6-2024

DAppSCAN: Building large-scale datasets for smart contract weaknesses in DApp Projects

Zibin ZHENG

Jianzhong SU

Jiachi CHEN

David LO
Singapore Management University, davidlo@smu.edu.sg

Zhijie ZHONG

See next page for additional authors

Citation

ZHENG, Zibin; SU, Jianzhong; CHEN, Jiachi; LO, David; ZHONG, Zhijie; and YE, Mingxi. DAppSCAN: Building large-scale datasets for smart contract weaknesses in DApp Projects. (2024). *IEEE Transactions on Software Engineering*. 50, (6), 1360-1373.
Available at: https://ink.library.smu.edu.sg/sis_research/8971

Author
Zibin ZHENG, Jianzhong SU, Jiachi CHEN, David LO, Zhijie ZHONG, and Mingxi YE

DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects

Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong and Mingxi Ye

Abstract—The Smart Contract Weakness Classification Registry (SWC Registry) is a widely recognized list of smart contract weaknesses specific to the Ethereum platform. Despite the SWC Registry not being updated with new entries since 2020, the sustained development of smart contract analysis tools for detecting SWC-listed weaknesses highlights their ongoing significance in the field. However, evaluating these tools has proven challenging due to the absence of a large, unbiased, real-world dataset. To address this problem, we aim to build a large-scale SWC weakness dataset from real-world DApp projects. We recruited 22 participants and spent 44 person-months analyzing 1,199 open-source audit reports from 29 security teams. In total, we identified 9,154 weaknesses and developed two distinct datasets, i.e., DAppSCAN-Source and DAppSCAN-Bytecode. The DAppSCAN-Source dataset comprises 39,904 Solidity files, featuring 1,618 SWC weaknesses sourced from 682 real-world DApp projects. However, the Solidity files in this dataset may not be directly compilable for further analysis. To facilitate automated analysis, we developed a tool capable of automatically identifying dependency relationships within DApp projects and completing missing public libraries. Using this tool, we created DAppSCAN-Bytecode dataset, which consists of 6,665 compiled smart contract with 888 SWC weaknesses. Based on DAppSCAN-Bytecode, we conducted an empirical study to evaluate the performance of state-of-the-art smart contract weakness detection tools. The evaluation results revealed sub-par performance for these tools in terms of both effectiveness and success detection rate, indicating that future development should prioritize real-world datasets over simplistic toy contracts.

Index Terms—Empirical Study, Smart Contracts, SWC Weakness, Dataset, Ethereum

1 Introduction

In 2015, Ethereum [1] introduced a revolutionary technology named smart contracts [2]. Smart contracts can be regarded as Turing-complete programs deployed on the blockchain. By utilizing smart contracts, developers can easily develop their decentralized applications (DApp). DApps are immutable, self-executed, without a centralized architecture, which guarantees the transparency and trustworthiness of DApps. These features make smart contracts widely used in many areas, e.g., finance [3] and gaming [4].

Unfortunately, a large number of security incidents related to Ethereum smart contracts have occurred and have caused billions of dollars in financial losses [5] in recent years. To increase the security of smart contracts, significant effort has been devoted to identifying and detecting security issues in smart contracts. For example, Chen et al. [6] introduced 20 kinds of smart contract defect by analyzing online Q&A posts. The DASP project [7] is a smart contract taxonomy that reports on 10 vulnerabilities. A notable blockchain security team named ConsenSys [8] summarized several common smart contract problems and provided a repository named the *Smart Contract Weakness Classification Registry (SWC Registry [9])*. There are 37 kinds of weaknesses in the SWC Registry as of April 2023. Based on classified weaknesses in the SWC Registry, numerous automated analysis tools employing various technologies have been developed in recent years [10], such as those based on program analysis [11], formal verification [12], fuzzing [13], and machine learning methods [14]. In general, the SWC Registry is still one of the most widely used weakness classifications and covers most of the weaknesses detected by existing analysis tools.

However, evaluating these tools is often challenging due to the scarcity of a large-scale labeled dataset. Based on our investigation of 20 academic papers, two common methods have been used to evaluate these tools: (1) manually labeling a small-scale dataset (typically comprising hundreds of contracts) and using it to evaluate the effectiveness (e.g., precision, recall) [15], and (2) using their tools to analyze a large-scale dataset (usually containing thousands of contracts) and then manually checking the correctness of some/all contracts which are labeled positive [16].

Both of these two methods have their limitations. In the *first* method, the selection of the dataset might be unfair or non-representative. Although many researchers have randomly selected and labeled hundreds of smart contracts to evaluate their tools, the majority of these contracts are toy contracts with significantly fewer lines of code than real-world DApp projects. Achieving good results on a dataset predominantly composed of toy contracts cannot guarantee the tool's effectiveness for real-world DApps. Furthermore, only a small proportion of contracts contain weaknesses among hundreds of contracts, which also impact the results. Regarding the *second* method, it can only determine the false/true positive rates of proposed tools; however, it cannot evaluate the tool's false/true negatives. Additionally, although the evaluation of prior tools includes

- *Zibin Zheng, Jiachi Chen, Jianzhong Su, Zhijie Zhong and Mingxi Ye are with School of Software Engineering, Sun Yat-sen University, China.*
  E-mail: {zhzibin, chenjch86}@mail.sysu.edu.cn
  E-mail: {sujzh3, zhongzhj3, yemx6}@mail2.sysu.edu.cn
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore.*
  E-mail: davidlo@smu.edu.sg
- *Jiachi Chen is the corresponding author.*

Manuscript received ; revised

a substantial number of smart contracts, approximately 96% of them are toy contracts that contain less than five on-chain transactions [17]. Consequently, the second method also has limited power to evaluate the tools considering a more realistic setting.

Based on the above motivations, the aim of this study is to construct a dataset that contains a substantial number of vulnerable smart contracts from real-world DApps. In the smart contract community, there are several well-known security teams, e.g., TrailofBits [18] and Slowmist [5], which provide audit services for DApp projects. Each DApp undergoes a comprehensive audit conducted by security experts, who then provide audit reports containing detailed defeat descriptions and their locations in DApps. Although audit reports effectively enhance the security of contracts, their cost is typically high. As a result, audited DApps are generally real-world projects rather than simple toy contracts.

In this paper, we recruited 22 participants (including 7 PhD and 15 master's students) and dedicated 44 person-months to manually analyze 1,199 open-source audit reports provided by 29 security teams. We finally summarized 9,154 weaknesses among these DApps. Given that most current smart contract research focuses on a few common weakness types, e.g., *Reentrancy* and *Integer Overflow and Underflow*, and that the majority of them can be found in the SWC Registry, we specifically highlighted 1,618 SWC weaknesses from 682 DApps. On average, each DApp has 58 Solidity smart contract files (39,904 in total) with 7,885 lines of code, and 66.3% of DApps have compiler versions higher than 0.6. For each SWC weakness, our dataset provides a description, location within the source code, and the associated audit report. We call this dataset **DAPPSCAN-SOURCE**, as the dataset provides the SWC weakness information at the source code level.

Note that a DApp typically consists of multiple Solidity files with complex dependencies, and it often relies on numerous open-source online libraries, such as `Safemath` in OpenZeppelin [19], which may not be included in the source code of the DApp project. Consequently, Solidity smart contract files in the DApp project might not be able to compile directly. To facilitate the use of the dataset, we developed a tool that can identify the dependency relationships of contracts within a DApp and insert missing library code into the contracts. Using this tool, we successfully compiled 6,665 smart contracts, and obtained their bytecode and related ABI information [20]. These contracts contain 888 SWC weaknesses, and we call this dataset as **DAPPSCAN-BYTECODE**.

Furthermore, based on DAPPSCAN-BYTECODE, we conducted an empirical study to evaluate the effectiveness of state-of-the-art smart contract weakness detection tools, i.e., Mythril [21], Slither [22], Security [23], Smartian [24], Sailfish [25] and eTainter [26]. We found that except for Slither that can successfully analyze 86% of contracts, other tools can only analyze a small portion of contracts in the dataset. Besides, among the contracts that these tools can analyze, only a few weaknesses are correctly detected. The results indicate that smart contract analysis tools should focus more on real-world situations rather than simple toy contracts.

