Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SURVEY PAPERS

Detecting Functional and Security-Related Issues in Smart Contracts: A Systematic Literature Review

Valentina Piantadosi* | Giovanni Rosa | Davide Placella | Simone Scalabrino | Rocco Oliveto

[1]University of Molise, Molise, Italy

Correspondence
*Corresponding author name, This is sample corresponding address. Email: valentina.piantadosi@unimol.it

Present Address
This is sample for present address text this is sample for present address text

Summary

Blockchain is a platform of distributed elaboration, which allows users to provide software for a huge range of next-generation decentralized applications without involving reliable third parties. Smart Contracts (SCs) are an important component in Blockchain applications: they are programmatic agreements among two or more parties that can not be rescinded. Furthermore, SCs have an important characteristic: they allow users to implement reliable transactions without involving third parties. However, the advantages of SCs have a price. Like any program, SCs can contain bugs, some of which may also constitute security threats. Writing correct and secure SCs can be extremely difficult because, once deployed, they can not be modified. Although SCs have been recently introduced, a large number of approaches have been proposed to find bugs and vulnerabilities in SCs. In this paper, we present a systematic literature review on the approaches for the automated detection of bugs and vulnerabilities in SCs. We survey 68 papers published between 2015 and 2020, and we annotate each paper according to our classification framework to provide quantitative results and find possible areas not explored yet. Finally, we identify the open problems in this research field to provide possible directions to future researchers.

KEYWORDS:
Blockchain, Smart Contracts

1 | INTRODUCTION

In 2008, Satoshi Nakamoto[1] introduced the formal idea of Blockchain as an infrastructural technology. After, Blockchain marked a wide range of industrial sectors, *i.e.*, Security, Privacy, Finance, Cloud Computing and the Internet of Things (IoT). Recently, Smart Contracts (SCs) emerged as a new kind of software programs enabled by the Blockchain. SCs are self-executed contracts in which users can define their agreements and trust relationships, that are archived in a Blockchain. Smart Contracts can guarantee integrity of transactions supplying automatized transactions without the supervision of an external financial system (*e.g.*, banks, tribunals, or notaries). These transactions are traceable, transparent and irreversible.

SCs are software: like normal software, they may contain functional problems (bugs), that may also have security implications (vulnerabilities). SCs are generally much smaller than typical software system. Differently from other software products, however, SCs are immutable, *i.e.*, once they are deployed they can never be modified. This makes the early detection of bugs and vulnerabilities not only desirable, like in other software products, but a necessity. A bug or a vulnerability in an SC can have a huge impact.

[0]**Abbreviations:** ANA, anti-nuclear antibodies; APC, antigen-presenting cells; IRF, interferon regulatory factor

The most notorious example is the one that occurred in 2016 in the Ethereum blockchain, *i.e.*, the DAO attack. A vulnerability in DAO, a popular SC, was exploited to steal 3.6M ETHs (the equivalent of $70 million)[2]. Thanks to both industrial and academic research, some techniques and tools are now available for verifying and testing Smart Contracts, to reduce the likelihood that they contain bugs or vulnerabilities. A recent systematic literature review by Vacca *et al.*[3] surveyed 96 articles to provide an overview of the software engineering challenges in the development of SCs. However, such a review provides a broad overview of the problems, without focusing on specific aspects. Liu *et al.*[4], instead, provide a survey focused on a facet of SC verification, *i.e.*, the detection of security problems in SCs. To the best of our knowledge, no previous work generically surveys the verification techniques available for SCs.

In this paper, we fill this gap by conducting a systematic literature review focused on the approaches for automatically analyzing SCs to detect both bugs and security vulnerabilities. Our review is steered by the following core research questions:

- **RQ$_1$**: *What techniques are used to test Smart Contracts?*
- **RQ$_2$**: *What is the scope of the testing activity?*
- **RQ$_3$**: *Which technologies are mostly targeted by the available approaches?*
- **RQ$_4$**: *How replicable are the approaches and studies conducted?*

To answer our research questions, we first define a classification framework to guide our review. We identified four main *dimensions* to take into account, *i.e.*, Scope, Approach, Targeted Technologies, and Reproducibility, and the *attributes*, *i.e.*, the aspects to analyze and their possible values.

We chose four digital libraries (IEEE, ACM, ScienceDirect, and SpringerLink) to extract the relevant primary studies. We collected an initial set of 1,800 articles, from which we selected only 68 final primary studies that were relevant for our literature review, after having applied inclusion and exclusion criteria. We found that the state-of-the-art approaches are mostly based on static analysis, they mostly target security issues, even if some weakness are still ignored. The large majority of the approaches we analyzed targets Ethereum-based SCs. Besides, we found that the replicability of the studies is still relatively low both in terms of the programs implementing the approaches (not shared in about two-thirds of the papers) and in terms of data used for the validation (not public in about a half of the papers).

The remainder of this paper is structured as follows. Section 2 provides the necessary background on the Blockchain technology and on SCs. Section 3 explains the adopted methodology, while Section 4 presents the article selection process. Section 6 reports the results achieved according to the classification framework presented in Section 5. The threats that could affect the validity of our study are discussed in Section 7. Finally, Section 8 summarizes our study.

2 | BACKGROUND

In this section we report the precise definition of *blockchain* and *smart contracts*. Furthermore, since we do not only focus on functional problems but also on security vulnerabilities, we also discuss known weaknesses related to smart contracts.

2.1 | Blockchain Technology

A Blockchain is built on top of a distributed ledger that uses the mechanism of the consensus and chains of blocks for assuring the immutability of transactions. The blocks typically contain *transactions* between pairs of users. To avoid consistency problems since many users can write at the same time new blocks on the chain, consensus protocols were introduced to regulate such an operation. For example, Proof of Work is a commonly used consensus protocol that reduces the risk of inconsistencies by forcing the nodes to solve a cryptographic challenge (inverting a one-way hash)[1].

A blockchain can be *permissioned* (private) or *permissionless* (public). In a *permissioned blockchain* only authorized user can enter the network to read/write the blockchain. These authorizations can be provided by a company or a group of companies. Examples of permissioned blockchains are Everledger[5], Ripple[6], and Eris. In *permissionless blockchain*, instead, any anonymous user can join the network as a node. Among the examples of *permissionless blockchains*, there are Bitcoin and Ethereum[7,8].

With the first generation of blockchain, it was not possible to create complex distributed applications. Ethereum introduced the second generation of blockchains, which allow for the deployment of customized Smart Contracts.

2.2 | Smart Contracts

Szabo *et al.*[9] defined the idea of **Smart Contract** (SC) as a runnable part of blockchains to execute the terms of an agreement with ease and security between two or more parties without the guarantee of a third party. The necessary condition is the satisfaction of pre-defined rules. The popularity of Smart Contract begins only with the birth of blockchains. For example, one person can send currency units to another person, if the first received currency units from a third party. In this case, the transitive property is satisfied[8].

Mainly, Smart Contracts can be divided in two categories:

- *smart contract code*: the code is created to be executed from the blockchain. It respects two constraints: the programming language and the features of the blockchain;
- *smart legal contract*: these type of contracts are related to legal, political and business institutions. they can be seen as a particular application of *smart contract code* with a combination of traditional legal language.

An SC is composed of an account balance, a private database, and some executable code. Therefore, SCs have a global state on the blockchain. An SC can (i) read and write on its private database, (ii) store money on its account balance, (iii) send and receive messages or money from users/other contracts, or (iv) create new contracts.

Different Blockchain platforms allow to deploy and use SCs. Each platform provides different features and high-level programming languages for implementing SCs. Example of such Blockchain platforms are:

- **Bitcoin**[1]. Bitcoin has a limited computational capability. It is based on a scripting language: *i.e.*, stack-based bytecode scripting language. The writing of contracts is oriented to a simple logic with multiple signatures to confirm the payment. Unfortunately, SCs can not have a complex logic because of the limitations related to the scripting language[10]. For example, the scripting language does not allow loops[11].
- **Ethereum**[11,12]. Ethereum is a very popular common platform for developing smart contracts. This public platform allows developers to define advanced and fully customized SCs thanks to its Turing-complete programming language Solidity. As in Bitcoin scripting language, the platform supports loops. Also, in Solidity, the code is written in a stack-based bytecode language and executed in the Ethereum Virtual Machine (EVM).
- **NXT**[13]. NXT uses templates to develop built-in smart contracts[10]. Its programming language shares the same issues like the one used in Bitcoin (non-Turing-complete programming language).

2.3 | Known vulnerabilities in Smart Contracts

Software systems are often affected by security vulnerabilities. The vulnerabilities found in any software system are collected in the CVE (*Common Vulnerabilities and Exposures*) list. The NIST (*National Institute for Standards and Technology*) and MITRE are responsible for maintaining and updating such a list of known vulnerabilities. MITRE also curates the CWE registry[14] (*Common Weakness Enumeration*), which reports the known possible security problems that may affect a software system.

Like any other software product, SCs can be affected by vulnerabilities as well. The *Smart Contract Weakness Classification Registry* (SWC Registry)[15] is an implementation of the weakness classification scheme proposed in EIP-1470 for Ethereum Smart Contracts. The SWC registry matches with the Common Weakness Enumeration (CWE) for terminologies and structure. Such a registry assigns to each weakness a unique SWC ID in the format `SWC-XXX`. Each SWC entry has the following attributes: (i) *title*, (ii) *relationships* between the CWE Base or Class type and the CWE variant, (iii) *description* of the impact, (iv) *remediation*, and (v) *references* to additional information on the weakness.

This registry contains 36 weaknesses related to smart contract systems. We provide in Appendix B a summary for all the SWC entries available as of February 2021.

3 | PLANNING OF THE SYSTEMATIC LITERATURE REVIEW

The *goal* of this Systematic Literature Review (SLR) is to understand the verification activity performed on Smart Contracts. We followed the process guidelines proposed by Keele *et al.*[16] to conduct our literature review. As a first step, we *planned* our literature review. Specifically, we assessed the necessity of a literature review in this field by searching for other systematic reviews about the same topic. Then, we defined the research questions and we precisely defined the review procedure.

3.1 | Related Work

Initially, we performed a preliminary search to find other relevant literature reviews in this field. We found a large number of surveys and literature reviews on Blockchain and Smart Contracts (for example, Casino *et al.*[17], Conoscenti *et al.*[18], and Shen *et al.*[19]). Moreover, several surveys related to smart contract security and verification can be found in literature.

Chen *et al.*[20] performed a literature survey focused on security-related aspects of Ethereum blockchain. In detail, their work deals with vulnerabilities, attacks, and defenses, giving insights and future research directions. Differently from such a study, we specifically focus on the verification aspect, and we do not consider only security-related problems, but also functional ones.

Di Angelo *et al.*[21] performed a survey of 27 tools for analyzing Ethereum SCs, focusing on their availability, maturity level, methods employed, and detection of security issues. As a result, they give recommendations for the development of new tools and also highlight the tools that they found particularly inspiring. Similarly, Feng *et al.*[22] performed a survey on the SC vulnerabilities detection tools, and they give insights on the limitations and development challenges for vulnerability detection tools on the Ethereum blockchain. Our study is complementary to these ones because we consider also approaches not implemented as publicly available tools. However, we define a dimension of our classification framework dedicated to the availability of the proposed approaches as tools.

Murray *et al.*[23] performed a survey of formal verification methods for SCs. They found that theorem proving methodology seems to be the most successful, but more work on formal verification for SCs is needed. Differently from such a survey, we do not focus on a single technique (*i.e.*, formal verification), but we try to cover all the available ones. Also in this case, we define a dimension of our classification framework for annotating the type of technique used by each approach.

Xu *et al.*[24] proposed a survey on vulnerability detection tools for Ethereum and EOSIO smart contract bytecode. They conclude that tools focused on bytecode are more versatile than those for high-level languages. Also, they report that there is still work to do to detect all the known types of vulnerabilities, and there is a lack of automated tools for results validation. Differently from such a study, our review does not specifically focus on a single technology (*e.g.*, Ethereum). Instead, we use the target technology as a dimension of our classification framework.

Vacca *et al.*[3] reported a Systematic Literature Review about the development issues that can be found while writing SCs. Such a review has a larger scope compared to ours: While there is partial overlap, we try to focus on specific aspects related to the automated approaches for the verification (*e.g.*, the type of techniques used) to provide a more in-depth overview of the topic.

Alharby *et al.*[25] provide a Systematic Mapping of the articles in the literature that target SCs. Also in this case, the authors did not specifically focus on verification approaches for SCs, but on SCs in general. One of the main findings of such a study is that the majority of scientific papers in the literature about SCs aim at defining methodologies for finding issues in them. This further justifies the need for a more in-depth analysis of such papers, to better understand the specific characteristics of the state-of-the-art approaches.

The survey most similar to ours is the one by Liu *et al.*[4]. The authors surveyed articles on security verification of SCs, and they analyze a total of 53 research papers. The main difference is that they specifically focus on SC securty and they also include generic correctness verification techniques. Also, for the security-related approaches, we define a mapping with a reference weakness enumeration (SWC).

3.2 | Research Questions

The Research Questions (RQs) that guided our Systematic Literature Review are the following:

- **RQ$_1$**: *What methodologies are used to verify Smart Contracts?* With this RQ we aim to understand if there are unexplored methodologies and issues to verify Smart Contracts. In this way, researchers can decide to experiment with them.
- **RQ$_2$**: *Which technologies are targeted by the available approaches?* With this research question, we want to classify approaches based on programming language and on type of Blockchain to find areas not sufficiently treated yet. This will allow researchers to decide what programming languages and Blockchains they should target.
- **RQ$_3$**: *How reproducible are the approaches and studies conducted?* With this final research question, we aim to learn to what extent the approaches and the results obtained in the studies conducted can be reproduced both in terms of data (*i.e.*, are the datasets used in the studies available?) and source code (*i.e.*, are the prototype tools implementing the approaches available?). The results will guide researchers to perform new studies using existing data and tools.

3.3 | Review Procedure

To conduct our literature review, we defined a formal *review process* (Figure 1). As a first step, we selected the relevant primary studies. To do this, we first performed a *pilot study*: this allowed us to both (i) determine an appropriate search query, and (ii) define the inclusion and exclusion criteria for the articles. Then, after selecting four digital libraries, we used the defined query to extract the first set of primary studies. Then, we used a two-stage filtering procedure: we first only focused on title and abstract, and later we carefully read the whole papers to check if they met our inclusion/exclusion criteria. We report the results of such a step in Section 4.

