Why Smart Contracts Reported as Vulnerable were not Exploited?

Tianyuan Hu [1], Jingyue Li [2], André Storhaug [2], and Bixin Li [2]

[1]Southeast University
[2]Affiliation not available

October 30, 2023

Abstract

As smart contracts process digital assets, their security is essential for blockchain applications. Many approaches have been proposed to detect smart contract vulnerabilities. Studies show that few of the reported vulnerabilities are exploited and hypothesize that many of the reported vulnerabilities are false positives. However, no follow-up study is performed to confirm the hypothesis and understand why the reported vulnerabilities are not exploited. In this study, we first collect 136,969 unique real-world smart contracts and analyze them using four vulnerability detectors, namely Oyente, SmartCheck, Slither, and SolDetector. Then, we apply Strauss' grounded theory approach to manually analyze the source code of the smart contracts reported as vulnerable to recognizing false positives and understand the reasons for false results. In addition, we analyze the transaction logs of the smart contracts reported as vulnerable to identifying and understanding their exploitations. Our results show that 75.37% of the 4,364 smart contracts reported as vulnerable are false positives, and eleven reasons are causing the false positives. After analyzing the 4,106,134 transaction logs of the contracts reported as vulnerable, we find that vulnerabilities of only 67 (0.015%) of the contracts have been exploited in history. We also identify six reasons for demotivating and preventing the attackers from exploiting the vulnerabilities. Our results reveal that state-of-the-art smart contract vulnerability detectors primarily treat the smart contracts as yet another application developed using Object Oriented (OO) languages when analyzing and reporting the smart contract vulnerabilities. Without considering the specific design principles of the Solidity programming language and the characteristics of smart contracts' application scenarios and execution environments, many of the reported vulnerabilities are not exploitable or not cost-effective to be exploited by adversaries.

Why Smart Contracts Reported as Vulnerable were not Exploited?

Tianyuan Hu, Jingyue Li, André Storhaug, Bixin Li

Abstract—As smart contracts process digital assets, their security is essential for blockchain applications. Many approaches have been proposed to detect smart contract vulnerabilities. Studies show that few of the reported vulnerabilities are exploited and hypothesize that many of the reported vulnerabilities are false positives. However, no follow-up study is performed to confirm the hypothesis and understand why the reported vulnerabilities are not exploited. In this study, we first collect 136,969 unique real-world smart contracts and analyze them using four vulnerability detectors, namely *Oyente*, *SmartCheck*, *Slither*, and *SolDetector*. Then, we apply Strauss' grounded theory approach to manually analyze the source code of the smart contracts reported as vulnerable to recognizing false positives and understand the reasons for false results. In addition, we analyze the transaction logs of the smart contracts reported as vulnerable to identifying and understanding their exploitations. Our results show that 75.37% of the 4,364 smart contracts reported as vulnerable are false positives, and eleven reasons are causing the false positives. After analyzing the 4,106,134 transaction logs of the contracts reported as vulnerable, we find that vulnerabilities of only 67 (0.015%) of the contracts have been exploited in history. We also identify six reasons for demotivating and preventing the attackers from exploiting the vulnerabilities. Our results reveal that state-of-the-art smart contract vulnerability detectors primarily treat the smart contracts as yet another application developed using Object Oriented (OO) languages when analyzing and reporting the smart contract vulnerabilities. Without considering the specific design principles of the Solidity programming language and the characteristics of smart contracts' application scenarios and execution environments, many of the reported vulnerabilities are not exploitable or not cost-effective to be exploited by adversaries.

Index Terms—Ethereum, smart contract, vulnerability detection, source code analysis

1 Introduction

Due to blockchains' monetary and anonymous nature, they are targets of adversaries. The security of smart contracts is critical because they may handle and store digital assets worth millions of dollars. The DAO hack [1] exploiting the reentrancy vulnerability in smart contract code resulted in a 60 million dollars loss. It is, therefore, imperative to prune out smart contracts' security problems before deploying them.

Many methods and corresponding tools, e.g., *Oyente* [2], *Securify* [3], *sFuzz* [4], *ContractFuzzer* [5], *ContraMaster* [6], *DefectChecker* [7], *EXGEN* [8], *HONEYBADGER* [9], have been proposed to detect smart contract vulnerabilities. Ren et al. [10] point out that the tools are evaluated by the tool authors using different datasets and metrics, which may result in biased conclusions. Several empirical studies were conducted by other researchers using manually annotated datasets or real-world smart contracts to evaluate these tools fairly. *SolidiFI* [11] is used to evaluate six tools [2], [3], [12], [13], [14], [15]. The results show that none of the tools detect all the injected bugs correctly, and all the evaluated tools report several false positives. Durieux et al. [16] evaluated nine vulnerability detectors [2], [3], [9], [12], [13], [14], [15], [17], [18], and found that 97% of the real-world contracts analyzed were labeled as vulnerable by the detectors. Perez and Livshits [19] analyzed 821,219 real-world contracts using six tools, namely *Oyente* [2], *ZEUS* [20], *MAIAN* [17], *Securify* [3], *TEETHER* [21], *Madmax* [22]. They classified the analyzed smart contracts as:

- **Reported as vulnerable:** A contract is reported as vulnerable if it is flagged as vulnerable by the vulnerability detector.
- **Truly vulnerable:** A contract is truly vulnerable if an external attacker could exploit its vulnerability.
- **Exploited:** A contract is exploited if a transaction on Ethereum's main network has triggered one of its vulnerabilities.

Results of the study by Perez and Livshits [19] show that, among the 73,62 contracts reported as vulnerable by at least two tools, only 463 contracts were exploited. They hypothesized that most reported vulnerabilities are false positives. However, no follow-up study tried to confirm the hypothesis and understand the reasons for the false positives. Christakis and Bird's survey in Microsoft [23] concluded that "90% of developers accept 5% false positives, 50% of developers accept 15% false positives, and only 24% of developers accept more than 20% false positives." Thus, it is vital to reduce smart contract vulnerability detectors' false positive rates. In this study, we are motivated to answer two research questions (RQs):

- **RQ1:** Whether smart contracts reported as vulnerable by existing tools are truly vulnerable, and why do the false positives occur?
  As aforementioned, the definition of true vulnerability by Perez and Livshits [19] is "the attacker

- *T. Hu and B. Li are with the School of Computer Science and Engineering, Southeast University, Nanjing, 211189, China.E-mail: tianyuan.hu@foxmail.com, bx.li@seu.edu.cn*
- *J. Li and A. Storhaug are with the Department of Computer science, Norwegian University of Science and Technology, Trondheim, Norway. E-mail: jingyue.li@ntnu.no, andre.storhaug@ntnu.no*

Manuscript received April 19, 2005; revised August 26, 2015.

could exploit the vulnerability," meaning the vulnerability could be executed by the adversary, and the execution could lead to security compromises. Accordingly, our definition of false positive is "the reported vulnerability could neither be executed by the adversary nor possibly lead to security compromise."

- **RQ2:** Whether the vulnerable smart contracts are exploited and what can prevent attackers from exploiting them?
  If a smart contract is found to be truly vulnerable, i.e., executable by the adversary and possibly lead to security compromise, we want to know if it has been executed at least once, although we may not know exactly the actual loss due to the execution. In addition, we want to identify commonalities between the vulnerable smart contracts that have and have not been executed.

To answer RQ1, we collected 136,969 unique real-world smart contracts and used four vulnerability detectors, namely, *Oyente* [2], *SmartCheck* [13], *Slither* [14], *SolDetector* [24] to label their vulnerabilities. We then analyzed the reported vulnerabilities of ten popular vulnerability types supported by at least two tools. We manually analyzed the source code of the smart contracts reported as vulnerable to judge whether they are truly vulnerable and adopted Strauss' grounded theory method [25] to identify the reasons for false positives. To answer RQ2, we collected the transaction logs of the reported vulnerable contracts and replayed their transactions on a full Ethereum node. We designed transaction log analysis rules to identify the vulnerability exploitation and used Strauss' grounded theory method to understand the contracts' exploitations and non-exploitations.

Results of the study confirmed the hypothesis by empirical studies, e.g., [16] and [19], that there are many false positives of reported vulnerable contracts. Among the 4,364 vulnerable contracts labeled by at least two of our evaluated tools, 3,272 are false positives. After analyzing the 4,106,134 transaction logs of the 4,364 smart contracts reported as vulnerable, we found vulnerabilities of only 67 contracts were exploited.

To understand the reasons for the low exploitation rate of the reported vulnerabilities, we applied the grounded theory analysis approach and analyzed the source code and transaction logs of the smart contracts. Through open and axial coding, we identified eleven reasons causing the vulnerability detectors to report false positives and six reasons that may have demotivated adversaries to attack the true vulnerabilities. Through selective coding, we identified two main themes of reasons for the low exploitation rate. One theme is that the studied vulnerability detectors mainly adapted existing approaches to analyze OO applications with weaknesses. Another theme is that the detectors have not sufficiently considered Solidity's principle as a contract-oriented programming language for applications running on EVM and blockchain to securely transfer assets or ether between supplier and client and minimize the gas cost of execution. The reported vulnerabilities can trigger false alarms and mislead smart contract developers without considering these characteristics.

The contributions of this study are:

- We empirically confirmed the hypothesis by many existing studies that a high false positive rate is a critical challenge of the state-of-the-art smart contract vulnerability detectors.
- We have identified eleven reasons for false positives, which are valuable for many vulnerability detectors to improve and reduce their false positive rates.
- We have identified six explanations for the non-exploitation of the vulnerable smart contracts, which can help security engineers to rank the reported vulnerability and to design possible mitigation strategies.
- We have derived theories regarding the reasons for the low exploitation rate and raised the alarm that the community working on the smart contract vulnerability detectors needs to consider the specific characteristics of the smart contract programming language and smart contracts' application scenario and execution environment when analyzing, reporting, and ranking the smart contract vulnerabilities.
- Last but not least, we have created a benchmark dataset containing 4,364 real-world Solidity smart contracts, which are manually labeled with ten types of vulnerabilities. The dataset is around 20 times bigger than the similar state-of-the-art benchmark [10]. The dataset can help evaluate the vulnerability detector tools more consistently and objectively and is available at https://github.com/1052445594/SC_UEE.

The rest of the paper is organized as follows. Section 2 introduces related work, and Section 3 presents the research design. The answers to RQ1 and RQ2 are given in Sections 4 and 5, respectively. Section 6 discusses the results and Section 7 concludes.

