# A Survey on Smart Contract Vulnerabilities: Data Sources, Detection and Repair

Hanting Chu[a], Pengcheng Zhang[a], Hai Dong[b], Yan Xiao[c], Shunhui Ji[a], Wenrui Li[d,*]

[a]*Hohai University, 8 Focheng West Rd, Nanjing, China*
[b]*School of Computing Technologies, RMIT University, Melbourne, Australia*
[c]*School of Computing, NUS University, Singapore*
[d]*Nanjing Xiao Zhuang University, Nanjing, China*

**Abstract**

Smart contracts contain many built-in security features, such as non-immutability once being deployed and non-involvement of third parties for contract execution. These features reduce security risks and enhance users' trust towards smart contracts. However, smart contract security issues still persist, resulting in huge financial losses. Contract publishers cannot fully cover contract vulnerabilities through contract version updating. These security issues affect further development of blockchain technologies. So far, there are many related studies focusing on smart contract security issues and tend to discuss from a particular perspective (e.g., development cycle, vulnerability attack methods, security detection tools, etc.). However, smart contract security is a complicated issue that needs to be explored from a multi-dimensional perspective. In this paper, we explore smart contract security from the perspectives of vulnerability data sources, vulnerability detection, and vulnerability defense. We first analyze the existing security issues and challenges of smart contracts, investigate the existing vulnerability classification frameworks and common security vulnerabilities, followed by reviewing the existing contract vulnerability injection, detection, and repair methods. We then analyze the performance of existing security methods. Next, we summarize the current status of smart contract security-related research. Finally, we summarize the state of the art and future trends of smart contract security-related research. This paper aims to provide systematic knowledge and references to this research field.

*Keywords:* Blockchains, Smart Contracts, Vulnerability Detection, Vulnerability Repair, Information Security.

## 1. Introduction

Blockchain is a distributed ledger that enables trusted value transfer in an environment of mutual distrust, which is a milestone in the history of human credit evolution [1]. Bitcoin, the hottest application of blockchain, can be operated by a limited number of scripts for transactions. With the emergence of Ethereum smart contracts, the blockchain technology has entered the era of programmable finance, where people can complete more customized transactions with the aid of smart contracts [2]. The concept of smart contracts was introduced by Szabo [3], which is a digital contract that can be executed over the Internet and designed to replace traditional paper-based contracts.

At present, the implementation of smart contracts relies heavily on the decentralized Ethernet virtual machines and the programming language represented by Solidity [4]. A blockchain system can be divided into a data layer, a network layer, a consensus layer, an incentive layer, a contract layer, and an application layer [5]. Compared with the other layers, the security threat of the contract layer has a closer relationship with the mechanism of the blockchainthe ensuing path explosion and symbolic path constraint insolvability problem [6]. This paper mainly focuses on the following security issues of the contract layer. First of all, a miner node can be selected to create a block without the need to verify if the node is located in a trusted execution environment. If the miner node is malicious, it can manipulate the block transaction order and cause security problems [7]. Secondly, smart contracts often need to call each other to achieve more complex functionality. However, calling an untrusted external contract may raise risks and errors. This can lead to potential security threats to the local contract if malicious code exists in the external contract [8]. Furthermore, since smart contracts on the blockchain can be written and published by any user with diverse programming abilities and development tools, there is no guarantee that a smart contract will be deployed on the chain without security vulnerabilities or flaws [9]. Finally, smart contracts also have some special mechanisms that traditional codes do not have, such as the gas mechanism of Ethereum smart contracts. These will lead to certain security risks specific to smart contracts on the blockchain. Due to the strong correlation of smart contracts with the access of financial currency, many attackers have been exploiting vulnerabilities of smart contracts for profits. In order to ensure data consistency and transaction traceability, smart contracts cannot be modified after deployment. Even if vulnerabilities are detected, they cannot be fixed via patching or version upgrading

*Corresponding author
*Email addresses:* htchu@hhu.edu.cn (Hanting Chu), pchzhang@hhu.edu.cn (Pengcheng Zhang), hai.dong@rmit.edu.au (Hai Dong), dcsxan@nus.edu.sg (Yan Xiao), shunhuiji@hhu.edu.cn (Shunhui Ji), wenrui_li@163.com (Wenrui Li )

Table 1: Smart Contract Security Incidents

| Serial Number | Time of Attack | Target of Attack | Amount of Damage |
|---|---|---|---|
| 1 | 2016-06-17 | The DAO | 60,000,000 USD |
| 2 | 2017-07-29 | Parity | 30,000,000 USD |
| 3 | 2018-04-22 | BeautyChain | 1,000,000,000 USD |
| 4 | 2018-04-22 | SmartMesh | 140,000,000 USD |
| 5 | 2020-02-15 | bZx[1] | 350,000 USD |
| 6 | 2020-02-18 | bZx[1] | 645,000 USD |
| 7 | 2020-04-19 | Lendf.Me | 24,696,616 USD |
| 8 | 2020-06-18 | Bancor | 135,229 USD |
| 9 | 2020-07-01 | VETH | 900,000 USD |
| 10 | 2020-08-04 | Opyn | 371,260 USDC |
| 11 | 2020-08-13 | YAM | 750,000 USD |
| 12 | 2020-12-28 | Cover Protocol | 3,000,000 USD |
| 13 | 2021-03-09 | DODO | 500,000 USD |
| 14 | 2021-05-05 | Value DeFi[1] | 5,817,780 USD |
| 15 | 2021-05-07 | Value DeFi[1] | 10,000,000 USD |
| 16 | 2021-06-24 | SharedStake | 500,000 USD |
| 17 | 2021-07-11 | Umbrella Network | 3,000,000 UMB +300,000 DVG |
| 18 | 2021-07-13 | DeFiPie | 124,999 BUSD |
| 19 | 2021-07-16 | THORChain[1] | 7,600,000 USD |
| 20 | 2021-07-23 | THORChain[1] | 8,000,000 USD |
| 21 | 2021-08-11 | Punk Protocol | 3,950,000 USD |
| 22 | 2021-08-29 | xToken | 4,500,000 USD |
| 23 | 2021-09-04 | DAO Maker | 4,000,000 USD |
| 24 | 2021-09-15 | Nowswap | 1,000,000 USD |
| 25 | 2021-10-14 | Compound | 68,800,000 USD |
| 26 | 2021-11-27 | dYdX | 211,000 USD |
| 27 | 2021-11-28 | Visor Finance | 975,720 USD |
| 28 | 2021-12-11 | Gelato Network | 744,000 USD |
| 29 | 2021-12-30 | SashimiSwap | 200,000 USD |
| 30 | 2022-03-20 | Umbrella Network | 700,000 USD |
| 31 | 2022-03-20 | Li.finance | 600,000 USD |
| 32 | 2022-03-27 | Revest Finance | 120,000 USD |
| 33 | 2022-03-30 | BasketDAOOrg | 1,200,000 USD |
| 34 | 2022-04-30 | Fei Protocol | 80,340,000 USD |

[1] An event with the same name but listed twice means that the event caused multiple attacks.

other than self-destruction due to the tamper-evident nature. In this regard, malicious attacks on smart contracts cannot be simply prevented. The most famous incident "The DAO" caused over $60 million financial losses due to a re-entry vulnerability [10]. This incident directly led to the subsequent Ethereum hard fork, an operation that conflicts with the "decentralized" nature of blockchains and caused huge controversy within the community. The financial losses caused by blockchain security incidents have been increasing annually since 2016, especially from 2020 onwards. We collected the major smart contract attacks since 2016, as shown in Table 1[1].

These smart contract attacks seriously threaten the development of this technology. The characteristics of smart contracts make it impossible for developers to maintain the existing vulnerability contracts. With the increasing attention on the security of smart contracts, the amount of related scientific studies is rising.

We systematically survey the papers focusing on smart contract security, vulnerability collection, vulnerability detection, vulnerability repair, and security protection published from 2015-2022. According to our survey, 49 papers that are believed to represent the state of the art in this field are selected and investigated. This paper concentrates on the overall perspectives of smart contract security, aiming to identify shortcomings in existing research and provide insights for solutions and future research directions.

**Contributions.** The main contributions of this paper are:

1. Comprehensive analysis of smart contract security issues and challenges. The security problems and challenges faced by existing smart contracts are comprehensively analyzed.
2. Systematic review of the existing smart contract security detection and defense methods. In terms of the existing security challenges of smart contracts, we analyze the existing vulnerability detection tools, performance evaluation methods, and vulnerability repair methods. The security assurance methods of smart contracts are summarized from the perspectives of vulnerability data sources, vulnerability detection, and vulnerability repair.
3. Elaboration on shortcomings of existing research and future research directions. The strengths and weaknesses of existing smart contract security methods are analyzed for various security challenges. In particular, we extensively examine available evaluation datasets, existing vulnerability repair methods and artificial intelligence-based vulnerability detection methods. Next, future research directions are indicated for addressing those weaknesses.

The rest of the paper is organized as follows: Section 2 provides an introduction to the relevant review work. Section 3 provides an overview of the background knowledge. Section 4 analyzes and evaluates the literature retrieved for this paper. Section 5 discusses the research questions based on the findings. Section 6 summarizes the existing research gaps and discusses future research directions. Finally, Section 7 concludes the entire paper.

## 2. Related Work

In recent years, many survey papers focusing on various aspects of smart contract security have been published. Kushwaha et al. [11] discuss Ethereum smart contract security vulnerabilities, detection tools, and vulnerability attacks and prevention mechanisms. This study does not consider the impact of deep learning-based detection methods on smart contract security. Harz et al. [12] investigate the verification tools and methods for the mainstream programming languages and distributed ledgers for smart contracts. However, they do not discuss smart contract vulnerabilities. Sayeed et al. [13] survey contract vulnerabilities and detection tools but do not consider how to prevent contract vulnerabilities. Wang et al. [14] summarize the research results on smart contract security published from 2015-2019, mainly focusing on how contracts can be maliciously exploited and attacked. Nevertheless, they not consider the perspective of contract risk defense. Huang et al. [15] conducted

[1] ETH DApp attacks. https://hacked.slowmist.io/?c=ETH

a literature review on smart contract security from the software lifecycle perspective. Surucu et al. [16] analyze the drawbacks of existing vulnerability detection efforts and discuss the possibility of applying machine learning to smart contract vulnerability detection. Still, this study does not analyze smart contract vulnerabilities and defense mechanisms. Perez et al. [17] explore the security and privacy issues of applying smart contracts to crowdsourced systems and the existing solutions to address the identified security and privacy issues. Atzei et al. [6] analyze existing smart contract vulnerabilities and categorize smart contract vulnerabilities into three perspectives: programming language, virtual machine, and blockchain.

The above studies mainly focus on vulnerability detection tools. It is well known that vulnerability datasets are a decisive factor in the performance assessment of the tools. The existing survey papers, however, have not acknowledged and investigated the impact of vulnerability datasets. Besides, most of the reviews do not fully cover existing vulnerability defense and repair solutions, which are critical mechanisms for smart contract security protection apart from vulnerability detection. In order to fill the above gaps in the field of smart contract security literature surveys, we conduct a new survey from the perspectives of vulnerability data sources, vulnerability detection, and vulnerability defense.