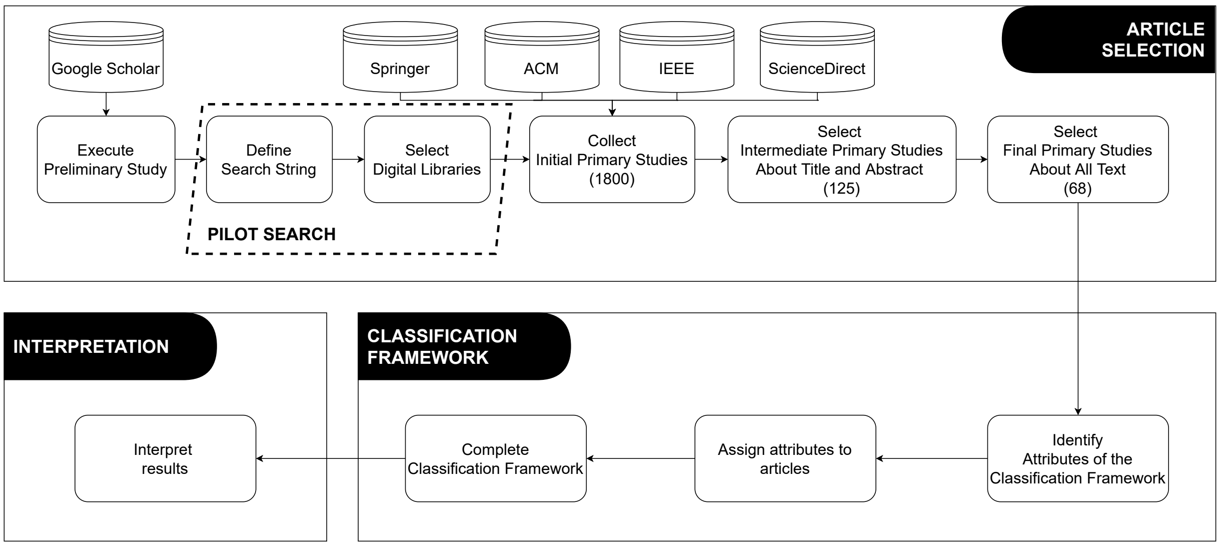

ARTICLE SELECTION
Google Scholar
Springer
ACM
IEEE
ScienceDirect
Execute Preliminary Study
Define Search String
Select Digital Libraries
PILOT SEARCH
Collect Initial Primary Studies (1800)
Select Intermediate Primary Studies About Title and Abstract (125)
Select Final Primary Studies About All Text (68)
INTERPRETATION
Interpret results
CLASSIFICATION FRAMEWORK
Complete Classification Framework
Assign attributes to articles
Identify Attributes of the Classification Framework

FIGURE 1 Systematic survey process

Then, we defined a *classification framework*: we based on the four research question to determine the *dimensions* of the framework (*i.e.*, which macro-aspects we wanted to take into account), and, for each of them, we defined specific *attributes* to determine for each paper. We explain such a procedure in Section 5. We used the classification framework to characterize each paper and to extract relevant information that allowed us to answer our research questions.

4 | ARTICLE SELECTION

As the first step in our Systematic Literature Review, we selected the primary studies to consider in it. To do this, we first determined the query that we could use to gather such a first set of possibly relevant studies. We tried to minimize the risk of (i) excluding relevant studies, and (ii) including irrelevant studies. Then, we used that query on several digital libraries to collect the first set of primary studies. We later refined the list of retrieved articles by using inclusion and exclusion criteria to select the final set of articles.

4.1 | Step 0: Query Definition

We first considered a single digital library, *i.e.*, Google Scholar, and we used a preliminary query, *i.e.*, "*testing of smart contract*". We chose Google Scholar at this phase since it indexes papers from specific digital libraries. As a result, we obtained numerous studies irrelevant for our SLR, including many studies regarding contextual applications of such technologies (*e.g.*, smart cities). This would have made the manual analysis infeasible. To narrow the search to our goal, we progressively refined such a query, using the results obtained as feedback. More specifically, we extracted from the relevant papers retrieved the key research terms that would allow us to precisely select that specific category of articles.

In this phase, we found that the term `test` and the phrase `"smart contract"` appeared in relevant articles. Using a query that combines both such parts, we still found a significant number of studies not relevant for our SLR. For example, the study by Yu *et al.*[26] proposes a parallelization technique for transitions, *i.e.*, not a technique for detecting problems in Smart Contracts. For this reason, we identified research terms that allowed us to further filter out such a kind of papers. We used terms commonly reflecting problems in terms of functional and non-functional (security-related) aspects, *i.e.*, `defect`, `bug`, `fault`, and `vulnerability`. To cope with corner cases, we used wildcards at the end of each term, to ensure that all the declensions of the words were considered, regardless of how the search engine was implemented. For the only phrase we included, *i.e.*, `"smart contract"`, we needed to specify also its plural version (`"smart contracts"`) since wildcards are generally not allowed in phrase queries.

As a result, we defined the following query:

```
test* AND ("smart contract" OR "smart contracts")
AND (defect* OR bug* OR vulnerabilit* OR fault*)
```

4.2 | Step 1: Initial Collection of Primary Studies

We choose an appropriate series of databases in order to increase the probability of finding highly relevant studies. We use the following digital libraries:

1. *ACM Digital Library*[27];
2. *IEEE eXplore*[28];
3. *ScienceDirect*[29];
4. *SpringerLink*[30];

For each digital library used, we executed the query reported in Section 4.1 and collected all articles published. We slightly adapted the query to each search engine since some of them required the use of a different syntax.

We obtained the following results: (i) 119 papers from IEEE; (ii) 245 papers from ACM; (iii) 534 papers from ScienceDirect; (iv) 902 papers from SpringerLink. In Figure 2 we show the distribution of primary studies found in the first step.

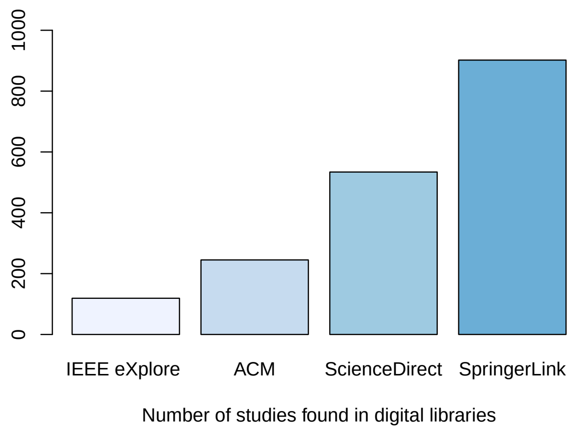


FIGURE 2 Distribution of primary studies found in the digital libraries

4.3 | Step 2: Study Selection

While the query we defined allowed us to considerably reduce the noise in the initial set of primary studies, some of them still were not entirely relevant. To exclude such studies, we manually analyzed the articles selected in the first step. Our selection was guided by the following inclusion and exclusion criteria:

- Include:
  - *IC1*: Papers defining an approach for detecting functional or security-related problems in SCs;
  - *IC2*: Papers written in English;
  - *IC3*: Peer-reviewed papers (*i.e.*, we exclude gray-literature).
- Exclude:
  - *EC1*: Surveys, literature reviews, books, book chapters, and magazine articles;
  - *EC2*: Papers which do not provide a validation;
  - *EC3*: Conference papers for which a journal extension was later published;

To apply such criteria, we performed two steps. First, we only analyzed the title and abstract of the papers: such a procedure allowed us to quickly exclude the papers that were clearly not relevant. Specifically, we could check the criteria *IC2*, *IC3*, *EC1*, and, partially, *IC1*. Then, to check the remaining criteria (*i.e.*, *IC1* and *EC2*), we read the complete remaining papers.

Given the 1,800 articles retrieved using the query previously defined on the digital libraries, we (i) selected only 125 studies after reading the title and abstract, and we (ii) selected 68 ones after carefully reading them.

4.4 | Editorial and Temporal Collocation

We provide below the editorial and temporal collocation of the primary studies selected for our SLR. We report in Table 1 the conferences and the journals in which the primary studies we selected appear, ordered by frequency.

The majority of the studies have been published at conferences (61 out of 68): indeed, 68 articles have been presented in 42 different conferences and only in 6 different journals. The most praised conferences (*i.e.*, 4 occurrences, reported in bold) in which are published studies on testing of Smart Contracts are: (i) *International Conference on Software Engineering* (ICSE) and *International Conference on Automated Software Engineering* (ASE). Instead, regarding journals, we found no relevant result given that we collect only 6 occurrences, where the assignment is one journal to one paper.

We report in Figure 3 the temporal collocation of the selected articles (*i.e.*, the number of articles published in each year, from 2015 to 2020). It is interesting to notice the step increase of the number of publications from 2018. Also, considering that we analyze only the first five months of 2020 it is interesting to see that there are already 10 articles in 2020 on the testing of Smart Contracts.

5 | CLASSIFICATION FRAMEWORK

To answer our research questions outlined in Section 3, we define a *classification framework* for the primary studies identified in Section 4. After having identified the dimensions, the attributes, and the values for our classification framework, we labelled each article according to it. While labelling, we annotated additional details for each non-boolean attribute that we could use to further improve the details of the classification, which could result in the introduction of new values for the attributes. We show in Figure 4 the final version of the classification framework, with *dimensions* (dark grey rectangles), *attributes* (white rectangles with solid borders), and *values* (rectangles with dashed borders), both the ones determined *a-priori* (white fill) and *a-posteriori* (light-blue fill). We first introduce the initial dimensions, attributes, and values of the framework (defined before starting the labeling process). Then, we describe the labeling process and the refinement of the classification framework.

5.1 | Identification of Dimensions, Attributes, and Values

We identified four *dimensions*, one for each research question: Approach (RQ$_1$), Scope (RQ$_2$), Targeted Technologies (RQ$_3$), and Reproducibility (RQ$_4$). For each dimension, we defined the *attributes* that we were interested in analyzing, based on our

TABLE 1 Editorial collocation of selected articles.

Type	Name	# selected articles
Conference	**International Conference on Automated Software Engineering (ASE)**	**4**
	International Conference on Software Engineering (ICSE)	**4**
	Conference on Computer and Communications Security (CCS)	3
	Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)	3
	International Conference on. Software Analysis, Evolution and Reengineering (SANER)	3
	Annual Computer Security Applications Conference (ACSAC)	2
	Asia-Pacific Software Engineering Conference (APSEC)	2
	International Workshop on Data Privacy Management (DPM)	2
	Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)	2
	Symposium On Applied Computing (SAC)	2
	International Symposium on Software Testing and Analysis (ISSTA)	2
	International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)	2
	Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)	1
	Annual International Conference on Computer Science and Software Engineering (CASCON)	1
	Annual International Conference on Privacy, Security, and Trust (PST)	1
	CCF China Blockchain Conference (CBCC)	1
	International Conference on Advanced Information Networking and Applications (AINA)	1
	International Conference on Blockchain (BLOCKCHAIN)	1
	International Conference on Certified Programs and Proofs (CPP)	1
	International Conference on Communication and Signal Processing (ICCSP)	1
	International Conference on Financial Cryptography and Data Security (FC)	1
	International Conference on Formal Engineering Methods (ICFEM)	1
	International Conference on Fundamental Approaches to Software Engineering (FASE)	1
	International Conference on Future Data and Security Engineering (FDSE)	1
	International Conference on Information Security Practice and Experience (ISPEC)	1
	International Conference on Internet of Things: Systems, Management and Security (IOTSMS)	1
	International Conference on Network and System Security (NSS)	1
	International Conference on Principles of Security and Trust (POST)	1
	International Conference on Program Comprehension (ICPC)	1
	International Conference on Runtime Verification (RV)	1
	International Conference on Smart Computing and Communication (SMARTCOM)	1
	International Conference on Software and Computer Applications (ICSCA)	1
	International Conference on Computational Science (ICCS)	1
	International Symposium on Formal Methods (FM)	1
	International Symposium on Leveraging Applications of Formal Methods (ISoLA)	1
	Network and Distributed Systems Security (NDSS)	1
	Symposium on Principles of Programming Languages (POPL)	1
	The Computer Security Foundations Symposium (CSF)	1
	The International Symposium on Software Reliability Engineering (ISSRE)	1
	Security Symposium (USENIX)	1
	Workshop on Programming Languages and Analysis for Security (PLAS)	1
	World Conference on Information Systems and Technologies (WORLDCIST)	1
Total publications		**61**
Journal	Journal Programming and Computer Software	1
	Programming Languages with Applications to Biology and Security	1
	Journal Blockchain Technology	1
	Journal Integrating Research and Practice in Software Engineering	1
	Journal of Network and Computer Applications	1
	IEEE Access	1
Total publications		**6**

research questions. Finally, for each attribute, we determined the possible values based on the possible values we expected to observe according to both the literature and our knowledge on the topic.

RQ$_1$: Approach. With the *Approach* dimension we aim at characterizing the approaches in terms of the basic techniques used for detecting issues in SCs. In addition, we want to characterize the approaches in terms of their scope, *i.e.*, in which contexts they can be used. The first obvious attribute we wanted to analyze was the base *methodology* used in the approaches defined to detect issues in SCs. Both static analysis (*e.g.*, symbolic execution) and dynamic analysis (*e.g.*, fuzzing) are typically used for detecting defects in traditional software systems. Subsequently, we want to understand the *properties* verified by the

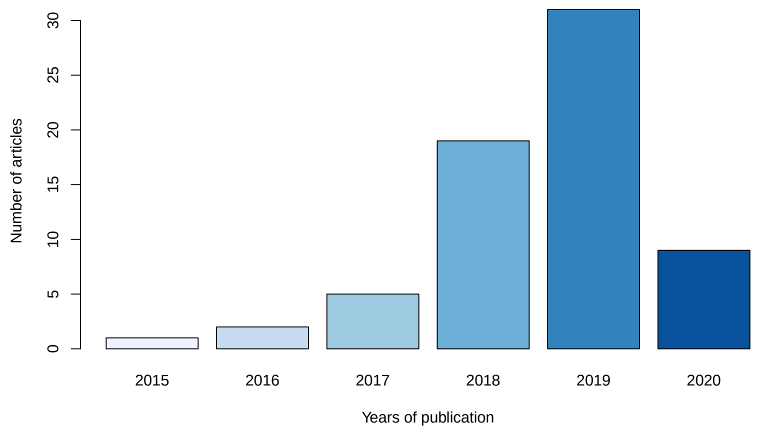


FIGURE 3 Distribution of research studies by years

approach. Since our review regards both normal bugs and vulnerabilities, we predetermined two possible values for such an attribute: *functional* for the former, and *security* for the latter. We also wanted to find specific issues with the related scopes that the approach could detect (*Issues*). While no specific taxonomy exists for SCs bugs, SWC provides a comprehensive list of possible weaknesses (see Section B) with their associated scopes. Therefore, the possible values we could assign to such an attribute is the complete list of scopes associated to each weakness, plus a new scope, *i.e.*, *Interaction beetween Smart Contracts*, dedicated to those vulnerabilities caused by an interaction between different smart contracts. To answer RQ_1, we also wanted to understand how many approaches used only a methodology and how many, instead, mixed two or more methodologies (*e.g.*, machine learning and dynamic analysis). Therefore, we also introduced the *Mixed* boolean attribute.

RQ_2: Targeted Technologies. With the *Targeted Technologies* dimension we aim at characterizing the approaches in terms of the technologies they were designed to work on. First, we wanted to understand the *Blockchain* targeted by the approach. To define the values for such an attribute, we considered the two most popular blockchain technologies supporting SCs at the time of the study, *i.e.*, Ethereum and Bitcoin. Also, in this case, we included *Others* as a possible value to include approaches defined for other blockchains. Each blockchain may support several high- or low-level programming languages to define SCs: we used the attribute *Programming Language* to identify the specific programming languages targeted by the approaches. To define the values, we took into account all the programming languages used to write smart contracts for the two blockchains we previously considered for the *Blockchain* attribute, *i.e.*, Solidity, YUL/YUL+, EVM Bytecode, Vyper (for Ethereum), and Bitcoin Scripting Language (for Bitcoin). Finally, as previously done for the *Approach* dimension, we checked if the approaches were able to target more than a programming language through the *Multilanguage* boolean attribute.