2 Related Work

The approaches to detect smart contract vulnerabilities can be classified into pattern matching, symbolic execution, dependency analysis, machine learning (ML), and fuzzing, as shown in Table 1.

2.1 Tools Using Pattern Matching Approaches

SmartCheck [13] translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns. *SolDetector* [24] is a static detection tool based on the knowledge graph of Solidity source code. For each smart contract to analyze, it constructs the knowledge graph containing the ontology and instance layers. Based on the knowledge graph, it uses the SPARQL [33] query to manipulate the knowledge graph and identify the defect.

2.2 Tools Relying on Symbolic Execution

Oyente [2] builds control flow graphs from smart contract bytecode to identify vulnerabilities using vulnerability patterns. By analyzing dependency diagrams of smart contracts, *ZEUS* [20] combines abstract interpretation and

TABLE 1: Smart Contract Vulnerability Detection Tools (*SC* represents source code; *BC* represents bytecode)

Year and ref.	Tool name	Vul. types covered	Inputs
Pattern Matching			
2018 [13]	SmartCheck	37 types	*SC*
2021 [24]	SolDetector	20 types	*SC*
Symbolic Execution			
2016 [2]	Oyente	6 types	*SC*
2018 [20]	ZEUS	7 types	*SC*
2018 [12]	Mythril	SWC Registry	*SC*
2018 [3]	Securify	37 types	*SC*/*BC*
2018 [21]	TEETHER	4 types	*BC*
2018 [17]	MAIAN	3 types	*SC*/*BC*
2019 [9]	HONEYBADGER	Honeypots	*BC*
2021 [7]	DefectChecker	8 types	*SC*
2022 [8]	EXGEN	4 types	*SC*
Data Flow Analysis			
2018 [18]	Osiris	Integer Vulnerability	*BC*
2018 [22]	MadMax	3 types	*BC*
2019 [14]	Slither	71 types	*SC*
2020 [26]	Clairvoyance	Reentrancy	*SC*
2020 [27]	Ethainter	5 types	*BC*
Machine Learning			
2019 [28]	GNN-based	3 types	*SC*
2020 [29]	ContractWard	6 types	Opcode
2021 [30]	VSCL	6 types	*BC*
Fuzzing			
2018 [5]	ContractFuzzer	7 types	*BC*+ABI
2018 [31]	Reguard	Reentrancy	*SC*/*BC*
2020 [4]	sFuzz	9 types	*BC*
2020 [32]	Ethploit	3 types	*SC*

symbolic execution to model smart contracts. As *ZEUS* analyzes artifacts at the low level virtual machine (LLVM) intermediate level, it cannot locate vulnerabilities at the Ethereum virtual machine (EVM) bytecode level. *Mythril* [12] uses symbolic execution and taint analysis to detect vulnerabilities. It performs decompilation and produces execution traces using a dynamic symbolic execution engine called Laser-EVM. However, *Mythril* is slow due to multiple symbolic executions. *Securify* [3] combines abstract interpretation and symbolic execution. The tool automatically classifies behaviors of a contract into three categories, compliance (matched by compliance properties), violation (matched by violation properties), and warning (no matches). Krupp et al. [21] first give a generic definition of vulnerable contracts and build *TEETHER*, a tool that employs symbolic execution to create an exploit automatically. However, *TEETHER* has difficulty solving hard constraints in execution paths and cannot simulate the blockchain behaviors very well, causing a loss of coverage. *MAIAN* [17] is a symbolic execution tool analyzing EVM bytecode. *MAIAN* classifies vulnerable contracts into three categories, namely, greedy, prodigal, and suicidal. *HONEYBADGER* [9] uses symbolic execution and pre-defined heuristics to expose honeypots. *DefectChecker* [7] symbolically executes the smart contract bytecode and generates their control flow graphs, stack events, and other features. Based on the information, it uses eight rules to detect different types of vulnerabilities. However, the public version of *DefectChecker* supports only Solidity 0.4.24. *EXGEN* [8] generates multiple transactions as exploits to vulnerable smart contracts and verifies the generated contracts' exploitability on a private chain with values crawled from the public chain. *Osiris* [18] is a framework that combines symbolic execution and taint analysis to detect vulnerabilities related to arithmetic operations in Ethereum smart contracts.

2.3 Tools Applying Data Flow Analysis

MadMax [22] is a gas-focused vulnerability detection tool consisting of a decompiler, which converts low-level EVM bytecode to code represented using an intermediate language. It then analyzes the code to detect out-of-gas vulnerabilities that require coordination across multiple transactions. *Slither* [14] is a highly scalable static analysis tool. It first converts Solidity smart contracts to an intermediate representation called SlithIR through control flow graph analysis. Then, it applies both data flow and taint analysis to detect vulnerabilities. *Clairvoyance* [26], [34] presents a static analysis tool that models cross-function and cross-contract behavior to detect the reentrancy vulnerability. Brent et al. [27] present *Ethainter* to detect composite vulnerabilities that escalate a weakness through multiple transactions. Based on the Datalog language [35] and the Soufflé Datalog engine [36], *Ethainter* constructs graphs containing data flow and control flow dependencies to identify vulnerabilities.

2.4 Tools Using Machine Learning Technologies

Zhuang et al. [28] use a graph neural network to classify vulnerable smart contracts. *ContractWard* [29] is machine learning-based vulnerability detection tool targeting six vulnerabilities. It employs three supervised ensemble classification algorithms, namely, XGBoost, AdaBoost, and Random Forest (RF), and two classification algorithms, namely, Support Vector Machine (SVM) and *k*-Nearest Neighbor (KNN). Their evaluations show that XGBoost is the best-performing classifier algorithm. *VSCL* [30] is a smart contract vulnerability detection framework that constructs a control flow graph (CFG) to understand program run time behavior. Further, n-gram and Term Frequency–Inverse Document Frequency (TFIDF) techniques are used to generate numeric values (vectors) to present features of smart contracts. Finally, the generated feature matrix is used as input for the deep neural network (DNN) model.

2.5 Tools Using Fuzz Testing

Fuzz testing [37] is an automated testing technique for analyzing computer programs. *ContractFuzzer* [5] is a fuzzing tool that generates random inputs to smart contracts according to the contracts' Application Binary Interface (ABI). *ContractFuzzer* defines a set of predefined test oracles that describes specific vulnerabilities. However, due to the randomness of the inputs, *ContractFuzzer*'s execution covers only limited system behavior. *ReGuard* [31] is a fuzzing tool to detect reentrancy vulnerabilities. It first converts the input to smart contracts into a C++ program via the Abstract Syntax Tree (AST) or CFG and generates random inputs to perform the fuzzing. *sFuzz* [4] employs an efficient, lightweight, adaptive strategy for selecting seeds to improve the fuzzing method based on random input generator [5]. *EthPloit* [32] adopts static taint analysis to generate exploit-targeted transaction sequences. It uses a dynamic seed strategy to pass hard constraints and an instrumented EVM to simulate blockchain behaviors. *ContraMaster* [6] is an oracle-supported dynamic exploit generation framework that can

mutate transaction sequences. It uses data flow, control flow, and dynamic contract state to guide its mutations of the transaction sequences.

2.6 Empirical Evaluations of Vulnerability Detectors

Although studies proposing new detection tools always provide evaluation results, the evaluations can be biased. The studies may use different terms and definitions of the same vulnerability and use datasets that favor their tools. Thus, other researchers performed empirical studies as shown in Table 2 to evaluate and compare the smart contract vulnerability detectors.

TABLE 2: Empirical Study of Vulnerability Detectors

Year and ref.	2020 [11]	2020 [16]	2021 [10]	2021 [19]
SmartCheck [13]	✓	✓		
Oyente [2]	✓	✓	✓	✓
ZEUS [20]				✓
Securify [3]	✓	✓		✓
Mythril [12]	✓	✓	✓	
Slither [14]	✓	✓		
Manticore [15]	✓	✓		
MAIAN [17]		✓		✓
Orisis [18]		✓	✓	
HONEYBADGER [9]		✓		
ContractFuzzer [5]			✓	
TEETHER [21]				✓
MadMax [22]				✓
sFuzz [4]			✓	

Ghaleb et al. [11] proposed *SolidiFI* to evaluate six static analysis tools [2], [3], [12], [13], [14], [15] using a dataset with injected vulnerabilities. They assumed that a vulnerability reported by most tools could not be a false positive. Experiment results on a set of 50 contracts injected with 9369 distinct vulnerabilities show that the evaluated tools do not detect several instances of vulnerabilities despite their claims of being able to detect such vulnerabilities. Only one tool, i.e., *Slither* [14], detected all injected reentrancy and TxOrigin vulnerabilities. Ghaleb et al. [11] also found that all evaluated tools have reported several false positives, ranging from 2 to 801 for different vulnerability types.

Ferreira et al. [38] presented *SmartBugs*, an extensible and easy-to-use execution framework that simplifies the execution of tools analyzing Solidity smart contracts. *SmartBugs* supports ten tools [2], [3], [9], [12], [13], [14], [15], [17], [18], [39] and provides two datasets of Solidity smart contracts. One dataset contains 143 annotated vulnerable contracts with 208 tagged vulnerabilities, and another contains 47,518 unique contracts collected through Etherscan [40]. However, the 47,518 real-world contracts are not manually labeled. By using *SmartBugs*, Durieux et al. [16] evaluated nine tools [2], [3], [9], [12], [13], [14], [15], [17], [18]. The evaluation was based on 69 annotated vulnerable smart contracts and all the real-world smart contracts in *SmartBugs*. The evaluation results showed that 97% of the real-world contracts were labeled as vulnerable. Durieux et al. [16] questioned that many reported vulnerabilities are false positives.

Ren et al. [10] evaluated six tools [2], [4], [5], [12], [18], [41], and proposed a unified standard to eliminate the evaluation biases. They constructed a benchmark suite with three datasets, including unlabeled real-world contracts (UR), contracts with manually injected vulnerabilities (MI), and confirmed vulnerable contracts (CV). The experiment results on these datasets demonstrated that different choices of experimental settings could significantly affect tool performance and lead to misleading or even opposite conclusions. The experiment results in [14] show that *SmartCheck* has more false positives than *slither*. However, Ren's study [10] gives opposite conclusions and shows that *SmartCheck* reports fewer false positives than *slither* on UR and MI datasets.