## 3. Background

### *3.1. Smart Contracts*

The concept of smart contract predates blockchain as a way to automate the construction of contractual protocols, dating back as far as 1995 when Nick Szabo published [3]. The definition refers to a promise defined in digital form that the participants can execute on a smart contract [18]. Similar to traditional software, smart contract technologies has a life cycle, containing five phases: design, development, deployment, invocation, and destruction. Since smart contracts cannot be changed once they are deployed, they do not require a maintenance phase in the traditional sense [7]. Compared with traditional contracts, smart contracts mainly address how to ensure the validity of the contract. While the validity of a traditional contract needs to be guaranteed by an institution such as a court of law, a smart contract codes the execution procedure. Once the conditions are met after deployment, the coded procedure can be executed automatically and cannot be interfered with by humans. The main scope of its application includes privacy, security, and decentralized functions, such as decentralized financial lending and decentralized crowdfunding [19].

### *3.2. Ethereum*

Ethereum is an open-source public blockchain platform, which supports a variety of high-level programming languages, by which developers can develop any decentralized applications (DApps) [20] on Ethereum. Ethereum can perfectly integrate blockchain and smart contracts. Ethereum not only inherits the characteristics of data disclosure, non-tamperability and decentralization of blockchains but also is Turing complete [21] compared to blockchains. Smart contract code is compiled into machine code that can be executed on Ethereum through the Ethereum virtual machine (EVM), which enables it to run on Ethereum. The EVM is an entirely isolated sandbox environment, so smart contracts have only very limited access to each other [22]. Since the information in the blockchain is open and transparent, while the information of each node is synchronized. Once a smart contract is successfully deployed, each node can execute the smart contract, and everyone can publish the smart contract on Ethereum. In order to prevent attackers from releasing malicious contracts, each operation performed by smart contracts on Ethereum will generate gas consumption.

Since smart contracts usually involve money transactions, it is crucial to secure them effectively as they can cause huge losses if they have security problems and are exploited by attackers.

### *3.3. Smart Contract Security*

Smart contract development is still at the early stage of development. Smart contracts are usually written by developers and deployed on Ether. Their life cycle is similar to that of software programs, so there will inevitably be some security problems [9]. Smart contracts cannot be changed once they are deployed. Even if security problems are found, they cannot be maintained by patching or version updating like traditional software. In other words, adding a self-destruct function not only increases the risk of contract attacks but also does not recover the economic loss already caused. A comprehensive audit of smart contract security is the most effective way to eliminate the security risks of smart contracts. However, no security audit can guarantee that a smart contract is 100% problem-free. Therefore, the repair work after the discovery of security problems is equally important.

## 4. Overview Methodology

The Systematic Literature Review (SLR) method provides an in-depth and broad overview of a particular area by searching and evaluating the existing literature [23]. In this paper, SLR is selected as the research method, given that the goal of this paper is to investigate and analyze the current state of art in smart contract security and provide directions for future research. We follow the SLR guidelines to conduct the literature analysis.

### *4.1. Research Questions*

The research questions need to indicate the goal of the review article. Smart contract security is a multi-dimensional issue that should be explored from a holistic perspective. For example, the diversity of vulnerability data is an important dimension to measure the feasibility of contract security solutions. Therefore, we explore smart contract security from four perspectives: current status of smart contract security, vulnerability data sources, vulnerability detection, and vulnerability defense. We distill the four security issues and their sub-problems as follows.

1. RQ1: Security status of smart contracts. RQ1.1: What are security issues faced by smart contracts? RQ1.2: What are typical smart contract vulnerabilities?
2. RQ2: Source of smart contract vulnerability data. RQ2.1: What are existing smart contract vulnerability datasets? RQ2.2: What are the major methods for generating smart contract vulnerability datasets? RQ2.3: How many vulnerabilities are covered in the existing datasets?
3. RQ3: Security detection methods for smart contracts. RQ3.1: What are the existing vulnerability detection methods? RQ3.2: What are the advantages and disadvantages of those detection methods?
4. RQ4: Defense mechanisms against vulnerabilities. RQ4.1: What are the existing vulnerability defense methods? RQ4.2: What are the strengths and limitations of those repair methods? RQ4.3: What vulnerabilities are covered by those vulnerability repair methods?

### *4.2. Literature Search*

This section focuses on how to select the relevant literature for answering the research questions above. Two major steps, namely comprehensive search and primary screening, are included.

#### *4.2.1. Comprehensive Search*

First, we conduct a comprehensive search to find references related to the research questions. We adopt the PIO (Population, Intervention, and Outcome) principle [24] to help us identify relevant keywords and databases specifically, where population represents terms related to technology and standards, and intervention means specific issues in the field. The search is performed using Population AND Intervention, with detailed PIO information shown in table 2.

To avoid the influence of preference factors on the research results, only the characters listed in table 2 are used as our query words. We target seven scientific databases as the sources for the literature search, as shown below.

- Google Scholar(https://scholar.google.com/)
- IEEE Xplore(https://ieeexplore.ieee.org/)
- DBLP(https://dblp.org/)
- Spring Link(https://link.springer.com/)
- ACM Digital Library(https://dl.acm.org/)
- Web of Science(https://www.webofscience.com/)
- EI Compendex(https://www.engineeringvillage.com/)

These databases contain relevant scientific results published in the field of computing. To conduct our research, we first manually remove the duplicated results retrieved. Although the concept of smart contracts was proposed as early as 1992, the smart contract application platform Ether was proposed by Vitalik Buterin in 2015. Hence, the starting year of our literature collection is 2015. In addition, to ensure the completeness of our search results, we also check the related references from the retrieved relevant papers and monitor the lists of papers recently accepted by relevant top conferences.

#### *4.2.2. Literature Screening*

After the above-mentioned search steps, we retrieved hundreds of papers. However, many of them are not quite relevant to the theme of this survey. Therefore, we screen the keywords, abstracts, main contributions, and conclusions of each paper and select the papers containing smart contract security related topics in those areas.

### *4.3. Quality Evaluation*

We identify the key information of each article by scanning its keywords, abstract, contribution, method, and conclusion, which facilitates us to classify the articles. As a result, we classify the literature into the classes of vulnerability data collection, vulnerability detection, and vulnerability remediation. In addition, to ensure the quality of our selected literature, we assessed the quality of the publications based on the *Computing Research and Education (CORE)* [25] ranking, which contains five categories: A*, A, B, C and No Grade. In addition, we also adopted the journal and conference ranking released by the China Computer Federation (CCF) to evaluate the quality of the publication venues [26], and marked them in Table 3. The ranking results of the two quality assessment methods are shown in Figure 1. Smart contract vulnerability detection-related papers accounted for the most significant proportion of 85.7% (42/49), of which 11 papers were published in the high-level platform, followed by smart contract vulnerability repair-related papers proportion of 10.2% (5/49), of which four papers published in the high-level platform, smart contract vulnerability injection-related papers less, accounting for 2% (1/49), but the paper published in the high-level platform. This paper argues that the main reasons for this uneven distribution are: first, the most effective way to avoid defects in smart contracts is smart contract vulnerability detection, and accurate, comprehensive smart contract vulnerability detection can directly prevent the existence of security risks in smart contracts. Second, the development of smart contracts is relatively short, and a sound smart contract security protection mechanism requires time to accumulate. So, there are still relatively few papers on smart contract vulnerability repair. Finally, smart contract vulnerability datasets with logos require the development of complete and comprehensive vulnerability identification rules, which is difficult, so work on smart contract vulnerability injection still needs to be accumulated.

### *4.4. Paper Distribution*

We collected information (including year of publication and source) of each research paper and divides all the papers in term of conference and journal articles (see Table 3). Next, we analysed the trend of publication number each year from 2015 to 2023. In 2015, smart contract was still in its early stage. The DAO incident in 2016 causing substantial financial losses began to attract attention of researchers. In 2017, research related

Table 2: Application of PIO Principle in Smart Contract Security

| Type | Concept | Keyword |
|---|---|---|
| Population | Technology and Standards Related Terminology | Model Checking OR Static Analysis OR Symbol Execution OR Formal Validation OR Abstract Semantic OR Fuzzing |
| Intervention | Specific Behaviors and Methods | Smart Contract Attack OR Smart Contract Safety{Security} OR Contract Verification OR Contract Analysis OR Contract Vulnerability OR Code Vulnerability OR Code Safety{Security} OR Contract Repair{Patch} |

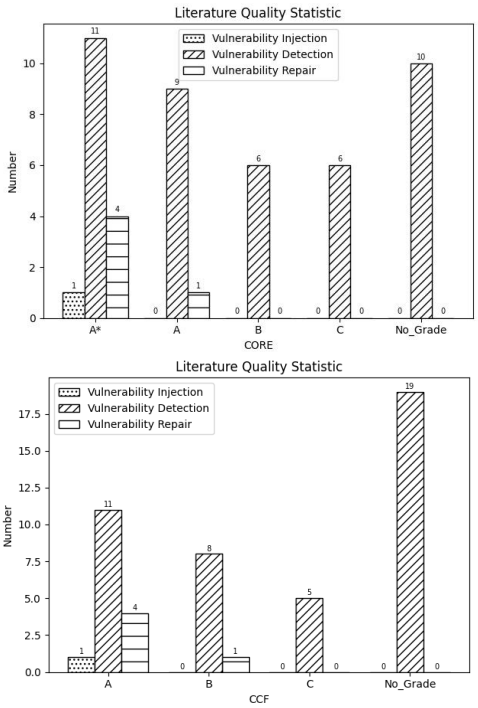


Figure 1: Literature Quality Statistic

to smart contract security began to appear. In 2018, the number of research papers on smart contract vulnerability detection increased rapidly, while experiencing a slight decrease in 2019. Starting from 2020, researchers are no longer limited to focusing on the single direction of vulnerability detection. Although the number of papers has reduced in the past two years, many high-quality papers were published.

Of the 49 papers investigated, most of the research was published in conferences, including 7 papers in top software engineering conferences and 6 in top security conferences. In addition, 3 papers were published in top software engineering journals. It is clear from the data that the amount of literature on smart contract security has been increasing. As an emerging area of research, the amount of literature is significant.

## 5. Findings

In this section, we present the major findings of the 49 articles related to the smart contract security of this paper. Table 3 presents an overview of all the articles, which includes the year, author, and a brief summary of each article. Then, we answer the four research questions based on those findings.

### *5.1. Existing Security Issues*

To answer RQ1, we found that smart contracts are designed for decentralized, secure, and trustworthy management, with features such as gas consumption mechanism, immutability, etc. These features are designed for specific application scenarios, but they also embed threats to smart contract security. Smart contracts mainly face security issues in the design phase and contract implementation phase.

In the smart contract design phase, the smart contract development language is still in the development stage, lacking a perfect specification mechanism. High-quality design documents are the prerequisite to guarantee the standard development of smart contracts. The application scenarios of smart contracts are diverse, and the requirements for designers in different application scenarios are also dynamic. In the face of such complex situations, if designers do not have a sound understanding of the requirements and design a flawed design solution, it will have a negative impact on the subsequent development work.

In the smart contract development phase, the programming ability of contract developers directly affects the quality of a smart contract. If a developer does not fully follow the design plan, it may lead to problems such as missing or incorrect functions of a contract. At the same time, developers who do not fully comply with the security programming guidelines of the contract language often write code that is not easily understood and maintained, which may also cause security problems of smart contracts.