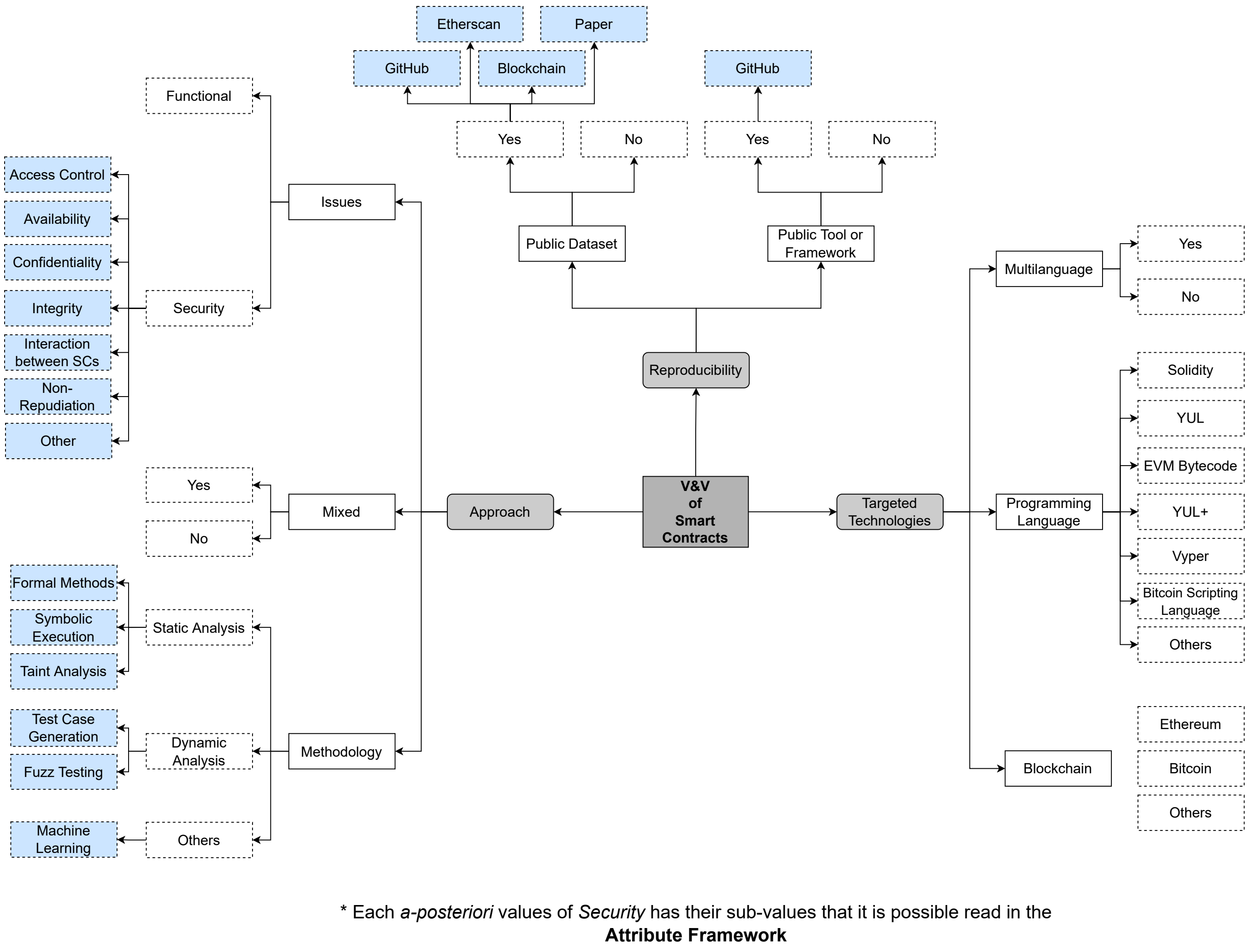
RQ_3: Reproducibility. With the fourth and final dimension, *Reproducibility*, we wanted to understand to what extent the approaches and the experiments presented in the primary studies we analyze are reproducible. We defined two boolean attributes: (i) *Public Dataset*, which allowed us to determine if the authors used a publicly accessible dataset of smart contracts for validating their approaches (including datasets released by the authors in the article), and (ii) *Public Tool or Framework*, for checking if the authors publicly released at least a prototype implementing their approach to allow future researchers to use it for comparison.

5.2 | Article Labeling and Framework Refinement

Three of the authors autonomously read and classified all the 68 primary studies identified by assigning one or more values for each attribute provided by the framework. The authors discussed the cases for which there was disagreement also with a fourth author to find consensus. While for most of the attributes we found that the pre-determined values were good enough to characterize the papers, we observed that these were insufficient for the following attributes:

- *Approach/Methodology*: some specific techniques recurred throughout several papers. For example, we found that *Fuzz Testing* was explicitly mentioned in seven articles. Therefore, we added to the framework the second level of possible values, *i.e.*, specific values for the three main values we initially determined. As for *Static Analysis*, we added *Formal*

FIGURE 4 The final version of the classification framework. The grey rectangles indicate the *dimensions*; the white ones with solid borders indicate the *attributes*. The rectangles with dashed borders indicate the *values*, with white fill indicating *a-priori* values, and the light-blue fill indicating *a-posteriori* values. Each a-posteriori value in "Security" has their own sub-values, available in Table B2, Table B3, and Table B4.



Methods, *Symbolic Execution*, and *Taint Analysis*; we added *Test Case Generation* and *Fuzz Testing* as sub-values of *Dynamic Analysis*; finally, we added a single sub-value for *Others*, *i.e.*, *Machine Learning*.

- *Approach/Issues*: We found that research articles were focused on two types of issues: *functional* and *security*. As for security, we add the second level of possible values, *i.e.*, the scope reached by each single SWC. Thus, we use the following values: *Access Control*, *Availability*, *Confidentiality*, *Integrity*, *Interaction between Smart Contracts*, *Non-Repudiation*, *Other*. We determined such scopes by analyzing the characteristics of the SWCs targeted in the research papers.

- *Reproducibility/Public Dataset*: we found that the papers that used publicly available dataset used five sources: *GitHub*, *Etherscan* (an analytics platform for the Ethereum blockchain), *Blockchain* (*i.e.*, they considered a subset of SCs deployed on a specific blockchain), and *Paper* (*i.e.*, they directly reported on the paper the SCs used for evaluation). Other papers reported a link to another kind of resource, *e.g.*, they self-hosted the dataset.

- *Reproducibility/Public Tool or Framework*: the authors that publicly released their approaches as tools or frameworks typically used *GitHub* for sharing the source code of their program; other papers reported a link to the program hosted through a different service (*e.g.*, private web page).

After this refinement of the taxonomy, two of the authors independently double-checked all the papers for such attributes to possibly label other papers according to this modified version of the framework. It is worth noting that, for the attributes that provide more levels of values (*i.e.*, *Approach/Methodology*, *Approach/Issues* and the two attributes of *Reproducibility*) we kept the more generic value we used in the first classification besides the specific ones we set in the second pass. For such attributes, we only kept the more generic values for the articles that used the specific non-recurring values we added while refining the classification framework.

6 | ANALYSIS OF THE RESULTS

In this section, we answer the research questions and summarize the literature on the V&V of Smart Contracts. We report in-depth results containing the full characterization of all 68 articles in Appendix A.

6.1 | RQ$_1$: What methodologies are used to verify Smart Contracts?

We analyze the results obtained for the **Approaches** dimension of the classification framework to characterize the methodologies used to face the challenges deriving from verificating the correctness of Smart Contracts. In addition, we analyze what are the problems faced in all studies targeting functional or security issues. Figure 5 shows the distribution of the methodologies used for each approach we analyzed. In Table 2 we report for each approach its respective reference. It can be noticed that *Static Analysis* is the most commonly used method and, more specifically, *Formal Methods* and *Symbolic Execution* are very common.

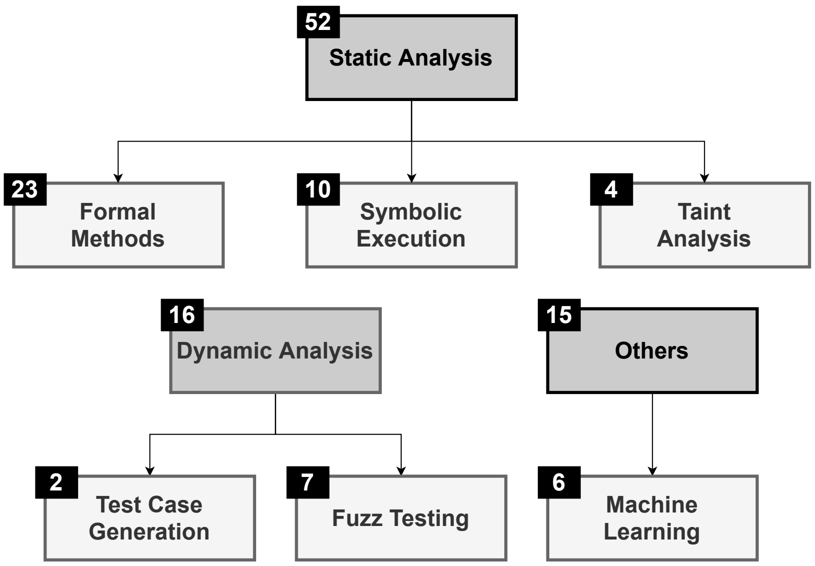


FIGURE 5 Distribution of the verification approaches for each defined classification category.

A lower number of approaches (16) use *Dynamic Analysis* techniques. In this case, *Fuzz Testing* is preferred over *Test Case Generation*. The former indicates the execution of smart contracts by using random and possibly unexpected values. In the context of Smart Contracts, a fuzzing approach identifies an issue when the violation of the predefined test oracles is detected, usually checking the result of transactions, log messages or blockchain accounts (*i.e.*, wallet amount). The latter indicates the generation of test input data following the SC specification. *Machine Learning* is still not very common (6 articles), but it is gaining popularity: all the articles that used ML-based approaches were published after 2019, with the single exception of Liu

TABLE 2 Reference of verification approaches.

Approach	Reference
Static Analysis	Whichmann *et al.*[31]
Formal Methods	Gerhart *et al.*[32]
Symbolic Execution	King *et al.*[33]
Taint Analysis	Boxler *et al.*[34]
Dynamic Analysis	Ball[35]
Test Case Generation	Myers[36]
Fuzz Testing	Gallagher *et al.*[37]
Others/Machine learning	Mitchell[38]

et al.[39]. We found that 79.1% of selected studies use only one methodology. Instead, 14 (20.9%) research articles use more than a methodology: 12 of them use only two methodologies, and 2 of them use a combination of three methodologies.

Static Analysis. Most of the approaches use formal methods, symbolic execution, or taint analysis. For example, Tsankov *et al.*[40] proposed SECURIFY, an approach that uses program analysis to prove the safety of Ethereum Smart Contract behaviour, with respect to a given property. SECURIFY performs a two-step analysis, where at first it extracts semantic information from the contract's dependency graph, then it checks compliance and security patterns over such semantic information. One of these studies is the one by Mossberg *et al.*[41] proposed Manticore, a symbolic execution framework for analyzing binaries and Smart Contract, exploring the state space of the targeted Smart Contracts. Gao *et al.*[42] proposed EasyFlow, a taint analysis based technique to detect overflow vulnerabilities at the EVM level. Their approach can detect the overflow level of the targeted Smart Contracts (*i.e.*, safe, potential overflow, manifested overflow, etc.) but also it can generate transactions to trigger potential overflows. Moreover, we found 15 out of the 52 research papers using static analysis approaches that did not match any of such categories of techniques. For example, the approach proposed by Lai *et al.*[43] detects integer overflow in SCs by using a set of custom XPath patterns defined on the basis of 11 different scenarios of integer overflow. Another example is the approach proposed by Grech *et al.*[44], which can find gas-focused vulnerabilities in Ethereum Smart Contracts. Their approach performs a static analysis at EVM bytecode level, decompiling and then reconstructing the program's higher-level semantics using abstract interpretation techniques and program analysis. Thus, 52 research papers are the sum of 15 research studies that did match any of such categories of techniques, 23 research studies that match with formal methods, 10 research studies match with symbolic execution and 4 research studies match with taint analysis.

Dynamic Analysis. In this category, we have approaches based on *Fuzz Testing* and *Test Case Generation*. These tools simulate the execution of a large number of transactions to find vulnerabilities in contracts. For example, ReGuard[45] uses fuzzing to automatically detect reentrancy bugs in Ethereum Smart Contracts. This is achieved by generating and executing random diverse transactions. Zhang *et al.*[46] proposed EthPloit, a test case generation tool focused on security exploits generation based on fuzzing, combined with static taint analysis. Specifically, EthPloit starts with static analysis to extract the ABI from byte code. Based on this, there is a test case generation phase, where also fuzzing is applied to optimize the generated test case. In the next steps, the test cases are selected (*i.e.*, successful exploits) and the reward function for the test case generator is adjusted. Besides such specific techniques, we found that 7 articles out of 16 define approaches based on Dynamic Analysis used techniques different from *Test Case Generation* and *Fuzz Testing*. For example, Chen *et al.*[47] proposed an approach to dynamically adjusts the gas costs of EVM operations according to the number of executions to avoid DoS attacks: indeed, such an approach aims at avoiding that failing to properly set the gas costs on Ethereum allows attackers to launch DoS attacks. Moreover, Nguyen *et al.*[48] proposed some behaviour-based methods to detect attack vectors on Ethereum Smart Contracts, such as vulnerabilities in the Solidity level. They implemented and then tested the ABBE tool by using a test suite composed of Smart Contracts containing vulnerabilities. As a result, their tool not only detects known attacks but also zero-day attack patterns.

Others. Most of the articles marked as *Others* (6 out of 15) use approaches mainly based on machine learning. For example, Momeni *et al.*[49] proposed a machine learning model that detects security vulnerabilities patterns in Smart Contracts. They used two static code analyzers to build a training dataset composed of 1,000 Smart Contracts verified on the Ethereum platform. Then, they trained an array of machine learning models such as Decision Trees, Random Forest, Neural Network, and SVM for different security vulnerabilities. As a result, their approach is able to find 16 different vulnerabilities with an average accuracy of 95%. On the other hand, some research paper belonging to *Others* category introduce techniques for mutation testing, *i.e.*, for measuring the quality of the test suite of the SCs. While such approaches do not directly check if a given SC contains bugs or vulnerabilities, they indirectly achieve this goal by finding blind spots for the test suite. Improving the test suite, in turn, possibly allows finding more issues in the SCs. For example, Chapman *et al.*[50] proposed Deviant, a mutation testing tool for Solidity. Deviant generates and runs mutants given a Solidity project to evaluate the effectiveness of the test suite according to the Solidity fault model. Li *et al.*[51] introduce MuSC, a mutation testing tool for Ethereum Smart Contracts implementing a set of novel mutation operators for the Solidity programming language. Another example of an approach labelled as *Other* is the work by Wang *et al.*[52]. The article introduces a high-availability and unified input Filter-based Secure Framework for Ethereum smart Contract (FSFC), designed to dynamically identify and discard bad inputs before they getting processed. In this way, the availability of the deployed smart contracts is not impacted avoiding suspending the contract service. Besides, Wang *et al.*[53] proposed a technique focused on transaction-related vulnerabilities. Their approach, VULTRON, can detect irregular transactions due to various types of adversarial exploits, building an oracle that can effectively distinguish transactions that are the results of malicious exploits from normal ones.

Mixed approaches. 14 out of 68 articles presented approaches that combine two or three categories of techniques to detect issues. The approach presented by He *et al.*[54] is a clear example of mixed approach: it combines *Symbolic Execution* (*i.e.*, Static Analysis), *Fuzz Testing*, and *Machine Learning*. Their approach combines a neural network with symbolic execution to define a technique able to automatically define fuzzing policies. Through such policies, a fuzzer generates inputs for unseen Ethereum SCs.