In addition, we found that there exist 82.3% of non-SWC weaknesses in collected audit reports. This is because the SWC registry mainly includes the classical and general weaknesses in smart contracts, e.g., *Reentrancy* and *Integer Overflow*, but the majority of the reported weaknesses in audit reports are non-security issues (such as code optimization suggestions) or non-classical issues (like functional bugs specific for each DApp), which are hard to be classified with general rules. Performing manual analysis on these non-classical weaknesses requires much more labor and potentially compromises data quality. Therefore, we focus only on the SWC weaknesses in this paper. Our dataset is only suitable for evaluating tools on classical and low-level smart contract weaknesses, but not on non-classical weaknesses (e.g., function bugs). Meanwhile, non-SWC weaknesses are also necessary for research because of their significant proportion in audit reports.

The main contributions of this paper are as follows.

- *Two large-scale SWC weakness datasets.* In this paper, we proposed two large-scale datasets from real-world DApp projects, i.e. DAPPSCAN-SOURCE and DAPPSCAN-BYTECODE. The dataset could aid further research on smart contract analysis. We open the whole dataset to the public at: https://github.com/InPlusLab/DAppSCAN/.
- *A tool to obtain compiled bytecode within DApps.* We proposed a tool that can automatically analyze dependencies between contracts, based on which we can insert missing public library code to generate the compiled smart contracts bytecode from DApp projects.
- *Empirical study of current tools.* We investigate the evaluation methods and dataset used by current smart contact analysis tools. We also assess state-of-the-art tools based on our dataset. The results demonstrate that most of their evaluations were based on toy contracts with a few lines of code and older compiler versions, and they have poor performance in detecting smart contract weaknesses in real-world DApps.

The organization of the rest of this paper is as follows. In Section 2, we provide the background knowledge of DApp projects, smart contract audit reports, and SWC Registry. Then, we highlight our motivation in Section 3. In Section 4 and 5, we introduce the details of the two datasets. In Section 6, we conducted an empirical study to evaluate the performance of state-of-the-art smart contract tools. Then, we provide the implications and describe threats to validity in Section 7. In Section 8, we elaborate on the related work. Finally, we conclude the whole study and mention future work in Section 9.

2 BACKGROUND

In this section, we briefly introduce background information about DApp projects, audit reports, and the SWC Registry.

2.1 Smart Contracts and DApp Projects

Smart contracts are Turing-complete programs running on the blockchain. Solidity [27] is one of the most well-known smart contract languages, which is an object-oriented language similar to Java and C++. It supports smart contracts that inherit code from other contracts and import code from other contract files. A DApp project always consists of multiple smart contracts and files, and its file structure is usually complicated due to the inheritance and import relationship. When developing a DApp project, developers always use frameworks (e.g., Truffle [28], Brownie [29]) to enhance development efficiency. These frameworks usually support import third-party contracts/library from Github directly; the code *import "@openzeppelin/contracts/token/ERC20/ERC20.sol"* can be used to download the ERC20 contract from Openzeppelin's Github repository directly, without the use of a code in the local environment. Furthermore, a DApp usually contains two parts, i.e., smart contracts and scripts. The smart contracts in the DApp might be deployed to multiple addresses, and these can interact with the scripts written in Python, Javascript, or other languages.

2.2 Smart Contract Audit Reports

In recent years, frequent security incidents that resulted in numerous financial losses motivated the birth of the smart contract audit service. The audit service provides a detailed code review, provided by security experts, to ensure security and potentially improve the performance of a smart contract. As of August 2022, there are 60 security teams that have been recommended by Etherscan [30], which are well known in blockchain ecology.

The audit process usually contains four steps [31]. First, an automated analysis step is performed using smart contract analysis tools. However, automated analysis is usually error-prone. Second, an in-depth manual code review is performed. This step is usually carried out by at least two security experts to ensure that all weaknesses are correctly discovered. Third, the security team contacts the customer and provides recommendations for security and optimization. The customers also confirm and repair the reported weaknesses during the consulting process. Finally, a security audit report is provided, which contains the descriptions, locations, and repaired versions (if provided) of weaknesses.

The price of an audit service varies and depends on the complexity of the contracts. Since auditing a DApp usually takes a long period of time and requires a high level of professional knowledge, the fees for these services are usually expensive.

2.3 Smart Contract Weakness Classification (SWC) Registry

In software engineering, weaknesses are errors that can lead to vulnerabilities, while vulnerabilities are mistakes that hackers can use to attack projects [32]. The smart contract weakness classification (SWC) registry [9], also known as EIP-1470 [33], is a specification introduced to classify common smart contract weaknesses. Each SWC weakness is included with a title, description, ways to fix it, and a specific SWC-ID. As of April 2023, 37 kinds of weakness have been collected in the SWC Registry. Most of the common issues reported in academic works can be linked to SWC weaknesses. For example, permission-less issues [34] are classified into two weaknesses, e.g., SWC-105 (Unprotected Ether Withdrawal) and SWC-106 (Unprotected SELFDESTRUCT Withdrawal). The SWC-ID and title can be found in Table 4; the details of the 37 kinds of SWC weakness can be found at: https://swcregistry.io/.

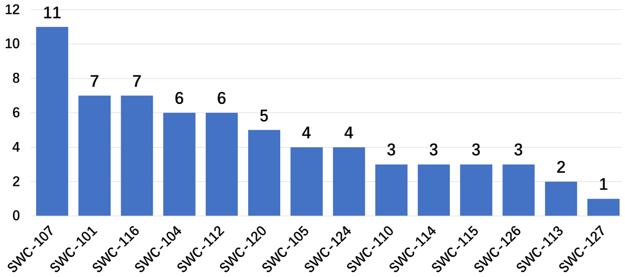

Fig. 1: The number of academic tools published in top conferences that can detect certain SWC weaknesses.

In particular, although some SWC weaknesses are either irrelevant from a security perspective or repaired in the recent Solidity compilers, they are worth researching. On the one hand, while SWC weaknesses include instances that may not pose a security threat (according to CWE [35]), non-security-related weaknesses still merit consideration. For example, SWC-135 (Code With No Effects) is frequently highlighted in audit reports, as addressing this problem could save costs for developers. On the other hand, although new Solidity versions address some weaknesses, there are a massive number of contracts with old Solidity versions running on Ethereum, and it is still possible for developers to deploy contracts with old versions to Ethereum. Thus, the SWC weaknesses being addressed by new Solidity versions (e.g., SWC-100 Function Default Visibility) still deserve attention.

3 Motivation

In this section, we investigate the dataset and evaluation methods used by current smart contract weakness detection tools. The results indicate a high demand for a large-scale, timely, and accurate dataset, which underscores the significance of our dataset.

3.1 Tools Collection and Analysis

We surveyed the major smart contract weakness detection tools published at top software engineering and security conferences, i.e., ASE, FSE, ICSE, ISSTA, S&P, CCS, USENIX Security, and NDSS. Note that a weakness may have a different name in different academic works. For example, a weakness known as "Timestamp Dependence" was first introduced by Luu et al. [16]. The same weakness was also reported by sfuzz [36] and named as "Block Number Dependency". This weakness is called "Weak Sources of Randomness from Chain Attributes" in the SWC Registry. Since the SWC Registry has the largest coverage of the kinds of Ethereum weaknesses, we use the names defined by the SWC Registry in this paper.

TABLE 1: 20 Tools published in top SE/Security conferences that can detect SWC weaknesses.

Tools	Venues	Detected SWC ID	Dataset and Evaluation
Oyente [16]	CCS'16	104, 107, 114, 120	M2{T2(19,366)}
Zeus [37]	NDSS'18	101, 104, 107, 113, 114, 115, 120	M2{T1(1,524)}
teEther [38]	Security'18	106, 112	M2{T2(38,757)}
Securify [23]	CCS'18	104, 105, 107, 112, 114, 124	M1{T1(100)} + T2(24,594)
ContractFuzzer [39]	ASE'18	107, 112, 116, 120, 126	M2{T1(6,991)}
sFuzz [36]	ICSE'19	101, 107, 112, 116, 120, 126	M2{T1(4,112)}
Sereum [40]	NDSS'19	107	M2{T2(24,594)}
Smartian [24]	ASE'19	101, 104, 105, 106, 107, 115, 116, 124, 127	M3{T1(58)} + M3{T1(72)} + T3(500)
ILF [41]	CCS'19	104, 105, 106, 112, 116	M2{T1(18,496)}
Harvey [42]	FSE'20	110, 124	M2{T1(17)}
extended Harvey [43]	ICSE'20	110, 124	M3{T1(17)}
CLAIRVOYANCE [44]	ASE'20	107	M2{T1(17,770)}
SOLAR [45]	ASE'20	107, 116, 120 , 126	M2{T1(25,983)}
VeriSmart [46]	S&P'20	101	M3{T1(60)}
VerX [47]	S&P'20	101, 110	M3{T1(12)}
ETHBMC [48]	Security'20	106, 112	M5{T2(2,194,650)}
SmartTest [49]	Security'21	101, 105, 106	M4{T1(443)}
Unknown Name [50]	FSE'21	101, 104, 107, 115, 116	M3{T1(176)} + T1(47,398)
Smartdagger [15]	ISSTA'22	107, 113, 116	M5{T1(47,398)} + M1{T1(594)}
eTainter [26]	ISSTA'22	113, 128	M1{T1(28)} + T2(60,612) + T3 (3,000)