Until now, there have been many studies on vulnerability classification frameworks. Atzei et al. [6] first classify smart contract vulnerabilities into three levels: programming language, virtual machine, and blockchain. Dika et al. [27] follow the classification of Atzei et al. and further classified the smart contract security issues into low, medium, and high-risk security levels. The Decentralized Application Security Project (DASP) summarized 10 high-risk Ethereum smart contract vulnerabilities. Chen et al. [28] define 20 smart contract flaws in terms of potential security, usability, maintainability, and reusability. Zhang et al. [29] propose a vulnerability classification framework called *JiuZhou* by extending *IEEE Standard Classification for Software Anomalies*, which summarizes 49 smart contract vulnerability types and defines their severity levels.

In this paper, we classify those smart contract vulnerabilities in terms of three levels: Solidity language, EVM virtual machine, and blockchain. We consider the vulnerability of smart

Table 3: Research Literature Summary

| Number | Year | Author | Specific Description |
|---|---|---|---|
| 1 | 2020 | Ghaleb et al | SolidiFI, an automated and systematic approach for evaluating static analysis tools for smart contracts. *ISSTA, CCF-A* |
| 2 | 2017 | Chen et al | GASPER, a model for automatically locating gas-consuming highs by analyzing the bytecode of smart contracts. *SANER, CCF-B* |
| 3 | 2018 | Nikolić et al | MAIAN, a dynamic analysis tool that uses symbolic analysis to detect three smart contract vulnerabilities. *ACSAC, CCF-B* |
| 4 | 2019 | Mossberg et al | Manticore, a dynamic symbolic execution framework, platform-independent symbolic restrictions. *ASE, CCF-A* |
| 5 | 2016 | Luu et al | Oyente, a symbolic execution detection tool for building contract control flow graphs at the bytecode level. *CCS, CCF-A* |
| 6 | 2018 | Torres et al | Osiris, an instrumentation tool that leverages symbolic execution and taint analysis at the bytecode level. *ACSAC, CCF-B* |
| 7 | 2018 | Tsankov et al | Securify, a security analyzer for smart contracts. *CCS, CCF-A* |
| 8 | 2019 | Feist et al | Slither, a static analysis framework that provides code inspection, code optimization, code understanding and code review. *WETSEB, CCF-Non* |
| 9 | 2018 | Tikhomirov et al | SmartCheck, a static analysis tool that converts source code into an XML intermediate representation for inspection. *WETSEB, CCF-Non* |
| 10 | 2018 | Kalra et al | ZEUS, a security analysis framework for smart contracts using abstract interpretation and symbolic model checking. *NDSS, CCF-B* |
| 11 | 2018 | Jiang et al | ContractFuzzer, a novel fuzzer for testing security vulnerabilities in ethereum smart contracts. *ASE, CCF-A* |
| 12 | 2021 | Torres et al | ConFuzzius, a hybrid test fuzzifier combining evolutionary fuzzy testing and constraint solving. *EuroS&P, CCF-Non* |
| 13 | 2020 | Wüstholz et al | Harvey, a grey-box fuzzy testing method for contract vulnerability mining. *FSE/ESEC, CCF-A* |
| 14 | 2018 | Liu et al | ReGuard, a dynamic analyzer for reentry errors in smart contracts. *ICSE-Companion, CCF-A* |
| 15 | 2020 | Nguyen et al | sFuzz, an adaptive fuzzing engine for EVM smart contracts. *ICSE, CCF-A* |
| 16 | 2019 | Fu et al | EVMFuzzer, a tool for detecting EVM vulnerabilities using differential fuzzy techniques. *FSE/ESEC, CCF-A* |
| 17 | 2018 | Zhou et al | SASC, a static analysis method for ethereum smart contracts. *NTMS, CCF-Non* |
| 18 | 2021 | Jiang et al | WANA, a scalable smart contract vulnerability detection tool based on Wasm bytecode symbolic execution. *QRS, CCF-C* |
| 19 | 2021 | Yu et al | ReDetect, a symbolic execution vulnerability detection tool for EVM bytecode level. *MSN, CCF-C* |
| 20 | 2020 | Wang et al | Artemis, an improved smart contract validation tool. *DSA, CCF-Non* |
| 21 | 2020 | Huang et al | EOSFuzzer, a generic black-box fuzz testing framework for detecting vulnerabilities in EOSIO smart contracts. *Internetware, CCF-Non* |
| 22 | 2020 | Ji et al | DEPOSafe, an automated detection tool for fake deposit vulnerabilities in smart contracts. *ICECCS, CCF-C* |
| 23 | 2019 | Fu et al | A symbolic execution model for multi-objective path-oriented search (MOPS) strategies based on path prioritization. *Access, CCF-Non* |
| 24 | 2018 | Tann et al | A Sequence Learning Approach to Detecting Smart Contract Vulnerabilities. *arXiv, CCF-Non* |
| 25 | 2022 | Hwang et al | CodeNet, a CNN-based vulnerability detection method. *Access, CCF-Non* |
| 26 | 2019 | Liao et al | SoliAudit, an approach to smart contract vulnerability assessment using machine learning and fuzzy testing. *IOTSMS, CCF-Non* |
| 27 | 2021 | Zhou et al | SC-VDM, a CNN-based lightweight smart contract vulnerability detection model. *DMBDA, CCF-Non* |
| 28 | 2021 | Eshghie et al | Dynamic, a monitoring framework for detecting re-entry vulnerabilities in smart contracts. *EASE, CCF-C* |
| 29 | 2021 | Lutz et al | ESCORT, a deep neural network based framework for detecting vulnerabilities in smart contracts. *arXiv, CCF-Non* |
| 30 | 2019 | Song et al | A method for detecting vulnerabilities in ethereum smart contracts using machine learning. *NSS, CCF-Non* |
| 31 | 2021 | Ashizawa et al | Eth2Vec, a machine learning-based static analysis tool for vulnerability detection in smart contracts. *BSCI, CCF-Non* |
| 32 | 2021 | Liu et al | Propose a vulnerability detection method that combines deep learning with expert models. *TKDE, CCF-A* |
| 33 | 2021 | Gao et al | SMARTEMBED, a Solidity detection method based on structural code embedding and similarity checking. *TSE, CCF-A* |
| 34 | 2020 | Zhuang et al | A method for smart contract vulnerability detection using graph neural networks. *IJCAI, CCF-A* |
| 35 | 2021 | Wu et al | Peculiar, a vulnerability detection method based on pre-training techniques and critical data flow graphs. *ISSRE, CCF-B* |
| 36 | 2022 | Mi et al | VSCL, an automated smart contract vulnerability detection framework using deep neural networks. *ICBC, CCF-Non* |
| 37 | 2020 | Gogineni et al | A Multi-Classification Technique for Learning Smart Contracts Based on AWD-LSTM Model. *IOP SciNotes, CCF-Non* |
| 38 | 2018 | Liu et al | S-GRAM, a new semantic-aware security auditing technology. *ASE, CCF-B* |
| 39 | 2022 | Huang et al | Developing a multi-task learning-based vulnerability detection model for smart contracts. *Sensors, CCF-Non* |
| 40 | 2020 | Wang et al | Contractward, a model for smart contract vulnerability detection using machine learning algorithms. *TNSE, CCF-Non* |
| 41 | 2021 | Yu et al | DeeSCVHunter, a modular and systematic deep learning framework to detect contract vulnerabilities. *IJCNN, CCF-C* |
| 42 | 2023 | Cai et al | A GNN-based vulnerability detection method for smart contracts is proposed. *JSS, CCF-B* |
| 43 | 2022 | Zhang et al | ASGVulDetector and BASGVulDetector to detect vulnerabilities from source code and bytecode perspective. *Future Internet, CCF-Non* |
| 44 | 2022 | Ye et al | Vulpedia, a detection method based on detection rules composed of vulnerability signatures. *JSS, CCF-B* |
| 45 | 2021 | Nguyen et al | SGUARD, a high precision overlay for smart contract vulnerability detection and remediation. *S&P, CCF-A* |
| 46 | 2021 | Rodler et al | EVMPATCH, a framework that supports automatic repair of contract errors based on bytecode rewriting. *USENIX Security, CCF-A* |
| 47 | 2020 | Yu et al | SCRepair, an automated smart contract repair algorithm using genetic programming search. *TOSEM, CCF-A* |
| 48 | 2020 | Zhang et al | SMARTSHIELD, an automatic bytecode correction method for fixing unsafe cases of unsafe code patterns in smart contracts. *SANER, CCF-B* |
| 49 | 2020 | Jin et al | Aroc, a generic smart contract fixer that automatically patches deployed contracts that are vulnerable to attacks. *TSE, CCF-A* |

Table 4: Smart Contract Vulnerability Types and Threat Levels

| Threat Level | Vulnerability Type |
|---|---|
| Solidity Language | Reentrancy |
| | Integer Overflow and Underflow |
| | Unchecked Request |
| | Unhandled Exception |
| | Unprotected Selfdestruct Instruction |
| | Uninitialized State Variables |
| | Locked Ether |
| EVM | Short Address Attack |
| | Storage Overlap Attack |
| Blockchain | Timestamp Dependency |
| | Transaction Order Dependency |
| | Replay Attack |

```
1  contract Reentrancy {
2    mapping (address => uint) public credit;
3    function donate(address to) payable public{
4      credit[to] += msg.value;
5    }
6    function withdraw(uint amount) public{
7      if (credit[msg.sender]>= amount) {
8        require(msg.sender.call.value(amount)());
9        credit[msg.sender]-=amount;
10     }
11   }
12   function queryCredit(address to) view public returns(uint){
13     return credit[to];
14   }
15 }
```

Figure 2: Reentrancy Vulnerability

contracts while taking into account the generality of vulnerabilities. The remaining of this subsection will analyze twelve major security vulnerabilities at these three levels in detail.

### *5.1.1. Solidity language level*

Solidity is a Turing-complete high-level language, which is the main language used by developers to write smart contracts. The security threats brought to smart contracts at the language level mainly originate from two aspects, i.e., design flaws of the Solidity language, and mistakes of developers during smart contract coding.

**Reentrancy Vulnerability**. This vulnerability also appears when contracts call each other, and its implementation principle is similar to the recursive call of a function [30]. An attacker takes advantage of a developer's negligence to make a program execute malicious code designed by the attacker repeatedly in a transaction until the gas is exhausted, thus causing huge financial losses. The root cause of the famous Ethereum hard fork DAO incident is the reentrancy vulnerability. As shown in Figure 2, the attacker attacks the contract to initiate a transaction by repeatedly calling the withdraw function until the victim's account balance is 0 or gas is depleted.

**Integer Overflow and Underflow Vulnerability**. When there is a computation operation on an integer variable in a statement or expression, an integer overflow and underflow occurs if the developer does not pay attention to the bounding value of the variable, resulting in a value that exceeds the upper

```
255  function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
256    uint cnt = _receivers.length;
257    uint256 amount = uint256(cnt) * _value;
258    require(cnt > 0 && cnt <= 20);
259    require(_value > 0 && balances[msg.sender] >= amount);
260
261    balances[msg.sender] = balances[msg.sender].sub(amount);
262    for (uint i = 0; i < cnt; i++) {
263        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
264        Transfer(msg.sender, _receivers[i], _value);
265    }
266    return true;
267  }
```

Figure 3: Integer Overflow and Underflow Vulnerability

or lower bound of the variable type and is different from the value expected by the developer [31]. If the developer fails to check that variable's final result before subsequent operations, financial losses can occur. As shown in Figure 3, there is no overflow judgment for *amount* on line 257. If an attacker makes *amount* overflow, then the attacker can bypass the code used to check the account balance on line 259. Through this vulnerability, an attacker can transfer a large number of tokens at a low cost.

