Turn the Rudder: A Beacon of Reentrancy Detection for Smart Contracts on Ethereum

***Abstract*—Smart contracts are programs deployed on a blockchain and are immutable once deployed. Reentrancy, one of the most important vulnerabilities in smart contracts, has caused millions of dollars in financial loss. Many reentrancy detection approaches have been proposed. It is necessary to investigate the performance of these approaches to provide useful guidelines for their application. In this work, we conduct a large-scale empirical study on the capability of five well-known or recent reentrancy detection tools such as Mythril and Sailfish. We collect 230,548 verified smart contracts from Etherscan and analyze 139,424 contracts after deduplication using the tools, which results in 21,212 contracts with reentrancy issues. Then, we manually examine the defective functions located by the tools in the contracts. From the examination results, we obtain 31 true positive contracts with reentrancy and 21,181 false positive contracts without reentrancy. We also analyze the causes of the true and false positives. Finally, we evaluate the tools based on the two kinds of contracts. The results show that more than 99.8% of the reentrant contracts detected by the tools are false positives with eight types of causes, and the tools can only detect the reentrancy issues caused by *call.value()*, 54.8% of which can be detected by the Ethereum's official IDE, Remix. Furthermore, we collect real-world reentrancy attacks reported in the past two years and find that the tools fail to find any issues in the corresponding contracts. Based on the findings, existing works on reentrancy detection appear to have very limited capability, and researchers should turn the rudder to discover and detect new reentrancy patterns except those related to *call.value()*.**

***Index Terms*—Smart contract, Reentrancy, Empirical study**

I. Introduction

Smart contracts are programs deployed on a blockchain [1]. Due to the decentralized and trusted authorities guaranteed by blockchain technology, smart contracts are widely used to develop decentralized applications (DApps) in a variety of domains such as games, government, and finance [2]. However, smart contracts are also restricted by the immutability of blockchain [3]. That is, a contract cannot be patched once deployed. A vulnerable contract could be leveraged by malicious attackers and result in serious problems, e.g., financial loss. Therefore, it is important to ensure the correctness of a contract before deploying it. This is a challenging task in practice for contract developers.

A number of vulnerabilities have been discovered for smart contracts from real-world attacks or through theoretical analysis [4]–[6]. For example, 36 types of vulnerabilities are recorded in the SWC registry [7], and the NCC Group [8] lists the top ten vulnerabilities, e.g., reentrancy and time manipulation. To enable developers to recognize and fix vulnerabilities, many approaches have been proposed to detect vulnerabilities in contracts [5], [9], [10]. Reentrancy is a vulnerability that has been extensively studied by existing approaches as it could lead to huge financial loss, e.g., the DAO attack [11] caused a loss of around 150 million dollars.

Existing reentrancy detection approaches mainly focus on the *call.value()*[1] operations in smart contracts. In Section II, we explain a reentrancy issue using an example (see Fig. 1). The reentrancy is caused by the delayed update of the state variable *userbalance* behind the call to *call.value()*. Reentrancy detection approaches aim to discover the payments that could be repeatedly incurred by external calls using various techniques such as symbolic execution [12], fuzzing [13], and neural networks [14]. Two recent empirical studies [15], [16] conducted two years ago revealed that there can be many false positives detected by the approaches at that time. The contracts used in these studies are written in Solidity version $\leq$0.6.0. In the past two years, the Solidity language has gone through several versions with significant changes [17]. Inspired by these studies, a number of approaches have been upgraded or newly proposed. However, there has been no study on the performance of state-of-the-art reentrancy detection approaches on the contracts developed in Solidity version $>$ 0.6.0. Moreover, it has not yet been confirmed whether existing approaches could detect other reentrancy issues in spite of those related to *call.value()*, as such reentrancy issues seem to be able to be detected by the official IDE, Remix [18].

In this work, we conduct a large-scale empirical study on the capability of existing approaches in detecting reentrancy from smart contracts. We collect the verified Solidity code of all 230,548 contracts from Etherscan [19] (a leading block explorer and analytics platform for Ethereum) on October 13, 2021, and select five well-known or recent tools such as Mythril [20] and Sailfish [21] that can locate the possibly defective functions with reentrancy issues in contracts. After filtering duplicate contracts with the same bytecode, we obtain 139,424 contracts. Then, we use the tools on the contracts, which results in 21,212 reentrant contracts. Next, we manually examine the defective functions of the reentrant contracts by recruiting 50 participants (including 27 undergraduates, 21 masters, and two PhDs). From the examination results, we build a set of 31 true positive contracts with reentrancy and a set of 21,181 false positive contracts without reentrancy. We also analyze the causes of the true and false positives. Using the two sets of contracts, we evaluate the tools. Furthermore, we test the tools on the contracts with reentrancy attacks

[1]This is a typical ether transfer function in Solidity. The function may have different forms in different versions of Solidity. In this paper, we use call.value() for simplicity.

reported in the past two years and also test the true positive contracts using Remix. The results are as follows: 1) more than 99.8% of the reentrant contracts detected by the tools are false positives with eight types of causes; 2) the true positive contracts are all related to *call.value()*, 54.8% of which can be detected by Remix; and 3) the tools fail to detect reentrancy issues from the recently attacked contracts. Based on the results, we conclude that existing works on reentrancy detection have poor performance and may be outdated, and researchers should shift their attention from *call.value()* to focus on discovering and detecting new reentrancy patterns.

The main contributions of this work are outlined below:

- We study the capability of five well-known or recent reentrancy detection tools on 139,424 smart contracts.
- We manually examine 21,212 reentrant contracts detected by the tools and build a set of 31 true positive contracts with reentrancy and a set of 21,181 false positive contracts without reentrancy. We further summarize eight types of causes that lead to false positives and find that all of the true positives are caused by *call.value()*.
- We evaluate the tools based on the two manually built sets of true and false positive contracts. Moreover, we test the true positives using Remix and test the tools on the contracts with recent reentrancy attacks. Based on the results, we provide insightful guidelines for researchers.
- We release our experimental data at GitHub [22], including the 230,548 contracts, the detection results of the tools, and the two sets of true and false positive contracts, to provide a benchmark for researchers to conduct future work on reentrancy detection.

The rest of the paper is organized as follows. Section II introduces smart contracts and reentrancy vulnerability. Section III describes our research methodology. Section IV presents the results. Section V discusses the results and describes two additional tests. Section VI analyzes the threats to validity of our study. Section VII reviews the related work. Section VIII concludes the paper and discusses future work.