Perez et al. [19] evaluated six tools [2], [3], [17], [20], [21], [22] on real-wold smart contracts and found many contradict results from different tools [19]. Taking the reentrancy vulnerability as an example, *Oyente* and *Securify* agree on only 23% of the contracts reported as vulnerable to reentrancy, while *ZEUS* does not agree with any other tools [19]. In addition, they analyzed more than 20 million Ethereum blockchain transactions and found that only 463 contracts related to six vulnerability types were exploited. Based on the evaluation results, they questioned whether the vulnerabilities reported by the evaluated tools were either false positives or not exploitable in practice.

Although the aforementioned empirical studies hypothesized that existing vulnerability detectors report many false positives, especially on real-world contracts, no follow-up study was performed to confirm the hypothesis and identify the reason for the false positives.

3 Research Design

3.1 Research Design to Answer RQ1

To answer RQ1, we designed the research flow as follows.

3.1.1 Step 1. Collect unique real-world smart contracts

We crawled all available smart contracts with at least one transaction from Etherscan [40] on 1st April 2022. As there are duplicated smart contracts, we filtered contracts for uniqueness with a similarity threshold of 0.9, calculated using the Jacard index [42]. This means that if two contracts' code shares more than 90% of the tokens, one of the contracts will be discarded. The low uniqueness requirement is due to the often large amount of embedded library code. If the requirement is set to high, the actual contract code will be negligible compared to the library code. Most contracts will be discarded, and the resulting dataset will contain mostly unique library code.

3.1.2 Step 2. Select vulnerability detectors

As shown in Table 1, there are many smart contract vulnerability detection tools. We use the following criteria to select the tools we focus on in this study.

- *The tools shall take smart contract Source Code as Input (SCI)*: As we want to confirm false positives and understand the reasons for them, we need to access the smart contract source code. Thus, we exclude tools that do not analyze Solidity smart contract source code.
- *The tools shall provide Vulnerability Localization (VL)*: To analyze the reported vulnerabilities precisely, we only consider tools that provide the location, i.e., code line number, of the vulnerabilities at the source

code level. The tools only label the smart contract as vulnerable without providing vulnerability location information are excluded.

- *The tools shall support multiple Solidity Versions (SV)*: When we crawl real-world smart contracts, we get smart contracts developed using various Solidity versions. If the tools support only a limited number of versions of Solidity, the smart contracts they can analyze are limited, meaning we cannot get sufficient vulnerable smart contracts to study. Thus, we require the tools to support several versions of Solidity.
- *The tools shall be efficient (Eff) to run*: The crawling of Etherscan [40] results in many smart contracts. If the detection tools are too slow to run, analyzing the smart contracts will take a long time. In addition, low-performance tools are unlikely to be used by industry practitioners. Thus, we exclude them from our study.
- *The tools shall be available to us (Available)*: Not all papers make their tools publicly available. We exclude tools we cannot access.

3.1.3 Step 3. Choose vulnerability types to focus and use the selected tools to analyze the chosen types of smart contracts

The vulnerability detection results from a particular tool can be biased by the tool's design flaws or bugs. As we want to identify generic reasons for false positives, we choose to analyze only the vulnerability types supported by at least two tools to reduce the possible biases introduced by a single tool.

After using the detectors to analyze the chosen smart contracts on the chosen vulnerability types, we get detection results containing vulnerability type names and locations.

3.1.4 Step 4. Analyze the vulnerabilities reported by the detection tools

The vulnerability detection tools report different vulnerabilities and their locations for the same smart contracts. Again, to avoid the biases caused by a single detector, we analyze only the smart contracts labeled as vulnerable to a particular vulnerability type by more than one tool. The chosen smart contracts are, hereafter, called cross-reported vulnerable contracts.

Analyzing detection reports aims to identify false positives and the corresponding reasons. We believe the reasons for the false positives are not limited to the imperfect implementations of the tools. There must be common and possibly unknown reasons that result in many tools reporting identical false positives. Thus, we choose to use Strauss' grounded theory approach [25], which is often used to identify generic and unknown theories from data. Strauss' grounded theory approach [25] is an iterative and recursive approach where the researchers must go back and forth until they achieve theoretical saturation. Our grounded theory analysis included several steps. First, we read the source code of each smart contract reported as vulnerable and classified them into two categories, i.e., truly vulnerable or false positive, in parallel with root cause analysis and open coding to code the reasons for the false results of each false positive. As a second step, these codes are grouped into conceptual categories through axial coding. We did a constant comparison and theoretical saturation to consolidate the reasons for the false positives across vulnerability types. The analysis ended when we could not derive more categories of reasons from the open codes. The axial coding resulted in eleven reasons for the false positives, which are explained in Section 4.5.

3.2 Research Design to Answer RQ2

The steps to answer RQ2 are as follows.

3.2.1 Step 1. Collect transaction logs

For all the cross-reported vulnerable contracts, we retrieve their transaction logs on Ethereum through the debug function of EVM, which supports replaying transactions and tracing transaction logs. The EVM's debug function is accessed through the Remote Procedure Call (RPC) provided by the Ethereum client.

3.2.2 Step 2. Analyze transaction logs to identify vulnerability exploitation

Step 4 to answer RQ1 identifies several false positives. For the false positive ones, we analyze their transaction logs to check if they are exploited. The purpose is to verify that our conclusions on the false positives are correct. We expect that there shall have no exploitation in the transactions logs of the false positive ones.

Step 4 to answer RQ1 also identifies true vulnerable smart contracts. We analyze these smart contracts' transactions to determine if the vulnerabilities have been exploited. We developed different detectors for each vulnerability type to analyze the vulnerability exploitation.

3.2.3 Step 3. Identify reasons for vulnerability exploitation

Step 2 finds many vulnerable smart contracts that are not exploited. We, again, use Strauss' grounded theory approach [25] to discover the possible reasons for this phenomenon. Besides the transaction logs, the extra data we analyze include the smart contracts' account types and balances. After the open coding and axial coding similar to what we did to answer RQ1, we derived several possible reasons, shown in Section 5.3.

In the end, we performed selective coding to connect reasons identified through axial coding of RQ1 and RQ2 to generate a coherent explanatory scheme. The selective coding resulted in two themes to explain the low exploitation rate of the reported vulnerabilities, namely, the weakness of adapting classical approaches to analyze smart contracts as OO applications and overlooking of specific characteristics of smart contract programming language and applications, which will be extensively recounted in Sections 4.5 and 5.3.

4 Results of RQ1

4.1 Collected Unique Smart Contracts

We crawled 2,217,692 smart contracts from Etherscan. From these contracts, 2,080,723 duplications were found, giving a duplication percentage of 93.82%. After duplication filtering, we got 136,696 unique smart contracts with 318,026,937 transactions (*before 2022.6.1, UTC+2 08:23:22*).

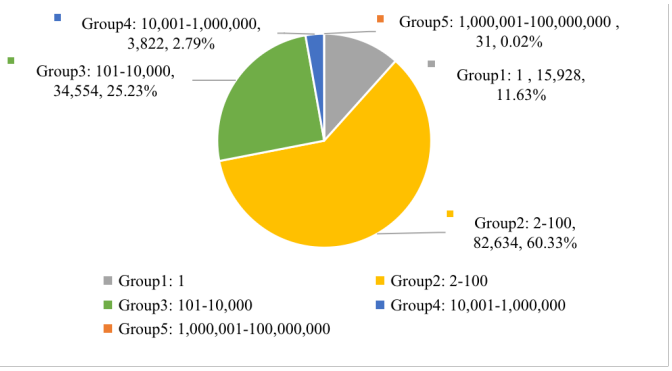

Fig. 1: Distribution of the Number of Transactions of the chosen Smart Contracts

Figure 1 shows the transaction information of these contracts and indicates that 88.37% of contracts have more than one transaction. Figure 1 also shows that the contracts have broad coverage of different numbers of transactions. Thus, we believe the chosen smart contracts are representative.

4.2 Selected Vulnerability Detectors

Based on the tool selection criteria, we choose to study four tools, namely, *Oyente* [2], *Smartcheck* [13], *slither* [14], *SolDetector* [24]. The chosen tools cover pattern matching, symbolic execution, and dependency analysis approaches. The other tools are excluded because they do not satisfy one or multiple selection criteria.

- *Source Code Input (SCI)*: *TEETHER* [21], *HONEYBADGER* [9], *Osiris* [18], *MadMax* [22], *Ethainter* [27], *VSCL* [30], *ContractFuzzer* [5], *sFuzz* [4] are excluded because they do not take source code as input. Even though the framework *SmartBug* [38] supports using *HONEYBADGER* [9] and *Osiris* [18] with the source code as input, we exclude them because they often report compilation errors, such as "Solc experienced a fatal error", when detecting real-world contracts and return null results. For example, when we run *HONEYBADGER* [9] and *Osiris* [18] with 100 real-world smart contracts, they reported 76 and 68 compilation errors respectively.
- *Vulnerability Localization (VL)*: *MAIAN* [17], GNN-based tool [28], *VSCL* [30], and *ETHPLOIT* [32] do not provide location information of the identified vulnerability and are, therefore, excluded.
- *Solidity Versions (SV)*: *Securify 2.0* [43] and *DefectChecker* [7] are excluded because they only support limited versions of Solidity. *DefectChecker* [7] aims at Solidity version 0.4.25, which is the most widely used version at the time of developing this tool. The updated tool *Securify 2.0* [43] only supports contracts written in Solidity after its version 0.5.8.
- *Efficiency (Eff)*: *Reguard* [31] needs to set the time limit for each contract detection because processing one smart contract can take more than 20 minutes. The study [7] demonstrated that *Mythril* is a slow tool and the maximum time to analyze a smart contract using *Mythril* is 2480.26s. We exclude *Reguard* and *Mythril* due to their slow performance.
- *Availability*: We cannot get access to the source or executable code of three tools [8], [20], [26], although we have contacted the paper authors and asked for the code.

A more structured summary of the reasons for excluding the detectors is shown in Table 8 in Appendix A.

4.3 Chosen Vulnerability Types

To choose vulnerability types supported by at least two tools, we did a mapping of the types between the tools and decided to focus on ten types of vulnerability, as shown in Table 3. It is worth noting that the vulnerability names in Table 3 are extracted from the detection results of the tools, which may be different from the names in the papers presenting the tools because the authors of the tools did not make the names to be consistent.

According to state-of-the-art books and literature, e.g., [44] and [45], the definitions and characteristics of the ten chosen vulnerability types are as follows.

4.3.1 *Unprotected Suicide (UpS)*

The *selfdestruct(address)* function can remove all bytecode from the contract address and sends all ether stored in this contract to the *address*. If a contract is vulnerable to UpS, attackers can self-destruct the contract and transfer all contract balances to an attacker-specified *address*. According to [44], a contract vulnerable to the UpS attack has the following characteristics.