**Unchecked Request Vulnerability**. This type of vulnerability occurs in connection with external calls to smart contracts [29]. If an external control calls data or addresses, an attacker can arbitrarily specify the call address, function, and parameters. Suppose an external transfer request is initiated without careful checksumming of the contract. In that case, an attacker could use this type of vulnerability to make the smart contract perform functions that the developer does not expect [6]. Such vulnerabilities could affect the contract's functionality, allowing even an unauthorized user to transfer tokens from the account and potentially cause financial losses.

**Unhandled Exception Vulnerability**. The vulnerability arises from calls between Ethernet smart contracts, which send tokens using statements such as *<address>.send*, *<address>.call.value*, or calling methods of other contracts using statements such as call. If there is an exception during the call, such as running out of gas, the call will terminate and roll back the state, returning false to the calling contract [32]. Suppose the caller uses a lower-level call statement and does not check this return value. In that case, the subsequent operation will continue, resulting in a method that is not implemented as expected by the developer.

**Unprotected Selfdestruct Instruction Vulnerability**. Because smart contracts cannot be deleted once deployed, the Solidity language provides a destruction function that requires the user to introduce a suicide function when developing a contract. Once a contract is faulty, property loss can be avoided by calling the suicide function to destroy the contract and transfer the Ether to the specified address. However, suppose a smart contract with a suicide function lacks permission control. In that case, any user can call the suicide function to kill the contract and transfer the contract's Ether to an address of their designation [33]. This error can affect the functionality of the contract and cause financial loss.

**Uninitialized State Variables Vulnerability**. The occurrence of this vulnerability is related to the state variables of a smart contract. If the smart contract does not initialize the state variables during the development phase, these variables will be

automatically assigned with default values [29]. If these uninitialized state variables can be accessed directly, these variables will point to unknown storage contents. This causes that the contract functions inconsistently compared to expectations and increases the risk of vulnerabilities.

**Locked Ether Vulnerability**. Smart contracts can be used to manage digital assets on a blockchain. In contrast to accounts on Ethereum, where transactions are made through private keys, smart contract accounts do not have their private keys and can only manage assets through code. Once a developer writes only the function of receiving ether but not the function of sending ether during the development of a smart contract, the ether in the contract will be locked permanently [34]. All the ether transferred into this contract cannot be transferred out, thus causing financial losses.

### *5.1.2. Virtual Machine Level*

The Ethereum virtual machine is the executor of the compiled bytecode of smart contracts. The virtual machine's design specification and bytecode are defined in the Ethereum Technical Yellow Book [35]. The virtual machine level brings smart contract security threats from two primary sources, i.e., the defects of the virtual machine's operation mechanism, and the security problems caused by the non-standard operation of different Ethereum platforms in the process of implementing the virtual machine.

**Short Address Attack Vulnerability**. The vulnerability arises from the auto-completion operation of the Ethernet virtual machine. The actual operation of smart contracts relies on the Ethernet virtual machine [14]. The input parameters required for the operation of functions in the contract will appear in the virtual machine in the form of fixed-length bytecodes. There are some pitfalls in this way of handling, as the function input parameters may be subject to short address attacks when they include address-based parameters with insufficient bits [6]. For example, when using a transfer function with the address and transfer amount as input parameters, an attacker can carefully construct a short address input with insufficient bits to make the auto-completion of the transfer amount exponentially larger, resulting in a large financial loss.

**Storage Overlap Attack Vulnerability**. The data in a smart contract is shared in the storage space. Different clients do not have the same implementation for the virtual machine, so there will be different running results of the contract data during the invocation [11]. Since the blockchain only synchronizes the transaction information between contracts and does not synchronize the runtime results, the difference in the runtime results is not easily detected. This type of vulnerability can lead to data overwriting in the contract, which may break the contract's functionality [36].

### *5.1.3. Blockchain Level*

Smart contracts achieve decentralization and tamper-proof for the blockchain, which provides a platform for the implementation of smart contracts. However, the features of the blockchain also bring certain security threats to smart contracts.

```
3   contract Timestamp{
4       uint public lastBlockTime;
5       function lucky() public payable{
6           require(msg.value == 100 wei);
7           require(lastBlockTime != block.timestamp); //block.timestamp
8           lastBlockTime = block.timestamp;
9           if(lastBlockTime % 10 == 5){
10              msg.sender.transfer(address(this).balance);
11          }
12      }
13  }
```

Figure 4: Timestamp Dependency Vulnerability

**Timestamp Dependency Vulnerability**. As the name implies, the vulnerability stems from the global variable of the smart contract, i.e., the timestamp. The timestamp of the block to which the smart contract belongs is available to developers as a global variable [27]. The timestamp is determined by the system at the time of mining, allowing for a deviation of 900 seconds so that miners can control the timestamp to some extent. If the functionality implemented by the smart contract changes with the timestamp, then a malicious miner can control the outcome of the smart contract to some extent to make it meet his/her needs. As shown in Figure 4, *block.timestamp* is used in the *Timestamp* contract as a condition for performing key operation. The contract decides whether to win by the timestamp of the block where the transaction is sent. Only the first transaction in each block is allowed to win. If the lowest bit of the block timestamp is 5, the sender of this transaction will win the prize. Since the miner has the right to set the timestamp anywhere from 0 to 900s, the miner can know in advance whether the lowest bit of the timestamp of the next block is 5 or not and then manipulate the result of the prize.

**Transaction Order Dependency Vulnerability**. This vulnerability is a security flaw that relies on the order of transaction execution and causes differences in transaction results [37]. Since transactions are packaged and sent to the transaction pool for storage, the miner nodes will select transactions from the transaction pool and put them into the newly generated blocks according to certain rules. The transactions and transaction orders selected by different miners are not fixed, so the order of transactions executed by the contract is uncertain. Miners generally give priority to transactions with high gas content and put them into the new blocks. An attacker can make such an attack by increasing the gas cost. The attacker can make his/her transactions written into the block before other transactions by increasing the gas cost, which affects the final result by influencing the contract's execution order.

**Replay Attack Vulnerability**. Because there is a hard fork of the public chain of Ethernet due to a major security event, many chains exist on Ethernet now [38]. As a result, if an attacker can predict the validation value used by a transaction, the attacker can replay that transaction on another chain [29]. Such vulnerabilities could lead to an attacker having the ability to replay the transaction on another chain, causing financial losses to the owner of the smart contract.

### 5.2. Existing Vulnerability Sources

To deal with the security threats in smart contracts, researchers have been analyzing contract vulnerabilities and developing various detection tools. However, how to evaluate the performance of these tools is also a problem worthy of study. An ideal evaluation benchmark can discover the blind spots of vulnerability detection tools, improve the performance of detection tools and broaden their usage scenarios, which is required by vulnerability analysis. To answer RQ2, we found that the existing construction methods of evaluation benchmarks mainly fall into three categories: manually constructing vulnerabilities, collecting vulnerabilities, and automatically generating vulnerabilities based on vulnerability injection technologies.

#### 5.2.1. Manually Constructed Vulnerabilities

To address the problem of lack of open source data, many studies choose to use manually constructed vulnerability datasets for benchmark assessments. A manually written benchmark dataset for evaluation ensures that each test sample contains the required vulnerability information and provides reasonable assurance that the scope of the assessment covers all the required vulnerabilities. However, over-reliance on manually generated data inevitably makes the evaluation results less objective and the models less capable of identifying diverse real vulnerabilities. In addition, manual vulnerability construction often has the problem of poor scalability and consumes labor and resources once the data needs to be extended.

#### 5.2.2. Manually Collected Vulnerability

Large-scale collection of vulnerability data in real environments and construction of objective vulnerability datasets as evaluation benchmarks are also a research hot spot. The method of manual vulnerability data collection ensures the authenticity of vulnerability data compared to manually constructed vulnerability data. However, subjectivity cannot be ignored during the collection process. In addition, not every kind of vulnerability data has an extensive distribution in the real environment. Relying entirely on manual collection and construction of datasets does not guarantee the adequacy and diversity of vulnerability data. A few researchers provide smart contract datasets with vulnerability labels. For example, *SmartContractSecurity* [39] provides 122 smart contracts containing 33 vulnerability categories. However, *SmartContractSecurity* does not classify these vulnerabilities. *Crytic* [40] provides a dataset containing 12 vulnerability categories. The NCC Group [41] proposes a classification of smart contract vulnerabilities based on the Decentralized Application Security Project (DASP) of smart contract vulnerability classification [42]. They provide 69 smart contracts containing 10 vulnerability categories. Apart from the aforementioned labelled datasets, Zhuang et al. [43] collect 42,000 unlabelled Ethereum smart contracts.

#### 5.2.3. Vulnerability Injection Method

To ensure that the vulnerability samples used for evaluation are sufficiently realistic, objective, and comprehensive in coverage, some studies have proposed to automate the construction of vulnerability datasets based on vulnerability injection and evaluate the performance of detection tools accordingly. In terms of technical implementation, vulnerability injection can be further divided into two main categories: finding sensitive locations in program code through static analysis techniques and using them to construct an objective vulnerability dataset by injecting vulnerability fragments in these locations; and inserting vulnerabilities into the source program by identifying situations where user-controlled inputs may trigger out-of-bounds reads and writes. Compared with the manual construction of vulnerabilities and manual collection of vulnerabilities, the method of constructing vulnerability datasets based on vulnerability injection removes the over-reliance on manual work. It can more easily construct vulnerability samples on a large scale to provide objective and realistic performance evaluation of vulnerability detection tools. At the same time, part of the vulnerability injection techniques also enable the customization of vulnerability samples to match the evaluation needs of various vulnerability detection tools flexibly. Asem et al. developed the first vulnerability injection tool for Ethereum smart contracts, **SolidiFi** [44]. This tool introduces targeted security vulnerabilities by injecting predefined vulnerability fragments into all potential locations of a smart contract. This tool can inject seven types of smart contract vulnerabilities, including reentrancy vulnerabilities.

#### 5.2.4. Summary

An ideal evaluation benchmark can discover the blind spots of vulnerability detection tools and improve the performance of detection tools, which is demanded by smart contract vulnerability analysis. Existing evaluation benchmark construction methods are mainly divided into three categories: manually constructed vulnerabilities, collected vulnerabilities, and automatically generated vulnerabilities based on vulnerability injection techniques. Their strengths and weakness are summarized as follows.

- Manually constructed smart contracts contain vulnerabilities with high diversity. However, this category of approaches may make the data set less objective.
- Collected vulnerability data ensures the authenticity of the constructed data sets. Its defect is that it cannot guarantee the sufficiency and diversity of the vulnerability data.
- Vulnerability injection based vulnerability data sets reduce the excessive dependence on manual work. The existing vulnerability injection methods mostly base on artificially formed vulnerability fragments. Although the volume of the data set can be guaranteed, the authenticity of the generated smart contract vulnerabilities cannot be insured.

### 5.3. Contract Security Detection Methods

Sections 5.1 and 5.2 describe the security issues faced by smart contracts and the existing vulnerability data sources. Understanding the main security issues faced by smart contracts can help developers develop practical detection tools. In addition, rich contract vulnerability data can help users effectively evaluate detection tools' performance. To answer RQ3, we

divide contract security detection tools into conventional and deep learning-based detection methods. Conventional detection methods are divided into symbolic execution, formal verification, and fuzzing. We will provide detail below. We analyze and introduce different types of methods and compare their advantages and disadvantages.