II. Background

A. Smart Contracts

Smart contracts are programs running on a blockchain. In general, smart contracts are written in Solidity [17], one of the most popular languages for smart contracts. The Solidity code of a smart contract is compiled into bytecode and then deployed on blockchain. In addition, the compilation process generates the application binary interface (ABI) to facilitate subsequent calls and analysis of smart contracts.

B. Reentrancy

From the 150m$ DAO attack in 2016 to the 80m$ Fei Protocol attack in 2022, the reentrancy vulnerability has caused huge financial loss. In this section, we describe how a reentrancy attack could happen with a simplified version of the DAO smart contract, as shown in Fig. 1. The example contract is developed for asset management. It uses the variable *userbalance* (line 2) to record the balance of each user and allows

```
1  contract SimpleDAO {
2      mapping (address => uint) public userbalance;
3      ...
4      function withdraw(uint amount) public{
5          if (userbalance[msg.sender]>= amount) {
6              require(msg.sender.call.value(amount)())
                   ;
7              userbalance[msg.sender]-=amount;
8          }
9      }
10 }
```

Fig. 1. Simple example of reentrancy

users to call the *withdraw()* function (line 4) to withdraw their balance. In the function, the contract first checks if the caller (represented by the address variable *msg.sender*) has enough balance (line 5); then, it transfers the requested *amount* of ether to the caller and subtracts the *amount* from the caller's balance recorded in the variable *userbalance*. However, Solidity introduces a special mechanism named the "fallback function". Users can write their own code in the fallback function and the function will be executed if a contract receives ether from other addresses. In the example case, the ether transfer function *call.value()* (line 6) will automatically call the fallback function of the caller contract and thus the caller can take over the control flow. Attackers can deploy malicious code in the fallback function to repetitively invoke the *withdraw()* function. Note that in the second invocation of *withdraw()*, line 7 has not been executed since the invocation begins at the *call.value()* function in line 6, and thus the *userbalance* has not been changed at this time. As a consequence, the "if" condition check (line 5) of the second invocation is passed and the victim contract will transfer ether to the caller repeatedly until the balance of the contract is drained.

III. Methodology

As shown in Fig. 2, our research methodology contains five main steps: 1) *Smart Contract Collection*, which collects smart contracts from Etherscan; 2) *Reentrancy Detection*, which detects reentrancy issues from the contracts using five selected tools; 3) *Manual Examination*, which manually examines whether the detected reentrant contracts are correct (i.e., true positive) or not (i.e., false positive); 4) *Cause Analysis*, which analyzes the causes of the true and false positive contracts; and 5) *Tool Evaluation*, which evaluates the tools based on the manually examined and analyzed results from the contracts.

A. Smart Contract Collection

We collect 230,548 smart contracts with verified Solidity code, ABI, and bytecode from Etherscan on October 13, 2021. Through preliminary analysis, there are a considerable number of duplicate contracts with the same bytecode. To reduce the workload of the subsequent steps of our study, we filter out the duplicates that have the same hash of bytecode. Consequently, we obtain 139,424 contracts. Table I summarizes the information of the contracts, including their average lines of code and the distribution of their versions.

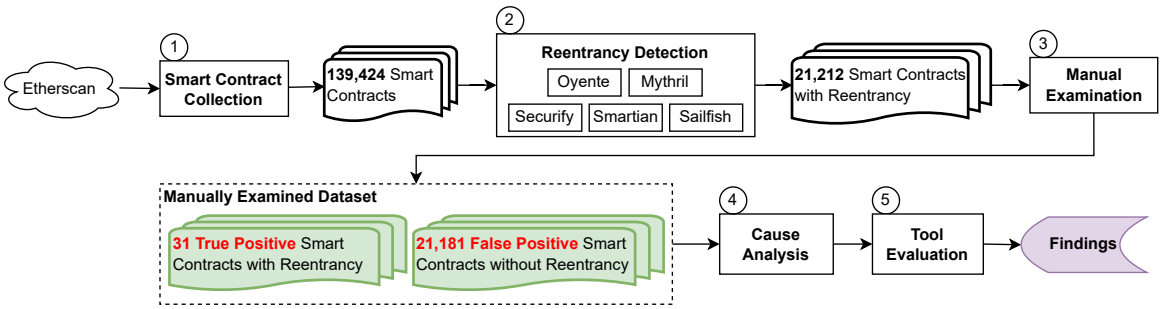

Fig. 2. The overview of our research methodology

TABLE I
SUMMARIES OF OUR SMART CONTRACT DATASET.

Key Information	Number
Total Contracts	230,548
Duplicated Contracts	139,424
Compiler Version <0.5	68,196
Compiler Version 0.5+	23,461
Compiler Version 0.6+	24,761
Compiler Version 0.7+	8,292
Compiler Version 0.8+	14,714
Average Line of Code in Duplicated Contracts	720.42

B. Reentrancy Detection

1) Tool Selection: To conduct our study, we need to select some representative tools that can detect reentrancy issues from contracts. In the empirical study conducted by Durieux et al. [15], they summarize a list of 35 vulnerability detection tools for smart contracts. We extend their list by searching for other state-of-the-art tools published in the literature or on the internet after that work, e.g., sFuzz [23], Smartian [24], Smartest [25], etc. However, not all of the tools are appropriate for our study, e.g., those that cannot detect reentrancy or cannot be applied to Solidity code. To select appropriate tools for this study, we define several criteria as follows.

- **Available and scalable**. The tool is publicly available and can be easily applied to a large set of contracts. In particular, the tool should support a command-line interface that is convenient for performing large-scale experiments.
- **Supports multiple versions of Solidity**. The tool can detect the contracts in our dataset that are written in multiple versions of Solidity.
- **Requires Solidity code only**. The tool only requires the Solidity code and its derivatives (e.g., ABI and Bytecode) as input without any other specification (e.g., a test suite annotated with assertions).
- **Ability to locate vulnerabilities**. The tool can locate the defective functions with reentrancy issues in a contract, which is important for practical use and can facilitate the manual examination task in this study.

