi.org/10.1145/3194113.3194115, URL https://dl.acm.org/doi/10.1145/3194113.3194115.

Torres, Christof Ferreira, Iannillo, Antonio Ken, Gervais, Arthur, State, Radu, 2021. ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In: 2021 IEEE European Symposium on Security and Privacy. EuroS&P, IEEE, Vienna, Austria, ISBN: 978-1-6654-1491-3, pp. 103–119. http://dx.doi.org/10.1109/EuroSP51992.2021.00018, URL https://ieeexplore.ieee.org/document/9581164/.

Torres, Christof Ferreira, Schütte, Julian, State, Radu, 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference. ACSAC '18, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450365697, pp. 664–676. http://dx.doi.org/10.1145/3274694.3274737.

Tsankov, Petar, Dan, Andrei, Drachsler-Cohen, Dana, Gervais, Arthur, Bünzli, Florian, Vechev, Martin, 2018. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. CCS '18, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450356930, pp. 67–82. http://dx.doi.org/10.1145/3243734.3243780.

Vidal, Fernando Richter, Ivaki, Naghmeh, Laranjeiro, Nuno, 2021. Revocation mechanisms for blockchain applications: A review. In: 2021 10th Latin-American Symposium on Dependable Computing. LADC, IEEE, ISBN: 978-1-6654-7831-1, pp. 01–10. http://dx.doi.org/10.1109/LADC53747.2021.9672577, URL https://ieeexplore.ieee.org/document/9672577/.

Vidal, Fernando Richter, Ivaki, Naghmeh, Laranjeiro, Nuno, 2022. Vulnerability detection for smart contracts: A systematic literature review - Supplementary material. URL https://zenodo.org/record/8109651.

Vidal, Fernando Richter, Ivaki, Naghmeh, Laranjeiro, Nuno, 2023. OpenSCV: An open hierarchical taxonomy for smart contract vulnerabilities.

W3C, 2011. xPath. URL https://www.w3.org/TR/xpath20/.

Wang, Zeli, Jin, Hai, Dai, Weiqi, Choo, Kim-Kwang Raymond, Zou, Deqing, 2020. Ethereum smart contract security research: survey and future research opportunities. Front. Comput. Sci. 15 (2), 152802. http://dx.doi.org/10.1007/s11704-020-9284-9, ISSN 2095-2228, 2095-2236, URL http://link.springer.com/10.1007/s11704-020-9284-9.

Wang, Haijun, Li, Yi, Lin, Shang-Wei, Ma, Lei, Liu, Yang, 2019. VULTRON: Catching vulnerable smart contracts once and for all. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results. ICSE-NIER, IEEE, Montreal, QC, Canada, ISBN: 978-1-7281-1758-4, pp. 1–4. http://dx.doi.org/10.1109/ICSE-NIER.2019.00009, URL https://ieeexplore.ieee.org/document/8805696/.

Wang, Wei, Song, Jingjing, Xu, Guangquan, Li, Yidong, Wang, Hao, Su, Chunhua, 2021. ContractWard: Automated vulnerability detection models for ethereum smart contracts. IEEE Trans. Netw. Sci. Eng. (ISSN: 2327-4697) 8 (2), 1133–1144. http://dx.doi.org/10.1109/TNSE.2020.2968505, URL https://ieeexplore.ieee.org/document/8967006/.

Wang Zexu and Wen, Bin, Ziqiang, Luo, Shaojie, Liu, 2021. M-A-R: A dynamic symbol execution detection method for smart contract reentry vulnerability. In: Dai Hong-Ning and Liu, Xuanzhe, Xiapu, Luo Daniel, Jiang, Xiao, Xiangping, Chen (Eds.), Blockchain and Trustworthy Systems. Springer Singapore, Singapore, ISBN: 978-981-16-7993-3, pp. 418–429.

Weiss, Konrad, Schütte, Julian, 2019. Annotary: A concolic execution system for developing secure smart contracts. pp. 747–766. http://dx.doi.org/10.1007/978-3-030-29959-0_36, URL http://link.springer.com/10.1007/978-3-030-29959-0_36.