1) A function containing the *selfdestruct(address)* function.
2) No access control prevents attackers from calling the *selfdestruct(address)* function to destroy the contract.

4.3.2 *TxOrigin (TO)*

In Solidity, *tx.origin* returns the address of the originating Externally Owned Account (EOA) of a transaction [45]. Using *tx.origin* for authorization could make a contract vulnerable if an authorized user calls into a malicious contract. An example attack exploiting the TO vulnerability is shown in Figure 2. The attacker first lures the owner of *VulnerableContract* to transfer ether to the *AttackContract*. After that, the *tx.origin* of this transaction is the *owner* of the *VulnerableContract*. Once the *AttackContract* receives ether, the fallback function of the *AttackContract* will be triggered. Because the *tx.origin* of this transaction is the *owner* of the *VulnerableContract*, the call from the *fallback* function to the *VulnerableContract* can pass the authorization check *if(tx.origin==owner)* and execute the *addr.transfer()* function, which will cause unexpected ether transfer. According to [45], a contract with TO vulnerability has the following characteristics.

1) *tx.origin* is used for authorization in a function.
2) There are critical operations after successful *tx.origin* authorization.

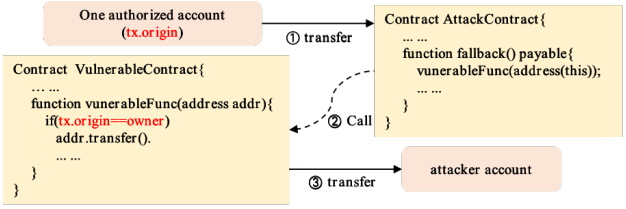

Fig. 2: An Attack Process Exploiting TO

TABLE 3: Mapping of the different vulnerabilities analyzed

	Oyente	SmartCheck	Slither	SolDetector
UpS	-	-	Suicidal	Unprotected Suicide
TO	-	Tx_Origin	Tx-Origin	TxOrigin
IOU	Integer Overflow/Underflow	-	-	Integer Overflow and Underflow
DC	-	-	Controlled-Delegatecall	DelegateCall
UcC	Callstack Depth Attack	Unchecked_Call	Unchecked-Send	Unchecked Send
RE	Re-Entrancy	-	Reentrancy	Reentrancy
FE	-	Locked_Money	Locked-Ether	Frozen Ether
NC	-	Transfer_in_Loop	Calls-Loop, Costly-Loop	Nested Call
TD	Timestamp Dependency	-	Timestamp	Dependency of timestamp
TOD	Transaction-Ordering Dependency	-	-	Transaction Order Dependency

4.3.3 Arithmetic Overflow and Underflow

An arithmetic overflow or underflow [18], [46], which is often also called Integer Overflow or Underflow (IOU), occurs when an arithmetic operation attempts to create a numeric variable value that is larger than the maximum value or smaller than the minimum value of the variable type. The popular IOU preventative technique is to use secure mathematical libraries, i.e., *SafeMath*, to replace the standard math operators, i.e., addition, subtraction, and multiplication. Thus, the arithmetic overflow or underflow may happen if a smart contract meets the following characteristic [44].

1) The arithmetic operation may pass a variable type's maximum or minimum value. However, the arithmetic operation is performed without using *SafeMath*.

```
1  contract AttackContract{
2      uint storageSlot0;//corresponds to deleLibrary in Dele
3      uint attack_storageSlot1;//corresponds to storageSlot1
4      address payable attacker_address;
5      function set(uint new_storageSlot1) public{
6          attack_storageSlot1=new_storageSlot1;
7          attacker_address.transfer(address(this).balance);
8      }
9  }
10 contract Dele{
11     address public deleLibrary;
12     uint public dele_storageSlot1=1; //modified by the
       attack_storageSlot1
13     constructor(address _deleLibrary) public payable{
14         deleLibrary=_deleLibrary;
15     }
16     function changeLibrary(address _deleLibrary) public
       payable{
17         deleLibrary=_deleLibrary;
18     }
19     function withdraw() payable public{
20         deleLibrary.delegatecall(abi.encodePacked(bytes4(
       keccak256("set(uint)"))));
21         msg.sender.transfer(dele_storageSlot1*1 ether);
22     }
23 }
```

Listing 1: An Example of DC Contract

4.3.4 DelegateCall (DC)

The function *address.delegatecall* allows a smart contract to dynamically load code from the target contract (*address*) at runtime. The code executed at the targeted address run in the context of the calling contract. Calling into untrusted contracts can be dangerous. The code at the target address can change storage values of the calling contract, e.g., to change the caller's contract balance [47], [48], because state-preserving of *delegatecall* refers to the storage slots rather than the variable name.

Listing 1 shows a smart contract *Dele* with the DC vulnerability and the *AttackContract* that can exploit it. When a user calls the *Dele* contract's function *withdraw()* to withdraw ether, it will execute the *delegatecall* in line 20 and calls the external function *set()*, which is in the *AttackContract*. The *set()* function assigns the variable *attack_storageSlot1* with the value *new_storageSlot1*, which points to the storage slot[1]. Consequently, the storage slot[1] in the calling contract *Dele* will be modified by *attack_storageSlot1*, meaning the value of *dele_storageSlot1* is overwritten to be *new_storageSlot1*. Thus, the *Dele* contract's function *withdraw*() may transfer extra ether to the attacker who controls the input parameter *new_storageSlot1*. According to [47], [48], [49], a contract vulnerable to the DC attack usually has the following characteristics.

1) A function containing the *delegatecall* function.
2) No access control on the function prevents the attacker from specifying the calldata or changing the target contract address.

4.3.5 Unchecked Call (UcC)

If a smart contract does not check the return value of a message call and assumes that the call is always successful, the failing of the call may lead to inconsistency between the logic of the program and the system state [22], [44], [50]. The functions *address.call()* and *address.send()* are often used to transfer ether, and they return a Boolean value indicating whether the call succeeds. The transaction that executes these functions may return a false value but will not revert if the external call fails. So, a smart contract with the UcC vulnerability has the following characteristic [22], [44], [50].

1) The functions *address.call()* or *address.send()* is used without result checking.

4.3.6 Reentrancy (RE)

In Ethereum, insecure use of *call()* function can lead to reentrancy attacks. In the reentrancy attack, a malicious contract calls back into the vulnerable contract before the first invocation of the vulnerable function is finished. If the state variable change is after the *call()* function, the unexpected reentrancy into the vulnerable contract will result in program execution and state variable change inconsistency. Figure 3 shows an attack process exploiting the RE vulnerability. An attacker creates the *AttackContract* to call the

VulnerableContract to transfer ether the attacker. In the *AttackContract*, there is a *fallback* function. Once *AttackContract* receives the ether, the *fallback* function will be triggered to call back into the *VulnerableContract* to perform the attacks, e.g., transfer more ether to the attacker's account before changing the account's balance.

1) A function transfers ether to another contract using the *call()* function.
2) The state variable change is after the *call()* function.

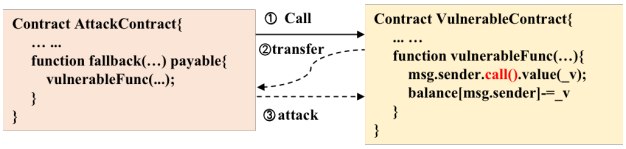

Fig. 3: An Attack Exploiting RE

4.3.7 Frozen Ether (FE)

A contract vulnerable to FE can receive ether but does not contain any functionalities to transfer ether. It relies on other contracts to transfer ether. However, if the contracts to be called to transfer ether are accidentally or intentionally terminated, the ether cannot be transferred from the contract and will be frozen. A contact with FE vulnerability has the following characteristic.

1) The contract can receive ether but cannot transfer ether by itself.

4.3.8 Nested Call (NC)

If a loop contains the gas-costly instruction but does not limit the loop iterations, the function containing the loop has a high risk of exceeding its gas limitation and causing an out-of-gas error [7]. An example of the gas-costly instruction is a non-zero value transfer as part of the CALL operation, which costs 900 gas [7]. According to [7], a contract vulnerable to NC attack has the following characteristics.

1) In the contract, dynamic data structures (e.g., array or mapping) or variables in the loop condition control the number of loop iterations.
2) The loop body contains gas-costly instructions, e.g., CALL operation.
3) No access control prevents an attacker from controlling the dynamic data structures or variables in the loop condition.

4.3.9 Timestamp Dependency (TD)

When mining a block, a miner has to set the timestamp for the block with the miner's local system time. The miner can vary this timestamp value by roughly 900 seconds while still having other miners accept the block [2]. Suppose the timestamp is used as a triggering condition to execute some critical operations [2], e.g., sending ether. In that case, miners can be incentivized to choose a timestamp that favors themselves. Thus, a contract vulnerable to TD has the following characteristic [2].

1) The contract uses the *timestamp* as the deciding factor for some critical operations, e.g., sending ether.

4.3.10 Transaction Order Dependency (TOD)

Miners decide the transaction order because transactions in the blockchain need to be packaged by miners before they are finally recorded on the chain. Malicious contract owners or attackers can exploit such order dependency. For example, if the contract is a game [19], which gives participants who submit a correct solution to a puzzle reward, a malicious contract owner could reduce the reward amount after the solution transaction is submitted. An attacker can watch the transaction pool and steal the correct answer. Then, he creates a transaction with the correct answer and gives a higher gas to get his answer packed in a block before the transaction of the answer provider is packed [45]. As there are many variations of TOD attacks, finding a precise characteristic of smart contracts vulnerable to TOD is challenging. According to [45], a high-level characteristic is as follows.

1) The contract may send out ether differently according to different values of a global state variable or different balance values of the contract.

4.4 Vulnerability Reported by Chosen Detectors

The results of analyzing the 136,969 smart contracts focusing on the ten vulnerability types are shown in Table 4. The data in the *Overlap* column of Table 4 show the contracts flagged as vulnerable by multiple tools. Results in Table 4 show that the tools reported a large number of IOU and FE-type vulnerabilities. As we plan to use Strauss' grounded theory approach to manually analyze each reported vulnerability, analyzing all the reported IOU and FE-type vulnerabilities will take an enormous time. Hence, we randomly select 100 contracts for the IOU and FE-type vulnerabilities to analyze, as shown in the *Selected Contracts* column in Table 4. In total, we analyzed 4,364 contracts reported as vulnerable.