Issues. We found that 43 studies out of 68 studies (64%) are focused on security, while only 24 studies (36%) target functional issues. Such a result was expected since vulnerabilities in SCs may have a relevant economic impact. The most analyzed scope is *Availability*, which has been studied in 37 research studies (55.2%) and it is associated to 12 SWCs (*i.e.*, SWC-128). In addition, 5 studies have been studied this scope without consider other scopes[55,56,57,47,58]. Another relevant scope is *Confidentiality*. Indeed, this scope has been studied in 30 papers (44.8%) and it is associated to 12 SWCs. Issues related to *Access Control* were considered in 20 research studies (*i.e.*, 29.8% of the total). The categories *Integrity* and *Non-Repudiation* are covered only by one article each (*i.e.*, the one by Kolluri *et al.*[59] and the one by Liu *et al.*[45], respectively). Finally, as for the scope we introduced, *i.e.*, *Interaction between Smart Contracts*, we found that it is quite widespread: 17 research studies (25.4%) targeted such an issue. We report in Figure 6 the distribution of the specific security vulnerabilities in research papers. The most studied weaknesses, *i.e.*, **SWC-101** (*Integer Overflow and Underflow*) and **SWC-107** (*Reentrancy*), have been addressed by the approaches defined in 19 articles. Instead, we found that 8 vulnerabilities located in the SWC Registry have not been explored by any study: such weaknesses are **SWC-103** (*Floating Pragma*), **SWC-117** (*Signature Malleability*), **SWC-118** (*Incorrect Constructor Name*), **SWC-121** (*Missing Protection against Signature Replay Attacks*), **SWC-122** (*Lack of Proper Signature Verification*), **SWC-127** (*Arbitrary Jump with Function Type Variable*), **SWC-130** (*Right-To-Left-Override control character (U+202E))*, and **SWC-131** (*Presence of unused variables*). This shows a gap in the literature concerning a relatively large portion of known vulnerabilities.

Vulnerabilities Missing from SWC. Reading and analyzing the research articles involved in the study, we found that some of the articles mention and address vulnerabilities missing from the SWC registry. Specifically, we found three vulnerabilities of such a kind. For convenience, we assign an unofficial incremental SWC-ID (i.e. SWC-Xi) to such new vulnerabilities. The first vulnerability we found is *External call to untrusted contract*, identified as **SWC-X1**. This vulnerability was studied by Momeni *et al.*[49]. Such a weakness manifests when a SC delegates some of its functionalities to an external contract that is inaccessible. Indeed, the implementation of such an external SC can have hidden malicious behaviours. While we found no SWC weakness related to such a vulnerability, we found other traces of such a weakness, beyond the previously mentioned article. A repository containing a catalogue of Smart Contracts formally verified by Runtime Verification and/or collaborators provides a comprehensive list of vulnerabilities, alternative to SWC, which includes *External Contract Referencing*[60], which is conceptually related to our **SWC-X1**. Another vulnerability we found is *Byte-manipulation*, identified as **SWC-X2**. Park *et al.*[61] proposed an approach for fixing such a vulnerability. On the Decentralized Application Security Project (DASP) Top 10, this vulnerability could be classified in the family of DoS attacks[62] . This vulnerability is due to the fact that the byte-wise splitting and merging operations cannot avoid performance penalties: for example, a large formula should be executed for every load into memory. The byte-wise splitting operation is used to operate on the 32-byte word to split it in 32 chunks of bytes (*i.e.*, returned from the 32-byte word). In this way, chunks can be stored in the local storage, which is a byte-addressable memory and can be represented as an array of bytes. Instead, the byte-wise merging operation is used to operate on 32 chunks of bytes to transform them into 32-byte words and store them in the local stack or in the global storage. These two types of memory are word-addressable, *i.e.*, they are based on an array of 32-byte words. The last vulnerability we found is the *Short Address Attack*,

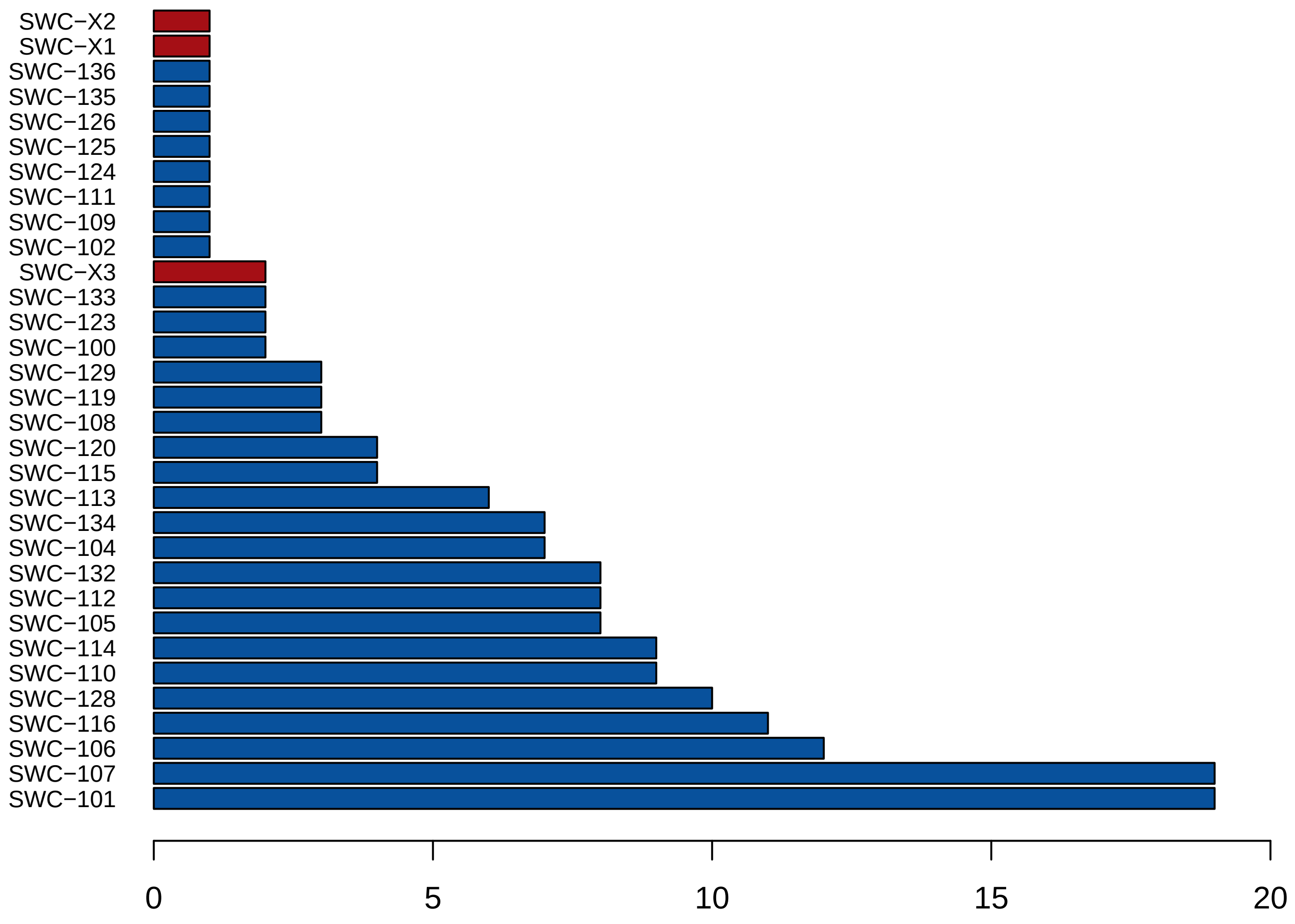


FIGURE 6 Number of papers (x-axis) targeting security vulnerabilities (y-axis).

identified as **SWC-X3**. Ashouri *et al.*[63] and Liao *et al.*[64] introduced approaches for detecting such a weakness in SCs. **SWC-X3** is discussed in the *Decentralized Application Security Project (DASP) Top 10 of 2018*[65] and thus can be a good candidate to become a new element of the SWC Registry. Some approaches (*e.g.*, the one proposed by Honig *et al.*[66]) base their approach on such a registry. Therefore, the fact that such vulnerabilities are missing from the SWC registry is a relevant issue.

Approaches Targeting Specific Vulnerabilities. There are two main methodologies used for detecting vulnerabilities in Smart Contracts: the majority of the approaches focus on *revealing* the presence of vulnerabilities, thus allowing developers to remove them, or on *proving* their absence (*e.g.*, KEVM[57], Securify[40], Zeus[67]); other approaches in this research area are able to automatically build exploits (*e.g.*, MAIAN[68], Manticore[41] and teEther[69]). Other approaches use alternative methodologies. ContractLarva[70] checks the execution of Smart Contracts, inserting some code inside the already written code to verify their behaviour. About a half of the approaches start the analysis with the bytecode of contracts. This is due to the lack of formal semantic in Solidity, but also from the changes in the behaviour according to the different versions of the compiler. For what concerns the used methods to test these Smart Contracts, the biggest part of the approaches used until now applies largely dynamic and static analysis, *e.g.*, SolAnalyser[71]. OYENTE[72] takes into account three particular vulnerabilities, *i.e.*, transaction-ordering dependent (TOD), Timestamp, and Reentrancy. Reentrancy vulnerability is one of the most dangerous and is the one that caused several attacks with the loss of various millions of dollars. Other approaches, like Manticore[41], SECURIFY[40], SmartCheck[73], try to catch a larger set of vulnerabilities. For what concerns formal verification, KEVM[57] is clearly the favourite for its maturity, with the inconvenience that its use requires expertise in the bytecode of EVM. SECURIFY[40] is the most advanced approach able to check formal guaranties. Osiris[74] is specialized in integer bugs, while MAIAN[68] analyzes three specific kinds of vulnerability, *e.g.*, the self-destruction of Smart Contracts. MAIAN is composed of two major components: symbolic analysis and concrete validation. In detail, MAIAN performs the symbolic analysis on the smart execution traces of smart contracts to find out trace vulnerabilities. Moreover, a concrete validation runtime is called to reduce the number of false positives. Concrete validation is used to execute the contract with the concrete values of the transactions returned by the symbolic analysis to verify if the property is verified in the concrete execution. If the concrete execution fails to return a violation of the trace property, the contract is marked as false positive, and true positive, otherwise.

Approaches Targeting Generic Vulnerabilities. It is interesting that, in two particular studies, the authors deal with security aspects without delving into specific vulnerabilities. For example, Liu *et al.*[39] defined a semantic-aware security auditing technique. This technique can be called S-GRAM for Ethereum because authors combine an N-gram language model and a lightweight static semantic labelling. In this way, potential vulnerabilities can be predicted by detecting irregular token sequences. Their results show that the S-GRAM is able to identify security issues. Zupan *et al.*[75] proposed a method and a prototype tool to generate secure Smart Contracts based on Petri Nets. Petri Nets is a visualizer of models for processes and workflows. The goal of their study is to create a secure smart contract template for easy deployment of a supported blockchain platform. Their main contribution is the creation of Smart Contracts via Petri Nets so that developers minimize logical errors.

Summary of RQ$_1$. Most of the approaches for verifying Smart Contracts are based on static analysis and, specifically, *Formal Methods* and *Symbolic execution* are the most frequently used techniques. Machine learning is used in only 6 articles, but all of them are recent, suggesting a growing interest in the field. Most of the articles we analyzed aim at verifying the security of SCs. Both research and practice in this field are not mature yet: we found that 8 weaknesses from the SWC registry are not addressed by any state-of-the-art approach, while 3 vulnerabilities addressed by some approaches are not included in SWC.

6.2 | RQ$_2$: Which technologies are targeted by the available approaches?

We answer this research question by analyzing the labelling results related to the **Targeted Technologies** dimension of our classification framework. We analyze all selected research papers to see what is the type of Blockchain used in every single study. We find that almost all the studies (62, *i.e.*, 92.5%) target SCs deployed on the Ethereum Blockchain. This is not surprising: previous studies (*e.g.*, Alharby *et al.*[25]) already reported that Ethereum is the most commonly used platform for deploying Smart Contracts. Therefore, it is natural that most of the approaches from the literature are meant to work on such a platform. We found only 4 research articles (5.9%) that target the Bitcoin Blockchain. Finally, we found only 3 articles (*i.e.*, 4.5%) that target different types of blockchain. Specifically, Wang *et al.*[76] define an approach for verifying SCs deployed on the Azure Blockchain, while Sergey *et al.*[55] target Zilliqa. Finally, ZEUS[67] targets both the Hyperledger Fabric and the Ethereum Blockchain. Figure 7 shows the distribution of Blockchain types.

We report in Figure 8 the labeling results for the targeted *programming language*. Solidity (47 articles, 70.1%) is the most targeted programming languages (*e.g.*, by SolAnalyser[71], Grossman *et al.*[77], and Zakrzewsky *et al.*[78]). Again, this is not surprising, since Solidity is the most widely used high-level programming language for writing Smart Contracts for Ethereum. In addition, we found particular attention in the literature to EVM Bytecode, with 23 (34.3%,) involved articles, *e.g.*, Bai *et al.*[79], Ma *et al.*[80] and Zhang *et al.*[81]. EVM Bytecode is the low-level language in which Ethereum SCs must be compiled to be executed; therefore, it is natural that some approaches specifically target such a language instead of other high-level languages. We found 6 studies (8.9%) that use both Solidity and EVM Bytecode[82,45,72,55,40,53]. Other programming languages are rarely targeted by the state-of-the-art approaches: 3 studies (*i.e.*, 4.5%) targeted Vyper[83,84,44], but they also target Solidity as well; 3 studies target the Bitcoin Scripting Language[85,86,87]. Only the approach proposed by Jiao *et al.*[84] uses a programming language different from the ones provided in our framework. Such a programming language is Bamboo[88], a language less popular than the alternatives for Ethereum SCs. Such a language is not mentioned in the Ethereum official documentation. The large majority of the papers we analyzed (58, *i.e.*, 86.6%) target a single programming language, while 9 of them (13.4%) target more than a programming language.

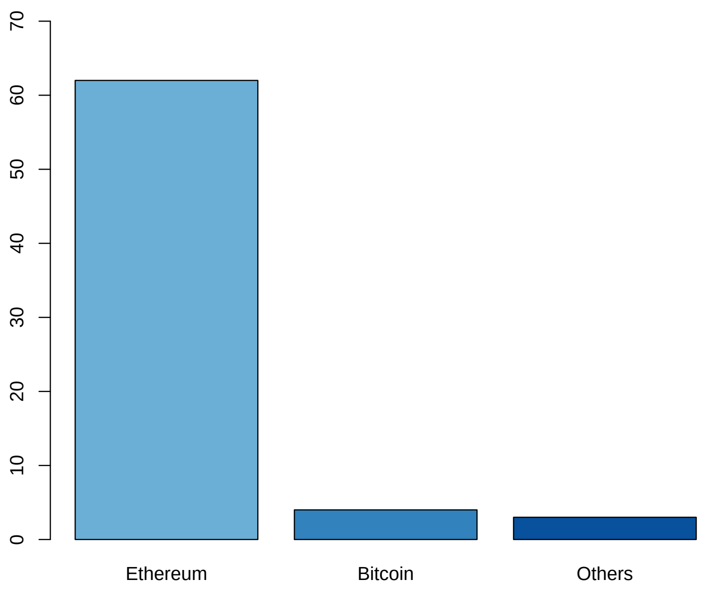


FIGURE 7 Distribution of Blockchain types.