TABLE 2: Types and methods used to evaluate tools

ID	Description
T1	Verified smart contracts collected from Etherscan
T2	Smart contract on Ethereum without the need for source code
T3	A selected smart contract dataset with high transactions or balances
T4	DApp projects collected from Github
M1	Manually label the vulnerabilities in a dataset before testing the tool
M2	Manually check all/some of the cases detected by the tool.
M3	Reuse previously labeled datasets
M4	Reuse contracts from CVE (https://www.cve.org/)
M5	Compare the performance with previous tools

3.2 Results

We totally find 20 tools published in top SE/Security conferences at the time we conducted our study. Table 1 lists their names, the venues in which they were published, the SWC weaknesses that can be detected using these tools, and the dataset and evaluation method they use. The number of tools that can detect certain SWC weaknesses is also given in Figure 1. From the figure, we can find that only 14 out of 37 SWC weaknesses could be detected using the current smart contract analysis tools. *Reentrancy* (SWC-107, 11 times) is the most popular weakness investigated in academia, followed by *Integer Overflow and Underflow* (SWC-101, 7 times) and *Block Values as a Proxy for Time* (SWC-116, 7 times). Most of the SWC weaknesses are currently not supported by these tools.

The datasets and evaluation methods used in these tools are listed in the last column of Table 1. There were four types of dataset (T1 to T4) and five kinds of method (M1 to M5) used to evaluate tools, which are listed in Table 2. Each evaluation method is represented in the form of "Method{Type(Size of Dataset)}". For example, M2{T1(1,524)} means that the dataset has 1,524 verified smart contracts (T1) [1]. Then, the authors manually checked some/all the cases detected by the tool (M2). The symbol "+" means that multiple evaluations were conducted. For example, M3{T1(176)} + T1(47,398) means that a previous labeled dataset (M3), which consists of 176 verified smart contracts (T1), was reused to evaluate the tool. Then, the tool was used to analyze 47,398 smart contracts (T1) without checking their correctness.

Only two works (Smartain [24] and eTainter [26]) used a selected dataset with high-value transactions or balances, which means that most of the works used toy contracts to evaluate their tools. For example, the dataset of SmartBug was reused by tools published in the past two years [15], [24], [50]. However, this dataset may include many out-of-date and toy contracts, as the average number of lines of code for this dataset is 204, and 99.8% of the contracts' compiler versions are lower than 0.6, while the latest Solidity compiler version is 0.8+.

Moreover, M2 was widely employed to evaluate the tools, though it could only verify the false/true positive results of a contract without reporting false/true negative results. While M1 can overcome this limitation, its dataset size is typically small. These limitations underscore the necessity for a dataset comprising a substantial number of labeled weakness cases.

4 The DAppSCAN-Source Dataset

The previous section highlighted the need for a dataset with: (1) a reasonable number of weakness cases, and (2) timely contracts from real-world DApp projects rather than toy contracts with old Solidity versions. In this section, we present the methodology employed to construct such a dataset.

[1] T1 means the first type of the dataset

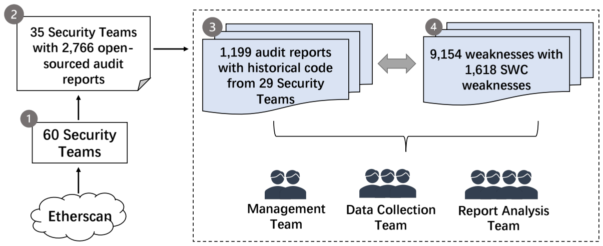
2
35 Security Teams with 2,766 open-sourced audit reports
1
60 Security Teams
Etherscan
3
1,199 audit reports with historical code from 29 Security Teams
4
9,154 weaknesses with 1,618 SWC weaknesses
Management Team
Data Collection Team
Report Analysis Team

Fig. 2: Overview architecture of finding SWC weaknesses from audit reports.

4.1 Methodology

Audit reports are documents that contain detailed security information about smart contracts. Figure 2 illustrates the process used to find SWC weaknesses in DApp audit reports. First, we obtained a list of 60 recommended security teams from Etherscan (as of August 2022) and found that there are only 35 teams that provide open-source audit reports with permissions for content extraction and redistribution. Then we manually reviewed the information on their homepages and discovered 2,766 open-source audit reports. Then, we include the audit reports that provide historical source code. After that, we include 1,199 audit reports from 29 security teams. Finally, we manually analyzed the selected audit reports and found 9,154 weaknesses; 1,618 of them are SWC weaknesses. These SWC weaknesses were derived from 682 DApps and 608 audit reports (we only focused on Solidity projects). In the following, we introduce the details of each step.

4.1.1 Participant Recruitment

Analyzing such a large number of audit reports is a labor-intensive task. To alleviate the workload, we need to assemble a team of participants familiar with Solidity program and SWC weaknesses. The first author's university has a blockchain laboratory with experienced researchers. Therefore, we sent invitations to the researchers in the laboratory to introduce the details of our work and the process of analyzing audit reports. We finally recruited 22 researchers for this task.

4.1.2 Roles of Team Members

Although all the recruited researchers claimed that they have rich experience in blockchain area, they may have had different understandings of SWC weaknesses and Solidity programming. Thus, we interviewed each member to understand their background. Based on these interviews, we found that three researchers had professional knowledge of smart contract weaknesses; they have already published several smart-contract-related works at top venues. Also, five researchers had limited background in Solidity weaknesses, but all had good experience in programming and blockchain. Regarding the other 14 researchers, they had good knowledge in smart contract programming but had not yet published related papers in top conferences/journals. On the basis of their background, we divided them into three groups.

- **Management team.** This team consisted of the three researchers with professional knowledge. Their duties included (1) launching online training on SWC weaknesses, (2) answering other members' questions, (3) reviewing audit report analysis results and refining the data, and (4) managing the work schedule.
- **Data collection team.** This team consisted of five researchers with a limited background in Solidity weaknesses. Their duties are described as step 2 and step 3 in Figure 2, namely, (1) finding out how many security teams released open-source audit reports and downloading all the audit reports; (2) reviewing the collected audit reports and locating the historical DApp versions containing SWC weaknesses as described in the audit reports, and (3) removing non-Solidity DApp projects, as our focus is Solidity-based DApps.
- **Report analysis team.** The remaining 14 members responded to analyzing the audit reports. They were required to (1) summarize the weaknesses described in the audit reports, (2) determine whether the weaknesses belonged to the SWC registry, and (3) if so, provide the SWC-ID and pinpoint its location within the source code.

4.1.3 Data Collection

The data collection team was responsible for collecting two types of data, i.e. audit reports and the codes of DApps. To collect audit reports, we examined the homepages of the security teams, as they typically provide a list of reports on their websites. Then we saved all of the audit reports as PDFs to avoid potential future link unavailability. In total, we collected 2,766 open-source audit reports from 35 security teams that release their audit reports with permissions for content extraction and redistribution. To ensure data completeness, two team members analyzed the reports of each security team.

For DApp codes, most weaknesses described in the audit reports had already been patched in the latest versions. Fortunately, audit reports usually highlight the code version they analyzed, e.g., a historical github version or a link on Etherscan/BscScan [51]. Thus, we manually reviewed the audit reports to find the corresponding code versions. As a result, we found 1,199 audit reports specifying the audited code version, which belong to 29 security teams. For each DApp in these audit reports, we also saved its code to prevent potential data loss and perform a double check to ensure correctness.

4.1.4 Data Analysis

This step was carried out by the report analysis team and was assisted by the management team. Before analyzing audit reports, 14 members of the report analysis team received training from the management team. The training content included: (1) information about the SWC Registry and examples of each SWC weakness, (2) how to analyze an audit report, and (3) the content of the analysis report they should submit. An analysis report contained a summary of the weakness, the type of SWC, and the location of the weakness in the source code. After training, 1,199 audit reports were distributed to the report analysis team in four rounds.