TABLE 4: Detection Results of Each Defect Type

	Oyente	SmartCheck	Slither	SolDetector	Overlap	Selected Contracts
UpS	-	-	218	1,046	137	137
TO	-	2,292	1,807	45	45	45
IOU	65,829	-	-	80,121	28,457	**100**
DC	-	-	1,186	24,227	924	924
UcC	940	2,0361	1,683	1,316	219	219
RE	314	-	31,287	2,031	97	97
FE	-	27,882	10,240	17,901	2,934	**100**
NC	-	807	15,702	33,393	473	473
TOD	4,298	-	-	8,814	913	913
TD	4,298	-	29,941	28,365	1,356	1,356

4.5 Results of Analyzing the Reported Vulnerabilities

After analyzing the 4,364 contracts, we identify eleven reasons for the vulnerability detectors to report false positives and show them in Figure 4.

Some of the reasons can be coded to scheme one, i.e., weaknesses of the detector in adapting approaches to analyze OO-based applications, which report vulnerabilities that are not reachable and triggerable.

4.5.1 Missing path feasibility analysis

SolDetector [24], *Slither* [14], and *DefectChecker* [7] overlook the path feasibility analysis and assume all code paths are reachable. Three false positives are caused by the fact the vulnerable function *selfdestruct* locates in an infeasible path

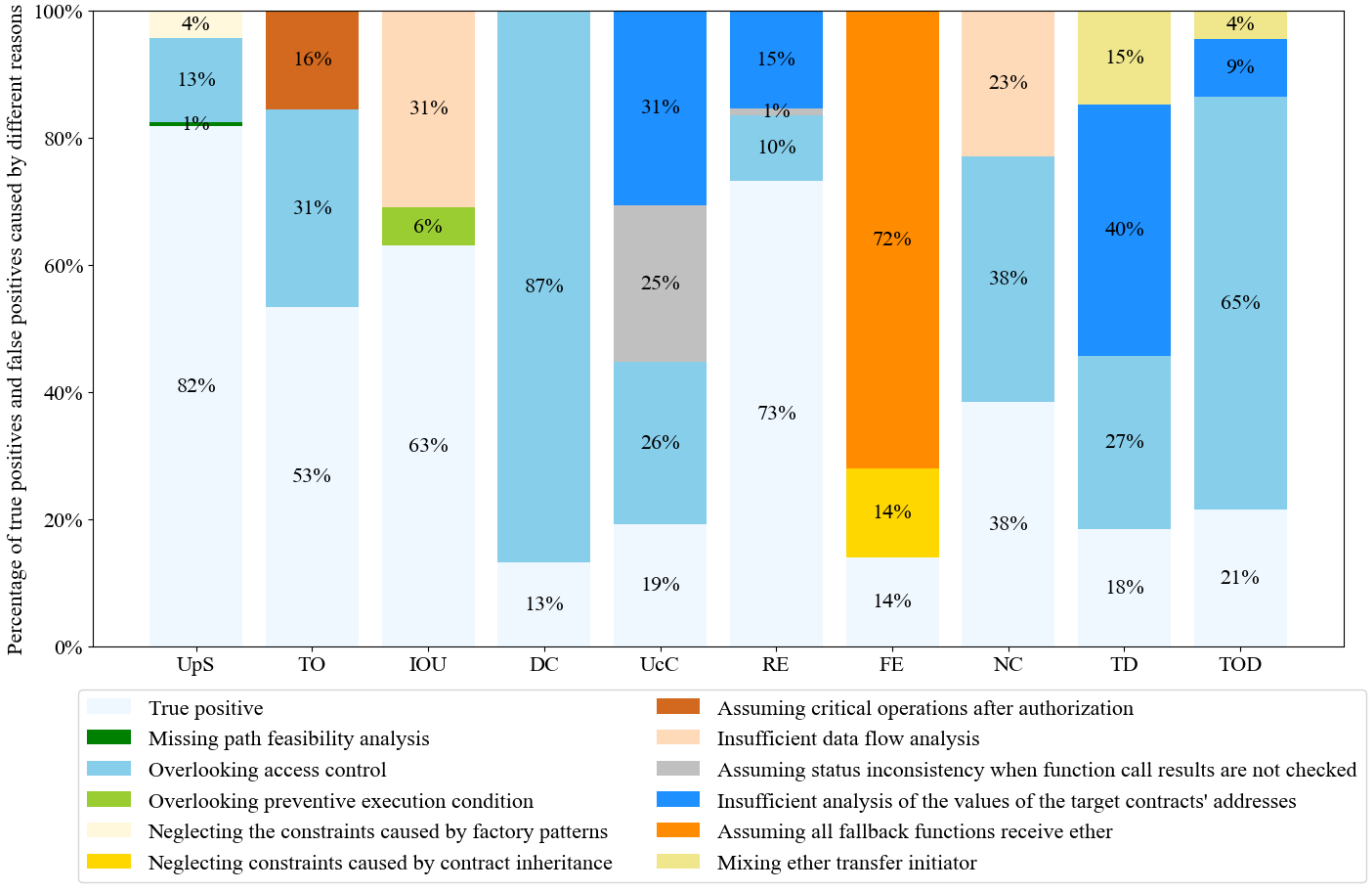

Fig. 4: The Reasons for False Positives

and will never be executed. The condition to execute the vulnerable *selfdestruct* is *require(cancel == 1)*. However, the initial value of *cancel* is 0, and no arithmetic operation changes this state variable value to be one.

4.5.2 Overlooking preventive execution condition

Vulnerability detection tools may report false positives because they do not check conditions that prevent attackers from triggering the vulnerability. For instance, in Listing 2, *balances[_from]* is greater than *_value* is checked before the arithmetic operation *balances[_from] -= _value* to avoid underflow changes of the storage value of *balances[_from]*. The sum of *balances[_to]* and *_value* is greater than *balances[_to]* is also checked before the arithmetic operation *balances[_to] += _value* to avoid the overflow changes of the storage value of *balances[_to]*.

```
function transferFrom(address _from, address _to, uint256
    _value) returns (bool success) {
    if (balances[_from] >= _value
       && allowed[_from][msg.sender] >= _value
       && balances[_to] + _value > balances[_to]) {
        balances[_to] += _value;
        balances[_from] -= _value;
        allowed[_from][msg.sender] -= _value;
        Transfer(_from, _to, _value);
        return true;
    } else { return false; }
}
```

Listing 2: A False Positive of IOU (Code from Contract: 0x98ca85c59dee34dbc26667eac04f13e39f5f765a)

4.5.3 Insufficient data flow analysis

Data flow analysis is essential to detect variable-related vulnerabilities, such as IOU and NC. By analyzing the 100 smarts reported as vulnerable to IOU, we found cases where variables involved in the arithmetic operation are fixed values or have a fixed range. The arithmetic operations on these variables will not lead to overflow or underflow.

The IOU false positives can be caused by insufficient data flow analysis within a function or across functions. Listing 3 shows an example of IOU false positives caused by insufficient data flow analysis within one function. In Listing 3, the arithmetic operation in line 2 is labeled as vulnerable to IOU. However, line 1 shows that the variable *allCards* has at least one element. Therefore, the length of *allCards* is always bigger than 1, which will not cause arithmetic underflow.

```
1 allCards.push(Card(ids[i],0,CardStatus.Tradable,upIndex));
2 idToCardIndex[ids[i]] = allCards.length - 1;
3 cardToOwer[ids[i]] = _address;
4 ownerCardCount[_address] = ownerCardCount[_address].add(1);
```

Listing 3: A False Positive of IOU (Code from Contract: 0x2919336f7a427de135dc515fc5004b083d171ba4)

In addition, specific operation relationships between different variables could also make the arithmetic operation secure. In Listing 4, *jackRewards=0.5*tournament* is always smaller than *tournament*. The arithmetic operation in line 16 is *tournament* minus *jackRewards*, which will not result in arithmetic underflow.

```
uint public winPercent = 55;
uint public losePercent = 35;
uint public jackWinPercent = 50;
function setJackWinPercent(uint _newJackWinPercent)
     onlyOwner external {
    jackWinPercent = _newJackWinPercent;
}
function startTournament() onlyOwner external {
    uint winRewards = Rewards * winPercent / 100;
    uint loserRewards = Rewards * losePercent / 100;
    uint addToJack = Rewards - winRewards - loserRewards;
    uint jackRewards;
    uint winCount = playerToWinCounts[winner];
    if (winCount == jackpotWinCount) {
      playerToWinCounts[winner] = 0;
      jackRewards = tournament*jackWinPercent/100;
      tournament -= jackRewards;
      //jackRewards=0.5*tournament<tournament
      winner.transfer(jackRewards);
    }
}
```

Listing 4: A False Positive of IOU (Code from Contract: 0x95be22039da3114d17a38b9e7cd9b3576de83924)

```
//Base contract
contract ERC20Mintable is IERC20 {
    uint8 private _decimals;
    constructor(uint8 decimals) public {
        _decimals = decimals;
    }
    function decimals() public view returns(uint8) {
        return _decimals;
    }
}
//Derived contract
contract Zion is ERC20Mintable, ERC20Detailed,  Pausable {
    uint256 public constant INITIAL_SUPPLY = 1000000000 *
     (10 ** uint256(decimals()));
}
```

Listing 5: A False Positive of IOU (Code from Contract: 0x01ad3c7da8364d3f73d8ba6deb88c2add26a7837)

```
uint public maxSACTx = 1;
function setMaxSACTx(uint256 newMaxSACTx) public
     onlyAuthorized {
    maxSACTx = newMaxSACTx;
}
function mintSAC(uint numberTokens) public payable {
    require(numberTokens > 0 && numberTokens <= maxSACTx);
    for(uint i = 0; i < numberTokens; i++) {
        ... ...
        payable(msg.sender).transfer(amount);
    }
}
```

Listing 6: A False Positive of NC (Code from Contract: 0x984f7b398d577c0adde08293a53ae9d3b6b7a5c5)

Some IOU false positives are caused by data flow analysis that overlooks smart contract inheritance. For example, in Listing 5, the arithmetic operation in line 13 is reported as vulnerable to IOU. However, the variable value comes from the function *decimals()*, which is initialized in base contract constructor (line 4) with the type of *uint8*. The maximum value of *uint8* is *255*. Therefore, the arithmetic operation in line 13 is secure because *INITIAL_SUPPLY* cannot reach the maximum of *uint256*.