Using the criteria above, we select five representative reentrancy detection tools from both industry and academia, as listed in Table II. Within industry, we select Mythril since it outperforms other tools [15]. Within academia, we select Oyente, Securify, Smartian, and Sailfish from the top conferences of software engineering or security. The selected tools use various techniques, e.g., symbolic execution, formal verification, and fuzzing, which are briefly described below:

Oyente [26] is one of the first smart contract analysis tools based on symbolic execution. It constructs the control flow graph of a contract and symbolically executes the contract to detect vulnerabilities by exploring as many execution paths as possible. Oyente serves as the basis of several other vulnerability detection tools such as Maian [27] and Osiris [28].

Mythril [20] is also a vulnerability detection tool based on symbolic execution, which combines taint analysis and control flow checking for more accurate detection. Mythril has been packaged as a commercial product by CONSENSYS [29].

Securify [30] is a vulnerability detection tool based on formal verification. It symbolically analyzes the dependency graph of a contract and checks compliance/violation patterns that capture sufficient conditions for proving whether a property holds or not. There are two versions of Securify, namely Securify(V1) and Securify(V2). We use both of them because they support different versions of Solidity.

Smartian [24] is a fuzzer that combines static analysis and dynamic analysis to detect vulnerabilities from contracts. It statically analyzes a contract to predict the transaction sequences that can lead to effective testing, and then uses such information to construct the initial seed corpus. During fuzzing, Smartian performs a lightweight dynamic data-flow analysis to effectively guide fuzzing.

Sailfish [21] is a scalable system for automatically finding state inconsistency bugs in smart contracts. In order to make the analysis tractable, Sailfish contains two phases: a lightweight exploration phase for reducing the number of instructions to analyze, and a refinement phase for generating extra constraints to approximate the side effects of whole-program execution and ensure the precision of the symbolic evaluation. Using these phases, Sailfish can efficiently detect state inconsistencies in smart contracts.

C. Experiment Setup

We use the selected tools to analyze smart contracts using Docker images, as listed in Table II. We use two methods to obtain the Docker images. For Oyente, Mythril, Securify(V1),

and Sailfish, we directly download their images from Dockerhub [31]. For Securify(V2) and Smartian, there are no images on Dockerhub, and so we build up their images according to the Dockerfiles[2] in their GitHub repositories. Note that Securify(V1) has multiple Docker images on Dockerhub, and different images can support the analysis of contracts with different versions of Solidity. We use multiple Docker images to increase the analysis range of the contracts.

We directly use Oyente, Mythril, Securify and Sailfish on the Solidity code of each contract. Smartian takes the bytecode and ABI of a contract as input, which are also collected from Etherscan. Referring to the time budget set in [15], we set an appropriate time budget, i.e., two minutes, for the five selected tools per contract. If the time budget is up, the tool stops the analysis and exports the analysis result.

Notice that some of the selected tools do not report the detected reentrancy issues as reentrancy. For example, Mythril reports two kinds of vulnerabilities, *External Call To User-Supplied Address* and *State access after external call*, when it detects a reentrancy. For each tool, we create a mapping of the reported vulnerabilities to reentrancy, as listed in Table II.

D. Manual Examination

In order to perform a deep analysis of the tools used in this study, we need to examine whether the detected reentrant contracts are correct. Recall that our selected tools can locate the defective functions in a contract, which can facilitate the examination task. In total, there are 31,720 defective functions detected from 21,212 contracts. Manually examining such a large number of functions is a heavy workload.

1) Participant Recruitment: To reduce the workload of the manual examination task, we need to recruit a relatively large number of participants and let each participant examine only a subset of contracts. Given a defective function of a contract, in order to judge whether the function has a reentrancy issue, the participants should be equipped with knowledge about Solidity and the reentrancy mechanism. It is not an easy job to recruit sufficient participants and ensure that they meet the requirements. In the lead co-author's affiliation, there are more than one hundred undergraduates and masters who could be potential participants. However, not all of the students are familiar with Solidity and reentrancy. Fortunately, in the first co-author's research group, there are a number of PhD students concentrated on research on the vulnerability detection of smart contracts, with 2-5 years of experience. The PhDs have good knowledge of Solidity and reentrancy. With the help of two experienced PhDs, we adopt a four-stage process to recruit participants from the undergraduates and masters as follows.

- **Invitation.** We send an invitation email to 70 undergraduates and 25 masters. In the email, we introduce our study and the manual examination task, and also ask the students whether they are willing to participate in the task. One week later, we receive positive feedback from 36 undergraduates and 23 masters.
- **Training**. We launch several online conferences with the 59 students and invite two PhDs with experience in smart contracts to introduce Solidity and the reentrancy mechanism with examples. Both PhDs also introduce two true positive reentrancy patterns and five false positive reentrancy patterns that they have either discovered from prior studies (e.g., [16]) or by themselves.
- **Self-learning.** After the training stage, we ask the students to further learn Solidity and reentrancy for two weeks by themselves using some suggested materials (e.g., the official Solidity documentation and online tutorials on reentrancy) or web search. We encourage the students to develop, deploy, and execute a contract with reentrancy using Remix.
- **Testing.** After the self-learning stage, we conduct a test to see whether the students have gained enough knowledge of Solidity and reentrancy to perform the examination task. We collect 15 contracts, of which five contracts have reentrancy issues. We use our selected tools on the contracts. The possibly defective functions with reentrancy in the five contracts are detected by at least one of the tools. We develop a website to facilitate the examination task for the students. The website randomly presents each contract in a set with the corresponding list of defective functions detected from the contract. We ask the students to examine the defective functions of a contract one by one and annotate a function that has a reentrancy issue with a specific string "⟨yes⟩ Reentrancy". After the test is completed, we evaluate the accuracy of each student's annotations. A total of 27 undergraduates and 21 masters achieve an accuracy of 100%, and they are chosen as participants for the manual examination task.

2) First Round of Manual Examination: We randomly divide the 48 participants into 24 groups. Each group contains two participants. We also randomly divide the 21,212 reentrant contracts detected by the tools into 24 subsets. 23 subsets contain 884 contracts, and one subset contains 880 contracts. Each subset is allocated to a participant group. The participants do not know which group they belong to. The aforementioned website is used to facilitate the manual examination task. The website randomly displays each contract in the subset allocated to a participant, along with a list of defective functions detected from the contract. When the participants click a defective function, the website can quickly skip to the function in the contract. We ask the participants to examine the defective functions one-by-one and annotate a function that has a reentrancy issue with a specific string "⟨yes⟩ Reentrancy".