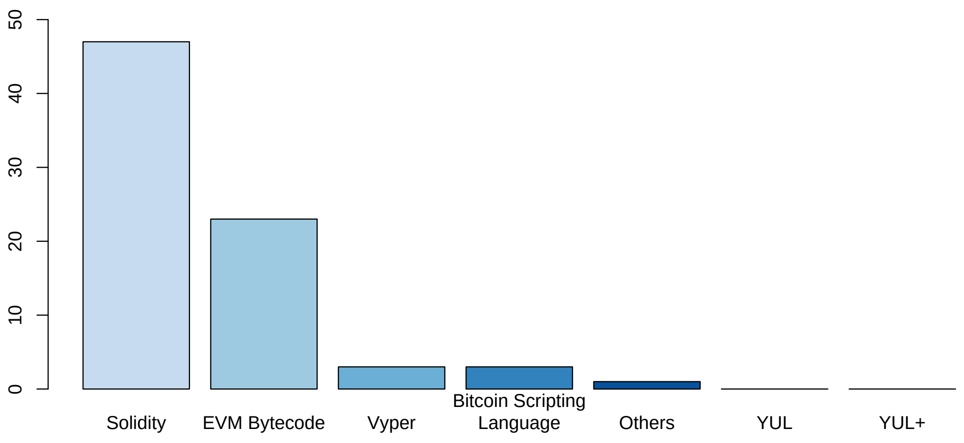


FIGURE 8 Distribution of the use of programming languages.

To provide a comprehensive view of the relationship between vulnerabilities and programming languages, we report the categorization provided through our classification framework in terms of such aspects in Table 3.

Summary of RQ_2. The large majority of articles target SCs developed for Ethereum and, more specifically, the ones written in Solidity. There is a need for new approaches for other commonly used Blockchains, such as Bitcoin and Hyperledger Fabric.

TABLE 3 Coverage with respect to the vulnerabilities observed and the different programming languages.

Dataset	Vulnerabilities																																								Programming Languages						
	Functional	SWC-100	SWC-101	SWC-102	SWC-103	SWC-104	SWC-105	SWC-106	SWC-107	SWC-108	SWC-109	SWC-110	SWC-111	SWC-112	SWC-113	SWC-114	SWC-115	SWC-116	SWC-117	SWC-118	SWC-119	SWC-120	SWC-121	SWC-122	SWC-123	SWC-124	SWC-125	SWC-126	SWC-127	SWC-128	SWC-129	SWC-130	SWC-131	SWC-132	SWC-133	SWC-134	SWC-135	SWC-136	SWC-X1	SWC-X2	SWC-X3	Solidity	EVM Bytecode	Vyper	YUL	YUL+	Bitcoin Scripting Language	Others
Akca *et al.*[71]			✓												✓		✓	✓												✓	✓					✓						✓						
Albert *et al.*[82]	✓																																									✓	✓					
Ashouri *et al.*[63]			✓						✓			✓				✓					✓	✓								✓				✓									✓					
Azzopardi *et al.*[70]	✓																																								✓							
Bhargavan *et al.*[89]						✓			✓																											✓							✓					
Chang *et al.*[90]	✓																																									✓						
Chapman *et al.*[50]	✓																																									✓						
Chatterjee *et al.*[83]	✓																																									✓		✓				
Chen *et al.*[47]																														✓												✓						
Dong *et al.*[91]									✓																																	✓						
El-Dosuky *et al.*[92]	✓																																									✓						
Feist *et al.*[93]							✓	✓	✓		✓										✓													✓								✓						
Fu *et al.*[94]			✓			✓		✓	✓			✓		✓		✓						✓			✓																	✓						
Gao *et al.*[42]			✓																																								✓					
Grech *et al.*[44]			✓																									✓		✓						✓						✓		✓				
Grossman *et al.*[77]	✓																																									✓						
He *et al.*[54]							✓	✓				✓		✓				✓																✓								✓						
Hirai *et al.*[56]								✓	✓																					✓													✓					
Honig *et al.*[66]	✓																																									✓						
Jiang *et al.*[95]							✓		✓			✓		✓				✓																		✓						✓						
Jiao *et al.*[84]	✓																																									✓		✓				✓
Kalra *et al.*[67]			✓			✓			✓						✓	✓		✓																								✓						
Klomp *et al.*[85]																																															✓	
Kolluri *et al.*[59]																											✓																✓					
Krupp *et al.*[69]							✓	✓						✓																																		
Lahbib *et al.*[96]	✓																																										✓					
Lai *et al.*[43]			✓																																							✓						
Li *et al.*[51]	✓																																									✓						
Liao *et al.*[64]		✓	✓			✓	✓	✓	✓	✓					✓	✓		✓				✓								✓											✓	✓						
Liu *et al.*[45]									✓																																	✓	✓					
Liu *et al.*[39]	✓																																									✓						
Luu *et al.*[72]									✓			✓				✓		✓																								✓	✓					
Ma *et al.*[80]	✓																																										✓					
Medeiros *et al.*[97]	✓																																									✓						
Momeni *et al.*[49]			✓			✓	✓	✓				✓		✓		✓	✓					✓																✓	✓			✓						
Mossberg *et al.*[41]	✓																																									✓						
Nguyen *et al.*[48]			✓					✓	✓					✓							✓															✓						✓						
Nikolic *et al.*[68]						✓		✓																										✓									✓					
Park *et al.*[61]			✓																																✓					✓			✓					
Peng *et al.*[98]			✓																												✓											✓						
Sergey *et al.*[55]																														✓												✓	✓					
Shishkin *et al.*[99]	✓																																									✓						
Song *et al.*[100]			✓						✓							✓		✓																								✓						
Tian *et al.*[101]									✓									✓																									✓					
Tikhomirov *et al.*[73]		✓		✓		✓	✓		✓	✓			✓	✓	✓		✓	✓												✓	✓			✓		✓						✓						
Torres *et al.*[74]			✓																															✓									✓					
Tsankov *et al.*[40]												✓		✓		✓									✓	✓								✓								✓	✓					
Wang *et al.*[102]									✓						✓	✓		✓																								✓						
Wang *et al.*[76]	✓																																									✓						
Wang *et al.*[52]			✓																																							✓						
Ye *et al.*[103]								✓	✓								✓	✓																									✓					
Zhang *et al.*[104]							✓	✓																										✓				✓				✓						
Zhang *et al.*[105]								✓	✓																																	✓						
Zhang *et al.*[81]			✓							✓					✓																												✓					

6.3 | RQ$_3$: How reproducible are the approaches and studies conducted?

We report the results of our labelling in terms of **Reproducibility** for the articles we analyzed. As for the *Public Dataset*, we observed that about half of the papers (31, *i.e.*, 46.3%) provide or use a publicly available dataset that allows independent replications of the results. Most of the studies that share a dataset (16, *i.e.*, 23.9%), use GitHub (*e.g.*, the studies by Peng *et al.*[98], Wang *et al.*[76], and Medeiros *et al.*[97]). Other studies (6, 9.0%) that provide the dataset through Etherscan with the date of access to the site (*e.g.*, SMARTSHIELD[81], the study by Chang *et al.*[90], and the study by Kolluri *et al.*[59]). Other 4 (6.0%) studies provide their datasets through different platforms (*e.g.*, Soliaudit[64] use Google Drive). Finally, 3 (*i.e.*, 4.5%) research studies directly provide the Smart Contracts used in their studies directly in the paper[91,47,85]. We report in Table 4 the size and the year of introduction of all the datasets available in the literature.

In terms of *Public Tool or Framework*, we found that only 25 (*i.e.*, 37.3%) research papers publicly share a prototype tool implementing the approach presented in the article. 21 out of 25 studies provide their tool through a GitHub repository[58,66,59]. 4 studies release their tool using an external provider (*e.g.*, self-hosted webpage), *i.e.*, Albert *et al.*[82], Klomp *et al.*[85], Luu *et al.*[72] and Securify[40]. Only one paper[98] provides its tool through both a GitHub Repository and another platform (Wandbox). We found two cases in which the tools were not publicly released, but for which a tool demo was provided as a YouTube video[1]. We report in Table 5 the links to all the tools we found in the articles we reviewed.

To provide a broader view of the experimental context of the studies, we annotated, for each paper, the approaches that the authors used as baselines to evaluate their approach. We show in Figure 9 the number of papers that used each baseline (we report

[1]The approach by Liu *et al.*[45] is available at https://youtu.be/XxJ3_-cmUiY, while the approach by Park *et al.*[61] is available at https://youtu.be/4XBcAclq0Vk

TABLE 4 Datasets of SCs in the literature.

Dataset	Size	Year
Akca *et al.*[71]	1,838 Smart Contracts	2019
Albert *et al.*[82]	24,000 Smart Contracts	2019
Ashouri *et al.*[63]	12 real-world and educational Smart Contracts	2020
Azzopardi *et al.*[70]	The ERC-20 token standard	2018
Bhargavan *et al.*[89]	112,802 Smart Contracts	2016
Chang *et al.*[90]	36,099 Smart Contracts	2019
Chapman *et al.*[50]	3 programs	2019
Chatterjee *et al.*[83]	36,764 real-world Ethereum Smart Contracts	2019
Chen *et al.*[47]	16,000 execution traces	2017
Dong *et al.*[91]	2 Smart Contracts	2019
El-Dosuky *et al.*[92]	Crawler per Smart Contracts	2019
Feist *et al.*[93]	36,000 Smart Contracts	2019
Fu *et al.*[94]	More than 10,000 Smart Contracts	2019
Gao *et al.*[42]	Repository software	2019
Grech *et al.*[44]	All programs available on the Ethereum Blockchain on April 9, 2018	2018
Grossman *et al.*[77]	3,444,354 blocks of the main Ethereum Blockchain	2017
He *et al.*[54]	18,498 Smart Contracts	2019
Hirai *et al.*[56]	Test suite	2017
Honig *et al.*[66]	2 Smart Contract projects	2019
Jiang *et al.*[95]	6,991 Smart Contracts	2018
Jiao *et al.*[84]	482 test	2020
Kalra *et al.*[67]	22,493 Smart Contracts	2018
Klomp *et al.*[85]	2 example scripts	2018
Kolluri *et al.*[59]	5,000 Smart Contracts	2019
Krupp *et al.*[69]	38,757 Smart Contracts	2018
Lahbib *et al.*[96]	1 smart contract named Access Control (AC) developed in the OrBAC access control model	2020
Lai *et al.*[43]	7,000 Smart Contracts	2020
Li *et al.*[51]	Truffle project with related contracts and test suite	2019
Liao *et al.*[64]	17,979 samples	2019
Liu *et al.*[45]	5 modified Smart Contracts	2018
Liu *et al.*[39]	For training set 43,553 deployed Smart Contracts and for test set 1,500 Smart Contracts	2018
Luu *et al.*[72]	19,366 Smart Contracts	2016
Ma *et al.*[80]	10 real-world Smart Contracts	2019
Medeiros *et al.*[97]	5 projects	2019
Momeni *et al.*[49]	13,745 Smart Contracts	2019
Mossberg *et al.*[41]	100 Ethereum Smart Contracts	2019
Nguyen *et al.*[48]	Test suite	2019
Nikolic *et al.*[68]	970,898 Smart Contracts	2018
Park *et al.*[61]	3 projects	2018
Peng *et al.*[98]	1,838 real Smart Contracts	2019
Sergey *et al.*[55]	4 kind of contracts used on Ethereum: ERC20, ERC721, auction and crowdfunding	2019
Shishkin *et al.*[99]	MiniDAO smart contracts autoprodotto	2018
Song *et al.*[100]	49,502 Smart Contracts	2019
Tian *et al.*[101]	20 Smart Contracts	2019
Tikhomirov *et al.*[73]	3 Smart Contracts	2018
Torres *et al.*[74]	1.2 million of Smart Contracts	2018
Tsankov *et al.*[40]	1 dataset with 24,594 Smart Contracts and 1 dataset of 100 Smart Contracts written in Solidity	2018
Wang *et al.*[102]	31,097 Smart Contracts	2019
Wang *et al.*[76]	All sample smart contracts that are shipped with Workbench + application policies on the Github repository of Azure Blockchain	2019
Wang *et al.*[52]	2 private chaing and several real-world Smart Contracts	2020
Ye *et al.*[103]	31,097 Smart Contracts in 561 files	2020
Zhang *et al.*[104]	49,522 Smart Contracts	2020
Zhang *et al.*[105]	100 randomly chosen real world Smart Contracts	2019
Zhang *et al.*[81]	2,214,409 real-world Smart Contracts	2020

only the baseline approaches used by at least two studies). In this figure, we only report studies that, in their evaluation, compare the technique introduced in the paper to a baseline. All the approaches commonly used as baselines are publicly available as tools on GitHub, except for ZEUS[67] (used by 5 papers), ReGuard[45], and KEVM[57] (both used by 2 papers).

OYENTE[72] is the most used baseline. This is probably due to the fact that such an approach was one of the first in the field and it has served as the starting point for several other projects, also because it is an open-source tool. OYENTE is basically a symbolic execution framework to help developers to find potential security bugs of the Ethereum Smart Contracts and works on the EVM. It consists of four modules, namely CFGBuilder, Explorer, CoreAnalysis, and Validator. The CFGBuilder constructs a Control Flow Graph of the contract, the contracts' CFG is provided to the Explorer, which symbolically executes the contract.

TABLE 5 Publicly available frameworks and tools for finding issues in Smart Contracts.

Type	Framework	Link
GitHub	Amani *et al.*[58]	https://git.io/JJxne
	Azzopardi *et al.*[70]	https://git.io/JJxZ6
	Feist *et al.*[93]	https://git.io/JJxn7
	Grech *et al.*[44]	https://git.io/JJxZb
	Grossman *et al.*[77]	https://git.io/JJxnP
	He *et al.*[54]	https://git.io/JJxnL
	Honig *et al.*[66]	https://git.io/JJxZu
	Jiang *et al.*[95]	https://git.io/JtD2r
	Jiao *et al.*[84]	https://git.io/JJxZz
	Kolluri *et al.*[59]	https://git.io/JJxnf
	Li *et al.*[51]	https://git.io/JtD2F
	Medeiros *et al.*[97]	https://git.io/JJxnS
	Mossberg *et al.*[41]	https://git.io/v7beQ
	Nguyen *et al.*[48]	https://git.io/JJxZa
	Nikolic *et al.*[68]	https://git.io/JJxct
	Sergey *et al.*[55]	https://git.io/JJxZd
	Tikhomirov *et al.*[73]	https://git.io/JJxn6
	Torres *et al.*[74]	https://git.io/JJxny
	Zhang *et al.*[105]	https://git.io/JJxnk
	Gao *et al.*[42]	https://git.io/Jtybh
	Peng *et al.*[98]	https://git.io/JtyNf
Other	Albert *et al.*[82]	http://costa.fdi.ucm.es/papers/costa/safevm.ova
	Klomp *et al.*[85]	https://git.science.uu.nl/r.klomp/BitcoinAnalysis
	Luu *et al.*[72]	https://www.comp.nus.edu.sg/loiluu/oyente.html
	Tsankov *et al.*[106]	https://securify.ch
	Peng *et al.*[98]	https://wandbox.org/permlink/PnaL6bO9zipKRuKu

Finally, the output is fed to the CoreAnalysis, which targets vulnerabilities such as Transaction-Ordering Dependence, Timestamp Dependence, Mishandled Exceptions, and Reentrancy Vulnerability. In the end, the Validator filters out false positives contracts flagged by the tool.

We report in Figure 10 a graph in which the nodes represents papers and the directed edges indicate a paper has used the approach defined in the pointed paper as a baseline. This map may guide future researchers to select the most appropriate baseline with which they can compare their novel approaches.