An example of insufficient data flow analysis that leads to NC false positives is in Listing 6, in which the maximum number of iterations equals the value of the variable *numberTokens*. The value of the variable *numberTokens* is smaller than *maxSACTx*, according to the *require* condition in line 6. The initial value of the variable *maxSACTx* is 1, and the value can be updated in the function *setMaxSACTx*, which is modified by the modifier *onlyAuthorized*. The variables in the *setMaxSACTx* cannot be maliciously incremented to exceed the block gas limit.

Other examples of NC false positives due to data flow analysis weaknesses are in Listings 7 and 8. In Listing 7, the number of loop iterations in line 7 will not be more than 79 according to lines 3 and 4 in the *reset* function and the *resetSend* function invoked in line 5. Thus, reporting this contact as vulnerable to NC is a false positive. Listing 8 shows an example that the maximum value of a variable is pre-defined even though the attacker can update the variable value. In the contract shown in Listing 8, the *session* is a struct which has multiple properties, including *investor*, *investorCount*, and *amountInvest*. An attacker can add a new element into the *session* by calling the function *invest*. However, the value of the property *investorCount* cannot be greater than the value of the global variable *MaxInvestor*, which is pre-defined as 20.

```
uint256 private InvsetIndex = 10000;
function reset() public {
   uint256 startIndex = InvsetIndex-79;
   uint256 endIndex = InvsetIndex;
   resetSend(startIndex,endIndex);
}
function resetSend(uint startIndex,uint endIndex) private{
   for(uint256 sendUser = startIndex;sendUser<=endIndex;
     sendUse = sendUser+1){
        ... ...
        address.transfer(amount);
  }
}
```

Listing 7: A False Positive of NC (Code from Contract: 0x123ba66d42ae85f7e9c911b375ed3dba078e94b7)

```
uint public constant MaxInvestor = 20;
function closeSession (uint _priceClose) public onlyEscrow{
    for (uint i = 0; i < session.investorCount; i++) {...}
    session.investorCount = 0;
}
function invest (bool _choose) public payable{
    require(msg.value >= minimunEth && session.investOpen);
    require(session.investorCount < MaxInvestor);
    session.investor[session.investorCount]=msg.sender;
    session.amountInvest[session.investorCount]=msg.value;
    session.investorCount+= 1;
}
```

Listing 8: A False Positive of NC (Code from Contract: 0xe9a3217b3e9c7384dd62c0159ab05ea00ab4093a)

A few reasons can be coded to scheme two, i.e., overlooking the characteristics of the Solidity programming language. Gavin Wood designed Solidity to support condition-oriented programming [51], which is a subdomain of contract-orientated programming and has the principle to "Never mix transitions with conditions." When developing smart contracts, it is common to set complicated conditions to be satisfied to allow the execution of the transitions. Without a lot more comprehensive analyses of smart contract conditions, the detectors will report the transitions as vulnerable, even if the conditions can prevent executing them.

4.5.4 Overlooking access control

Vulnerability detection tools report false positives because they overlook the access control mechanisms that prevent the contract from being attacked. The false positives caused by overlooking access control happen in detecting eight vulnerability types, including UpS, TO, DC, UcC, RE, NC, TD, and TOD. Example of scenarios in that extra access control checks the identities of critical functions' caller or controls a critical variable are as follows.

① The *if/require* condition is set, such as *require(msg.sender == owner)*, before critical operations are

called. For example, the access control on the function containing the UpS vulnerability prevents the attacker from exploiting it. In Listing 9, the function *closeStableCoin* checks the caller's identity using the *require* condition in line 2 before executing *selfdestruct*.

```
1 function closeStableCoin() public {
2     require(whitelist.isSuperAdmin(msg.sender), "Only
       SuperAdmin can destroy Contract");
3     selfdestruct(msg.sender); // admin is the admin address
4 }
```

Listing 9: A False Positive of UpS (Code from Contract: 0xb8836928b5b2431c89646dce0bda19508cf9c2d5)

Another false-positive example is a smart contract labeled vulnerable to TO if *tx.origin* is used for authorization. The false alarm comes from the fact that multiple-level access control can protect the contract. The developer not only uses *tx.origin* for authorization but also checks the identity of the *msg.sender* and *recipient*, such as *require(owner == tx.origin && msg.sender == tx.origin, "Not token owner")*, which can prevent external contracts from calling the current contract and defend against the TO attack.

② In Solidity, modifiers are used to modify the behavior of a function. A modifier usually contains code, e.g., code to check the user's identity, and a special symbol "_". When executing the function claimed using the modifier, the functions' code will be inserted at the location of the symbol "_" in the modifier. If the modifier's code to check the user's identity is located before "_", the functions' code inserted will be protected by the identity checking. Otherwise, the functions' code can be called by any user.

As shown in Listing 10, even though *tx.origin* is used for authorization, the modifier *onlyMain* checks the identity of *msg.sender* before executing the function *addBrick*, which prevents intermediate contracts from being used to call the current contract [45]. In 13 false positive results of the RE type vulnerability, the code of the modifier checks whether the *msg.sender* is *tx.origin* or *owner*, which makes the recursive call from another contract impossible.

```
1 modifier onlyMain() {  require(msg.sender == main); _;}
2 function addBrick(uint _value) external onlyMain returns (
      bool success){
3     require(_value >= 10 ** 16);
4     require(owner == tx.origin);
5     return true;
6 }
```

Listing 10: A False Positive of TO (Code from Contract: 0x65e871cd0e132e14b3bd9569199dcb436c752b2f)

③ The visibility attributes, such as *private* and *internal*, of critical function restrict *external* functions from calling it. In Listing 11, the contract *BZxProxy* is a derived from the contract *Proxiable*. The function *_replaceContract* contains a DC vulnerability in line 9, of which the visibility is *internal*. In Solidity, internal functions or variables can only be used internally or by derived contracts. Therefore, it is secure for the function *replaceContract* in the derived contract *BZxProxy* to use the function *_replaceContract* with the modifier *onlyOwner*. In this study, we assume that the data from an authenticated caller are always authentic. Thus, the attacker cannot exploit the DC vulnerabilities if the external contract address and calldata used to call the external contract are controlled by the authenticated caller.

```
1  contract Ownable {
2      modifier onlyOwner() {require(msg.sender == owner); _;}
3  }
4  contract BZxStorage is Ownable {... ...}
5  contract Proxiable {
6      mapping (bytes4 => address) public targets
7      function initialize(address _target) public;
8      function _replaceContract(address _target) internal {
9          require(_target.delegatecall(0xc4d66de8, _target),
        "Proxiable::_replaceContract: failed");
10     }
11 }
12 contract BZxProxy is BZxStorage, Proxiable {
13     function replaceContract(address _target)  public
        onlyOwner{
14         _replaceContract(_target);
15     }
16 }
```

Listing 11: A False Positive of DC (Code from Contract: 0x86343be63c60ce182d8b5ac6a84f0722d8d61ae5)

Listing 12 shows a false positive case of the NC vulnerability, in which the loop iterations is determined by the input parameters *_tos* of the function *transferETH*. Here, the modifier *onlyOwner* requires the function to be called only by authenticated users. As we believe the contract owner will not input a large array to disable the function maliciously, we label this case as a false positive.

```
1 function transferETH(address[] _tos) public onlyOwner
     returns (bool) {
2     require(tos.length > 0);
3     require(address(this).balance > 0);
4     for(uint32 i=0;i<_tos.length;i++){
5        _tos[i].transfer(address(this).balance/_tos.length);
6     }
7     return true;
8 }
```

Listing 12: A False Positive of NC (Code from Contract: 0xe62e6e6c3b808faad3a54b226379466544d76ea4)

④ Across control is performed across-functions. For NC, some false positives are caused by access control across multiple functions or modifiers. In Listing 13, the loop in line 7 is labeled with the NC vulnerability, and the max number of loop iteration is equal to the length of variable *landmarks* that the function *totalSupply* can access. The variable *landmarks* is global and can be modified through the function *createLandmark*. However, the function *createLandmark* is modified by the modifier *onlyCOO*, which requires that the caller is the authenticated user *coo*. Therefore, the attacker cannot arbitrarily increase the elements in the variable *landmarks* to exploit the function *buy*.

```
1  uint256[] private landmarks;
2  function totalSupply() public view returns (uint256) {
3      return landmarks.length;
4  }
5  function buy(uint256 _tokenId) public payable {
6      require(msg.sender != address(0));
7      for (uint i = 0; i < totalSupply(); i++) {
8          uint id = landmarks[i];
9          landmarkToOwner[id].transfer(feeGroupMember);
10     }
11 }
12 modifier onlyCOO() {  require(msg.sender == coo); _; }
13 function createLandmark(uint256 _tokenId) public onlyCOO {
14     ... ...
15     landmarks.push(_tokenId);
16 }
```

Listing 13: A False Positive of NC (Code from Contract: 0xeb35a696af4cf2e18203781db1c7607adbabc251)

4.5.5 Assuming critical operations after authorization

One main characteristic of TO is having critical operations, e.g., sending ether, after successful authorization. If there is no critical operation following successful authentication, we label the contracts reported vulnerable to TO as false positives. We labeled seven TO false positives because successfully bypassing the authentication will not bring critical risks. The example codes are : *if(tx.origin != owner()) require(block.timestamp <mintTimestamp, "You're early")*, and *if (tx.origin == owner()) return false;* .

Although Solidity supports a few OO programming language principles, such as inheritance, its implementation of the OO features can be different. Two of the reasons explained in Sections 4.5.6 and 4.5.7 can also be coded to scheme two, i.e., overlooking the characteristics of the Solidity programming language.

4.5.6 Neglecting the constraints caused by factory patterns

Factory pattern is one of the most used design patterns in Java. "In the factory pattern, instead of directly creating instances of objects, a single object (the factory) does it for you" [52]. Solidity supports the factory pattern, and smart contracts are the objects. A factory in Solidity is a contract (called **main contract**) that can deploy multiple instances of other contracts (called **template contracts** in this paper) at runtime.