Once the manual examination task is completed, we obtain three sets of contracts: 1) $commPs$: the set of contracts that have at least one function annotated by both participants in a group; 2) $commNs$: the set of contracts that have no function annotated by both participants in a group; and 3) $Diffs$: the set of contracts with different annotations given by both participants in a group (i.e., one participant annotates at least one function while the other participant does not annotate any

[2]A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

TABLE II
FIVE REPRESENTATIVE REENTRANCY DETECTION TOOLS USED IN OUR STUDY

Tool	Technique	GitHub Repository URL	Docker	Reentrancy Patterns
Oyente	Symbolic execution	https://github.com/enzymefinance/oyente	qspprotocol/oyente-0.4.25	Re-Entrancy Vulnerability
Mythril	Symbolic execution	https://github.com/ConsenSys/mythri	mythril/myth	External Call To User-Supplied Address State access after external call
Securify	Formal verification	V1: https://github.com/eth-sri/securify	qspprotocol/securify-0.4.25 qspprotocol/securify-usolc	DAO DAOConstantGas
		V2: https://github.com/eth-sri/securify2	Dockerfile	Benign Reentrancy Reentrancy with constant gas Gas-dependent Reentrancy No-Ether-Involved Reentrancy
Sailfish	Formal verification	https://github.com/ucsb-seclab/sailfish	holmessherlock/sailfish:latest	DAO
Smartian	Fuzzing	https://github.com/SoftSec-KAIST/Smartian	Dockerfile	Reentrancy

function). There are 97, 20,626, and 489 contracts contained in $commPs$, $commNs$, and $Diffs$, respectively.

3) Review of the Manual Examination Results: Considering that the participants may not be proficient in smart contracts, we ask the two experienced PhDs involved in the training process to review the participants' examination results. At first, we randomly sample 377 contracts from $commNs$, which is a statistically significant sample size considering a confidence level of 95% and a confidence interval of 5%. Since there is no annotation added to the contracts in $commNs$, both PhDs need to examine the defective functions of the contracts by themselves. They independently examine each of the contracts. After the examination process, both PhDs do not annotate any function of the contracts, meaning that the participants' judgements on the contracts are all correct. This result probably thanks to the introduction of false positive patterns of reentrancy during the training process. Next, we ask both PhDs to review the participants' annotations of the 97 contracts in $commPs$. They perform the review independently. If the annotations of a contract are correct, they label the contract as 1; otherwise 0. Consequently, there are 83 contracts labeled as 0 by both PhDs, indicating that the participants fail to accurately identify reentrancy issues from the contracts. Based on the two groups of results, we are confident in the annotations of contracts in $commNs$, while we lack confidence in the annotations of contracts in $commPs$ and the contracts with disagreement in $Diffs$.

Although the reentrancy issues identified by the participants are not reliable, the participants accurately identify a large number of false positive contracts without reentrancy, which greatly reduces the number of contracts that need to be subsequently examined by the experienced PhDs.

4) Second Round of Manual Examination: According to the review results above, we ask the two experienced PhDs to re-examine the 586(=97+489) contracts in $commPs$ and $Diffs$ using a card sorting approach [32]. Both PhDs first independently examine and annotate the defective functions of each contract, similar to the examination task described in Section III-D2. After the examination process, there are 15 contracts with different annotations. By discussing the disagreements together, both PhDs reach a consensus. Finally, we obtain a set of 31 true positive contracts with reentrancy, denoted as TPs, and a set of 21,181(=20,626+586-31) false positive contracts without reentrancy, denoted as FPs.

E. Cause Analysis

To better understand the true and false positive contracts, we further ask the two PhDs to analyze the causes of the contracts. To reduce the workload, we randomly sample 377 contracts from FPs, which is a statistically significant sample size considering a confidence level of 95% and a confidence interval of 5%. Since we do not have a predefined set of all possible causes, we ask both PhDs to perform cause analysis using two substeps. One PhD first analyzes the cause of each of the 408(=377+31) contracts and records the cause using a short description. The other PhD then reviews the recorded causes. In cases of disagreement, both PhDs discuss the cause to reach a common decision. From the analysis results, the 31 true positive contracts are all related to the *call.value()* function, while there are eight types of causes that lead to the 377 false positive contracts, as listed in Table V.

F. Tool Evaluation

Using the two sets of true and false positive contracts with different causes, we evaluate the five tools used in this study. In spite of the overall performance in terms of precision (see Table IV), we perform a deep analysis by counting the numbers of true and false positive contracts detected by the tools, with respect to each cause type, as listed in Table V.

IV. RESULTS

In this section, we present and discuss the results of the analytical procedure described in Section III.

A. Automated Analysis Results

Table III presents the automated analysis results of the five tools on our dataset. For each tool, the first column (RE) is the number of reentrant contracts reported by the tool, and the second column (Analyzed) is the number of contracts successfully analyzed by the tool.

Analysis Scope. Solidity is a new and frequently updated language; the new features, along with the frequent updates, limit the scope of analysis of vulnerability detection tools. In particular, the versions of Solidity above 0.5.0 have major

TABLE III
THE ANALYSIS RESULTS OF FIVE VULNERABILITY DETECTION TOOLS

		Oyente		Mythril		Securify (V1)		Securify (V2)		Smartian		Sailfish		Total	
Version	**Num.**	**RE**	**Analyzed**	**RE**	**Analyzed**	**RE**	**Analyzed**	**RE**	**Analyzed**	**RE**	**Analyzed**	**RE**	**Analyzed**	**RE**	**Analyzed**
<0.5	68,196	513	56,289	11,962	40,006	2,324	48,715	0	37	15	59,511	1,403	43,541	14,967	66,170
0.5+	23,461	0	576	1,877	15,954	63	13,761	1,693	15,726	3	18,075	641	17,401	3,805	22,350
0.6+	24,761	0	54	1,263	14,988	0	127	797	4,497	2	15,356	244	4,789	2,040	20,470
0.7+	8,292	0	7	273	3,363	0	7	2	71	2	4,155	1	61	275	5,542
0.8+	14,714	0	13	125	7,035	0	29	0	44	0	3,148	0	14	125	8,683
Total	139,424	513	56,939	15,500	81,346	2,387	62,639	2,492	20,375	22	100,245	2,289	65,806	21,212	123,215

changes in their grammar [17], so that the analysis scope of some tools (e.g., Oyente, Securify(V1), and Securify(V2)) is limited in specific versions. Accordingly, we divide our dataset (139,424 smart contracts) into five parts, as shown in Table III.