Summary of RQ$_3$. About half of the papers analyzed shared their datasets, while about a third of them publicly release a prototype tool implementing the defined approach. We observed that, naturally, open-source tools are more likely to be used as baselines in other studies and to be extended.

6.4 | Discussion and Open Problems

Smart Contract technology is a quite new topic in the research context. During the last few years, blockchain technology has grown rapidly involving new tools and application contexts other than just being used as a virtual currency. This implies that there is a growing need for new techniques and approaches that support the new features introduced in the upcoming new versions of the software system behind the specific SC technology. There are security vulnerabilities that are still not discovered (zero-day vulnerabilities), even if some approaches specifically target this kind of security issues[48]. Note that our review was specifically focused on approaches targeting Smart Contracts. Some issues, however, might be external to the SCs, *e.g.*, they may be related to the structure of the network. For example, Xu *et al.*[107] (which we excluded from our literature review) proposed an approach

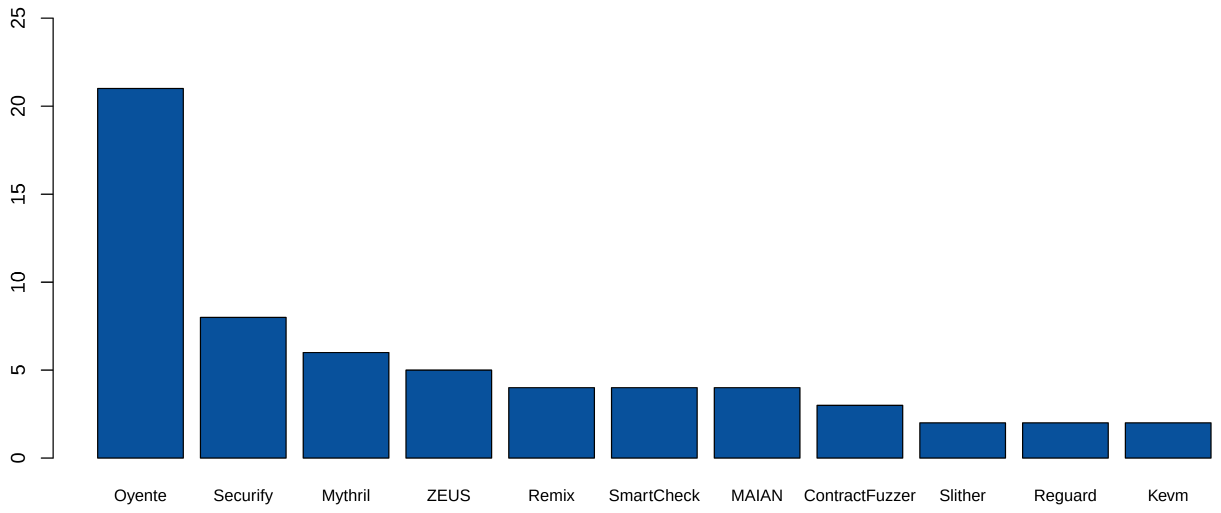


FIGURE 9 Usage of approaches as baselines in other studies. We only report the ones used by at least two papers.

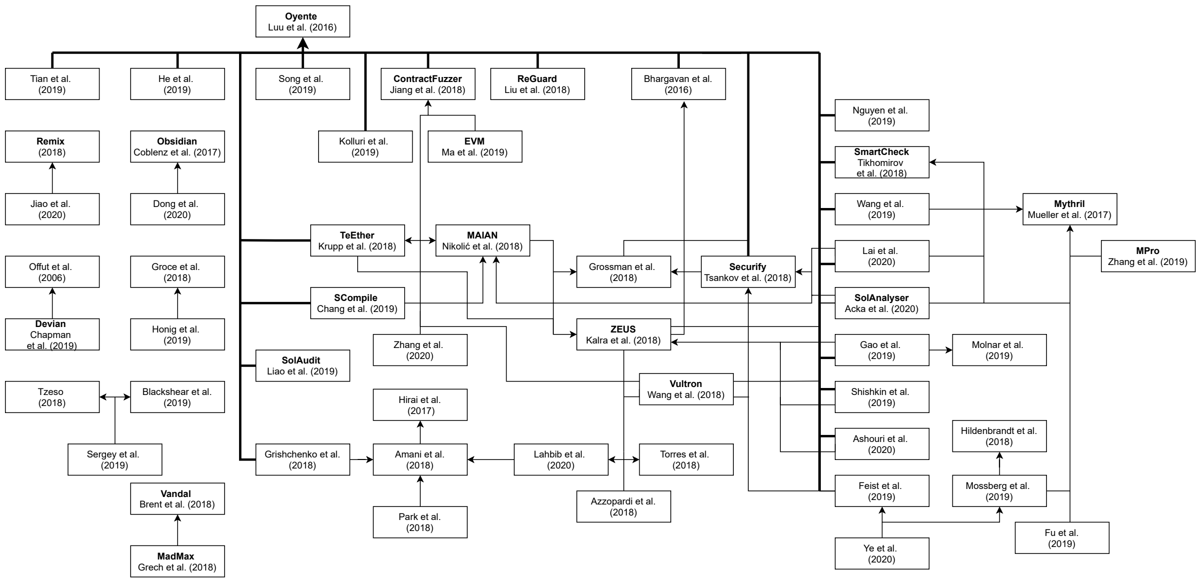


FIGURE 10 Map of the baselines used for the evaluation of each approach proposed in the papers we analyzed in our literature review.

for the detection of eclipse attacks. This kind of attack enables malicious actors to isolate a system user by taking control of all outgoing connections. To avoid this kind of security issue, specialized middlewares can be adopted, such as specialized firewalls for the monitoring of the overall network traffic. We discuss below some gaps highlighted by our literature review that future research should try to fill.

Support for Non-Ethereum Blockchains. As Ethereum and his language Solidity were firstly introduced and widely used, most of the existing approaches are focused only on these technologies. The fact that the vast majority of the approaches target only a single Blockchain technology (*i.e.*, Ethereum shows an important gap in the literature). While the same is partially true for other areas of Software Engineering research (*e.g.*, Java is often the most targeted programming languages for many types of approaches), it can be noticed that the technologies behind SCs are much more different among each other than general-purpose programming languages. Different blockchain provides different features, thus enabling possibly different bugs and vulnerabilities. While an approach devised for Java could be adapted to a broader class of object-oriented programming languages with a limited effort, the same might not be true for approaches that target SCs. For example, the Bitcoin Scripting Language is not Turing-complete: approaches working on Ethereum-based languages may not be directly adapted to such a profoundly different language. Therefore, we believe that future studies should try to target other very common blockchain technologies, such as Bitcoin and Hyperledger[108].

Reference Datasets of Bugs and Vulnerabilities. Most of the papers we analyzed defined their own datasets or they re-used previously defined small datasets including SCs with the specific bugs and vulnerabilities targeted by the approach they introduced. There is a lack of a comprehensive state-of-art dataset including both buggy and vulnerable SCs that can be used as a universal benchmark for evaluating novel approaches. Also, it can be useful to have a public dataset of vulnerable Smart Contracts referred to the security vulnerabilities reported in the SWC registry. In some cases, methods for Smart Contracts generation are used. For example, to train their approach based on machine learning, Momeni *et al.*[49] built a dataset of vulnerable Smart Contracts using two static analysis based approaches. Finally, our analysis highlighted that there is no universally recognized registry for vulnerabilities of SCs yet. The SWC registry is an attempt in this direction, but we found that it does not cover some important weaknesses targeted by state-of-the-art approaches. At the moment, the information about weaknesses in SCs is scattered, and future work should be aimed at providing a unified registry, similar to MITRE's CWE[14].

Technological Support for Testing Smart Contracts. Since Smart Contracts are based by design on distributed systems (*i.e.*, blockchains), it is difficult to apply classic testing techniques and methodologies. *Testnets* are fully working blockchains publicly available only for testing purposes. For Ethereum, one of the most used ones is *Kovan*[109] . Using such an approach requires set-up and deploy a fully working blockchain: the need to perform such operations may hinder the adoption of testing altogether. Using modern containerization tools such as Docker[110] and Kubernetes can help to deploy a minimal working example of a specific Smart Contract only to run tests. An example is the approach proposed by Chen *et al.*[86], which involves containers to build a simulated blockchain network for testing purposes.

Beyond Isolated Smart Contracts. Some SCs partially or totally rely on external SCs to work properly through the *off-chain* transactions. Therefore, some Smart Contracts may have properties across multiple interacting contracts[70]. Extending testing approaches to these contexts can be an important future direction in the field.

7 | THREATS TO VALIDITY

Our Systematic Literature Review may be affected by threats to validity for every single step. Specifically, we describe below the threats regard (i) the selection of the primary studies, including the manual evaluation for including/excluding articles, and (ii) the classification framework.

7.1 | Article Selection

The first threat regards the definition of the search query has a primary role in a Systematic Literature Review. Using a query too narrow may exclude interesting articles from the review, while using a query too broad may increase the burden of manually discarding a great number of false positives. To reduce the risk of both such problems, we defined the query after having analyzed the first set of valid articles. However, it is still possible that some relevant articles might have not been included in the first set of primary studies.

Another important threat regards the choice of inclusion and exclusion criteria. In our work we intentionally did not include grey literature, *i.e.*, we only considered peer-reviewed articles. While this choice could result in the exclusion of interesting articles introducing relevant approaches, it allows us to have confidence in the acceptable quality of the selected studies. It is worth noting that, since the topic we surveyed is relatively new, we did not exclude short papers *a-priori*. Instead, we always excluded the papers that did not provide any validation (even a minimal one) to avoid completely untested ideas.

7.2 | Classification Framework

The definition of dimensions, attributes and values of our classification framework may have an impact on the results. For example, excluding a recurring type of approach may result in hiding some relevant piece of information. To define the dimensions and the attributes, we strictly based on the research questions that guided our SLR. To reduce the risk of missing relevant values, we first defined an initial set of values, but we refined our classification framework after first labelling the primary studies. This allowed us to introduce more detailed values.

Manually reading and classifying the articles according to the classification framework we defined requires subjectively interpret parts of the articles. Only part of the information was explicitly mentioned by the authors: for example, some of the primary studies analyzed did not clearly mention the base methodology used (*e.g.*, static or dynamic analysis) or the targeted technology (*e.g.*, they assumed it based on the targeted programming language, or vice versa). Most noticeably, some work did not explicitly mention the CWE ID of the weaknesses targeted by the approach (*e.g.*, they used their names). To reduce the risk of introducing biases due to subjective classifications, three of the authors independently classified all the articles. A fourth author was involved when consensus could not be achieved.

8 | CONCLUSION

Smart Contracts are an important component of some blockchains (*e.g.*, Ethereum and Bitcoin). They allow users to write programmatic agreements among two or more parties that can not be rescinded. SCs may be affected by bugs and security vulnerabilities but, differently from other types of programs, they can not be fixed and updated later because of their nature of agreements. We conducted a Systematic Literature Review on the approaches defined to detect functional and security-related issues in SCs to (i) understand what is the state of the art in this field, and (ii) to find possible open problems that future research should address. Our review process was based on the use of a classification framework that allowed us to extract relevant information from the primary studies we selected.

First of all, we found that most of the approaches in the literature are based on *static analysis*, and, among them, *formal methods* are generally preferred. While such approaches generally have scalability issues that hamper their usage for large software systems, they are well-suited for SCs, which are generally small and, most importantly, need to be correct from the beginning. Other techniques are less popular: machine learning-based approaches are gaining popularity, but few of them are available.

Another interesting finding is that most of the approaches focus on security. While a catalogue of possible weaknesses exists (SWC), we observed a misalignment between such a catalogue and the literature. A subset of weaknesses is targeted by existing approaches (most notably, *Reentrancy* and *Integer Overflow and Underflow*), while 8 SWC weaknesses are not taken into account by any existing approach. Moreover, we found 3 weaknesses targeted by existing approaches that do not appear in the SWC catalogue.

In terms of targeted technologies, we found a noticeable imbalance in the state of the art: the large majority of the approaches target *Ethereum*-based SCs. Also, most of such approaches target the most popular programming language for SCs in Ethereum, Solidity. However, we believe that this is not a major issue: A good amount of approaches exist for the EVM Bytecode, the low-level language in which all the high-level languages must compile to be executed. Therefore, all languages can potentially be supported.

Finally, we observed that only half of the papers we analyzed using publicly available datasets for their experiments. Moreover, only a third of them publicly release at least a prototype tool with their approach to foster the replicability of the experimental results. We found clear evidence (besides the anecdotical beliefs) that releasing a tool has a positive effect on the research community: most of the baselines used for assessing the worth of new approaches are constituted publicly available ones. In other words, a good approach that is not available may be hard to reproduce and, therefore, it is rarely used as a baseline. A clear example of this is Oyente[72], one of the first tools released for detecting issues in SCs. Such an approach was not only largely used as a baseline, but it was used as a base for many other approaches.

We also identified three principal future directions in this field. Future research should be aimed at (i) defining new approaches targeting SCs for blockchains different from Ethereum, (ii) building a reference dataset for evaluating approaches for detecting issues in SCs, and (iii) finding issues related to the interaction among different SCs.

References

1. Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* 2008: 1–9.

2. Falkon S. Tuhe Story of the DAO - Its History and Consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee; .

3. Vacca A, Di Sorbo A, Visaggio CA, Canfora G. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software* 2021; 174: 110891. doi: 10.1016/j.jss.2020.110891

4. Liu J, Liu Z. A survey on security verification of blockchain smart contracts. *IEEE Access* 2019; 7: 77894–77904.

5. Everledger. https://everledger.io/; .

6. Ripple. https://ripple.com/; .

7. Alharby M, Van Moorsel A. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372* 2017.

8. Buterin V. On public and private blockchains (2015). *URL: https://blog. ethereum. org/2015/08/07/on-public-and-private-blockchains* 2015.

9. Szabo N. Formalizing and securing relationships on public networks. *First Monday* 1997.

10. Lewis A. A gentle introduction to smart contracts. *Retrievd from https://bitsonblocks. net/2016/02/01/a-gentle-introduction-to-smart-contracts* 2016.

11. Buterin V, others . A next-generation smart contract and decentralized application platform. *white paper* 2014; 3(37).

12. Wood G, others . Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 2014; 151(2014): 1–32.

13. NXT. https://www.jelurida.com/nxt; .

14. CWE Registry. https://cwe.mitre.org/; .

15. SWC Registry. https://cwe.mitre.org/; .

16. Keele S, others . Guidelines for performing systematic literature reviews in software engineering. tech. rep., Technical report, Ver. 2.3 EBSE Technical Report. EBSE; https://cdn.elsevier.com/promis_misc/525444systematicreviewsguide.pdf: 2007.

17. Casino F, Dasaklis TK, Patsakis C. A systematic literature review of blockchain-based applications: current status, classification and open issues. *Telematics and informatics* 2019; 36: 55–81.

18. Conoscenti M, Vetro A, De Martin JC. Blockchain for the Internet of Things: A systematic literature review. In: IEEE. ; 2016: 1–6.

19. Shen C, Pena-Mora F. Blockchain for cities—a systematic literature review. *Ieee Access* 2018; 6: 76787–76819.

20. Chen H, Pendleton M, Njilla L, Xu S. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 2020; 53(3): 1–43.

21. Di Angelo M, Salzer G. A survey of tools for analyzing Ethereum smart contracts. In: IEEE. ; 2019: 69–78.

22. Feng X, Wang Q, Zhu X, Wen S. Bug searching in smart contract. *arXiv preprint arXiv:1905.00799* 2019.

23. Murray Y, Anisi DA. Survey of formal verification methods for smart contracts on blockchain. In: IEEE. ; 2019: 1–6.

24. Xu J, Dang F, Ding X, Zhou M. A Survey on Vulnerability Detection Tools of Smart Contract Bytecode. In: IEEE. ; 2020: 94–98.