TABLE 3: Statistical information of DAPPSCAN-SOURCE.

Key Information	Numbers
Total audit reports	608
Total DApps	682
Total Solidity files	39,904
Average Solidity files in a DApp	58
Average Line of Code in a DApp	7,885
Compiler Version 0.4+	104
Compiler Version 0.5+	123
Compiler Version 0.6+	164
Compiler Version 0.7+	48
Compiler Version 0.8+	240
Other Compiler Version	3

- **First Round.** In the first round, 188 out of 1,199 (15.7%) audit reports were distributed to the report analysis team (almost 27 reports per member), and each audit report was analyzed by two members independently. To ensure independence, the analysis team members were not allowed to share their results and dataset with others. The results were then integrated by the management team. Then, the management team compared and checked the results to summarize some common issues, e.g., formatting errors and misunderstandings of SWC weaknesses. Next, the report analysis team received the second stage of training on these common issues. After training, the members were required to self-check their results and compare their results with another member. If the result was different, they were required to discuss it together and provide a final result. The management team provided technical help throughout the process.
- **Second and Third Rounds.** In the second and third rounds, 163 (13.6%) and 284 (23.7%) audit reports were distributed to the report analysis team, respectively. Similarly to the first round, each audit report was also independently analyzed by two members. Then, the results were collected by the management team and compared for consistency. The difference in the results would lead to the results being returned, and the report analysis team members would be required to discuss the difference and further provide a consistent result. If they were unable to reach an agreement, the management team would help to make a final decision.
- **Fourth Round.** All the remaining audit reports were distributed in this round. After the second and third rounds, we found that the correctness rate increased rapidly with the progress of the project. Thus, in the fourth round, we did not double check. Instead, the management team randomly sampled and checked 10% of the analysis results. The sampling results showed the good quality of the analysis.

4.2 Results

4.2.1 Dataset

Through manual data analysis of audit reports, we totally found 9,154 weaknesses. In this work, we focus on SWC weaknesses and build up DAPPSCAN-SOURCE dataset based on the audit reports containing at least one SWC weaknesses. There are three parts of the DAPPSCAN-SOURCE dataset, i.e., audit reports, the codes of DApps, and analysis reports.

Audit Reports. Overall, we found SWC weaknesses in 608 audit reports, which have the corresponding codes of DApps. Note that some audit reports provide the codes before and after the revision, which are both collected to facilitate follow-up research.

DApps. Since an audit report might review multiple DApps, we totally found 682 DApps from the 608 audit reports. Table 3 lists some key information about DApps. Each DApp has about 58 Solidity smart contract files (39,904 in total) with 7,885 lines of code on average. In general, **the real-world DApp projects in our dataset are much more complicated than the smart contracts analyzed by previous academic tools.** For example, the average line of code in the widely used SmartBug dataset [52] is only 204, and more than 90% of their contracts are used out-of-date compiler versions (0.4+). While Solidity version 0.8+ was the most common version, about 35.2% ($\frac{240}{682}$) in our dataset, and it was also the latest compiler version as of May 2023. Version 0.6.1 was published on Jan. 2020, and it was also a widely used version in real-world smart contract development at the time of conducting this study; about **66.3%** ($\frac{452}{682}$) of DApps' compiler versions in our dataset were higher than 0.6, while the corresponding number on SmartBug dataset was only **0.02%**. (The distribution of compiler versions in the SmartBug Dataset was: 0.4+: 43,664 (93.2%); 0.5+: 3,147 (6.7%); 0.6+: 11 (0.02%); others: 25 (0.05%) [52].)

Analysis Reports. Based on the collected codes of DApps, we convert audit reports to our designed analysis reports, which are more suitable for automated analysis. Analysis reports contain a set of JSON files that record the analysis results. Each JSON file corresponds to a Solidity file and records the SWC weaknesses in it. Fig. 3 shows an example of the content of the JSON data. The attribute "filePath" records the path of the corresponding Solidity file, while the attribute "SWCs" contains the descriptions of the SWC weaknesses in the smart contract. There are three elements in each SWC weakness, that is, the SWC's *category*, its corresponding *function* (if exists), and the *lineNumber*. The *category* provides the type of SWC weakness. The *function* provides the function name to which the weakness code belongs. However, some SWC weaknesses do not correspond to a function, but to a whole contract or Solidity files. For example, the "Outdated Compiler Version" weakness can be related to all Solidity files of the DApp. In such cases, we set the *function* as N/A in the JSON. The *lineNumber* provides the detailed location of the SWC weakness in the Solidity file, which may include multiple lines of codes or even the whole Solidity file.

4.2.2 SWC in DApps

Totally, there are 1,618 SWC weaknesses in DAPPSCAN-SOURCE that cover 34 (out of 37) kinds of SWC weakness. Among them, eight kinds of SWC weakness commonly appear in DApps (more than 100 times). The third column (D1) of Table 4 presents the number of SWC weaknesses included in DAPPSCAN-SOURCE dataset.

SWC-135 (Code With No Effects) was the most common SWC weakness identified. This weakness typically results from redundant code and does not cause severe security

```
{
 "filePath": "Hacken-Tenderize/contracts/
 tenderizer/integrations/livepeer/Livepeer.sol",
 "SWCs": [
  {
   "category": "SWC-Unchecked Call Return Value",
   "function": "_withdraw",
   "lineNumber": "L122"
  },
  {
   "category": "SWC-Unchecked Call Return Value",
   "function": "_claimSecondaryRewards",
   "lineNumber": "L162 - L164"
  }
 ]
}
```

Fig. 3: A sample of the analysis report in JSON format.

issues. However, it reduces code readability and increases gas consumption.

SWC-101 (Integer Overflow and Underflow) and SWC-107 (Reentrancy) ranked as the second and third most common weaknesses. These two weaknesses are notorious in Ethereum since they have caused significant financial losses. For example, the well-known "The DAO" security incident [53] is caused by Reentrancy. The high frequency of these two SWC weaknesses highlights the seriousness of smart contract security issues.

For 15 SWC weaknesses, we found fewer than 10 cases in our dataset; many of them had already been addressed in the latest Solidity compiler or IDE can generate warnings that prevent developers from making these mistakes. For instance, SWC-118 (Incorrect Constructor Name) can be regarded as a typographical error in the code. Before Solidity version 0.4.22, the name of the constructor function had to be the same as the contract name. Thus, a typographical error in the constructor function could cause serious security problems. For example, a contract could be named *BuyToken* but the constructor's name could be *buyToken*, in this case, the attacker could invoke *buyToken* to perform malicious behavior (e.g., modifying essential state variables). Furthermore, the introduction of SWC-111 (Deprecated Solidity Functions) and SWC-109 (Uninitialized Storage Point) weaknesses will also occur a warning in Solidity IDE, e.g., the newest version of Remix [54]. Thus, using the latest version of the Solidity compiler and IDE can increase the security of contracts.

In addition, DAPPSCAN-SOURCE does not contain three uncommon kinds of SWC weakness, SWC-136 (Unencrypted Private Data On-Chain), SWC-130 (Right-To-Left-Override control character (U+20E)), and SWC-127 (Arbitrary Jump with Function Type Variable).

5 THE DAPPSCAN-BYTECODE DATASET

In the previous section, we obtained a smart contract dataset directly from DApp projects, which provides the real conditions of SWC weaknesses in audit reports for real-world DApps, such as the distribution of weaknesses. In particular, DAPPSCAN-SOURCE is suitable for evaluating tools that do not require compilation or testing, such as emerging large-language models [55]. However, due to the complexity of DApps, directly analyzing DApps can be challenging for most existing tools since they rely on contract compilation or testing. As shown in Figure 4, DApps typically consist of multiple contract files with complex dependencies. On the one hand, most existing tools cannot directly analyze the whole DApp project since they cannot find the main contacts of the DApp. On the other hand, the DApp relies on a significant number of public libraries (e.g., `SafeMath` in OpenZeppelin [19]), which are not included in the original DApp projects and impede compilation.

TABLE 4: The number of SWC weaknesses in two datasets.