Oyente and Securify(V1) fail to analyze most of the smart contracts above version 0.5.0, and Securify(V2) can only analyze smart contracts above version 0.5.8. Sailfish fails to analyze smart contracts with a version above 0.7.0. Note that there are a few smart contracts above version 0.5.0 that can be successfully analyzed by Oyente, which might be because the contracts do not use the new features of the later versions. This phenomenon also appears with Sailfish, Securify(V1) and Securify(V2). In particular, the analysis scopes of Mythril and Smartian are less affected by the version of a smart contract, being able to successfully analyze most of the smart contracts in our dataset. In addition to the limitation of versions, the other reason for analysis to fail is that the analysis does not successfully terminate when an external timeout is reached, in which case we kill the process. Consequently, there are 15,778, 1,891, and 2,522 contracts that have no results exported by the tools Mythril, Securify(V1), and Securify(V2). The successful analysis rates of the tested tools show a decreasing trend when applied to higher versions of Solidity, which is ranging from 97% ($\frac{66,170}{68,196}$) in <0.5 to 60% ($\frac{8,683}{14,714}$) in 0.8+.

In order to analyze as many contracts as possible, we combine the automated analysis results of all tools for manual checking. However, there still exist 16,209 smart contracts that fail to be analyzed by any tool.

Reported Reentrancy Rate. As listed in Table III, the five tools report a total of 21,212 reentrant contracts. The reported rate, i.e., $\frac{\#reported\_reentrant\_contracts}{\#successfully\_analyzed\_contracts}$, varies by tool. Mythril successfully analyzes 81,346 contracts and 19% are reported as reentrancy issues. These 15,500 issues occupy 78% of all reported reentrancy issues. Smartian successfully analyzes the greatest number of smart contracts but reports the smallest number of reentrancy issues. There are only 22 reported reentrancy issues in 100,245 contracts that are successfully detected by Smartian. This is likely because the patterns used to detect reentrancy are different. These patterns are decisive for vulnerability detection tools. In the next section, we summarize and propose some effective patterns for vulnerability detection tools to detect reentrancy issues.

Reported Reentrancy Distribution. From Table III, we find that the reported reentrancy rate decreases as the version of Solidity updates, ranging from 22.6% ($\frac{14,967}{66,170}$) in <0.5 to 1.4% ($\frac{125}{8,683}$) in 0.8+. There may be three reasons for this: 1) the developers' awareness of preventing reentrancy issues is strengthened, as many companies have been victims of the reentrancy vulnerability and several related works have been proposed for detecting it; 2) codebases for preventing reentrancy have been proposed and are widely used, such as the *ReentrancyGuard* proposed by openzepplin [33]; and 3) the new types of reentrancy are hard for existing tools to detect. In the next section, we conclude and discuss some new types of reentrancy that may enable developers to write safer contracts or propose more effective vulnerability detection tools.

TABLE IV
THE NUMBER OF REENTRANT CONTRACTS (REPORTED NUM.) REPORTED BY FIVE TOOLS, AND THE NUMBER OF TRUE POSITIVES (TP NUM.).

Tool	TP Num.	Reported Num.
Oyente	23	513
Mythril	23	15,500
Securify(V1)	12	2,387
Securify(V2)	2	2,492
Smartian	4	22
Sailfish	1	2,289
Total	31	21,212

B. Precision of Tools

According to the manual examination, we evaluate the precision of the detection tools. As shown in Table IV, all tools have very low precision in revealing reentrancy vulnerabilities. Sailfish and Mythril are the worst two, whose precision is less than 0.2%. Smartian achieves the highest precision among these tools. However, it only reports 22 reentrancy vulnerabilities and 4 (18.1%) are true positives. In general, the precision of the above tools is too low to satisfy real-world applications in detecting reentrancy vulnerabilities.

C. False Positives of Tools

In this subsection, we summarize the common reasons for the false positives of the tested tools.

1) Permission control: Symbolic execution-based tools, e.g., Oyente and Mythril, usually use a specific pattern to detect vulnerabilities. However, they fail to consider the user's permission when checking a control flow path. There are three ways that could lead to permission control-based false positives, i.e., identify control, address control, and reentrancy lock (a defense mechanism against reentrancy).

TABLE V
CAUSE ANALYSIS RESULTS OF THE FALSE POSITIVES DETECTED BY FIVE TOOLS. THE LAST COLUMN 'TOTAL' IS THE UNION OF THE FALSE POSITIVES REPORTED BY THE TOOLS.

Cause Type	Oyente	Mythril	Securify(V1)	Securify(V2)	Smartian	Sailfish	Total
Permission Control (Identity Control)	6	152	12	12	0	1	170
Permission Control (Address Control)	2	16	2	14	0	3	30
Permission Control (Reentrancy Lock)	0	2	0	6	0	0	6
No State Change After External Call	2	99	2	14	0	0	116
Change State Variable without Financial Risk	6	12	8	21	0	3	43
Special Transfer Value	0	4	0	7	0	1	10
Reentrancy by Transfer/Send	0	1	0	8	0	0	9
Non-callable Function	2	7	25	1	0	1	33

```
1  modifier onlyOwner{
2      require(msg.sender == owner);
3      _;
4  }
5  ...
6  function execute( address _to, uint _value, bytes
       _data) external onlyOwner {
7      ...
8      _to.call.value(_value)(data);
9  }
```

Fig. 3. Code example: Identify control based permission control

```
1  contract DaiSavingsEscrow{
2      address private daiAddress = 0
           x6B175474E89094C44Da98b954EedeAC495271d0F;
3      IERC20 public dai = IERC20(daiAddress);
4      function register(...) public {
5          ...
6          dai.transferFrom(msg.sender, vault,
               payment);
7          ...
8        }
9    }
```

Fig. 4. Code example: Address based permission control

1. Identity Control: The first type of false positive is caused by the ignorance with regard to identity control of the contract caller. Figure 3 shows a code snippet of a real-world smart contract that is falsely detected to have reentrancy vulnerability by Oyente and Mythril. The ether transfer function *call.value()* in line 8 of the contract is detected to lead to reentrancy vulnerability. The function execution enables the caller to transfer the amount of *_value* ether to the address *_to*. Malicious users could attack this smart contract from the fallback function in the smart contract deployed at the *_to* address if they can call the example function. However, this function is not callable from arbitrary addresses due to the *onlyOwner* modifier at line 1, which allows only the contract owner to execute the function. Therefore, the function can not be reentered by other malicious smart contracts.