25. Alharby M, Aldweesh A, Moorsel vA. Blockchain-based smart contracts: A systematic mapping study of academic research (2018). In: IEEE. ; 2018: 1–6.

26. Yu W, Luo K, Ding Y, You G, Hu K. A Parallel Smart Contract Model. In: International Conference on Machine Learning and Machine Intelligence. ; 2018: 72–77.

27. ACM Digital Library. https://dl.acm.org; .

28. IEEE eXplore. https://ieeexplore.ieee.org/Xplore/home.jsp; .

29. Science Direct. https://www.sciencedirect.com; .

30. Springer Link. https://link.springer.com; .

31. Wichmann BA, Canning A, Clutterbuck D, Winsborrow L, Ward N, Marsh DWR. Industrial perspective on static analysis. *Software Engineering Journal* 1995; 10(2): 69–75.

32. Gerhart S, Craigen D, Ralston T. Observations on industrial practice using formal methods. In: IEEE. ; 1993: 24–33.

33. King JC. A new approach to program testing. *ACM Sigplan Notices* 1975; 10(6): 228–233.

34. Boxler D, Walcott KR. Static taint analysis tools to detect information flows. In: The Steering Committee of The World Congress in Computer Science, Computer .... ; 2018: 46–52.

35. Ball T. The concept of dynamic analysis. In: Springer. ; 1999: 216–234.

36. Myers GJ. The Art of Software Testing.

37. Gallagher T, Landauer L, Jeffries B. *Hunting security bugs.* Microsoft Press . 2006.

38. Mitchell TM. Does machine learning really work?. *AI magazine* 1997; 18(3): 11–11.

39. Liu H, Liu C, Zhao W, Jiang Y, Sun J. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In: IEEE. ; 2018: 814–819.

40. Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Buenzli F, Vechev M. Securify: Practical security analysis of smart contracts. In: ; 2018: 67–82.

41. Mossberg M, Manzano F, Hennenfent E, et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: IEEE. ; 2019: 1186–1189.

42. Gao J, Liu H, Liu C, Li Q, Guan Z, Chen Z. Easyflow: Keep ethereum away from overflow. In: IEEE. ; 2019: 23–26.

43. Lai E, Luo W. Static analysis of integer overflow of smart contracts in ethereum. In: ; 2020: 110–115.

44. Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2018; 2(OOPSLA): 1–27.

45. Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B. Reguard: finding reentrancy bugs in smart contracts. In: IEEE. ; 2018: 65–68.

46. Zhang Q, Wang Y, Li J, Ma S. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. In: IEEE. ; 2020: 116–126.

47. Chen T, Li X, Wang Y, et al. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In: Springer. ; 2017: 3–24.

48. Nguyen QB, Nguyen AQ, Nguyen VH, Nguyen-Le T, Nguyen-An K. Detect Abnormal Behaviours in Ethereum Smart Contracts Using Attack Vectors. In: Springer. ; 2019: 485–505.

49. Momeni P, Wang Y, Samavi R. Machine Learning Model for Smart Contracts Security Analysis. In: IEEE. ; 2019: 1–6.

50. Chapman P, Xu D, Deng L, Xiong Y. Deviant: A Mutation Testing Tool for Solidity Smart Contracts. In: IEEE. ; 2019: 319–324.

51. Li Z, Wu H, Xu J, Wang X, Zhang L, Chen Z. MuSC: A Tool for Mutation Testing of Ethereum Smart Contract. In: ; 2019: 1198-1201.

52. Wang Z, Dai W, Choo KKR, Jin H, Zou D. FSFC: An input filter-based secure framework for smart contract. *Journal of Network and Computer Applications* 2020; 154: 102530.

53. Wang H, Li Y, Lin SW, Ma L, Liu Y. Vultron: catching vulnerable smart contracts once and for all. In: IEEE. ; 2019: 1–4.

54. He J, Balunović M, Ambroladze N, Tsankov P, Vechev M. Learning to fuzz from symbolic execution with application to smart contracts. In: ; 2019: 531–548.

55. Sergey I, Nagaraj V, Johannsen J, Kumar A, Trunov A, Hao KCG. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages* 2019; 3(OOPSLA): 1–30.

56. Hirai Y. Defining the ethereum virtual machine for interactive theorem provers. In: Springer. ; 2017: 520–535.

57. Hildenbrandt E, Saxena M, Rodrigues N, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In: IEEE. ; 2018: 204–217.

58. Amani S, Bégel M, Bortin M, Staples M. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: ; 2018: 66–77.

59. Kolluri A, Nikolic I, Sergey I, Hobor A, Saxena P. Exploiting the laws of order in smart contracts. In: ; 2019: 363–373.

60. External Contract Referencing. https://git.io/Jt44o; .

61. Park D, Zhang Y, Saxena M, Daian P, Rosu G. A formal verification tool for Ethereum VM bytecode. In: ; 2018: 912–915.

62. Denial of Service. https://www.dasp.co/#item-5; .

63. Ashouri M. Etherolic: a practical security analyzer for smart contracts. In: ; 2020: 353–356.

64. Liao JW, Tsai TT, He CK, Tien CW. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In: IEEE. ; 2019: 458–465.

65. Short Address Attack. https://www.dasp.co/#item-9; .

66. Honig JJ, Everts MH, Huisman M. Practical Mutation Testing for Smart Contracts. In: Springer. 2019 (pp. 289–303).

67. Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: Analyzing Safety of Smart Contracts.. In: ; 2018: 1–12.

68. Nikolić I, Kolluri A, Sergey I, Saxena P, Hobor A. Finding the greedy, prodigal, and suicidal contracts at scale. In: ; 2018: 653–663.

69. Krupp J, Rossow C. teether: Gnawing at ethereum to automatically exploit smart contracts. In: ; 2018: 1317–1333.

70. Azzopardi S, Ellul J, Pace GJ. Monitoring smart contracts: Contractlarva and open challenges beyond. In: Springer. ; 2018: 113–137.

71. Akca S, Rajan A, Peng C. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In: IEEE. ; 2019: 482–489.

72. Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: ; 2016: 254–269.

73. Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. Smartcheck: Static analysis of ethereum smart contracts. In: ; 2018: 9–16.

74. Torres CF, Schütte J, State R. Osiris: Hunting for integer bugs in ethereum smart contracts. In: ; 2018: 664–676.

75. Zupan N, Kasinathan P, Cuellar J, Sauer M. Secure Smart Contract Generation Based on Petri Nets. In: Springer. 2020 (pp. 73–98).

76. Wang Y, Lahiri SK, Chen S, et al. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In: Springer. ; 2019: 87–106.

77. Grossman S, Abraham I, Golan-Gueta G, et al. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages* 2017; 2(POPL): 1–28.

78. Zakrzewski J. Towards verification of Ethereum smart contracts: a formalization of core of Solidity. In: Springer. ; 2018: 229–247.

79. Bai X, Cheng Z, Duan Z, Hu K. Formal modeling and verification of smart contracts. In: ; 2018: 322–326.

80. Ma F, Fu Y, Ren M, et al. EVM*: from offline detection to online reinforcement for ethereum virtual machine. In: IEEE. ; 2019: 554–558.

81. Zhang Y, Ma S, Li J, Li K, Nepal S, Gu D. SMARTSHIELD: Automatic Smart Contract Protection Made Easy. In: IEEE. ; 2020: 23–34.

82. Albert E, Correas J, Gordillo P, Román-Díez G, Rubio A. SAFEVM: a safety verifier for Ethereum smart contracts. In: ; 2019: 386–389.

83. Chatterjee K, Goharshady AK, Goharshady EK. The treewidth of smart contracts. In: ; 2019: 400–408.

84. Jiao J, Lin SW, Sun J. A Generalized Formal Semantic Framework for Smart Contracts.. In: ; 2020: 75–96.

85. Klomp R, Bracciali A. On symbolic verification of Bitcoin's script language. In: Springer. 2018 (pp. 38–56).

86. Chen C, Qi Z, Liu Y, Lei K. *Using Virtualization for Blockchain Testing*: 289-299; 2018

87. Bigi G, Bracciali A, Meacci G, Tuosto E. Validation of decentralised smart contracts through game theory and formal methods. In: Springer. 2015 (pp. 142–161).

88. Bamboo: a language for morphing smart contracts. https://github.com/CornellBlockchain/bamboo; .

89. Bhargavan K, Delignat-Lavaud A, Fournet C, et al. Formal verification of smart contracts. In: ; 2016: 91–96.

90. Chang J, Gao B, Xiao H, Sun J, Cai Y, Yang Z. sCompile: Critical path identification and analysis for smart contracts. In: Springer. ; 2019: 286–304.

91. Dong C, Li Y, Tan L. A New Approach to Prevent Reentrant Attack in Solidity Smart Contracts. In: Springer. ; 2019: 83–103.

92. El-Dosuky MA, Eladl GH. DOORchain: deep ontology-based operation research to detect malicious smart contracts. In: Springer. ; 2019: 538–545.

93. Feist J, Grieco G, Groce A. Slither: a static analysis framework for smart contracts. In: IEEE. ; 2019: 8–15.

94. Fu M, Wu L, Hong Z, Zhu F, Sun H, Feng W. A critical-path-coverage-based vulnerability detection method for smart contracts. *IEEE Access* 2019; 7: 147327–147344.

95. Jiang B, Liu Y, Chan W. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: IEEE. ; 2018: 259–269.

96. Lahbib A, Wakrime AA, Laouiti A, Toumi K, Martin S. An Event-B Based Approach for Formal Modelling and Verification of Smart Contracts. In: Springer. ; 2020: 1303–1318.

97. Medeiros H, Vilain P, Mylopoulos J, Jacobsen HA. SolUnit: a framework for reducing execution time of smart contract unit tests. In: ; 2019: 264–273.

98. Peng C, Akca S, Rajan A. SIF: A Framework for Solidity Contract Instrumentation and Analysis. In: IEEE. ; 2019: 466–473.

99. Shishkin E. Debugging Smart Contract's Business Logic Using Symbolic Model Checking. *Programming and Computer Software* 2019; 45(8): 590–599.

100. Song J, He H, Lv Z, Su C, Xu G, Wang W. An efficient vulnerability detection model for ethereum smart contracts. In: Springer. ; 2019: 433–442.

101. Tian Z. Smart Contract Defect Detection Based on Parallel Symbolic Execution. In: IEEE. ; 2019: 127–132.

102. Wang S, Zhang C, Su Z. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2019; 3(OOPSLA): 1–29.

103. Ye J, Ma M, Peng T, Xue Y. A Software Analysis Based Vulnerability Detection System For Smart Contracts. In: Springer. 2020 (pp. 69–81).

104. Zhang Q, Wang Y, Li J, Ma S. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. In: ; 2020: 116-126.

105. Zhang W, Banescu S, Pasos L, Stewart S, Ganesh V. MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract. In: IEEE. ; 2019: 456–462.

106. Tsankov P. Security analysis of smart contracts in datalog. In: Springer. ; 2018: 316–322.

107. Xu G, Guo B, Su C, et al. Am I eclipsed? A smart detector of eclipse attacks for Ethereum. *Computers & Security* 2020; 88: 101604.

108. Hyperledger Foundation. https://www.hyperledger.org; .

109. Kubern. https://kovan-testnet.github.io/website/; .

110. Docker. https://www.docker.com; .

111. Alt L, Reitwießner C. Smt-based verification of solidity smart contracts. In: Springer. ; 2018: 376–388.

112. Bai X, Cheng Z, Duan Z, Hu K. Formal modeling and verification of smart contracts. In: ; 2018: 322–326.

113. Gao J, Liu H, Li Y, et al. Towards automated testing of blockchain-based decentralized applications. In: IEEE. ; 2019: 294–299.

114. Gao J. Guided, Automated Testing of Blockchain-Based Decentralized Applications. In: ICSE '19. IEEE Press; 2019: 138–140

115. Grishchenko I, Maffei M, Schneidewind C. A semantic framework for the security analysis of ethereum smart contracts. In: Springer. ; 2018: 243–269.

116. Kasampalis T, Guth D, Moore B, et al. IELE: A rigorously designed language and tool ecosystem for the blockchain. In: Springer. ; 2019: 593–610.