When using the factor pattern, the *selfdestruct* in the instances created from the template contract cannot destruct the main contract. The vulnerability detection tools report six UpS false positives because they do not consider such a constraint. For example, in Listing 14, *SwapperFactory* is a main contract that creates the template contract objects multiple times and destruct the objects by calling the function *destroy*. Therefore, the *selfdestruct* in the template contract cannot be exploited by attackers.

```
contract SwapperFactory {//Main contract
  function performSwap(address payable user){
     Swapper swapper = createClone(user, srcToken, dstToken
     , uniqueId);
     swapper.destroy(user);
  }
  function createClone( address user, address srcToken,
     address dstToken, string memory uniqueId) private
     onlyAdmin() returns (Swapper) {
    bytes32 salt = computeCloneSalt(user, srcToken,
     dstToken, uniqueId);
    bytes memory bytecode = getCloneBytecode();
    address payable cloneAddress = computeAddress(salt);
    if (!isContract(cloneAddress)) {
      assembly {
        cloneAddress := create2(0, add(bytecode, 0x20),
     mload(bytecode), salt)
      }
    }
    return Swapper(cloneAddress);
  }
}
contract Swapper {  //Template contract
  function destroy(address payable user) external {
    selfdestruct(user);
  }
}
```

Listing 14: A False Positive of UpS (Code from Contract: 0xaa548618371d95cb0cb211bbbf37b26be3c744cc)

4.5.7 Neglecting constraints caused by contract inheritance

Solidity supports inheritance between smart contracts. A contract can inherit multiple contracts. The contract from which other contracts inherit is known as a base contract, while the contract which inherits the features of the base contracts is called a derived contract. "With the inheritance construct, the derived contract inherits all the methods, functionality, and variables of the base contract and can extend a base contract with additional functionality" [45].

In 14 false positive FE cases, the contracts tagged as vulnerable to FE are base contracts containing no transfer operation. However, the contract inherits these base contracts implemented ether transferring. Thus, these base contracts are not vulnerable to FE attacks. In addition, these base contracts do not have an account on Ethereum's main network and own no ether. Therefore, these base contracts will not lock the ether, meaning the FE attack will not negatively impact these base contracts.

The application scenarios of smart contract applications are to transfer assets or ether between suppliers and clients. Without being able to manipulate or sabotage the asset transfer maliciously, the likelihood of security compromise or giving benefits to attackers by executing the code, which seems to be vulnerable from the OO programming analysis perspective, is low. The reasons for false positives presented from Sections 4.5.8 to 4.5.11 can also be coded to the second theme, i.e., overlooking smart contract application scenarios.

4.5.8 Insufficient analysis of the values of the target contracts' addresses

Calling an external contract or transferring ether to an address can be dangerous if the attacker controls the contract or the target address. However, in some cases, the target contracts' addresses are hard-coded, fixed, or under the complete control of the contract owner. Weak analysis of the target contracts' values resulted in RE, TD, and TOD false positives.

In 12 contracts that are reported as vulnerable to RE, the target contracts' addresses are hard-coded. The hard-coded address may be a global variable used in multiple functions. Defining the target address as *immutable* can also freeze the value of the addresses. Therefore, an unexpected call from the *fallback* function in the target contract will not happen. For TD and TOD, if the recipient address is fixed or fully controlled by the contract owner, the attack will not get profit by attacking the vulnerability even if an attacker can manipulate the timestamp or determine the order of function calls and transactions. As shown in Listing 15, the contract creator initialized the *fundRaiser* in the constructor as the receiver of the ether, and no function can modify this variable. The attackers cannot get benefits even if they can manipulate the timestamp. Thus, we label the code in Listing 15 as a TD false positive case.

```
function Crowdsale(address fundRaiserAccount){//Constructor
    fundRaiser = fundRaiserAccount;
}
modifier afterIcoDeadline() { if (now >= icoDeadline) _; }
function withdrawFunds() afterIcoDeadline public{
    require(fundRaiser == msg.sender);
    fundRaiser.transfer(address(this).balance);
    emit FundTransfer(fundRaiser, address(this).balance);
}
```

Listing 15: A False Positive of TD (Code from Contract: 0xd75ee6c853ce690668c923e7ba7b8411ca81db46)

4.5.9 Assuming status inconsistency when function call results are not checked

If there is no status change after calling the functions *send()* and *call()*, it is not risky even though the result of the message call is not checked. There is no status change following the message call in 14 UcC false positives. The detection tools tag them as vulnerable to UcC because they assume that function calls without result checking will always lead to status inconsistency. Listing 16 shows an example of such a false positive. The function *executeCall()* transfers ether by the *call()* function in line 4. The variable *underExecution* is set to avoid the recursive calling from the *_target*. The initial value of *underExecution* is *false* (line 2). It will change to be *true* (line 3) before executing transferring by *call()* and turn back to *false* after transferring (line 5). Therefore, no status change follows the execution of the function *call()* because the value of *underExecution* is always false regardless the function call *call()* fails or not.

```
1 function executeCall()external onlyAllowedManager(){
2     require(underExecution == false);
3     underExecution = true; // Avoid recursive calling
4     _target.call.gas(_suppliedGas).value(_ethValue)(
       _transactionBytecode);
5     underExecution = false;
6   }
```

Listing 16: A False Positive of UcC (Code from Contract: 0xa5fd1a791c4dfcaacc963d4f73c6ae5824149ea7)

4.5.10 Assuming all fallback functions receive ether

If the contract can receive ether but cannot transfer it by itself, the vulnerability detectors tag it vulnerable to FE. The detectors assume that all contracts shall host ether and, therefore, shall be able to transfer ether. However, 66 of the 100 contracts tagged as vulnerable to FE will never lock ether because their fallback functions cannot or refuse to accept ether. A contract usually has one fallback function. The fallback function is executed on a call to the contract if none of the other functions match the given function signature or if no data was supplied at all and there is no receive ether function [53]. The fallback function can be declared using *function()*, *fallback()*, or *receive()* in different Solidity versions. In order to receive ether, the fallback function must be marked *payable*, as shown in line 1 in Listing 17. The contract cannot receive ether if it does not contain the fallback function or if the fallback function is not marked as *payable*, as shown in line 2 in Listing 17. The approach to refuse ether is to insert the *revert()* into the fallback function as shown in lines 3 and 4 in Listing 17. Once the contract receives the ether, this transaction will revert. Therefore, any transaction transferring ether to these contracts will fail, and these contracts will not receive any ether.

```
1 function() payable public{ }//Accept ETH
2 function() public{ }//Don't accept ETH
3 function () payable public{ revert(); }//Don't accept ETH
4 receive() external payable { revert(); }//Don't accept ETH
```

Listing 17: Fallback Functions

4.5.11 Mixing ether transfer initiator

To detect TOD vulnerability, *Oyente* and *SolDetector* focus on the ether flow because TOD may lead to undesirable outcomes when dealing with ether [2]. *Oyente* labels a contract as vulnerable to TOD if it sends out ether differently when the order of transactions changes. *Oyente*, *Slither*, and *SolDetector* label the contract as vulnerable to TD if the block *timestamp* is used as the condition to send ether. However, *Oyente*, *Slither*, and *SolDetector* all ignore the scenario that the ethers transferred to a user after the timestamp checking may come from the user himself. For example, many contracts vulnerable to TD or TOD are wallet contracts that support users to purchase or withdraw tokens within the specified time range. If the time range passes, the ether to purchase the token will be returned to the user. Such ether transfer after the timestamp checking is not harmful because the ether is returned to its initial owner. Listing 18 shows an example, which is contract *OpportyPresale* for token transaction. Users can send a message with *msg.value* to the contract *OpportyPresale* to purchase token. *OpportyPresale* contains a fallback function labeled as vulnerable to TD, and the vulnerability is in line 6. Once this contract receives ether, the fallback function will be triggered to verify whether the transaction meets the conditions regarding timestamp (*now >endDate)* and amount of ether (*msg.value >= 0.3 ether*). The contract will return the ether to the token purchaser if the transaction is not within the valid time.

```
1  contract OpportyPresale is Pausable {
2      function() whenNotPaused public payable {
3          require(state == SaleState.SALE);
4          require(msg.value >= 0.3 ether);
5          require(whiteList[msg.sender].isActive);
6          if (now > endDate) {
7              state = SaleState.ENDED;
8              msg.sender.transfer(msg.value);
9              return ;
10         }
11     }
12 }
```

Listing 18: A False Positive of TD (Code from Contract: 0xca67e92833c2de6bf3a444127fa0c60092255bf4)

Our identified reasons for false positives illustrate that the existing characteristics of smart contract vulnerabilities in state-of-the-art literature are insufficient. Based on the results of RQ1, we extend the characteristics of the ten types of vulnerabilities and show the extensions in Figure 5. We hope the extended characteristics can help vulnerability detector developers design and implement better tools.

5 Results of RQ2

As mentioned in Section 3.2, we collect the transaction logs of the contracts that are reported as vulnerable and analyze them. Figure 6 shows a fragment of an example transaction log containing two log blocks. Each block reflects the running state of EVM, in which:

- *pc* is the program counter.
- *op* represents a low-level machine language consisting of a series of instructions, each of them representing an operation.
- *gas* represents the remaining gas.
- *gasCost* refers to the gas consumption of the current opcode.
- *depth* of call stack indicates the depth of nested calls, which has a maximum value 1024.
- *stack* is an internal place where temporary variables, such as local variables, intermediate calculation results, and return addresses, are stored.

	Known Characteristics of the Vulnerabilities	Extended characteristics to reduce false positives
UpS	1) A function containing the *selfdestruct* function. 2) No access control prevents attackers from calling the *selfdestruct(address)* function to destroy the contract.	3) The *selfdestruct* function is not invoked through the factor pattern. 4) The *selfdestruct* function is not in an infeasible path.
TO	1) *tx.origin* is used for authorization in a function. 2) There are critical operations after successful *tx.origin* authorization.	3) *tx.origin* is the only authorization of the critical operation.
IOU	1) The arithmetic operation may reach the maximum or minimum size of a type. However, the arithmetic operation is performed without using SafeMath.	2) No arithmetic operation prevents variables from reaching the boundary of the integer type. 3) No execution condition prevents the overflow/underflow results from being written into the storage.
DC	1) A function containing the *delegatecall* function. 2) No access control on the function prevents the attacker from specifying the calldata or changing the target contract address.	
UcC	1) The functions *call()* or *send()* is used without result checking.	2) The failed call can be triggered. 3) The failed call will lead to status inconsistencies between state variables in the contract and actual transfer operations.
RE	1) function transfers ether to another contract with the *call()* function. 2) The state variable change is after the *call()* function.	3) No access control prevents attackers from specifying the target address to receive money. 4) The target contract's address is not a hard-code address nor is only initialized in the constructor.
FE	1) The contract can receive ether but cannot transfer ether by itself.	2) The vulnerable contract is the main contract rather than the base contract. 3) The *fallback* function will not reject to receive ether.
NC	1) In the contract, dynamic data structures (e.g., array or mapping) or variables in the loop condition control the number of loop iterations. 2) The loop body contains gas-costly instructions, e.g., CALL operation. 3) No access control prevents an attacker from controlling the dynamic data structures or variables in the loop condition.	4) No data flow controls on the dynamic data structures or variables in the loop condition.
TD	1) The contract uses the timestamp as the deciding factor for some critical operations, e.g., sending ether.	2) The ether transferred to callers is from the contract rather than the callers.
TOD	1) The contract may send out ether differently according to different values of a global state variable or different balance values of the contract.	2) The ether transferred to callers is from the contract rather than the callers.