2. Address Control: The code snippet in Fig. 4 shows a case of false positive type, which is caused by the limited access control of the contract address. This contract is detected to have reentrancy vulnerability in the *register()* function. There is an external call in line 6, where the function calls the *transferFrom()* function from the contract in the address *dai*. However, when we look into the address variable, it can be seen that the address *dai* is assigned with a hard code address in line 2 and line 3 and cannot be modified by other functions. Therefore, the smart contract corresponding to the address *dai* is the smart contract recorded in the address on Ethereum. As a result, the external call in line 6 can not be reentered by other malicious smart contracts.

3. Reentrancy Lock: Figure 5 demonstrates another case of a false positive caused by reentrancy lock. The function in line 15 is detected to have a reentrancy vulnerability based on the external call in line 17. It seems that a smart contract corresponding to the *Token* address could deploy malicious code to reenter the *withdraw()* function to obtain additional profit. However, the function is actually safe against reentrancy attacks due to the modifier *nonReentrant*. This modifier is declared in line 6, where the modifier checks the state of a variable *_notEntered* before the execution of the function and sets it to be "False" (line 7 and 8). If the check fails, the *require* statement will stop the execution of this transaction and roll back to the state before the transaction executes. The "_" in line 9 is a placeholder for the body of the function and line 10 will only be executed after the execution of the function body is finished. Note that before the execution of line 10, the value of variable *_notEntered* is "False" during the execution of the function body. As a result, if any external smart contract tries to reenter *withdraw()* for the second time before finishing the first execution, the check statement in line 7 will fail and the transaction will be stopped immediately. Therefore, the smart contract is safe against reentrancy attacks.

2) No State Change After External Call: If a function could be executed by external calls several times in a single transaction, the contract is regarded as reentrant contract by the previous reentrancy detection tools. However, in some situations, the external calls may not involve financial acts and no state will be changed after the external call. For example, Fig. 6 declares a function *getTokenBal()* to inquire the balance of address *who* in the token contract *tokenAddr* (line 9). However, the state of the contract (i.e., ether balance, storage variable, etc.) will not be changed after the external call of the *balanceOf()* function. Therefore, even if a malicious token *t* reenters this function, it cannot affect the execution of the smart contract.

```
contract ReentrancyGuard {
    bool private _notEntered;
    constructor () internal {
        _notEntered = true;
    }
  modifier nonReentrant() {
      require(_notEntered);
      _notEntered = false;
      _;
      _notEntered = true;
    }
}
contract GovernanceVesting is ReentrancyGuard {
    ...
  function withdraw() public nonReentrant {
      ...
      IERC20(Token).transfer(governanceAddress,
          governanceFunds);
      Withdrawn = true;
    }
}
```

Fig. 5. Code example: Reentrancy lock based permission control

```
  contract ForeignToken {
      function balanceOf(address _owner) constant
          public returns (uint256);
      ...
    }
  contract Bitcash {
      ...
      function getTokenBal(address tokenAddr,
          address who) constant public returns (uint
          ){
          ForeignToken t = ForeignToken(tokenAddr);
          uint bal = t.balanceOf(who);
          return bal;
        }
    }
```

Fig. 6. Code example: No state change after external call

3) Change State Variable without Financial Risk: Some smart contracts may change state variables after the external call, making them potentially vulnerable to reentrancy attacks. However, not all state changes lead to reentrancy vulnerability. Take the function in Fig. 7 as an example. The code in line 3 defines an external call to address '*token*', and then the code in line 4 assigns it to a state variable *tokens*. There is no following code in the function, and this function is not called in any other functions in the contract[3]. Reentering this function in line 3 will not cause financial risk because the *transferFrom()* function is to transfer tokens from the first parameter (*msg.sender* in this case) to the second parameter (*this* smart contract in the case). Thus, *transferFrom()* is for the function caller to transfer tokens to this smart contract, and reentering this function only increases the balance of this smart contract. In this case, no financial risk exists in the contract.

4) Special Transfer Value: This type of false positive is caused by the ignorance of parameter semantics in the ether

[3]The full Solidity code of the contract can be found in 0x046ec9bb312c51425f7a00b2ab7525afe7db52e

```
  function depositToken(address token, uint amount)
      {
      ...
      if (!Token(token).transferFrom(msg.sender,
          this, amount)) throw;
      tokens[token][msg.sender] = safeAdd(tokens[
          token][msg.sender], amount);
    }
```

Fig. 7. Code example: Change state variable without financial risk

```
  function tradeEthVsDAI(uint numTakeOrders, uint
      numTraverseOrders, bool isEthToDai, uint
      srcAmount) public payable {
      ...
      if (isEthToDai) {
          require(msg.value == srcAmount);
          wethToken.deposit.value(msg.value)();
          ...
      } else ...
    }
```

Fig. 8. Code Example: Special Transfer Value

transfer function. For example, the function in Fig. 8 is detected to be reentrant due to the ether transfer function in line 5. However, the amount parameter is *msg.value*, which means the ether transferred to the external address *wethToken* is the ether amount received by the function *tradeEthVsDAI*. In this sense, reentering this function would not cause financial loss since it does not affect the balance of the smart contract.

5) Reentrancy by transfer/send: The gas system is a special mechanism introduced by Ethereum to limit the resource consumption of smart contracts. Once the smart contract runs out of gas, the execution will be terminated and all states will be rolled back. Unlike the *call.value()* function, *transfer()* and *sen()d* will change the maximum gas limitation to 2,300 units when the recipient is a contract. For the *transfer()* function in Fig. 9, the 2,300 gas limits is insufficient for a write operation to any storage variable, which means that the attackers cannot raise reentrancy attack.