Wesley Scott and Christakis, Maria, A., Navas Jorge, Richard, Trefler, Valentin, Wüstholz, Arie, Gurfinkel, 2022. Verifying solidity smart contracts via communication abstraction in SmartACE. In: Finkbeiner Bernd and Wies, Thomas (Ed.), Verification, Model Checking, and Abstract Interpretation. Springer International Publishing, Cham, ISBN: 978-3-030-94583-1, pp. 425–449.

Wohlin, Claes, 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14. ACM Press, New York, New York, USA, ISBN: 9781450324762, pp. 1–10. http://dx.doi.org/10.1145/2601248.2601268.

Wu, Hongjun, Zhang, Zhuo, Wang, Shangwen, Lei, Yan, Lin, Bo, Qin, Yihao, Zhang, Haoyu, Mao, Xiaoguang, 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering. ISSRE, IEEE, Wuhan, China, ISBN: 978-1-6654-2587-2, pp. 378–389. http://dx.doi.org/10.1109/ISSRE52982.2021.00047, URL https://ieeexplore.ieee.org/document/9700296/.

Xing, Cipai, Chen, Zhuorong, Chen, Lexin, Guo, Xiaojie, Zheng, Zibin, Li, Jin, 2020. A new scheme of vulnerability analysis in smart contract with machine learning. Wirel. Netw. (ISSN: 1572-8196) http://dx.doi.org/10.1007/s11276-020-02379-z.

Xu, Zhiwu, Wen, Cheng, Qin, Shengchao, 2018. State-taint analysis for detecting resource bugs. Sci. Comput. Program. (ISSN: 01676423) 162, 93–109. http://dx.doi.org/10.1016/j.scico.2017.06.010, URL https://linkinghub.elsevier.com/retrieve/pii/S0167642317301314.

Xue, Yinxing, Ye, Jiaming, Zhang, Wei, Sun, Jun, Ma, Lei, Wang, Haijun, Zhao, Jianjun, 2022. xFuzz: Machine learning guided cross-contract fuzzing. IEEE Trans. Dependable Secure Comput. (ISSN: 1545-5971) 1–14. http://dx.doi.org/10.1109/TDSC.2022.3182373, URL https://ieeexplore.ieee.org/document/9795233/.

Yaga, Dylan, Mell, Peter, Roby, Nik, Scarfone, Karen, 2018. Blockchain technology overview. Technical Report, National Institute of Standards and Technology, Gaithersburg, MD, http://dx.doi.org/10.6028/NIST.IR.8202, URL https://doi.org/10.6028/NIST.IR.8202 https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf.

Yang, Zhongju, Zhu, Weixing, Yu, Minggang, 2023. Improvement and optimization of vulnerability detection methods for ethernet smart contracts. IEEE Access (ISSN: 2169-3536) 11, 78207–78223. http://dx.doi.org/10.1109/ACCESS.2023.3298672.

Yao, Yao, Li, Hui, Yang, Xin, Le, Yiwang, 2022. An improved vulnerability detection system of smart contracts based on symbolic execution. In: 2022 IEEE International Conference on Big Data (Big Data). IEEE, ISBN: 978-1-6654-8045-1, pp. 3225–3234. http://dx.doi.org/10.1109/BigData55660.2022.10020730.

Ye, Jiaming, Ma, Mingliang, Lin, Yun, Ma, Lei, Xue, Yinxing, Zhao, Jianjun, 2022. Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures. J. Syst. Softw. (ISSN: 01641212) 192, 111410. http://dx.doi.org/10.1016/j.jss.2022.111410.

Yu, Lei, Lu, Junyi, Liu, Xianglong, Yang, Li, Zhang, Fengjun, Ma, Jiajia, 2023. PSCVFinder: A prompt-tuning based framework for smart contract vulnerability detection. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering. ISSRE, IEEE, ISBN: 979-8-3503-1594-3, pp. 556–567. http://dx.doi.org/10.1109/ISSRE59848.2023.00030.