ID	Title	# SWC in D1	# SWC in D2
135	Code With No Effects	284	161
101	Integer Overflow and Underflow	203	123
107	Reentrancy	138	86
104	Unchecked Call Return Value	118	201
102	Outdated Compiler Version	115	20
103	Floating Pragma	105	90
128	DoS With Block Gas Limit	103	49
114	Transaction Order Dependence	87	47
100	Function Default Visibility	74	8
131	Presence of unused variables	51	46
116	Block values as a proxy for time	48	3
105	Unprotected Ether Withdrawal	42	15
108	State Variable Default Visibility	36	6
119	Shadowing State Variables	34	0
113	DoS with Failed Call	29	7
129	Typographical Error	23	0
120	Weak Sources of Randomness from Chain Attributes	23	5
123	Requirement Violation	14	0
112	Delegatecall to Untrusted Callee	12	10
134	Message call with hardcoded gas amount	9	0
126	Insufficient Gas Griefing	9	1
124	Write to Arbitrary Storage Location	9	0
111	Use of Deprecated Solidity Functions	9	0
115	Authorization through tx.origin	8	1
110	Assert Violation	7	0
122	Lack of Proper Signature Verification	5	2
125	Incorrect Inheritance Order	4	1
117	Signature Malleability	4	1
133	Hash Collisions With Multiple Variable Length Arguments	3	2
121	Missing Protection against Signature Replay Attacks	3	1
118	Incorrect Constructor Name	3	0
106	Unprotected SELFDESTRUCT Instruction	3	1
132	Unexpected Ether balance	2	0
109	Uninitialized Storage Pointer	1	1
136	Unencrypted Private Data On-Chain	0	0
130	Right-To-Left-Override control character (U+202E)	0	0
127	Arbitrary Jump with Function Type Variable	0	0
/	Total	1,618	888

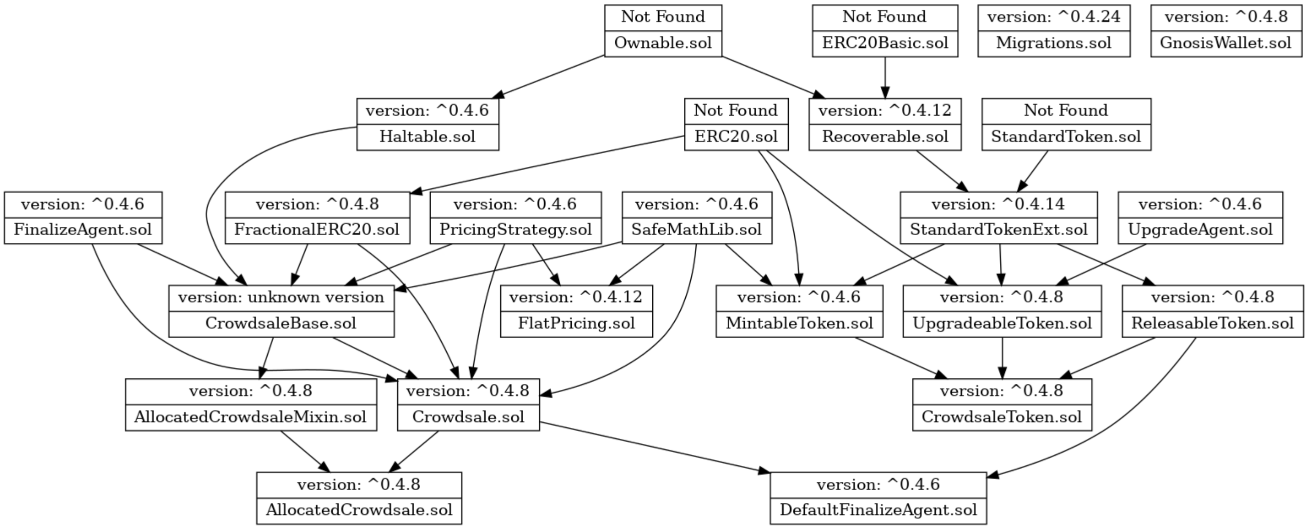

Fig. 4: Dependency relations of a DApp sample.

To this end, we aim to build a compilable dataset to facilitate the evaluation of existing tools. We propose a tool for compiling DApps that can identify dependency relationships and compile any contracts within DApps. In addition, we have manually collected a list of public libraries that each DApp relies on, based on which we perform automatic compilation and further build the DAPPSCAN-BYTECODE dataset.

5.1 Tools to Compile DApps

We present a tool for parsing the dependency relations of DApps and performing compilation. This tool comprises three critical steps, namely dependency analysis, library collection, and contract compilation.

5.1.1 Dependency Analysis

To better understand the structure of DApps, we first automatically identify the dependency relations in DApps to help users gain a better understanding of them and identify missing library contracts. Specifically, we scan all Solidity files in the DApp project, excluding irrelevant configuration files. For each Solidity file, we identify its dependencies based on the corresponding source code and construct a directed acyclic graph representing the relations of the DApp (dependency graph). Furthermore, each node with no outdegree in the dependency graph represents a *main contract* to be compiled and deployed by DApp developers. Although we are capable of analyzing all contracts in the DApp, the following procedures are based on the leaf nodes since other contracts are redundant and overlap with them.

5.1.2 Library Collection

With the dependency relations of each DApp, we can identify missing library contracts such as ERC20 in OpenZeppelin. We manually collect these missing library contracts from Github and automatically insert them into the DApp project. Specifically, we identify the source of each missing library contract based on its library name (i.e. root path) and the corresponding contract path to determine its version. Different versions of the same library usually have a distinct structure. Additionally, we save these library contracts in the same directory as the DApp project.

5.1.3 Contract Compiling

With the structure of the DApps constructed and the missing dependencies filled, we can compile the main contracts in the DApps. This compiling procedure consists of two steps, including the identification of the Solidity version and the generation of an output file (a JSON file) that contains the compiled result. In the first step, we parse the main contract and identify its Solidity version. In particular, we omit identifying the Solidity version of its dependencies, assuming that their versions are the same as the main contract. This is because the inconsistency of the Solidity version among contracts would make compilation fail, and we do not include these cases in our dataset. After identification, we use the corresponding Solidity compiler to compile the main contracts and obtain their bytecode and ABI information.

5.2 Results

We have successfully compiled 6,665 smart contracts from 682 DApps, with 888 SWC weaknesses as DAPPSCAN-BYTECODE dataset. The distribution of these SWC weaknesses is illustrated in the fourth row (D2) of Table 4. 25 out of 37 SWC weaknesses can be found in this dataset, and eight SWC weaknesses appeared more than 40 times.

This compilable dataset contains notorious SWC weaknesses on Ethereum that can lead to huge financial losses, including SWC-104 (Unchecked Call Return Value), SWC-101 (Integer Overflow and Underflow), and SWC-107 (Reentrancy). Additionally, the high frequency of these SWC weaknesses shows the seriousness of smart contract security issues. Our dataset can assist future research in achieving a more realistic evaluation of the tools that claim to detect these critical weaknesses.

Some weaknesses only contain less than 10 cases in our compilable dataset due to: (1) the sparseness of weaknesses in real-world contracts, as we discuss in Section 4.2; (2) the compilation errors caused by incomplete DApp project. More specifically, the compilation of contracts fails mainly due to two reasons. The first reason is that some of the contracts in the project are missing, which leads to compilation failure of contracts inherited from the missing ones. The second reason is the misconfiguration of the contract version. The mismatch between the contract being compiled and the contract being inherited leads to a compilation failure.

Note that the bytecode compiled from a DApp project might contain the same weakness. For example, there are three Solidity contracts in a DApp, namely A, B, and C, and contract A has a Reentrancy issue. Both A+B (B inheriting from A) and A+C (C inheriting from A) are compilable and can generate a bytecode with Reentrancy. This is also the reason why the number of SWC-104 in DAPPSCAN-BYTECODE is larger than that in DAPPSCAN-SOURCE in Table 4. We do not remove these cases, as their bytecode is different, and both of them may exist in real-world scenarios.

6 TOOLS EVALUATION

In this section, we selected state-of-the-art weakness detection tools and use them to conduct experiments on the DAPPSCAN-BYTECODE dataset. To this end, we evaluated the effectiveness of the tools on weakness detection.

6.1 Tool Selection

To carry out our study, we selected representative tools for weakness detection. Although many tools have been proposed in recent years, not all of them are suitable for our study. Therefore, we define several criteria to filter appropriate tools:

- **State-of-the-art.** The tools are published in top-tier software engineering and security venues or well known in industry (e.g., having thousands of stars on GitHub) before May 2023.
- **Avaliable and scalable.** The tools are publicly available and convenient for analyzing large-scale smart contracts (e.g., supporting a command-line interface).
- **Supporting multiple Solidity versions.** Since the smart contracts in our dataset are of multiple Solidity versions, the tools should have the ability to analyze the smart contracts in various Solidity versions.
- **Only requiring code as input.** The input of tools only requires the Solidity code, ABI or Bytecode of smart contracts. For example, we exclude some tools (e.g., Echidna [56]) since they require users to provide manual designed rules (e.g., invariants or specification) for automated analysis.