6) Non-callable Function: There are two types of bytecode in Ethereum: runtime bytecode and creation bytecode. Creation bytecode contains the information that will never be executed after the contract is deployed on a blockchain, e.g., the constructor function. However, Mythril fails to identify this situation. When the source code is used as the input, Mythril simply compiles the source code to creation bytecode and uses it for vulnerability detection. This gap between creation

```
  function _withdraw(address from,address payable to
      ,address token,uint256 amount) internal {
      ...
      if(token == address(0)) {
          to.transfer(amount);
      } else ...
    }
```

Fig. 9. Code example: Reentrancy by transfer/send

TABLE VI
REAL REENTRANCY ATTACKS ON ETHEREUM

No.	Time	Lost	Attacked Projects	Description
1	2020/04	$3.5M	Uniswap	A vulnerability when adopting ERC777 tokens
2	2020/04	$25M	Lendf.Me	A vulnerability when adopting ERC777 tokens
3	2020/11	$2M	Akropolis	A combination of reentrancy attack and flash loan attack to exploit the saving pools
4	2020/11	$8M	OUSD	A combination of reentrancy attack and flash loan attack by utilizing mint logic flaw in its contracts
5	2021/07	$0.1M	DeFiPie	A combination of reentrancy attack and flash loan attack to withdraw almost all available liquidity from the protocol
6	2021/08	$18M	Cream Finance	A combination of reentrancy attack and flash loan attack to exploit a vulnerability in AMP token contract
7	2022/03	$1M	Bacon Protocol	A logic error in its lend() routine to allow hacker can get more lending credits
8	2022/03	$0.1M	Revest Finance	Bad design in the minting-related functions — do not strictly follow the check-validation-interaction model when transferring the ERC1155 token.

bytecode and runtime bytecode results in a type of false positive located in the constructor function. Since this type of function will not be recorded on the blockchain, it will never be called and be attacked by malicious attackers.

D. Distribution of False Positives in Each Tool

According to the sampled false positives, we present the distribution of the false positives reported by each tool in Table V. Since Smartian only reports 15 false positives and none of these sampled false positives is reported by Smartian, we only discuss the false positives reported by other tools. We have the following observations: 1) The items in *Permission Control* occupy the most, 55% ($\frac{170+30+6}{377}$). This is because there are several ways to deal with the permission of the contracts, which are hard for analyzers to handle. This result also indicates the most pressing problem to be solved by future work. 2) These tools also report two types of false positives, *No State Change After External Call* and *Change State Variable without Financial Risk*, which occupy 42% ($\frac{116+43}{377}$). These two types relate to the functions in other contracts, which increases the difficulty of analysis. As for other types of false positives (i.e., *Special Transfer Value*, *Reentrancy by Transfer/Send* and *Non-callable Function*), they occupy 14% ($\frac{10+9+33}{377}$) of the false positives.

V. DISCUSSION

A. Reentrancy in the wild

With the development of the smart contract ecosystem, reentrancy vulnerability has become much rarer in recent years. First, dozens of tools are developed to detect reentrancy. For example, the Ethereum's official IDE (Remix) warns of risk when the check-effects-interaction pattern is detected in a smart contract. In addition, as one of the most famous attacks on Ethereum, the DAO attack (the first smart contract reentrancy attack) has been widely introduced in smart contract tutorials, e.g., books, blogs, and videos. Thus, developers are well educated in avoiding reentrancy caused by *call.value()*.

Unfortunately, most academic works still focus on reentrancy caused by *call.value()*, which might be the wrong direction based on today's Ethereum ecosystem. Based on the SlowMist hacked repository[4], only eight real reentrancy attacks happened in Ethereum after 2020, as shown in Table VI. It is not difficult to find that these real reentrancy attacks are much more complicated compared with the traditional *call.value()* related reentrancy issues detected by most tools. These eight attacks could be classified into three groups. First, the No. 1, 2, and 8 reentrancy attacks are caused by based designs when using ERC tokens [34], e.g., ERC777 and ERC1155. The bad design provides opportunities for hackers to transfer the tokens in these contracts. Second, the No. 3-6 reentrancy attacks should be combined with a flash loan[5], as this kind of reentrancy attack usually needs a large amount of capital to affect some key values of smart contracts. For example, buying almost all the tokens in a liquidity pool might lead to a logic error in contracts. Third, the No. 7 reentrancy attack was led by a logic error in the contract. In this case, attackers could obtain more lending credits than they paid.

The aforementioned reentrancy attacks are very complicated. Detecting them needs a high level of professional skill and a deep understanding of reentrancy attacks, rather than using a simple detection pattern. Evidence can be found in the case of the Akropolis smart contract (No. 3 attack), which was audited by two professional smart contract audit teams. However, neither of them found the reentrancy issues. This shows that some real reentrancy attacks are much more complicated and need more advanced methods to be detected.

B. IDE Warning of Reentrancy

With the development of the smart contract language, the Ethereum's official IDE, i.e., Remix, has an integrated static analyzer to check for vulnerabilities, bad development practices, etc. The analyzer provides warnings for potential reentrancy vulnerability if the check-effects-interaction pattern is detected in the smart contract. A natural question is whether the detected true positive contracts could be warned when developing them with Remix. We test all of the 31 true positives using Remix, and find that 17 (54.8%) of them are warned to have a potential reentrancy vulnerability.

VI. THREATS TO VALIDITY

Threats to internal validity relate to the errors in the implementation of the reentrancy detection tools and the bias of participants in the manual examination of the reentrant contracts detected by the tools and the cause analysis of true (*resp.* false) positive contracts with (*resp.* without) reentrancy.

[4]https://hacked.slowmist.io/en/

[5]Flash loans allow users to borrow and settle loans instantaneously in a single transaction without providing any collateral.

We implement the tools using their Docker images or Dockerfiles released at GitHub. For the manual examination task, we perform two rounds to reduce the heavy workload of 21,212 contracts. In the first round, we recruit 48 participants willing to participate in our task and ensure that they have relatively sufficient knowledge of Solidity and reentrancy to perform the task, adopting a four-stage process (Section III-D1). In the second round, we ask two PhDs with experience in Solidity and reentrancy to review the examination results of the first round and re-examine the contracts with low-quality results. For the cause analysis task, we ask both PhDs to collaboratively analyze the causes of true and false positive contracts. To avoid the subjective bias of a single person, the manual examination and cause analysis tasks for each contract are performed by two participants (including the two PhDs). It should be pointed out that the participants might miss some possible reentrancy issues in contracts that they do not know or have not been trained with.

Threats to external validity relate to the generalizability of the results. The objective of this study is to investigate the capability of existing works on reentrancy detection for smart contracts. We collect a large dataset of 230,548 contracts with Solidity code from Etherscan and select five well-known or recent reentrancy detection tools according to some key criteria for practical use (Section III-B1). We exclude some popular tools such as Slither and sFuzz as they either cannot be applied to Solidity code or cannot locate the possibly defective functions in contracts. We acknowledge that our results obtained using five tools cannot reflect all existing works on reentrancy detection. However, the results can help researchers gain a good understanding of the limitations of existing works and motivate researchers to address new reentrancy issues in spite of those related to *call.value()*.