*5.3.1. Symbolic Execution*

The principle of symbolic execution is to abstract the external input into symbolic values, abstract the program in a smart contract into an execution tree, and then traverse the execution tree based on the external input values and the semantics of the program. The main idea of symbolic execution is to convert uncertain inputs into symbolic values during execution to drive the execution of the program. Symbolic execution is also divided into two types: static symbolic execution and dynamic symbolic execution.

In terms of static execution, **Oyente** [45] is the first static analysis detection tool proposed. Oyente takes the bytecode of a smart contract and the state of Ether as input and explores the control flow graph information of the contract during symbolic execution, and performs vulnerability detection by path constraints and other information. This paper introduces Oyente to detect four types of vulnerabilities: reentrancy, conditional contention, timestamp dependency, and unhandled exceptions, and the authors subsequently supplement the open source code with code for integer overflow vulnerabilities. The types of vulnerabilities detected by Oyente are not comprehensive. However, as the first tool available for smart contract vulnerability detection, it provides sound foundation for subsequent research. **Osiris** [46], an improved version of Oyente, is proposed by Torres et al. to detect integer-like vulnerabilities in smart contracts using symbolic execution methods, including security issues such as integer overflow, integer underflow, and value truncation due to improper type conversion. Chen et al. [47] propose a static analysis tool **GASPER** based on symbolic execution, which automatically locates contracts with high gas consumption for analysis by analyzing smart contracts at the bytecode level. This tool can automatically discover 3 types of gas consumption patterns, i.e., SLOAD (load a byte from memory), STORE (save a byte to memory) and BALANCE (get account balance operation). **Mythril** [48] is a symbolic execution engine proposed by Consensys et al. This approach combines taint analysis and control flow inspection on top of symbolic execution to allow vulnerability analysis of contracts at the bytecode level. Mythril can be used to detect 14 types of vulnerabilities such as reentrancy, integer overflow and underflow, and timestamp dependencies. **WANA** [49] proposed by Jiang et al. is a generic symbolic execution engine for Wasm bytecode, which can support vulnerability analysis of EOSIO smart contracts. **ReDetect** [50] is a symbolic execution-based detection tool proposed by Yu et al. for detecting reentrancy vulnerabilities in smart contracts at the EVM bytecode level. **Artemis** is an improved smart contract validation tool [51]. Artemis is built on the Oyente symbolic execution framework. To support the detection of new types of vulnerabilities, Artemis extends its vulnerability detection module to support the analysis of new vulnerability patterns, which can be used to detect four types of vulnerabilities including dangerous delegate calls. **DEPOSafe** [52] is an automated detection tool for false deposit vulnerabilities in smart contracts. DEPOSafe includes both symbolic execution-based analysis and dynamic verification based on behavioral modeling. DEPOSafe feeds the bytecode of a smart contract and its contract address through a pipeline consisting of a static detector and dynamic verifier components to generate security reports.

Dynamic symbolic execution, also known as hybrid symbolic execution, improves detection accuracy by generating constrained program inputs through identified paths. Nikolić et al. propose a tool called **MAIAN** [33], which detects vulnerabilities by tracing the execution path of a contract via analyzing multiple invocations during its lifecycle (each run of a contract is called a single invocation). MAIAN can detect three types of vulnerabilities: greedy contracts, self-destructive contracts, and prodigal contracts. **Manticore** is a dynamic symbolic execution framework [53] that implements a platform-independent generic symbolic execution engine that makes no assumptions about the underlying execution model and operates and manages the program based on the lifecycle of the state. Compared to other detection tools that analyze a single contract, Manticore supports the analysis of multiple contracts simultaneously. Fu et al. [54] propose a multi-objective path search (**MOPS**) strategy based on path priority. First, it obtains the code regions with security threats and their critical paths by improving Mythril. Then, a multi-objective-oriented path search strategy is proposed to guide dynamic symbolic execution to cover critical paths quickly. Finally, security rules are described and corresponding detection logic is proposed for different vulnerability classes.

*5.3.2. Formal Verification*

Formal verification is an effective technique to verify that a program conforms to predefined design properties and security specifications. Traditional verification techniques ascertain the logic of programs and code by describing them in a logical or descriptive language. Next, these techniques apply mathematical logic proofs to reason about their actual behavior to test whether the program meets the functional requirements of the intended design.

**ZEUS** [55] is an automated formal verification tool for smart contracts. Zeus translates Solidity source code into the LLVM intermediate language and uses XACML to write verification rules on top of it. It further uses the SeaHorn verifier for formal verification. Zeus designs five security vulnerability detection rules that can determine the security of the target program in the process of formal verification. **Securify** [56] is a security analyzer for the bytecode level of smart contracts. Securify obtains semantic information from the bytecode of smart contracts and describes the semantic facts in Datalog syntax. After inferring the semantic information, Securify checks it against the predefined security property rules. The security attribute rules are divided into obedience mode and violation mode, and the security of the contract is checked by matching the semantic information with the security attribute rules.

*5.3.3. Fuzzing*

Fuzzing is a popular and effective software testing technique that detects anomalies by feeding a large amount of unexpected data to a smart contract for potential vulnerability discovery. Compared with the complex design of symbolic execution and formal verification methods, fuzzing is simple and efficient. It helps uncover more profound vulnerabilities in smart contracts. It is tested during operation of smart contracts.

**ContractFuzzer** [34] is the first framework applying fuzz testing techniques to smart contracts for vulnerability detection. It analyzes the ABI specification of smart contracts, generates inputs that match the syntax of the contract invocation under test, and defines the characteristics of different defects for detecting contract defects. ContractFuzzer can be used to detect seven types of vulnerabilities, including re-entry, timestamp dependency, transaction order dependency, etc. Among them, for the re-entry vulnerability, ContractFuzzer designs a specific attack contract and detects it by invoking the specific attack contract to the contract under test, which in turn triggers the re-entry vulnerability of the contract. **Echidna** [57] is a smart contract fuzzy testing framework published by Trail of Bits, which performs analysis and fuzzy execution in the smart contract source code, and fuzzy tests the contract under test by generating random transaction data that meet the contract invocation specification. **ConFuzzius** is a hybrid test fuzzifier combining evolutionary fuzz testing and constraint solving proposed by Torres et al. [58]. **Harvey** is a grey-box fuzzy testing method for smart contract vulnerability mining proposed by Wüstholz et al. [59]. Harvey generates simple call sequences by obtaining the dependencies of different functions seen on global variables to improve the impact of sequences on program coverage. **ReGuard** [30] is a dynamic analyzer for reentrancy errors in smart contracts. ReGuard uses fuzzy test-based techniques to generate random and diverse transaction data as possible attacks. ReGuard then dynamically identifies potential re-entry vulnerabilities in smart contracts by logging critical execution traces. **sFuzz** [60] is a fully automated engine for testing against smart contracts . sFuzz generates smart contract execution traces through fuzzy testing and uses vulnerability analysis pattern analysis to discover potential vulnerabilities in a contract. **EOSFuzzer** [61] is a generic black-box fuzzy testing framework for detecting vulnerabilities in EOSIO smart contracts. EosFuzzer consists of four parts: fuzzy input generator, fuzzy executor, Wasm VM Instrumentation, and vulnerability detection engine. **EVMFuzzer** [62] is the first tool to detect EVM vulnerabilities using differential fuzzy testing. The core idea of EVMFuzzer is to continuously generate seed contracts and make them available to the target EVM and the benchmark EVM to find as many inconsistencies between execution results as possible and eventually discover output cross-references of vulnerabilities.

*5.3.4. Other Traditional Technology*

In addition to the symbolic execution, formal verification, and fuzzy testing techniques mentioned above, program analysis and taint analysis are often adopted for smart contract vulnerability detection. Program analysis is mainly divided into static program analysis and dynamic program analysis. Static program analysis specifically analyzes a program through its control flow and data flow information, while dynamic program analysis requires further access to a program's operation information. Taint analysis is a unique program analysis technique used to achieve more accurate program analysis by labeling critical data and tracing its flow direction.

**Slither** [36] is a static analysis framework that provides code detection, code optimization, code understanding, and code review. Slither performs lexical and syntactic analysis at the source code level of smart contracts. It uses abstract syntax trees to generate inheritance graphs, control flow graphs, and contract expressions and creates an intermediate language called SlitherIR. This intermediate language implements all static program analysis work at the intermediate language level, which helps the analysis framework to extend to different high-level languages and types. **SASC** [63] is a static program analysis tool for smart contract vulnerability detection. SASC generates a topology graph by performing syntactic topology analysis through smart contract invocation relationships and then identifies potential security risks in a contract by marking the location of logical risks on the topology graph. **SmartCheck** [64] is also a static analysis tool for smart contracts. SmartCheck converts smart contract source code into an XML-based parse tree as an intermediate representation (IR) and performs smart contract vulnerability checking using XPath schema queries on the IR.

Apart from the aforementioned three tools, other tools such as Oyente [45], Mythril [48], etc. Combine taint analysis with different data flow analysis techniques to improve the tools' vulnerability detection accuracy.

*5.3.5. Deep Learning-based Approaches*

Deep learning is a branch of artificial intelligence that uses algorithms for autonomous learning of data with the ability to improve itself. Deep learning models have been regarded as black boxes, where the user is entirely unaware of how the model learns. There is no way to intervene in the model's output manually. Applying deep learning techniques to smart contract vulnerability detection is a trendy topic.

Tann et al. [65] use a sequence learning approach to detect smart contract vulnerabilities, which is the earliest approach to applying deep learning in the area of smart contract vulnerability detection. Liao et al. [66] propose a vulnerability detection method called **SoliAudit**. SoliAudit uses machine learning and fuzzy testing for smart contract vulnerability assessment. Gogineni et al. [67] propose a multi-classification technique based on Average Stochastic Gradient Descent Weight-Dropped LSTM (AWD-LSTM) models for learning smart contracts by using two neural network models to learn the semantic information of the input data and classify them. Gao et al. [68] propose a Solidity code checking method named **SmartEmbed** based on code embedding and similarity checking, which can be used for similar contract code detection, error detection, and contract verification. Zhuang et al. [43] propose a degree-free graph convolutional neural network (DR-GCN) and a temporal message propagation network (TMP). The overall flow