Based on above-mentioned criteria, we selected six representative weakness detection tools whose techniques are varied, including symbolic execution, formal verification, and fuzzing. The tools are listed in 5 and are briefly described as follows.

TABLE 5: Six representative vulnerability detection tools used in our study.

Tool	GitHub Repository	Docker Image
Mythril	github.com/ConsenSys/mythril	mythril/myth
Slither	github.com/crytic/slither	trailofbits/eth-security-toolbox
Securify	github.com/eth-sri/securify2	Dockerfile
Smartian	github.com/SoftSec-KAIST/Smartian	Dockerfile
Sailfish	github.com/ucsb-seclab/sailfish	holmessherlock/sailfish:latest
eTainter	github.com/DependableSystemsLab/eTainter	Manual construction

Mythril [57] is a symbolic executor that combines taint analysis and control flow check to accurately detect smart contract vulnerabilities. In particular, Mythril has been packaged as an industry product by Consensys [8].

Securify [23] is a vulnerability detection tool based on Datalog analysis. It first abstracts the dependency graph from smart contracts, then verifies the pre-defined compliance/violation patterns to prove whether a property or vulnerability holds or not.

Slither [22] is a Solidity static analysis framework, which first converts Solidity smart contracts into an intermediate representation (IR) called SlithIR and then further verifies the contract properties for vulnerability detection and other in-depth analysis. In particular, Slither is a well-known industry tool with over 3.9k stars on GitHub.

Smartian [24] detects smart contract weaknesses based on fuzzing. It first statically analyzes the contract to initialize high quality seed corpus. Based on the seeds, Smartian performs a lightweight dynamic data-flow analysis during testing to select more effective seeds for improving its efficiency.

Sailfish [25] detects state-inconsistency bugs such as SWC-107 (Reentrancy) and SWC-114 (Transaction order dependence) in smart contracts. It first converts the contract into a storage dependency graph, queries it to explore vulnerability, and then uses symbolic evaluation to prune potential false alarms.

eTainter [26] utilizes static taint tracking to detect gas-related vulnerabilities (e.g., SWC-128 (DoS With Block Gas Limit)) in the bytecode of smart contracts. It tracks taints through the contract state variable and the data flow from contract entry points, while using domain-specific optimizations to improve precision.

6.2 Experiment

6.2.1 *Setup*

Specifically, we directly downloaded the Docker images of Mythril, Slither and Sailfish from Dockerhub [58]. For Securify and Smartian, we built their Docker images according to the Dockerfiles in their GitHub repositories.

Based on Docker images, we used the selected tools to analyze smart contracts in DAPPSCAN-BYTECODE. Specifically, according to the input of the tools, we perform Mythril, Slither, Securify and Sailfish on the Solidity code of smart contracts, and perform Smartian and eTainter on the Bytecode of smart contracts. To determine the time budget of the tools, we randomly sampled 364 smart contracts (a statistically significant sample size considering a confidence level of 95% and a confidence interval of 5%) from the DAPPSCAN-BYTECODE dataset and evaluated the execution time of the tools. Then we found that 90.3% ($\frac{1972}{364*6}$) analysis tasks could be finished within five minutes. Thus, we set an execution time budget of five minutes for each smart contract analysis task. If the time budget is up,

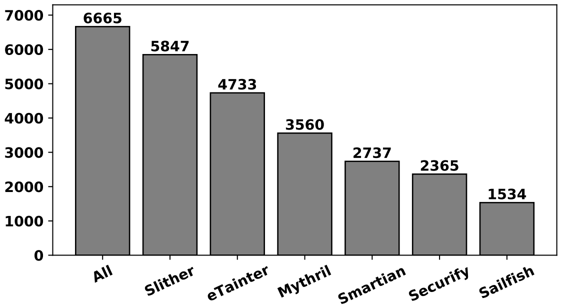

Fig. 5: The number of smart contracts being successfully analyzed by each tool.

we stop the execution and export the analysis results. For fair evaluation, we ran the selected tools with a default configuration similar to the setting followed by Durieux et al. [52]. All experiments were conducted on a Ubuntu machine with an Intel Core i9-10980XE CPU (36 cores and 72 threads) and 256 GB of memory.

6.2.2 Result Analysis

We evaluate the effectiveness of the tools based on the automated analysis results and provide some insightful findings.

Successful Analysis Rate. Fig. 5 demonstrates the number of smart contracts that are successfully analyzed by each tool. We notice that Slither obtains the highest successful analysis rate, about 88% ($\frac{5,847}{6,665}$), this is likely because Slither is a relatively mature industrial product with continuous updates, while other academic tools receive little upgrade once proposed. We also find that other tools fail to analyze many smart contracts. We perform further analysis and illustrate the reasons as follows: (1) The analysis scopes of tools are limited by Solidity versions. Specifically, although we have selected the tools that support analyzing smart contracts with multiple versions, there are still some versions that are not supported by tools (e.g., the newly proposed versions after the tools are proposed). (2) Some design flaws in tools lead to failure of analysis, such as the tool running out of memory during analysis and throwing error.

Weakness Detection. Based on the labels in our dataset, we perform a file-level evaluation on the tools' analysis results. Specifically, if the tools report the weaknesses in the vulnerable Solidity files with corresponding labels, we consider that the tools successfully detect the true weaknesses. Table 6 shows the results of the analysis, which only shows the SWC weaknesses whose number is greater than 0 and can be detected by at least one tool. In this table, if the element in the table is represented as a number, it means that the tool for the corresponding column can detect the SWC-ID weakness of the corresponding row. In total, the tools support the detection of 21 kinds of SWC weakness. Slither supports detecting the most kinds of SWC weakness, and some tools (e.g., Sailfish and eTainter) only support detecting a few kinds of SWC weakness.

In general, the tools demonstrate their limited effectiveness in detecting SWC weaknesses within audit reports. Next, we illustrate the detection results in detail from three perspectives: True Positive, False Positive, and False Negative.

True Positive. For the 21 kinds of SWC weakness that the tools support detection, we notice that the tools only successfully detect a few true SWC weaknesses in audit reports (see the ”Union” column), most of which are reported by Slither. The three most weaknesses detected by the tools are SWC-103 (Floating Pragma), SWC-135 (Code With No Effect), and SWC-107 (Reentrancy) with successful detection rates of 91.1% ($\frac{82}{90}$), 77.8% ($\frac{105}{135}$), and 55.4% ($\frac{46}{83}$), respectively.

False Positive. Overall, the tools report a huge number of false positive weaknesses. Specifically, some kinds of SWC weakness such as SWC-103 (Floating Pragma), contain more than 1,000 false positives reported by tools. The limited performance (e.g., inaccurate detection mechanism) of tools mainly results in false positives. For example, reentrancy could be prevented by *Reentrancy Lock*, which is not considered by tools and leads to false positives [59]. In particular, this result is similar to work [60], which found the high false positive rate of tools.

False Negative. For some kinds of SWC weakness, the tools show their high false negative rates. In particular, some kinds of SWC weakness are commonly seen in audit reports but only few of them are successfully detected by tools, including SWC-104 (Unchecked Call Return Value), SWC-114 (Transaction Order Dependence), SWC-128 (DoS With Block Gas Limit) and SWC-131 (Presence of Unused Variables). These results demonstrate the limitations of tools to protect real-world DApps from weaknesses.

Unsupported Weaknesses. As shown in Table 6, 7 kinds of SWC weakness in audit reports are not supported detecting by any tool. Among them, SWC-102 (Outdated Compiler Version) is the most common weakness in audit reports, while other kinds of SWC weakness are small in number. We perform further analysis on these kinds of SWC weakness and find that they are difficult to design detection patterns. For example, the Solidity compiler is updated frequently, making it hard for tools to follow the latest compiler and detect SWC-102 weaknesses.

7 Discussion

7.1 Implications

7.1.1 Limitations of Current Academic Tools

In Section 3, we list 20 academic tools that could detect SWC weaknesses. We tried to use these tools on the DAPPSCAN-SOURCE dataset proposed in Section 4. However, we found that none of these academic tools was able to support the analysis of raw DApp projects. The key reason for this failure was that most DApps were developed using a smart contract framework, such as Truffle [28], Brownie [29], or Hardhat [61], Embark [62]. There are 58 Solidity files on average in the DApps of our dataset. Smart contracts can inherit from other contracts, and the file structure of DApp projects is usually complicated due to the import relationship (for details, see Section 2.1). We found that only Slither [22], a tool from the industrial team named TrailofBits [18] supported the analysis of DApps developed using smart contract frameworks. This finding may also hint that academic work should focus more on the real-world development process.