Yu, Xingxin, Zhao, Haoyue, Hou, Botao, Ying, Zonghao, Wu, Bin, 2021. DeeSCVHunter: A deep learning-based framework for smart contract vulnerability detection. In: 2021 International Joint Conference on Neural Networks. IJCNN, IEEE, Shenzhen, China, ISBN: 978-1-6654-3900-8, pp. 1–8. http://dx.doi.org/10.1109/IJCNN52387.2021.9534324, URL https://ieeexplore.ieee.org/document/9534324/.

Zeng, Qingren, He, Jiahao, Zhao, Gansen, Li, Shuangyin, Yang, Jingji, Tang, Hua, Luo, Haoyu, 2022. EtherGIS: A vulnerability detection framework for ethereum smart contracts based on graph learning features. In: 2022 IEEE 46th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, Los Alamitos, CA, USA, ISBN: 978-1-6654-8810-5, pp. 1742–1749. http://dx.doi.org/10.1109/COMPSAC54236.2022.00277, URL https://ieeexplore.ieee.org/document/9842713/.

Zhang, Zhuo, Lei, Yan, Yan, Meng, Yu, Yue, Chen, Jiachi, Wang, Shangwen, Mao, Xiaoguang, 2022a. Reentrancy vulnerability detection and localization: A deep learning based two-phase approach. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, ISBN: 9781450394758, pp. 1–13. http://dx.doi.org/10.1145/3551349.3560428, URL https://dl.acm.org/doi/10.1145/3551349.3560428.

Zhang, Jiashuo, Li, Yue, Gao, Jianbo, Guan, Zhi, Chen, Zhong, 2023. Siguard: Detecting signature-related vulnerabilities in smart contracts. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings. ICSE-Companion, IEEE, ISBN: 979-8-3503-2263-7, pp. 31–35. http://dx.doi.org/10.1109/ICSE-Companion58688.2023.00019.

Zhang, Qingzhao, Wang, Yizhuo, Li, Juanru, Ma, Siqi, 2020a. EthPloit: From fuzzing to efficient exploit generation against smart contracts. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, London, ON, Canada, ISBN: 978-1-7281-5143-4, pp. 116–126. http://dx.doi.org/10.1109/SANER48275.2020.9054822, URL https://ieeexplore.ieee.org/document/9054822/.

Zhang, Shuai, Wang, Meng, Liu, Yi, Zhang, Yuhan, Yu, Bin, 2022b. Multi-transaction sequence vulnerability detection for smart contracts based on inter-path data dependency. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security. QRS, IEEE, Guangzhou, China, ISBN: 978-1-6654-7704-8, pp. 616–627. http://dx.doi.org/10.1109/QRS57517.2022.00068, URL https://ieeexplore.ieee.org/document/10062352/.

Zhang, Lejun, Wang, Jinlong, Wang, Weizheng, Jin, Zilong, Su, Yansen, Chen, Huiling, 2022c. Smart contract vulnerability detection combined with multi-objective detection. Comput. Netw. (ISSN: 13891286) 217, 109289. http://dx.doi.org/10.1016/j.comnet.2022.109289.

Zhang, Pengcheng, Xiao, Feng, Luo, Xiapu, 2019a. SolidityCheck : Quickly detecting smart contract problems through regular expressions. URL https://arxiv.org/abs/1911.09425.

Zhang, Pengcheng, Xiao, Feng, Luo, Xiapu, 2020b. A framework and DataSet for bugs in ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, ISBN: 978-1-7281-5619-4, pp. 139–150. http://dx.doi.org/10.1109/ICSME46990.2020.00023.

Zhang, Pengcheng, Xiao, Feng, Luo, Xiapu, 2020c. A framework and DataSet for bugs in Ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, Adelaide, SA, Australia, ISBN: 978-1-7281-5619-4, pp. 139–150. http://dx.doi.org/10.1109/ICSME46990.2020.00023, URL https://ieeexplore.ieee.org/document/9240706/.

Zhang, Rui, Xue, Rui, Liu, Ling, 2019b. Security and privacy on blockchain. ACM Comput. Surv. (ISSN: 0360-0300) 52 (3), http://dx.doi.org/10.1145/3316481.