chart of the method is shown in Fig 5. Firstly, the control flow and data flow information is extracted from the smart contract code, followed by extracting representative key function calls or variables as nodes and call relationships between functions as edges and then normalizing the proposed graph. This method supports reentrancy vulnerabilities, timestamp-dependent vulnerabilities, and infinite loop vulnerabilities. Zhou et al. [69] propose a lightweight convolutional neural network (CNN)-based smart contract vulnerability detection model (SC-VDM) that automatically detects vulnerabilities in smart contracts on lightweight computers without expert knowledge. Eshghie et al. [70] propose a monitoring framework named **Dynamit** for detecting reentrancy vulnerabilities in smart contracts. Dynamit classifies transaction data by obtaining feature information from transaction data and using a machine learning model. Wang et al. [71] propose a smart contract vulnerability detection model (Contractward) using machine learning that can be used to detect six types of smart contract vulnerabilities, including reentrancy vulnerabilities. Song et al. [72] propose a model for detecting smart contract vulnerabilities using machine learning techniques. Ashizawa et al. [73] propose a static analysis tool, **Eth2Vec**, which learns smart contract code by bytecode, assembly code, and abstract syntax trees to identify smart contract vulnerabilities. Mi et al. [74] propose an Automating Vulnerability Detection in Smart Contracts with Deep Learning (**VSCL**) model that converts bytecode to sequence code by decompiling. It then generates new sequences using control flow graphs and depth-first search algorithms. Finally it uses deep neural networks for vulnerability analysis and detection. Hwang et al. [75] propose a vulnerability detection method called **CodeNet**. Feature extraction by converting smart contract bytecodes into images and inputting the extracted images into a CNN model for vulnerability detection. It can detect four types of vulnerabilities, including reentrancy vulnerabilities. Lutz et al. [76] propose the first deep neural network (DNN)-based smart contract vulnerability detection framework, **ESCORT**, which supports lightweight migration learning for invisible security vulnerabilities. Wu et al. [77] propose a method, **Peculiar**, to detect smart contract vulnerabilities based on critical data flow graphs using pre-training techniques, which is mainly used to detect re-entry vulnerabilities. Liu et al. [78] propose a linguistic model-based contract vulnerability prediction technique, **S-GRAM**, which predicts potential vulnerabilities of contracts by serializing contract information and analyzing it using statistical linguistic models. Yu et al. [79] design a modular and systematic deep learning vulnerability detection framework, **DeeSCVHunter**, which contains four preprocessing modules, embedding, training, and evaluation to detect reentrancy vulnerabilities and timestamp-dependent vulnerabilities. Zhang et al. [80]. propose two static analysis methods, **ASGVulDetector** and **BASGVulDetector**, to detect vulnerabilities in Ethereum smart contracts from the source code and bytecode perspectives. Ye et al. [81]. propose a detection method called **Vulpedia** by extracting structured program features from vulnerable and non-vulnerable contracts as vulnerability signatures and constructing a systematic detection method based on detection rules composed of vulnerability signatures.

The above methods usually perform feature learning for a single representation of information. To learn a more comprehensive set of smart contract vulnerability features, Liu et al. [68] propose combining graph neural networks with custom expert knowledge for smart contract vulnerability detection, which can be used to detect reentrancy vulnerabilities, timestamp-dependent vulnerabilities, and infinite loop vulnerabilities. Huang et al. [82] propose a multi-task learning-based smart contract vulnerability detection model, which improves the detection capability of the model by setting auxiliary tasks to learn more directional vulnerability features. Cai et al. [83] propose a GNN-based approach for smart contract vulnerability detection. A graphical representation of smart contract functions is constructed by combining an abstract syntax tree (AST), a control flow graph (CFG), and a program dependency graph (PDG). Program slicing is employed to normalize the graph. Finally, a bidirectional gated graph neural network model with a hybrid attention pool is used to identify potential vulnerabilities in smart contract functions.

*5.3.6. Summary*

Vulnerability detection is one of the most active research directions in smart contract security. It is well acknowledged as one of the most effective ways to prevent contracts from being attacked [36]. Existing smart contract vulnerability detection methods are mainly divided into conventional detection methods and deep learning-based detection methods. Among the conventional detection methods, fuzz testing, symbolic execution and formal verification are the three most commonly used techniques. Most existing literature focuses on fuzz testing, symbolic execution, formal verification and deep learning-based detection methods.

- Fuzzy testing techniques rely on the runtime information of smart contract programs. The targeted vulnerabilities must be path reachable to make the vulnerability detection more accurate. The challenge lies in how to generate better input to improve the coverage of a program.
- Symbolic execution techniques can explore the execution path of a program more precisely by collecting and solving constraints. Such techniques can even analyze the dependencies between transactions to solve the appropriate sequence of transactions. However, this type of methods can be stranded by contracts with complex constraints and long sequences of transactions.
- Formal verification is capable of analyzing more semantic information and ensuring that a contract program matches the targeted design. Nevertheless, custom vulnerability detection modeling relies on knowledge of domain experts, and the vulnerabilities detected are not necessarily realistic.
- Deep learning-based detection methods do not rely on experts to develop detection rules, which have higher scalability in comparison to conventional approaches. However, since users usually treat deep learning models as

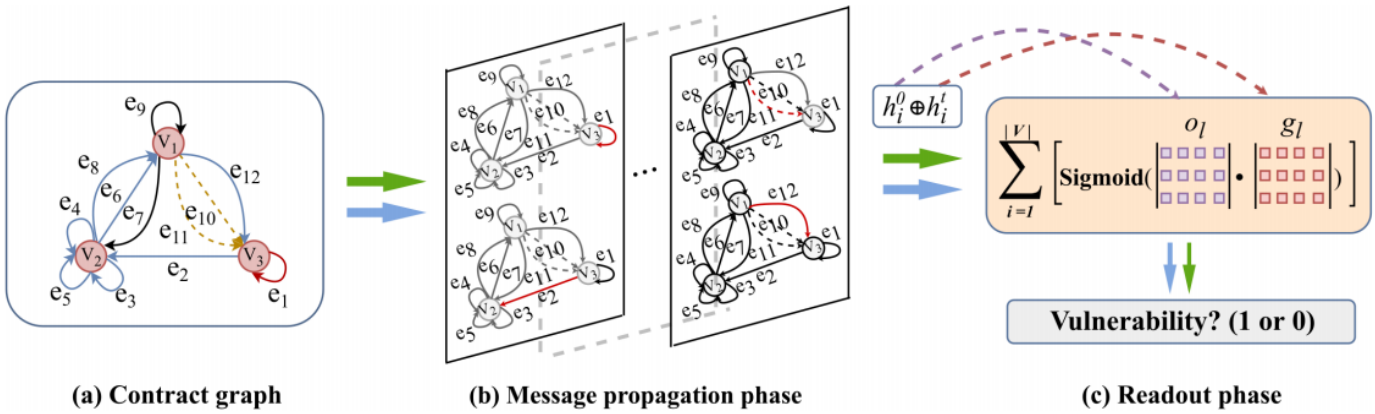


Figure 5: Workflow of TMP

black boxes, they are completely unaware how the model learns and there is the problem of poor interpretability.

### 5.4. Vulnerability Defense Method

Smart contract vulnerability defense is an important part of smart contract security. Due to the immutability of smart contracts, traditional software vulnerability repair methods cannot be applied to smart contracts. This also makes it challenging to repair smart contract vulnerabilities. In response to RQ4, we mainly divide smart contract vulnerability defense methods into two aspects: security programming and vulnerability repair.

#### 5.4.1. Secure Programming

Since smart contracts cannot be modified once being deployed on Ethereum, it becomes especially important to write more secure code that does not contain vulnerabilities. The existing work is divided into two main classes: developing a more secure third-party library for Solidity, and developing a more secure language for writing smart contracts. Both of them reduce security risks by advancing the quality of smart contract code.

OpenZeppelin has developed a variety of third-party code libraries [2] for Solidity, which include ERC standard tokens, access rights control, secure arithmetic runs, etc. Smart contract developers can simply import these code bases during the contract development process with these libraries. One of OpenZeppelin's most famous third-party libraries is **Safemath**, which can be referenced in contracts to prevent integer overflow vulnerabilities when performing arithmetic operations.

**Vyper** [84] is a Python-influenced programming language tailored specifically for smart contract development. The Vyper language does not provide features such as Modifiers, Class inheritance, Function overloading, Infinite-length loops, etc. which can easily cause contract security problems. The Vyper language adds Bounds and overflow checking, support for signed integers and decimal fixed-point numbers, and other features to improve the security of smart contracts. Digital Asset Modeling Language (**DAML**) [85] is a high-performance programming language for developing and deploying distributed applications in a blockchain environment. The DAML language has a variety of built-in modules that developers can call by simply importing the modules they need, avoiding input errors that developers make in the process of writing code. DAML language has a variety of built-in modules that developers can call by simply importing the modules they need, avoiding input errors that developers make while writing code. In addition, the DAML language provides a locking mode where only designated users can lock assets through active and authorized operations. When a contract is locked, some or all of the choices specified on that contract may not be executed, which improves the security of smart contracts.

#### 5.4.2. Vulnerability Repair

At the smart contract source code level, Yu et al. [86] propose the first generic automated contract repair method, **SCRepair**, which uses a parallel genetic repair algorithm. Given a smart contract with security vulnerabilities and a test suite, this tool is able to repair the given vulnerabilities by performing a random search on the contract. In addition, taking into account the gas consumption, the generated patches are sorted by gas and used to generate gas-optimized security contracts. Nguyen et al. [87] develope a method, **SGUARD**, to automatically transform smart contracts. SGUARD first collects symbolic execution traces of smart contracts, and then statically analyzes the collected traces to identify potential vulnerability types in the contracts. Next, based on the contract's AST, it identifies the source code corresponding to the vulnerability and apply a specific fix pattern for each vulnerability type to fix the contract vulnerability.

At the smart contract bytecode level, Zhang et al. [88] propose a bytecode correction system called **SMARTSHIELD** to fix three smart contract vulnerabilities. The workflow of this method is shown in Figure 6. SMARTSHIELD first analyzes the AST and EVM bytecodes of the contract to extract bytecode-level semantic information. Next, based on the extracted semantic information, it fixes the insecure control flow and data operations. Finally, it generates the corrected EVM bytecode and a fix report. Rodler et al. [89] propose a frame-

[2] https://docs.openzeppelin.com/contracts/4.x/

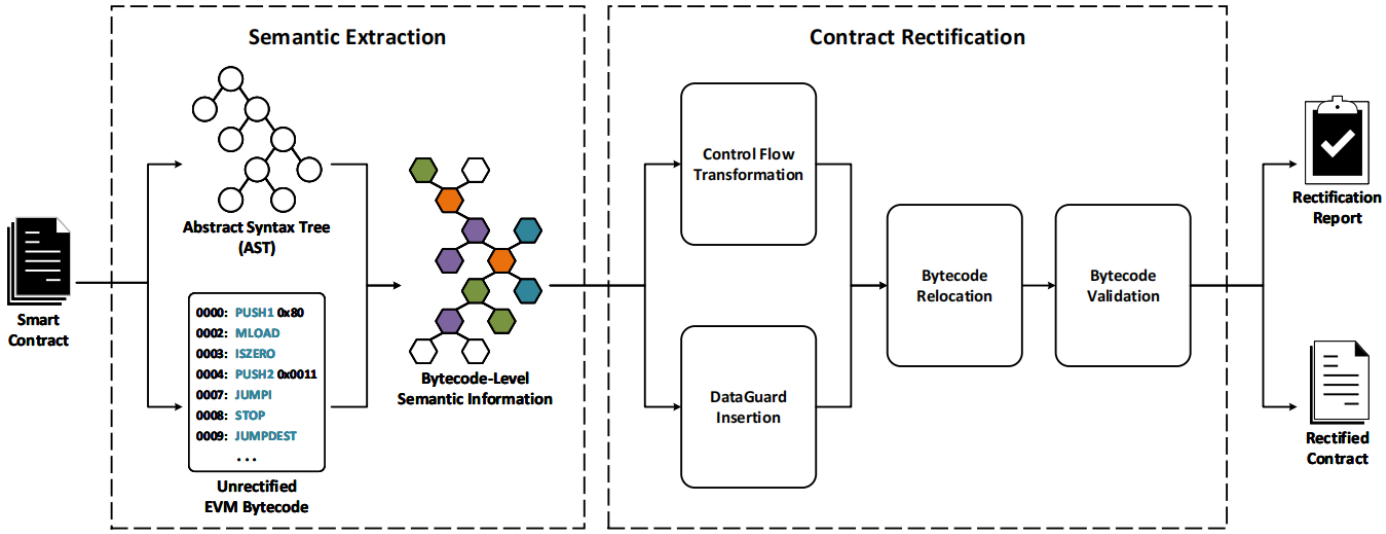


Figure 6: Workflow of SMARTSHIELD

work called **EVMPATCH** that can immediately and automatically patch faulty smart contracts. This framework consists of four main components: a vulnerability detection engine, a bytecode rewriter, a patch testing component, and a contract deployment component. EVMPATCH utilizes a bytecode rewriter to ensure that patches are minimally invasive and that newly patched contracts are compatible with the original contract. It is verified that EVMPATCH can fix both integer overflow and permission control flaws.