TABLE 6: Six representative vulnerability detection tools used in our study. "All" means the number of all vulnerable Solidity files. "/" means the tool can not detect the corresponding weakness. "Union" means the number of the union of all detected vulnerable Solidity files by tools.

ID	Mythril			Securify			Slither			Smartian			Sailfish			eTainter			Union	All
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	
101	2	71	90	/	/	/	/	/	/	0	105	92	/	/	/	/	/	/	2	92
103	/	/	/	37	1298	53	82	4132	8	/	/	/	/	/	/	/	/	/	82	90
104	0	8	60	0	114	60	13	538	47	0	26	60	/	/	/	/	/	/	13	60
105	/	/	/	0	6	14	0	56	14	0	4	14	/	/	/	/	/	/	0	14
106	0	20	1	0	8	1	0	15	1	0	12	1	/	/	/	/	/	/	0	1
107	15	115	68	1	13	82	45	459	38	0	0	83	0	31	83	/	/	/	46	83
108	/	/	/	0	124	4	/	/	/	/	/	/	/	/	/	/	/	/	0	4
109	/	/	/	0	322	1	0	59	1	/	/	/	/	/	/	/	/	/	0	1
110	0	106	0	/	/	/	/	/	/	0	350	0	/	/	/	/	/	/	0	0
112	0	8	6	0	1	6	0	19	6	/	/	/	/	/	/	/	/	/	0	6
113	0	26	7	/	/	/	/	/	/	0	4	7	/	/	/	0	0	7	0	7
114	/	/	/	1	37	41	/	/	/	/	/	/	1	56	41	/	/	/	2	42
115	0	10	1	/	/	/	1	1	0	0	2	1	/	/	/	/	/	/	1	1
116	0	126	3	0	28	3	0	305	3	0	6	3	/	/	/	/	/	/	0	3
119	/	/	/	0	40	0	0	331	0	/	/	/	/	/	/	/	/	/	0	0
120	/	/	/	/	/	/	0	24	5	/	/	/	/	/	/	/	/	/	0	5
124	0	1	0	0	248	0	/	/	/	0	5	0	/	/	/	/	/	/	0	0
125	/	/	/	/	/	/	0	72	1	/	/	/	/	/	/	/	/	/	0	1
128	/	/	/	/	/	/	1	167	48	/	/	/	/	/	/	0	10	49	1	49
131	/	/	/	0	232	44	2	222	42	/	/	/	/	/	/	/	/	/	2	44
135	/	/	/	/	/	/	105	2735	30	/	/	/	/	/	/	/	/	/	105	135
100	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	6
102	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	19
117	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	1
121	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	1
122	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	2
126	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	1
133	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	2

7.1.2 *Automatic Software Engineering for Smart Contracts*

In this paper, we have proposed a dataset that contains 1,618 SWC weaknesses from 682 DApps. The scale of our dataset can be used to perform some automatic software engineering research. For example, about 86.2% of the DApps in our dataset are projects on Github or Gitlab. For each SWC weakness, we have provided the location of the vulnerable code, which can be used to automatically locate/predict weaknesses. In addition, the fixed code recorded in the related Github commit in audit reports can be utilized to develop code repair technologies.

7.1.3 *Non-SWC Weaknesses in Audit Reports*

Due to the limited labor force, we only labeled codes with SWC weaknesses, as the SWC Registry covered a wide range of smart contract issues and is commonly used in existing research (as shown in Table 1). However, 82.3% ($\frac{7,536}{9,154}$) of the weaknesses in the audit reports of our dataset are not included in the SWC Registry.

To study non-SWC weaknesses, we sampled and analyzed 366 weaknesses, a statistically significant sample size considering a confidence level of 95% and a confidence interval of 5% referring to [59]. After performing manual analysis, we summarize their main conditions as follows: (1) The reported items are not related to the security of smart contracts, such as the gas optimization and the code standardization suggestions; (2) The reported weaknesses are special functional/logic bugs related to the specific business of each DApp, which cannot be included in SWC registry as a general rule. For example, the token transfer logic is incorrectly implemented in the deposit business so that the attacker can obtain a deposit certificate without transferring any tokens [63]. Both of them are hard to classify with general rules. Furthermore, SWC registry has not been updated since 2020, so SWC registry cannot cover these new and non-classical weaknesses.

Given that the majority of items in audit reports are non-SWC weaknesses, we believe they are also significant for research. Thus, we also provide descriptions and related audit reports in our dataset, which may be helpful for future research. In future work, we will conduct an in-depth empirical study on these weaknesses and summarize typical and common issues to facilitate subsequent research.

7.2 Threats to Validity

7.2.1 *Internal Validity*

In this study, we proposed two types of datasets, i.e., DAppScan-Source and DAppScan-Bytecode. Both of them required significant labor, which may involve some errors. Specifically, in DAppScan-Source, we manually analyzed 1,199 audit reports and identified SWC weakness-related information from them. InDAppScan-Bytecode, we manually collected the missing third-party libraries. To ensure accuracy, all manual processes were performed by at least two experienced researchers. Additionally, the dataset is open-source on GitHub. **If users identify any errors, they can submit issue reports through Github, and we will update our dataset promptly.**

7.2.2 *External Validity*

The Ethereum ecosystem is rapidly evolving. For example, Solidity v0.6.1 was released in January 2020, while the latest version as of April 2023 is Solidity v0.8.20. There are many differences between the various compiler versions, and some kinds of SWC weakness may disappear due to updates of the Solidity compiler. Although developers are allowed to choose older Solidity compiler versions, there is a trend towards the use of newer compilers. Fortunately, most of the DApps we collected in this paper used compiler versions greater than 0.6+. In the future, we will update our dataset to ensure that it remains up to date.

8 Related Work

In recent years, many smart contract analysis tools have been developed, and several studies have investigated their efficacy.

Durieux et al. [52] conducted an empirical study based on nine automated smart contract analysis tools. To compare and reproduce their research, they create two datasets: (1) 69 annotated vulnerable smart contracts with 115 vulnerabilities (97 of them were tagged as one of the DASP vulnerabilities) that were used to evaluate the precision of tools; and (2) 47,518 open-source smart contracts collected from Etherscan to compare different tools. According to their results, only 29 out of 69 vulnerabilities were detected by the tools. Among the nine tools, Mythril obtained the highest accuracy, but this was only 27%. The nine tools flagged 97% of smart contracts in the large-scale dataset as vulnerable, which shows the high positives of current tools.

Perez et al. [60] evaluated whether vulnerable smart contracts detected by academic tools can be actually exploited. First, they used six academic tools to analyze 821,219 contracts, and 23,327 of them were flagged as vulnerable by at least one tool. Then, the authors performed bytecode-level analysis to confirm potential exploits. They found that only 463 (1.98%) of them had been exploited since their deployment. The balances held in these contracts were 8,487 ETH (0.27%), whereas the total balance in the 23,327 vulnerable contracts was 3 million ETH. These results reflect that the most vulnerabilities reported in academic tools are either false positives or not exploitable.

Yi et al. [64] built a blockchain vulnerability dataset, which contained three levels, i.e., file-level, text-level, and code-level vulnerabilities. To build the dataset, they collected commit messages, issue reports, and pull requests from Github. Then, they conducted an empirical study to investigate common blockchain vulnerability types, blockchain-specific patch code patterns, and susceptible blockchain modules. Their results showed that modules related to consensus, wallet, and networking were more vulnerable than other modules; and about 70% vulnerabilities on blockchain systems were similar to those in traditional projects. The remaining 30% of the vulnerabilities were classified into 21% blockchain-specific vulnerability patterns.

Zhang et al. [65] systematically investigated 516 unique real-world smart contract vulnerabilities from *Code4rena* [66] contests and real-world exploit reports, 74.6% of which are exploitable. They further studied the scope of detectable vulnerabilities and found that existing tools could only deal with about 20.5% of real-world exploitable vulnerabilities. For the remaining undetectable vulnerabilities, they categorized them into seven types and performed in-depth analysis, such as root causes and distributions.