Zheng, Gavin, Gao, Longxiang, Huang, Liqun, Guan, Jian, 2021. Ethereum Smart Contract Development in Solidity. Springer Singapore, Singapore, ISBN: 978-981-15-6217-4, http://dx.doi.org/10.1007/978-981-15-6218-1, URL http://link.springer.com/10.1007/978-981-15-6218-1.

Zhou, Zhi-Hua, 2012. Ensemble Methods: Foundations and Algorithms (Chapman & Hall/CRC Machine Learning & Pattern Recognition), first ed. Chapman and Hall/CRC, ISBN: 1439830037, pp. 1–236.

Zhou, Qihao, Zheng, Kan, Zhang, Kuan, Hou, Lu, Wang, Xianbin, 2022. Vulnerability analysis of smart contract for blockchain-based IoT applications: A machine learning approach. IEEE Internet Things J. (ISSN: 2327-4662) 9 (24), 24695–24707. http://dx.doi.org/10.1109/JIOT.2022.3196269.

Zhu, Huijuan, Yang, Kaixuan, Wang, Liangmin, Xu, Zhicheng, Sheng, Victor S., 2023. GraBit: A sequential model-based framework for smart contract vulnerability detection. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering. ISSRE, IEEE, ISBN: 979-8-3503-1594-3, pp. 568–577. http://dx.doi.org/10.1109/ISSRE59848.2023.00024.

Zhuang, Yuan, Liu, Zhenguang, Qian, Peng, Liu, Qi, Wang, Xiang, He, Qinming, 2020. Smart contract vulnerability detection using graph neural network. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence. International Joint Conferences on Artificial Intelligence Organization, California, ISBN: 978-0-9992411-6-5, pp. 3283–3290. http://dx.doi.org/10.24963/ijcai.2020/454.

Zou, Weiqin, Lo, David, Kochhar, Pavneet Singh, Le, Xuan-Bach D, Xia, Xin, Feng, Yang, Chen, Zhenyu, Xu, Baowen, 2019. Smart contract development: Challenges and opportunities. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2019.2942301.

Fernando Vidal is a Ph.D. student at the University of Coimbra, Portugal. His research interests are related to blockchain technology. Fernando has been publishing at international conferences, addressing some of his findings of blockchain technology, such as vulnerabilities in smart contracts and revocation. In addition, Fernando was invited by the Advances in Science, Technology and Engineering Systems Journal (ASTESJ) magazine and IEEE Potentials to be one of the reviewers of the blockchain submissions. Fernando has applied his acquired knowledge, helping companies implement blockchain technology through consulting.

Naghmeh Ivaki received the Ph.D. degree from the University of Coimbra, Portugal. Currently, she is an assistant professor and a full member of the Software and Systems Engineering Group (SSE) of the Center for Informatics and Systems (CISUC), Department of Informatics Engineering, University of Coimbra. She specializes in the scientific field of Informatics Engineering, with a particular focus on the dependability, security, and safety assessment of computer systems. In her field of specialization, she has authored more than 45 peer-reviewed publications and participated in several national and international research projects.

Nuno Laranjeiro Nuno Laranjeiro received the Ph.D. from the University of Coimbra, Portugal, where he is currently an Associate Professor. His research focuses on dependable and secure software services, and his interests include experimental dependability evaluation, fault injection, robustness of software and web services, web services interoperability, and software security. He is currently mostly involved in developing new techniques toward more reliable and secure cloud systems and microservices, in developing techniques for evaluating the reliability and security of blockchain smart contracts and using machine learning techniques for software fault and vulnerability management in the software development lifecycle. He has contributed, as an author, reviewer, and program committee member, to leading conferences and journals in the dependability and services computing areas. Nuno has been involved in the organization of several international events, including several editions of the International Symposium on Software Reliability Engineering (ISSRE) and also the Dependable and Secure Services Workshop/Track (as main chair) jointly organized with the IEEE World Congress on Services. He participated in international research projects, including several H2020 and FP7 projects (e.g., ADVANCE, DEVASSES, ATMOSPHERE, EUBrasilCloudFORUM) and FP7 projects (CRITICAL STEP, CECRIS).