The above approaches all target vulnerability fixing for undeployed off-chain smart contracts. In contrast, Jin et al. [90] propose a generic smart contract fixer named **Aroc** that patches vulnerable deployed on-chain contracts. Aroc first generates a patch contract containing security rules based on the fixed template and deploys it to the blockchain, Next, the contract with the security vulnerability is bundled with the patch contract so that subsequent transactions need to invoke the patch contract before invoking the original contract. In this way, the smart contract is repaired to ensure that transactions that trigger potential vulnerabilities are blocked in the patch contract.

#### *5.4.3. Summary*

Vulnerability defense is an important means to combat smart contract attacks. It is also one of the important goals of security research. Smart contracts have the characteristics that 1) the code cannot be modified once the it is deployed and 2) they are run in a decentralized environment. In this regard, its security defense is more difficult than traditional programs. Existing work mainly focuses on two research directions of smart contract security defense, namely, smart contract security programming and vulnerability repair.

- Smart contract security programming is mainly divided into developing safer third-party libraries and safer smart contract writing languages. Both of them aim to reduce smart contract security risks by improving the quality of code. However, this type of method cannot deal with all smart contract vulnerabilities. It can only provide protection for smart contracts in the development stage, which cannot interfere with deployed smart contracts.

- Vulnerability repair methods are mainly divided into on-chain repair and off-chain repair. Developers can adopt this technique to improve the security of smart contracts. Although on-chain repair can fix deployed smart contracts, this type of method also destroys immutability of smart contracts.

## 6. Discussion

We discuss limitations of the existing research on smart contract security and future research directions.

### *6.1. Inadequacy of Existing Research*

#### *6.1.1. Inadequacy of Data Sources*

The existing open source datasets suffer from low data volume and insufficient coverage of vulnerabilities. The manually constructed smart contract datasets rely heavily on human factors and cannot objectively reflect the real performance of tools. Hence, few people adopt this approach to build datasets and conduct evaluations. The manual collection approach ensures the authenticity of the data compared to the manual construction approach. Nevertheless, the distribution of the data volumes among the vulnerability types is sparse, which has a certain deviation compared to the real environment. The automated injection approach can avoid the subjective influence of human factors on dataset construction. Researchers can construct vulnerability samples in large quantities to ensure the variety of vulnerabilities in the generated contracts. However, compared to real contracts, the vulnerability fragments injected by tools are manually constructed, which often encounter the problem of poor authenticity. Hence, the quality of automated generated contracts still needs to be improved.

Table 5: Comparison of the main methods of vulnerability data collection

| Main Technology | Tools | Number | Type | Method Comparison |
|---|---|---|---|---|
| Manually Constructed | - | - | - | **Advantages**, increase the richness of vulnerability types. **Disadvantages**, the evaluation results are not objective enough, and the interference of human factors cannot be avoided. |
| Manually Collected | SmartContractSecurity | 122 | 37 | **Advantages**, ensures the authenticity of the data. **Disadvantages**, the adequacy and diversity of the number and types of vulnerabilities cannot be guaranteed. |
| | DASP | 69 | 10 | |
| | crytic | 25 | 12 | |
| Vulnerability Injection | SolidiFi | 50 | 7 | **Advantages**, it avoids over-reliance on manual labor and ensures the richness and diversity of data distribution. **Disadvantages**, the quality of generated contracts needs to be improved. |

We summarize the existing vulnerability data sources in terms of the amount of vulnerability data and the types of vulnerabilities included, and analyze the advantages and disadvantages among these methods. The results are shown in Table 5.

#### *6.1.2. Inadequacy of Existing Vulnerability Detection Methods*

Smart contract vulnerability detection is one of the most extensively studied directions in smart contract security. Existing vulnerability detection methods are mainly divided into two aspects: traditional detection and deep learning-based detection. Symbolic execution, formal verification, and fuzzy testing are the three mainstream methods of traditional smart contract vulnerability detection, which have their advantages and disadvantages. Symbolic execution techniques need to rely on constraint collection and solution, how to explore a more comprehensive execution path, and dependencies between transactions with the knowledge of the sequence of transactions. The problem of state space explosion makes it difficult to solve the complex constraints, which seriously limits the accuracy of detection. Formal verification can obtain more semantic-level information. However, the models used for vulnerability detection is often restricted by limited human experience, the low level of automation of such tools, and the problem of unreachable paths. Fuzzy testing is simple and efficient. However, fuzzy testing techniques rely heavily on contract runtime information. Fuzzy testing can successfully discover vulnerabilities only if the path is reachable. The technical difficulty of fuzzy testing lies on generating better quality input information to improve the test coverage.

Traditional detection relies heavily on manual rules. It has the problem of poor extendibility. Attackers can deliberately design dangerous contracts that bypass detection performed by traditional tools based on these designed detection rules, thus causing financial losses. On contrast, deep learning-based detection methods do not need experts to develop detection rules. They are more extendable than traditional detection methods. However, because of the black-box nature of deep learning models, the results detected by the models cannot explain the cause.

We summarize the most representative research work and vulnerability detection tools and analyze the advantages and disadvantages of these approaches in terms of the auxiliary approach, the analysis level of the tools, the number of detectable vulnerabilities, and whether they are open source. The results are shown in Table 6.

#### *6.1.3. Inadequacy of Existing Vulnerability Defense Methods*

For vulnerability defense of smart contracts, existing research focuses on two aspects: secure programming and vulnerability remediation.

Most researchers focus on secure programming in the smart contract language level to reduce the security risk of contracts. Secure programming is mainly divided into developing a more secure third-party code base in solidity language and using a more secure language to write smart contracts. These two methods can reduce the risk of introducing security vulnerabilities in the development phase of smart contracts. Still, this approach cannot solve the smart contract vulnerability defense problem. Once these third-party inventories in security vulnerabilities or new programming language contain vulnerability flaws, these flaws will impact smart contract security.

In the area of smart contract security, it is important to discover abnormal contract behavior and flaws in a smart contract. It is even more challenging to fix contract errors and ensure the correctness of deployed contracts in a timely manner. Existing smart contract fixes focus on providing fixes for non-deployed smart contracts and new smart contract vulnerabilities are constantly being discovered. However, even if all known vulnerabilities are fixed, these off-chain fixes cannot help once new vulnerability issues arise for contracts that are running on Ether. **Aroc** [90] can fix the deployed contract by bundling the problematic contract with the patch contract for invocation. Still, the extendibility is limited because it relies on the repair template to generate the patch contract. In addition, this method breaches the tamper-evident property of the contract and the fairness of the contract participants. How to balance contract security and fairness requires researchers to conduct more profound research.

We summarize the existing vulnerability remediation tools and analyze the advantages and disadvantages of these approaches in terms of the analysis level of the tools, the number of fixable vulnerabilities and whether they are open source. The results are shown in Table 7.

#### *6.1.4. Inadequacy of Existing Review Studies*

Parizi et al. [32] empirically evaluate open source smart contract detection tools for smart contract vulnerabilities on Ether, including four tools, i.e., Oyente, Mythril, Security, and SmaerCheck. This review does not systematically analyze and classify contract vulnerabilities and detection tools. Sayeed et al. [13] classify smart contract vulnerabilities into four types

Table 6: Comparison of the main methods of vulnerability detection tools

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

Table 7: Comparison of vulnerability repair tool methods

| Main Technology | Tools | Analysis Level | Number | Open Source | Method Comparison |
|---|---|---|---|---|---|
| Off-Chain Repair | SCRepair | Source Code | 4 | Yes | **Advantages**, uulnerability can be fixed with a specific fix template **Disadvantages**, increase gas consumption for new contracts. |
| | SGUARD | Source Code | 4 | Yes | |
| | SMARTSHIELD | Bytecode | 3 | No | |
| | EVMPATCH | Bytecode | 2 | Yes | |
| On-Chain Repair | Aroc | Source Code | 4 | No | **Advantages**, it can fix deployed smart contracts. **Disadvantages**, it can break the tamper-evident nature of the contract and affect fairness. |

using attack principles and systematically analyze seven smart contract vulnerabilities and ten vulnerability detection tools. Kushwaha et al. [11] systematically review the security vulnerabilities in the Ethereum blockchain. They discuss the corresponding prevention mechanism by analyzing the existing smart contract vulnerability attack mechanism, and analyze 25 vulnerability detection tools.

Compared with the existing review literature, the strength of this paper is that we have conducted a more comprehensive analysis through three aspects of smart contract security, i.e., vulnerability detection, vulnerability sources, and vulnerability repair. This paper discusses security issues and security assurance methods at different stages, from contract design, implementation, and testing to operation. It summarizes ten mainstream vulnerabilities from three levels: language, virtual machine, and blockchain. We examine 49 vulnerability injection, detection, and repair methods, and analyze the shortcomings of existing research results.

### *6.2. Future Research Direction*

This paper summarizes the following three future research directions based on our above discussion on the shortcomings of existing smart contract security methods.

1. **Combining secure programming specifications and risk detection blocking.** Since smart contracts cannot be changed once being deployed, it is important to ensure they are secure and reliable. As the existing smart contract development language Solidity is in the development stage, many challenges still have not yet been solved. Developing a more secure programming specification can reduce the contract security risk in the development stage. In addition, all-round security testing can be conducted by means of vulnerability testing for developed and undeployed smart contracts, and real-time operational monitoring and analysis can be conducted for deployed smart contracts. Timely security risk detection and blocking for post-deployment contracts can ensure the security of the contracts at runtime. Combining secure programming specifications with security risk blocking ensures the security of smart contracts at all stages, from design and development to operation.

2. **Constructing large-scale and high-diversity baseline vulnerability assessment datasets.** Due to the existing vulnerability data collection methods, the constructed benchmark datasets generally have problems such as insufficient data volume and uneven distribution of vulnerability types. In order to build a benchmark dataset with a sufficient amount and rich variety of vulnerability data, we may consider using automatic vulnerability injection technologies to improve the quality of the vulnerability fragments for injection and ensure the authenticity of the generated smart contracts under the premise of manually collecting real smart contract vulnerability data. This approach avoids the subjectivity of manual construction and enriches the types of vulnerabilities in the dataset. The final goal is to build labelled benchmark datasets containing multiple vulnerability types to facilitate the evaluation of

vulnerability detection and remediation tools.