Di et al. [67] performed a large-scale tool evaluation on 248,328 distinct Ethereum smart contracts. They utilized the open-source framework Smartbugs [68] to conduct the automated analysis on the bytecode of smart contracts. Consequently, while the tools reported a total of 1,307,686 potential weaknesses, over time (block height) the number of reported vulnerabilities is decreasing and the tools are degrading to varying degrees.

Chaliasos et al. [69] evaluated five state-of-the-art automated security tools based on 127 high-impact real-world attacks. They also surveyed 49 developers and auditors who work on leading DeFi protocols. Their findings indicated that the tools could only detect about 8% of the attacks, which are all related to reentrancy vulnerabilities. Existing tools do not adequately address logic-related bugs and vulnerabilities in the protocol layer, which are viewed by practitioners as significant threats.

Differences to our work: Compared to other works, our work provided a larger-scale and labeled dataset containing 9,154 weaknesses with 1,618 SWC weaknesses in the form of DApp projects (rather than individual contracts as previous works). In particular, we also proposed a tool that could automatically compile DApp projects and provided a compilable-contract dataset to facilitate contract analysis. Furthermore, we conducted extensive experiments and highlighted the limitations of current tools for analyzing real-world smart contracts in DApps.

9 Conclusion and Future Works

In this paper, we proposed two large-scale datasets with SWC weaknesses to facilitate the evaluation of smart contract analysis tools. To construct the first DAPPSCAN-SOURCE dataset, we recruited participants to analyze 1,199 audit reports. Our dataset contains a total of 9,154 weaknesses, with 1,618 highlighted as SWC weaknesses, providing their descriptions and locations in the source code. The SWC weaknesses in our dataset originated from 682 DApps. On average, each DApp had 58 Solidity smart contract files with 7,885 lines of code, and 66.3% of DApps used compiler versions higher than 0.6. The second dataset, DAPPSCAN-BYTECODE, is compiled from the DAPPSCAN-SOURCE dataset using a tool that automatically parses the dependency relations of smart contracts within DApps. This dataset comprises 6,665 smart contracts with 888 SWC weaknesses. Finally, based on the second dataset, we conducted an empirical study to assess the effectiveness of six state-of-the-art tools in SWC weakness detection. The results reveal poor performance in both effectiveness and successful detection rate, highlighting the need for tools capable of detecting real-world DApp smart contracts.

In the future, we plan to build a website to display the results of this paper, such as the frequency and security events of SWC weaknesses. The website will facilitate the search and downloading of weakness cases. Moreover, we aim to include additional audit reports provided by security teams and will collect and analyze these reports regularly to expand our dataset. In addition, we also aim to study the non-SWC weakness and summarize common issues to facilitate future research.

References

[1] (Aug., 2022) Ethereum.org. [Online]. Available: https://www.ethereum.org/

[2] (Aug., 2022) solc,. [Online]. Available: https://github.com/ethereum/solidity

[3] (Sept., 2020) Peer to Peer Lending Blockchain Platform. [Online]. Available: https://consensys.net/blockchain-use-cases/decentralized-finance/

[4] (Aug., 2022) Cryptokitties. [Online]. Available: https://www.cryptokitties.co/

[5] (Aug., 2022) Slowmist. [Online]. Available: https://www.slowmist.com/en

[6] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining Smart Contract Defects on Ethereum," *IEEE Transactions on Software Engineering*, 2020.

[7] (Aug., 2022) Dasp10. [Online]. Available: https://dasp.co

[8] (Aug., 2022) Consensys: Blockchain technology solutions. [Online]. Available: https://www.consensys.net/

[9] (Aug., 2022) Smart contract weakness classification registry. [Online]. Available: https://github.com/SmartContractSecurity/SWC-registry

[10] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–44, 2021.

[11] 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*, 2018, pp. 664–676.

[12] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 912–915.

[13] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3560429

[14] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.

[15] Z. Liao, Z. Zheng, X. Chen, and Y. Nan, "Smartdagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 752–764.

[16] 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*. ACM, 2016, pp. 254–269.

[17] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[18] (Aug., 2022) Trailofbits. [Online]. Available: https://www.trailofbits.co

[19] OpenZepplelin. (Aug., 2022) Openzepplelin safemath library. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/SafeMath.sol

[20] Alchemy. (2022, March) What is an abi of a smart contract? examples and usage. [Online]. Available: https://www.alchemy.com/overviews/what-is-an-abi-of-a-smart-contract-examples-and-usage

[21] C. Teams. (2022, Augest) Mythx plans. [Online]. Available: https://mythx.io/plans/

[22] 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)*. IEEE, 2019, pp. 8–15.

[23] 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*, 2018, pp. 67–82.

[24] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 227–239.

[25] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 161–178.

[26] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Etainter: Detecting gas-related vulnerabilities in smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 728–739. [Online]. Available: https://doi.org/10.1145/3533767.3534378

[27] (2023) Solidity. [Online]. Available: https://soliditylang.org/

[28] (Aug., 2022) Truffle. [Online]. Available: https://trufflesuite.com/

[29] (Aug., 2022) Brownie. [Online]. Available: https://github.com/eth-brownie/brownie

[30] Etherscan. (2022, Augest) Audit teams on etherscan. [Online]. Available: https://etherscan.io/directory/Smart_Contracts/Smart_Contracts_Audit_And_Security

[31] S. Finance. (2022, Augest) Audit process of solidity finance. [Online]. Available: https://solidity.finance/

[32] CWE. (April., 2023) Cwe-frequently asked questions (faq). [Online]. Available: https://cwe.mitre.org/about/faq.html

[33] G. Wagner. (2018, September) Smart contract weakness classification (swc). [Online]. Available: https://eips.ethereum.org/EIPS/eip-1470

[34] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 716–727.

[35] (2023) Common weakness enumeration (cwe). [Online]. Available: https://cwe.mitre.org/about/index.html

[36] 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*, 2020, pp. 778–788.

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

[38] J. Krupp and C. Rossow, "{teEther}: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.

[39] 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)*. IEEE, 2018, pp. 259–269.

[40] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *NDSS*, 2019.

[41] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[42] 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*, 2020, pp. 1398–1409.

[43] ——, "Targeted greybox fuzzing with static lookahead analysis," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 789–800.

[44] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1029–1040.

[45] Y. Feng, E. Torlak, and R. Bodík, "Summary-based symbolic evaluation for smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1141–1152.

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

[47] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1661–1677.

[48] J. Frank, C. Aschermann, and T. Holz, "{ETHBMC}: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2757–2774.

[49] S. So, S. Hong, and H. Oh, "{SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1361–1378.

[50] M. Ren, F. Ma, Z. Yin, Y. Fu, H. Li, W. Chang, and Y. Jiang, "Making smart contract development more secure and easier," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1360–1370.

[51] (Aug., 2022) Bscscan: The explore for binance blockchain. [Online]. Available: https://bscscan.com/

[52] 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*, 2020, pp. 530–541.

[53] CoinDesk. (Aug., 2022) The dao attack: Understanding what happened. [Online]. Available: https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/

[54] "Remix," 2022. [Online]. Available: https://remix.ethereum.org/

[55] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *arXiv preprint arXiv:2309.05520*, 2023.

[56] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 557–560. [Online]. Available: https://doi.org/10.1145/3395363.3404366

[57] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," in *9th Annual HITB Security Conference (HITBSecConf)*, vol. 54, 2018.

[58] "Docker hub," 2023. [Online]. Available: https://hub.docker.com/

[59] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '23, 2023.

[60] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1325–1341.

[61] (Aug., 2022) Hardhat. [Online]. Available: https://hardhat.org/

[62] (Aug., 2022) Embark. [Online]. Available: https://framework.embarklabs.io/

[63] (2022) Incorrect token transfer. [Online]. Available: https://github.com/InPlusLab/DAppSCAN/blob/main/DAppSCAN-source/audit_report/BlockSec-blocksec_coordinape_v1.1_signed/report.pdf

[64] X. Yi, D. Wu, L. Jiang, K. Zhang, and W. Zhang, "Diving into blockchain's weaknesses: An empirical study of blockchain system vulnerabilities," *arXiv preprint arXiv:2110.12162*, 2021.

[65] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 615–627.

[66] (2023) Code4rena. [Online]. Available: https://code4rena.com/

[67] M. di Angelo, T. Durieux, J. F. Ferreira, and G. Salzer, "Evolution of automated weakness detection in ethereum bytecode: a comprehensive study," *arXiv preprint arXiv:2303.10517*, 2023.

[68] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[69] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits, "Smart contract and defi security: Insights from tool evaluations and practitioner surveys," *arXiv preprint arXiv:2304.02981*, 2023.