APPENDIX

A COMPLETE CATEGORIZATION

TABLE A1 Article characterization

Study	Reproducibility				Targeted Technologies												Approaches																		
	Public Dataset		*Public tool or framework*		*Blockchain*			*Programming Languages*							*Multilanguage*		*Methodology*									*Mixed*		*Issues*							
																												Functional	*Security*						
	Yes	No	Yes	No	Ethereum	BitCoin	Others	Solidity	EVM Bytecode	Vyper	YUL	YUL+	BitCoin Scripting Language	Others	Yes	No	DYNAMIC ANALYSIS	Test Case Generation	Fuzzing Testing	OTHERS	Machine Learning	STATIC ANALYSIS	Formal Methods	Symbolic Execution	Taint Analysis	Yes	No		Access Control	Availability	Confidentiality	Integrity	Interaction between SCs	Non-Repudiation	Other
Akca *et al.*[71]	✓			✓	✓			✓								✓		✓				✓				✓				✓	✓	✓	✓		✓
Albert *et al.*[82]		✓	✓		✓			✓	✓						✓								✓				✓	✓							
Alt *et al.*[111]		✓		✓	✓			✓								✓							✓				✓		✓	✓	✓	✓			✓
Amani *et al.*[58]		✓	✓		✓				✓							✓							✓				✓			✓					✓
Ashouri *et al.*[63]		✓		✓	✓				✓							✓			✓								✓		✓	✓	✓	✓	✓		✓
Azzopardi *et al.*[70]	✓		✓		✓			✓								✓							✓				✓	✓							
Bai *et al.*[112]		✓		✓	✓				✓							✓							✓				✓	✓							
Bhargavan *et al.*[89]		✓		✓	✓				✓							✓							✓				✓			✓	✓	✓	✓		✓
Bigi *et al.*[87]		✓		✓		✓							✓			✓				✓			✓			✓		✓							
Chang *et al.*[90]	✓			✓	✓			✓								✓								✓			✓	✓							
Chapman *et al.*[50]	✓			✓	✓			✓								✓				✓							✓	✓							
Chatterjee *et al.*[83]		✓		✓	✓			✓		✓					✓							✓					✓	✓							
Chen *et al.*[86]		✓		✓		✓							✓			✓				✓							✓	✓							
Chen *et al.*[47]	✓			✓	✓			✓								✓	✓										✓			✓					✓
Dong *et al.*[91]	✓			✓	✓			✓								✓	✓										✓						✓		✓
El-Dosuky *et al.*[92]	✓		✓		✓			✓								✓	✓				✓					✓		✓							
Feist *et al.*[93]	✓		✓		✓			✓								✓						✓					✓			✓	✓	✓	✓		✓
Fu *et al.*[94]		✓		✓	✓			✓								✓								✓			✓		✓	✓	✓	✓	✓		✓
Gao *et al.*[113]		✓		✓	✓			✓								✓				✓					✓	✓		✓							
Gao *et al.*[42]		✓		✓	✓				✓							✓									✓		✓		✓	✓	✓	✓			
Gao[114]		✓		✓	✓			✓								✓	✓					✓				✓		✓							
Grech *et al.*[44]	✓		✓		✓			✓		✓					✓							✓					✓		✓	✓	✓	✓	✓		✓
Grishchencko *et al.*[115]		✓		✓	✓				✓							✓							✓				✓	✓							
Grossman *et al.*[77]		✓	✓		✓			✓								✓	✓						✓			✓		✓							
He *et al.*[54]	✓		✓		✓			✓								✓			✓		✓			✓		✓				✓	✓	✓			✓
Hildenbrandt *et al.*[57]		✓		✓	✓				✓							✓							✓				✓			✓					✓
Hirai *et al.*[56]	✓			✓	✓				✓							✓							✓				✓			✓			✓		✓
Honig *et al.*[66]	✓		✓		✓			✓								✓				✓							✓	✓							
Jiang *et al.*[95]		✓	✓		✓			✓								✓			✓			✓				✓				✓	✓	✓	✓		✓
Jiao *et al.*[84]	✓		✓		✓			✓		✓				✓	✓							✓					✓	✓							
Kalra *et al.*[67]		✓		✓	✓		✓	✓								✓							✓				✓		✓	✓	✓	✓	✓		✓
Kasampalis *et al.*[116]		✓		✓	✓			✓								✓							✓				✓		✓	✓	✓	✓			
Klomp *et al.*[85]	✓		✓			✓							✓			✓							✓				✓	✓							
Kolluri *et al.*[59]	✓		✓		✓				✓							✓			✓					✓		✓						✓	✓		
Krupp *et al.*[69]		✓		✓	✓				✓							✓								✓			✓			✓	✓	✓			✓
Lahbib *et al.*[96]		✓		✓	✓			✓								✓							✓				✓	✓							
Lai *et al.*[43]	✓			✓	✓			✓								✓						✓					✓		✓	✓	✓	✓			
Li *et al.*[51]	✓		✓		✓			✓								✓				✓							✓	✓							
Liao *et al.*[64]	✓			✓	✓			✓								✓			✓		✓					✓			✓	✓	✓	✓	✓		✓
Liu *et al.*[45]		✓		✓	✓			✓	✓						✓				✓								✓						✓		
Liu *et al.*[39]	✓			✓	✓			✓								✓					✓	✓				✓		✓							
Luu *et al.*[72]	✓		✓		✓	✓		✓	✓						✓									✓			✓			✓	✓	✓	✓		✓
Ma *et al.*[80]		✓		✓	✓				✓							✓							✓				✓	✓							
Medeiros *et al.*[97]	✓		✓		✓			✓								✓				✓							✓	✓							
Momeni *et al.*[49]		✓		✓	✓			✓								✓					✓	✓				✓			✓	✓	✓	✓	✓		✓
Mossberg *et al.*[41]	✓		✓		✓			✓								✓								✓			✓	✓							
Nguyen *et al.*[48]	✓		✓		✓			✓								✓	✓										✓		✓	✓	✓	✓	✓		✓
Nikolic *et al.*[68]		✓	✓		✓				✓							✓	✓										✓			✓		✓			✓
Park *et al.*[61]	✓			✓	✓				✓							✓							✓				✓		✓	✓	✓	✓			
Peng *et al.*[98]	✓		✓		✓			✓								✓						✓					✓		✓	✓	✓	✓			✓
Sergey *et al.*[55]	✓		✓				✓	✓	✓						✓							✓					✓			✓					✓
Shishkin *et al.*[99]	✓			✓	✓			✓								✓							✓				✓	✓							
Song *et al.*[100]		✓		✓	✓			✓								✓					✓						✓		✓	✓	✓	✓	✓		✓
Tian *et al.*[101]		✓		✓	✓				✓							✓								✓			✓			✓	✓	✓	✓		✓
Tikhomirov *et al.*[73]	✓		✓		✓			✓								✓						✓					✓			✓	✓	✓	✓		✓
Torres *et al.*[74]		✓	✓		✓				✓							✓								✓	✓	✓			✓	✓	✓	✓			
Tsankov *et al.*[40]		✓	✓		✓			✓	✓						✓								✓				✓		✓	✓	✓	✓			✓
Wang *et al.*[102]		✓		✓	✓			✓								✓							✓				✓			✓	✓	✓	✓		✓
Wang *et al.*[76]	✓			✓			✓	✓								✓							✓				✓	✓							
Wang *et al.*[53]		✓		✓	✓			✓	✓						✓					✓							✓		✓	✓	✓	✓	✓		✓
Wang *et al.*[52]		✓		✓	✓			✓								✓				✓							✓		✓	✓	✓	✓			
Ye *et al.*[103]		✓		✓	✓				✓							✓						✓					✓			✓	✓	✓	✓		✓
Zakrzewski *et al.*[78]		✓		✓	✓			✓								✓							✓				✓	✓							
Zhang *et al.*[104]	✓			✓	✓			✓								✓		✓	✓						✓	✓				✓		✓			✓
Zhang *et al.*[105]		✓	✓		✓			✓								✓								✓			✓						✓		✓
Zhang *et al.*[81]	✓			✓	✓				✓							✓						✓					✓		✓	✓	✓	✓			✓
Zupan *et al.*[75]		✓		✓	✓			✓								✓							✓				✓	✓							

B SWC CATALOG OF VULNERABILITIES FOR SMART CONTRACTS

TABLE B2 Security vulnerabilities in Smart Contracts (first part)

			Scope						
SWC-ID	**Name**	**Summary**	Access Control	Availability	Confidentiality	Integrity	Interaction between SCs	Non-Repudiation	Other
SWC-100	Function Default Visibility	Function are set to public by default. If a developer does not change its visibility, the attacker can make unauthorized changes.							✓
SWC-101	Integer Overflow and Underflow	An arithmetic operation generates an output that is out of the range of the representable number (both for the upper limit and for the lower limit).							
SWC-102	Outdated Compiler Version	The compiler version is too old.							✓
SWC-103	Floating Pragma	Contracts are distributed with a different compiler version and different flags of those tested.							✓
SWC-104	Unchecked Call Return Value	The call return value is not checked and the execution will continue also after an exception. If the call fails accidentally or for malicious actions, this can return an unexpected behaviour.		✓		✓			
SWC-105	Unprotected Ether Withdrawal	For missing access controls, malicious parties can take some or all Ether from the contract account.							✓
SWC-106	Unprotected SELFDESTRUCT Instruction	For missing access controls, malicious parties can trigger the self-destruction of the contract.							✓
SWC-107	Reentrancy	A malicious contract calls back into the calling contract during another invocation of the function, causing different invocations of the function.					✓		
SWC-108	State Variable Default Visibility	The visibility label can allow erroneous assumptions on who can access to the variable.							✓
SWC-109	Uninitialized Storage Pointer	Uninitialized local storage variables are linked to unexpected storage locations in the contract.		✓	✓	✓			
SWC-110	Assert Violation	The `assert()` function is used to assert invariants, but this can be violated if in the contract there is a bug that takes in a invalid state or the assertion is not correct.							✓
SWC-111	Use of Deprecated Solidity Functions	The use of deprecated functions and operators can reduce the code quality.							✓
SWC-112	Deletagecall to Untrusted Callee	Delegatecall is a message call but the code at the target address is executed in the context of the calling contract. The target address could be not trusted.		✓	✓	✓			
SWC-113	DoS with Failed Call	If an external call fails, the contract can be verified a DoS condition.		✓	✓	✓			
SWC-114	Transaction Order Dependence	This vulnerability is also called race condition. A race condition vulnerability is occured when code is dependent by the order of the transactions submitted to it.		✓	✓	✓			
SWC-115	Authorization through `tx.origin`	The global variable `tx.origin` is associated to the address of who sent the transaction. If the authorized account call a malicious contract, a contract could become vulnerable. Consequently, a call to the vulnerable contract might seem correct because it has been performed from an authorized sender.							✓
SWC-116	Block values as a proxy for time	The access to time values can be unsafe (*i.e.*, `block.timestamp` and `block.number`). For example, regards to `block.timestamp`, malicious miners can change timestamps in the blocks.		✓	✓	✓			
SWC-117	Signature Malleability	Signature can be valid also after the alteration. The change can be light because a malicious user can alter some values.	✓		✓	✓			

TABLE B3 Security vulnerabilities in Smart Contracts (second part)

			Scope						
SWC-ID	**Name**	**Summary**	Access Control	Availability	Confidentiality	Integrity	Interaction between SCs	Non-Repudiation	Other
SWC-118	Incorrect Constructor Name	The constructor name can cause security issues if this does not correspond to the contract name.	✓	✓	✓				
SWC-119	Shadowing State Variables	This vulnerability is verified if there are multiple definitions on the contract and function level.							✓
SWC-120	Weak Sources of Randomness from CA	A weak source of randomness is very dangerous in Ethereum because the miner can extract many blocks in a short time.	✓		✓				✓
SWC-121	Missing Protection against SRA	For having a major usability or for saving gas cost, it needs to verify the signature in Smart Contracts and to have a secure implementation against Signature Replay Attacks. An attacker could obtain the message hash contained in the contract, which does not store all processed message hashes.	✓		✓	✓			
SWC-122	Lack of Proper Signature Verification	In smart contract systems, users can sign off-chain messages because these systems can retrieve the authenticity of signed messages before of the processing. By the way, these systems cannot directly interact with smart contracts, but some of them can assume the validity of a signed message causing a vulnerability.				✓			✓
SWC-123	Requirement Violation	The `require()` construct is used to validate external inputs of a function. External inputs can be provided by callers and can be returned by callees. The first case is called *precondition violations* and this is verified when in the contract there is a bug that returns an external input or when the requisit condition is too strong.							✓
SWC-124	Write to Arbitrary Storage Location	The smart contract have to allow the writing only to authorized users in sensitive storage locations. If a malicious user can write in these location, then the authorization can be bypassed.	✓	✓	✓	✓			✓
SWC-125	Incorrect Inheritance Order	Given the multiple inheritance, Solidity can have a Diamond Problem: the base contracts define the same function and the child contract do not know what have to call. For having a normal behavior, base contracts should have different priorities.				✓			
SWC-126	Insufficient Gas Griefing	This type of attack are performed on contracts that take data to use it in a sub-call on another contract. The sub-call fails, but the execution could be continue. If we have a relayer contract the 'forwarder' can censor transactions with the sufficient gas, but it cannot be used for the success of sub-call.							✓
SWC-127	Arbitrary Jump with Func. Type Variable	The user can arbitrary change the function type variable and the execution of variables can be randomly. If the user is malicious, he can point to a function type variable in any instructions without respect required validations.							✓
SWC-128	DoS With Block Gas Limit	To call functions in smart contracts, there is the need to have a certain amount of gas. If the execution cost of a function is greater than the limit of block gas, it is possible to have a Denial of Service.	✓	✓					✓
SWC-129	Typographical Error	This error is verified when the operation is defined in a wrong way (*e.g.*, the operation is +=, but accidentally it is become =+).							✓

TABLE B4 Security vulnerabilities in Smart Contracts (third part)

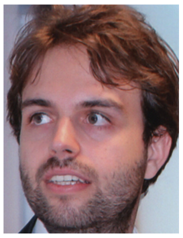
			Scope						
SWC-ID	**Name**	**Summary**	Access Control	Availability	Confidentiality	Integrity	Interaction between SCs	Non-Repudiation	Other
SWC-130	Right-To-Left-Override control character	Attackers can use the character U+202E to force RTL text rendering and to change the real intent of a contract.	✓					✓	
SWC-131	Presence of unused variables	A best practice is to avoid unused variables, because computations can increase and consume unnecessary gas. Anothe problem is the decrease of the code readability.							✓
SWC-132	Unexpected Ether Balance	If the Ether balance is incorrect, smart contracts can behave in an erroneous way.		✓					
SWC-133	Hash Collisions With Multiple VarLen Args.	It is possible to have an hash collision, if the `abi.encodePacked()` has multiple variable length arguments.	✓						
SWC-134	Message call with hardcoded gas amount	There is a setting on the forward of gas (*i.e.*, 2300 gas) of the `transfer()` and `send()`. If the gas cost increase significantly on instructions, smart contracts can break.	✓	✓	✓				
SWC-135	Code With No Effects	The developer can produce a code with no effects in Solidity.							✓
SWC-136	Unencrypted Private Data On-Chain	Malicious users can read contract transactions to understand values stored in the state's contract. Thus, it is important to have private variable.				✓			✓

AUTHOR BIOGRAPHY

Valentina Piantadosi was born in Isernia (Italy) on November 1st, 1993. She received (magna cum laude) a Master's Degree in Software System Security from the University of Molise (Italy) in 2018 defending a thesis on Software Reliability and Testing, advised by Prof. Rocco Oliveto. She received a Bachelor's Degree in Computer Science from the University of Molise in 2016 defending a thesis on Software Refactoring, advised by Prof. Rocco Oliveto. She is currently a Ph.D. Student. at the Department of Bioscences and Territory of University of Molise, advised by Prof. Rocco Oliveto. Her research interests include Vulnerability Detection, Testing and Machine Learning.

Giovanni Rosa received the bachelor's degree in Computer Science from the University of Molise, defending a thesis entitled "Are Developers Good in Code Review?" advised by Prof. Rocco Oliveto and co-advised by Prof. Jens Krinke, from the University College of London. Afterwards, he received the master's degree in Software Systems Security also from the University of Molise defending a thesis entitled "Evaluating SZZ Implementations Through a Developer-informed Oracle" advised by Prof. Rocco Oliveto and co-advised by Prof. Gabriele Bavota, from the Università della Svizzera Italiana and Dr. Simone Scalabrino, from the University of Molise. During the master's degree, he also obtained a scholarship to work on a research project about machine learning techniques for the automatic analysis of biomedical data.

Davide Placella was born in Isernia (Italy) on May 15th, 1995. He received a Master's Degree with magna cum laude in Software System Security from the University of Molise (Italy) in 2020 writing a thesis in Software Reliability and Testing, advised by Prof. Rocco Oliveto. He also received a Bachelor's degree in Computer Science from the University of Molise in 2018 writing a thesis in Numerical calculation, advised by Prof. Giovanni Capobianco. Currently, he is a Cyber Security Consultant at Avanade Italy. His current work include Vulnerability Detection, data protection, data governance, risk management, identity protection, IAM and GDPR compliance.



Simone Scalabrino is a Research Fellow at the University of Molise, Italy. He has received his MS degree from the University of Salerno, and his PhD degree from the University of Molise, defending a thesis on automatically assessing and improving source code readability and understandability. His main research interests include code quality, software testing, and empirical software engineering. He has received three ACM SIGSOFT Distinguished Paper Awards at ICPC 2016, ASE 2017, and MSR 2019. He is co-founder and CSO of datasound, a spin-off of the University of Molise. More information available at: https://dibt.unimol.it/sscalabrino/.

Rocco Oliveto is a Full Professor at the University of Molise (Italy). He is the Rector's Delegate for placement, internship and technology transfer. Since October 2020 he is the Deputy Director of the Department of Biosciences and Territory. He is the founder of the Software Engineering and Knowledge Engineering (STAKE) Lab of the University of Molise. Prof. Oliveto is co-author of over 150 papers on topics related to software traceability, software maintenance and evolution and empirical software engineering. In 2018 he was second in the world, in the consolidator category, in an eight-year bibliometric study (2010-2017), conducted to identify the 20 best early stages, consolidator and experienced software engineering researchers. He has received several awards for his research activity, including 5 ACM SIGSOFT Distinguished Paper Award and 2 Most Influential Paper Award. Prof. Oliveto participated in the organization and was a member of the program committee of several international conferences in the field of software engineering. Since 2018 he has been CEO of Datasound srl, a spin-off of the University of Molise that was created to conceive, design and develop innovative recommendation systems to be applied in different contexts.