3. **Combining traditional detection methods with deep learning methods.** For the smart contract security problem, most existing research uses a single traditional or deep learning method for contract vulnerability detection. To achieve broader coverage of contract vulnerability detection, we may consider the organic combination of traditional detection and deep learning methods. This paradigm can realize targeted extraction of feature information for different contract vulnerability features, integration of the smart contract's syntax and semantic information, and construction of a complete contract security analysis framework for more accurate smart contract security analysis.
4. **Combining on-chain and off-chain repair.** For well-developed smart contracts, it is necessary to strengthen the security protection of smart contracts based on vulnerability repair. The existing research mainly focuses on how to repair the detected vulnerabilities before contract deployment. For the non-deployed smart contracts, off-chain repair solutions with better performance are required to ensure contract security. For the deployed contracts, on-chain repair technologies need to be further developed to update the patch contract in real time to ensure contract security. In this regard, comprehensive contract security repair frameworks are demanded to protect smart contract security from both off-chain and on-chain.

Overall, future research on smart contract security can be carried out in the above directions, including customizing the corresponding specification mechanism for different research goals, creating large-scale evaluation datasets with reasonable diversity, solving existing security problems while gradually improving the security of existing methods, and fixing vulnerabilities both off-chain and on-chain, to achieve a full range of smart contract security protection.

### *6.3. Threat to Validity*

Two potential limitations may exist in the methodology of this review: (1) limitations in the scope of the publication search and (2) lack of accuracy and completeness in the data extraction process.

First, to ensure the fairness of the research literature selection process, we develop a literature search strategy and define keywords and search terms that enable us to search for relevant literature. However, selecting keywords is somewhat subjective, which may lead to omitting some relevant studies. In addition, we select a set of mainstream computer science databases as the sources of the literature search to cover as many relevant studies as possible. Nonetheless, although the concept of smart contracts emerged relatively early, the academic research on smart contracts is still in its infancy stage. The relevant industry websites may contain more current information than the academic websites. Hence, our findings may have missed some industry conducted studies. Second, the data extraction results may be somewhat inaccurate. Due to the short development time of smart contracts, there is no official specification of vulnerability definition. Consequently, existing studies have diverse definitions or descriptions regarding smart contract security problems. When we categorise and summarize the 45 relevant articles, we find that some studies lack sufficient information to adequately describe the characteristics of vulnerabilities. The vulnerability test samples and experimental settings in some works are also inadequately presented, which can affect the accuracy of our analysis.

## 7. Conclusion

This paper surveys the main features of smart contracts and analyzes the main threats faced by smart contracts at three levels: language, virtual machine and blockchain. It explores smart contract security assessment and analysis with a comprehensive set of dimensions. These include a review of the source of vulnerability data and the existing ways of collecting vulnerability data, followed by introducing the progress of existing smart contract security research work on vulnerability injection, vulnerability detection and vulnerability defense. Next, we analyze the advantages and disadvantages of these research techniques, upon which four future research directions for smart contract security research are summarized.

## References


[1] M. Alharby and A. Van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

[2] R. Gupta, S. Tanwar, F. Al-Turjman, P. Italiya, A. Nauman, and S. W. Kim, "Smart contract privacy protection using ai in cyber-physical systems: tools, techniques and challenges," *IEEE access*, vol. 8, pp. 24746–24772, 2020.

[3] N. Szabo, "Formalizing and securing relationships on public networks," *First monday*, 1997.

[4] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 2–8, IEEE, 2018.

[5] Y. Yuan, F.-Y. Wang, *et al.*, "Blockchain: the state of the art and future trends," *Acta Automatica Sinica*, vol. 42, no. 4, pp. 481–494, 2016.

[6] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*, pp. 164–186, Springer, 2017.

[7] Y. Liu, F. R. Yu, X. Li, H. Ji, and V. C. Leung, "Blockchain and machine learning for communications and networking systems," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1392–1431, 2020.

[8] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-enabled smart contracts: architecture, applications, and future trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 11, pp. 2266–2277, 2019.

[9] M. Kaulartz and J. Heckmann, "Smart contracts–anwendungen der blockchain-technologie," *Computer und Recht*, vol. 32, no. 9, pp. 618–624, 2016.

[10] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski, "Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack," *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.

[11] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic review of security vulnerabilities in ethereum blockchain smart contract," *IEEE Access*, 2022.

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

[13] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24416–24427, 2020.

[14] Z. Wang, H. Jin, W. Dai, K.-K. R. Choo, and D. Zou, "Ethereum smart contract security research: survey and future research opportunities," *Frontiers of Computer Science*, vol. 15, no. 2, pp. 1–18, 2021.

[15] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, "Smart contract security: A software lifecycle perspective," *IEEE Access*, vol. 7, pp. 150184–150202, 2019.

[16] O. Sürücü, U. Yeprem, C. Wilkinson, W. Hilal, S. A. Gadsden, J. Yawney, N. Alsadi, and A. Giuliano, "A survey on ethereum smart contract vulnerability detection using machine learning," *Disruptive Technologies in Information Sciences VI*, vol. 12117, pp. 110–121, 2022.

[17] A. J. Perez and S. Zeadally, "Secure and privacy-preserving crowdsensing using smart contracts: Issues and solutions," *Computer Science Review*, vol. 43, p. 100450, 2022.

[18] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," in *International conference on financial cryptography and data security*, pp. 494–509, Springer, 2017.

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

[20] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[21] D. Vujičić, D. Jagodić, and S. Ranić, "Blockchain technology, bitcoin, and ethereum: A brief overview," in *2018 17th international symposium infoteh-jahorina (infoteh)*, pp. 1–6, IEEE, 2018.

[22] S. Corbet, B. Lucey, and L. Yarovaya, "Datestamping the bitcoin and ethereum bubbles," *Finance Research Letters*, vol. 26, pp. 81–88, 2018.

[23] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," tech. rep., Technical report, ver. 2.3 ebse technical report. ebse, 2007.

[24] M. Petticrew and H. Roberts, *Systematic reviews in the social sciences: A practical guide*. John Wiley & Sons, 2008.

[25] "Computing research and education." https://www.core.edu.au/.

[26] "The china computer federation." https://www.ccf.org.cn/.

[27] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pp. 530–541, 2020.

[28] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[29] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in ethereum smart contracts," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 139–150, IEEE, 2020.

[30] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 65–68, IEEE, 2018.

[31] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International conference on financial cryptography and data security*, pp. 79–94, Springer, 2016.

[32] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," *arXiv preprint arXiv:1809.02702*, 2018.

[33] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference*, pp. 653–663, 2018.

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

[35] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[36] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15, IEEE, 2019.

[37] M. Di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON)*, pp. 69–78, IEEE, 2019.

[38] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*, pp. 459–474, IEEE, 2014.

[39] "Smartcontractsecurity. smart contract weakness classification and test cases.." https://swcregistry.io/.

[40] "Examples of solidity security issues.." https://github.com/crytic/not-so-smart-contracts.

[41] "The ncc group." https://www.nccgroup.com/.

[42] "Decentralized application security project.." https://dasp.co/.

[43] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network.," in *IJCAI*, pp. 3283–3290, 2020.

[44] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 415–427, 2020.

[45] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269, 2016.

[46] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 664–676, 2018.

[47] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pp. 442–446, IEEE, 2017.

[48] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," *HITB SECCONF Amsterdam*, vol. 9, p. 54, 2018.

[49] B. Jiang, Y. Chen, D. Wang, I. Ashraf, and W. Chan, "Wana: Symbolic execution of wasm bytecode for extensible smart contract vulnerability detection," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 926–937, IEEE, 2021.

[50] R. Yu, J. Shu, D. Yan, and X. Jia, "Redetect: Reentrancy vulnerability detection in smart contracts with high accuracy," in *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*, pp. 412–419, IEEE, 2021.

[51] A. Wang, H. Wang, B. Jiang, and W. K. Chan, "Artemis: An improved smart contract verification tool for vulnerability detection," in *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, pp. 173–181, IEEE, 2020.

[52] R. Ji, N. He, L. Wu, H. Wang, G. Bai, and Y. Guo, "Deposafe: Demystifying the fake deposit vulnerability in ethereum smart contracts," in *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 125–134, IEEE, 2020.

[53] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1186–1189, IEEE, 2019.

[54] M. Fu, L. Wu, Z. Hong, F. Zhu, H. Sun, and W. Feng, "A critical-path-coverage-based vulnerability detection method for smart contracts," *IEEE Access*, vol. 7, pp. 147327–147344, 2019.

[55] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts.," in *Network and Distributed System Security Symposium*, pp. 1–12, 2018.

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

[57] "Echidna: A fast smart contract fuzzer.." https://github.com/crytic/echidna.

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

[59] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of*

*Software Engineering*, pp. 1398–1409, 2020.

[60] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 778–788, 2020.

[61] Y. Huang, B. Jiang, and W. K. Chan, "Eosfuzzer: Fuzzing eosio smart contracts for vulnerability detection," in *12th Asia-Pacific Symposium on Internetware*, pp. 99–109, 2020.

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

[63] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, IEEE, 2018.

[64] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pp. 9–16, 2018.

[65] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," *arXiv preprint arXiv:1811.06632*, 2018.

[66] J.-W. Liao, T.-T. Tsai, C.-K. He, and C.-W. Tien, "Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pp. 458–465, IEEE, 2019.

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

[68] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[69] K. Zhou, J. Cheng, H. Li, Y. Yuan, L. Liu, and X. Li, "Sc-vdm: A lightweight smart contract vulnerability detection model," in *International Conference on Data Mining and Big Data*, pp. 138–149, Springer, 2021.

[70] M. Eshghie, C. Artho, and D. Gurov, "Dynamic vulnerability detection on smart contracts using machine learning," in *Evaluation and Assessment in Software Engineering*, pp. 305–312, 2021.

[71] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.

[72] J. Song, H. He, Z. Lv, C. Su, G. Xu, and W. Wang, "An efficient vulnerability detection model for ethereum smart contracts," in *International Conference on Network and System Security*, pp. 433–442, Springer, 2019.

[73] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts," in *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pp. 47–59, 2021.

[74] F. Mi, Z. Wang, C. Zhao, J. Guo, F. Ahmed, and L. Khan, "Vscl: Automating vulnerability detection in smart contracts with deep learning," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–9, IEEE, 2021.

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

[76] O. Lutz, H. Chen, H. Fereidooni, C. Sendner, A. Dmitrienko, A. R. Sadeghi, and F. Koushanfar, "Escort: ethereum smart contracts vulnerability detection using deep neural network and transfer learning," *arXiv preprint arXiv:2103.12607*, 2021.

[77] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 378–389, IEEE, 2021.

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

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

[80] Y. Zhang and D. Liu, "Toward vulnerability detection for ethereum smart contracts using graph-matching network," *Future Internet*, vol. 14, no. 11, p. 326, 2022.

[81] J. Ye, M. Ma, Y. Lin, L. Ma, Y. Xue, and J. Zhao, "Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures," *Journal of Systems and Software*, vol. 192, p. 111410, 2022.

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

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

[84] "Vyper." https://vyper.readthedocs.io/.

[85] "Daml." www.digitalasset.com.

[86] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart contract repair," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–32, 2020.

[87] T. D. Nguyen, L. H. Pham, and J. Sun, "Sguard: towards fixing vulnerable smart contracts automatically," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1215–1229, IEEE, 2021.

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

[89] M. Rodler, W. Li, G. O. Karame, and L. Davi, "{EVMPatch}: Timely and automated patching of ethereum smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1289–1306, 2021.

[90] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, "Aroc: An automatic repair framework for on-chain smart contracts," *IEEE Transactions on Software Engineering*, 2021.