VII. Related Work

A. Reentrancy Detection Tools

Reentrancy is one of the most notorious vulnerabilities, first discovered in 2016 [26], and continually discussed since. Due to the significance of this security issue, many detection tools have been proposed to prevent smart contracts from being attacked, including dynamic testing tools and static analyzers. For example, ContractFuzzer [35], sFuzz [23], and Smartian [24] use dynamic testing [36] technologies to trigger reentrancy issues. ContractFuzzer is the very first fuzzing tool, which develops test oracles [37] for the reentrancy vulnerability. Due to its inefficiency, Nguyen et al. propose an adaptive strategy in sFuzz to guide fuzzer toward executing unreached paths. However, these tools pay little attention to the characteristics of smart contracts. Choi et al. claim that some paths can only be reached by critical transaction sequences and thus propose Smartian.

Oyente [26], Securify [30], Slither [38], and Zeus [39] use static analysis technologies to discover reentrancy vulnerability. In 2016, Luu et al. propose Oyente which first addresses the problem of reentrancy detection. Oyente constructs the control flow graph of contracts and identifies reentrancy issues by risky patterns [26]. Due to the lack of context information, Oyente is imprecise. Tsankov et al. propose Securify to extract semantic information from smart contracts and perform analysis to check the existence of predefined patterns. Furthermore, Feist et al. propose Slither to perform automatic analysis [38]. Bose et al. put forward Slither to handle state-inconsistency bugs (e.g., reentrancy) and achieve a better performance [39].

B. Benchmark in Smart Contracts

A benchmark is a foundation for comparing different methods [40]. However, only a few benchmarks have been proposed in the area of smart contracts. *VeriSmartBench*[6] is one of these benchmarks, which contains examples for all CVE cases. Unfortunately, this repository contains very few examples of reentrancy issues. *SolidiFi-benchmark* [15] is another benchmark which contains 31 examples with reentrancy issues, but many of them are many artificial and toy examples.

Efforts have been made in other empirical studies to find different aspects of insight. Durieux et al. [15] conduct a large-scale evaluation of automated analysis tools. Rather than focusing on specific vulnerabilities, they pay close attention to the effectiveness and efficiency of distinct tools. In addition, Perez et al. [41] focus on evaluating whether vulnerable contracts have been exploited. They reveal that only a small proportion of smart contract vulnerabilities have been profited from. Xue et al. [16] reveal five *path protective techniques* which are relevant with the false alarm of reentrancy detection. Yet, these approaches only cover a portion of Solidity versions and cannot be used to draw a comprehensive conclusion.

VIII. Conclusions and Future Work

In this paper, we conduct a large-scale empirical study to investigate the performance of existing works on reentrancy detection for smart contracts. We collect 230,548 contracts from Etherscan and select five well-known or recent tools that can locate the possibly defective functions with reentrancy issues in a contract. By manually examining 21,212 reentrant contracts detected using these tools, we obtain 31 true positive contracts with reentrancy and 21,181 false positive contracts without reentrancy. We also analyze the causes of the true and false positives. Using the two sets of contracts with different causes, we evaluate the tools. Two additional tests are conducted to evaluate the tools on a number of contracts with recent reentrancy attacks and to verify whether the detected reentrancy issues can be identified by the Ethereum's official IDE, Remix. The results show that the tools have a extremely high false positive rate of more than 99.8% and that the tools can only detect reentrancy related to *call.value()*, 54.8% of which can be detected by Remix. Based on the results, we suggest that researchers turn to discovering and detecting new reentrancy issues in spite of the classical and simple patterns related to *call.value()*. In future work, we plan to improve existing reentrancy detection tools by reducing false positives based on our summarized causes and to study the reentrancy issues reported in recent attacked contracts.

[6]https://github.com/soohoio/VeriSmartBench

REFERENCES

[1] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[2] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[3] F. Hofmann, S. Wurster, E. Ron, and M. Böhmecke-Schwafert, "The immutability concept of blockchains and benefits of early standardization," in *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*. IEEE, 2017, pp. 1–8.

[4] 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)*. IEEE, 2020, pp. 125–134.

[5] F. Ma, Z. Xu, M. Ren, Z. Yin, Y. Chen, L. Qiao, B. Gu, H. Li, Y. Jiang, and J. Sun, "Pluto: Exposing vulnerabilities in inter-contract scenarios," *IEEE Transactions on Software Engineering*, 2021.

[6] C. Ferreira Torres, M. Baden, R. Norvill, B. B. Fiz Pontiveros, H. Jonker, and S. Mauw, "Ægis: Shielding vulnerable smart contracts against attacks," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 584–597.

[7] "Swc registry," 2022. [Online]. Available: https://swcregistry.io/

[8] "The ncc group," 2022. [Online]. Available: https://www.dasp.co/

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

[10] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, "Zether: Towards privacy in a smart contract world," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 423–443.

[11] 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.

[12] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[13] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[14] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, p. e00938, 2018.

[15] 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.

[16] 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.

[17] "Solidity," 2022. [Online]. Available: https://docs.soliditylang.org/en/v0.8.16/index.html

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

[19] "Etherscan," 2022. [Online]. Available: https://etherscan.io/

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

[21] 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.

[22] "Our experimental data," 2022. [Online]. Available: https://github.com/BeaconOfReentrancy/ReentrancyDataset

[23] 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.

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

[25] 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.

[26] 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*, 2016, pp. 254–269.

[27] 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*, 2018, pp. 653–663.

[28] 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.

[29] "Consensys," 2022. [Online]. Available: https://consensys.net/

[30] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[31] "Docker hub," 2022. [Online]. Available: https://hub.docker.com/

[32] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[33] "Reentrancyguard," 2022. [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/security

[34] R. Norvill, B. Fiz, R. State, and A. Cullen, "Standardising smart contracts: Automatically inferring erc standards," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2019, pp. 192–195.

[35] B. Jiang, Y. Liu, and W. 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.

[36] F. A. Hanson, "Testing testing," in *Testing Testing*. University of California Press, 1993.

[37] L. Baresi and M. Young, "Test oracles," 2001.

[38] 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.

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

[40] F. Perazzi, J. Pont-Tuset, B. McWilliams, L. Van Gool, M. Gross, and A. Sorkine-Hornung, "A benchmark dataset and evaluation methodology for video object segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 724–732.

[41] 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.