Fig. 5: Extended Characteristics of Smart Contract Vulnerabilities

- *memory* is a temporary place to store data, of which a contract obtains a freshly cleared instance for each message call [53].
- *storage* is a key-value store. Data in storage are stored permanently between function calls and transactions. For instance, the global variables declared in the smart contract are stored in the *storage*.

Figure 6 contains two log blocks. Code in the first block compares whether *A* (line 2047) and *B* ((line 2048) are equal, and the comparison result is pushed onto the stack of the second log block (line 2065).

Fig. 6: A Fragment of of Transaction Log

5.1 Analyzed Transaction Logs

In the analysis, we excluded transaction logs of TD and TOD vulnerabilities because these two types of vulnerabilities exploit the mining process. Therefore, information in the transaction log cannot reflect the exploitation. The 219 contracts vulnerable to UcC has 5,450,975 transactions, which is too many for us to replay all of them and perform the ground theory analysis. We randomly selected 100 UcC contracts reported as vulnerable, which have 145,469 transactions. The numbers of transactions analyzed for the eight vulnerability types are shown in Table 5.

We designed analysis rules as shown in Figure 7 to analyze vulnerability exploitation. The analysis rules contain *opcode*, information to search in *stack* or *storage*, and additional constraints. Each of the rules is explained as follows.

5.1.1 Unprotected Suicide (UpS)

The EVM opcode SELFDESTRUCT destroys contracts. The SLEFDESTRUCT opcode used to be called SUICIDE, but SUICIDE was deprecated due to the negative associations of the word [45]. It is insecure if the attacker exploits SELFDESTRUCT. However, it is challenging to identify whether a user is malicious. In this study, we define the contract's creator as benign and assume any other users who destroy the contract they do not own are malicious.

5.1.2 TxOrigin (TO)

Attacks exploiting the TO vulnerability usually follow the attack process shown in Figure 2. We designed the TO

TABLE 5: Number of Transactions analyzed

Vulnerabilities	UpS	TO	IOU	DC	UcC	RE	FE	NC	Total
Number of Analyzed Transactions	33,920	77,934	131,643	1,683,694	145,469	5,362	396,127	1,631,985	4,106,134

	Opcode	Stack/Storage	Additional Constraints
UpS	① SELFDESTRUCT		caller is not the contract creator
TO	① CALL② ORIGIN ③ EQ	(1) target_address=stack[-2]=contract_address (2) origin_address=stack[-1] (3) eq_result=stack[-1]=1	
IOU	① ADD/SUB/MUL ②SSTORE	(1) expected_value= ADD/SUB/MUL(stack[-1], stack[-2]) (2) storage_value=actual_value	actual_value=stack[1]!=expected_value
DC	① DELEGATECALL②SSTORE③SLOAD	(2) stack[-1]=key1, depth1, value1 (3) stack[-1]=key2, depth2, value2	depth1=depth2+1 *and* key2=key1 *and* value1=value2
UcC	① CALL ②¬ (ISZERO and JUMPI) ③SSTORE	(1) call_result=stack[-1]=0	
RE	① CALL ② CALL	(1) callee_address=stack[-2]=attacker_address (2) callee_address=stack[-2]=contract_address	
FE	①No CALL/ DELEGATECALL/ CREATER/SELFDESTRCT/SUICIDE		balance of the contract > 0
NC			error=out of gas

Fig. 7: Analysis Rules of Transaction Logs

analysis rule according to that attack process. To find a call from the *fallback* function in the attacker's contracts, we search the CALL instruction ①, in which the target address is the vulnerable contract address (1). Then, we look for the ORIGIN instruction ②, in which the origin address (2) is usually used for authorization. Finally, we identify the EQ instruction ③ for the authorization and expect the authorization result to succeed (3).

5.1.3 Arithmetic Overflow and Underflow (IOU)

The arithmetic overflow is usually caused by arithmetic instructions ADD or MUL, and the arithmetic underflow may happen when executing the SUB instruction. We first find the log block containing the arithmetic instruction ADD, MUL, or SUB ① and get two operands from the stack to calculate the expected result (1) of the arithmetic operation. Then we compare the actual value (2) with the expected value. If the actual value that has been pushed onto the stack is not equivalent to the expected value, it means that the arithmetic overflow or underflow has occurred. Torres et al. point out that not every overflow is considered harmful because the compiler may also introduce it for optimization purposes [54]. Thus, we only trace the overflow or underflow that has updated the blockchain state. If the error result flows into a SSTORE instruction ②, we will label the transaction as IOU exploitation.

5.1.4 DelegateCall (DC)

The *delegetacall* is insecure if the state of the called contract affects the calling contract. At the transaction log level, we recognize DC exploitation according to the storage state caused by the *delegatecall*. If a *DELEGATECALL* ① causes a storage variable modification by SSTORE ② when executing an external contract, and this storage variable is used in the calling contract by SLOAD ③, we label the transaction as DC exploitation. When conducting an external call, e.g., CALL or DELEGATECALL, *depth* increases by one. To distinguish the external and the current contract call, we check if the depth value (*depth1*) of SSTORE (2) is exact one bigger than *depth* value (*depth2*) of SLOAD (3). When the delegatecall call ends, the *depth* turns back to the value it has had when DELEGATECALL is executed [19]. To identify the storage variable used in the calling contract but modified in the external contract, we check if the two storage variables share the same key and value at different depths. It is worth noting that the external call should be secure if the external contract address and the calldata are specified by the contracts' owners. Therefore, we must manually check if the initiator of the identified DC exploitation is the contract's owner to exclude false positive exploitation.

5.1.5 Unchecked Call (UcC)

The *send()* and *call()* functions are used to send ether and are compiled into the EVM CALL instructions. The CALL instruction results are pushed onto the stack, where 0 means failure and 1 means success. The CALL result is stored in the log block where the depth of the trace turns back to the value it has had when the CALL instruction is executed [19]. If CALL ① results in value 0 (1) and the SSTORE changes the storage status ③ without executing an opcode to check the CALL result ②, we will flag the transaction as an UcC exploitation.

5.1.6 Reentrancy (RE)

As shown in Figure 3, a reentrancy attack usually calls ether transferring functions in the vulnerable contract to trigger the malicious fallback function in the malicious contract. Therefore, an RE exploitation has at least two CALL instructions ①② in one transaction. The target address of the first CALL is the attack's contract address (1), and the target address of the second CALL is the vulnerable contract's address (2). The state in the contract is updated by executing the SSTORE ③ instruction.

5.1.7 Frozen Ether (FE)

There are several reasons for funds being locked in a contract. Perez et al. [19] focus on the case that the contract relies on an external contract to transfer ether, but the external contract does not exist any longer. In this study, the vulnerable contracts are labeled by static analysis tools, which cannot know whether the contract being relied on to transfer ether is destroyed. Therefore, if the contract balance is not 0 and the contract's transaction logs do not contain any instruction supporting transferring ①, we flag the contract as a FE exploitation. A potential issue of our analysis rule is that no ether transfer in the contract's transaction does not mean the contract cannot transfer ether. Thus, the analysis rule may report false positives of FE exploitation.

5.1.8 Nested Call (NC)

In Ethereum, when transactions fail due to gas shortage, the transaction log will contain error messages, such as *"error":"out of gas"*. To search NC exploitation, we first look for the transaction log containing an error tag, and the reason is "out of gas." Gas shortage can also be caused by other failed transfer transactions irrelevant to NC. These unrelated transactions usually do not call any function, and their transaction logs contain mostly only one PUSH1 instruction. To exclude unrelated transactions and reduce false positives, we manually check if the nested call is the reason for the transaction failures.

5.2 Identified Vulnerability Exploitation

If at least one contract's transaction on Ethereum's main network is identified as executed by our analysis rules, we count the contract as exploited. The numbers of identified exploitations are shown in Table 6. We do not find any exploitation of contracts we labeled as false positives, which confirms our conclusions on the false positives in answering RQ1. For the identified true positive vulnerable contracts, their exploitations are explained below.

Unprotected Suicide (UpS). We found 19 contracts vulnerable to UpS that have been exploited, meaning 19 contracts were destructed but not by their owner. Listing 19 shows an example of the exploited contracts. The contract is related to a game, and the function *takeAGuess()* of the contract is to let users take a guess after transferring 0.0001 ether. If the number a user inputs equals the *winningNumber*, the function will transfer 90% of the contract's balance to the user, then kill the contract and return the remaining balance to the contract owner. The transaction details of the exploited UpS are shown in Figure 8 in Appendix A, which shows that an attacker input the number nine that satisfied the if condition in line 3 and then executed the *selfdestruct()* function successfully and got 0.0468 ether from the contract.

```
1 function takeAGuess(uint _myGuess) public payable {
2     require(msg.value == 0.0001 ether);
3     if (_myGuess == winningNumber) {
4         msg.sender.transfer((this.balance*9)/10);
5         selfdestruct(owner);
6     }
7 }
```

Listing 19: contract Vulnerable to UpS that was exploited (Code from Contract: 0x3ac0d29eaf16eb423e07387274a05a1e16a8472b

An interesting finding is that some contracts still have balances even though the contracts have been self-destructed. The reason is that the contract account will not disappear even if the contract is destroyed. The contract account can receive ether but does not support transferring ether, leading to the locked ether. An example in Figure 9 in Appendix A shows that the transactions of a self-destructed contract, in which 0.033 ether are locked.

TxOrigin (TO). The transaction log analysis does not reveal exploitation of the TO vulnerabilities.

Arithmetic Overflow and Underflow (IOU). We found 15 occurrences of arithmetic overflows. Listing 20 shows the source code of an example contract that is exploited. The transaction has an invocation of the function *transport()*, in which an arithmetic operation was conducted based on the function *addDungeonRewards()*, which calculated the reward for different *originDungeonId* in the *dungeons* without using SafeMath.

```
1  DungeonToken public dungeonTokenContract;
2  function transport() external payable {
3      // ** STORAGE UPDATE **
4      // Increment the accumulated rewards for the dungeon.
5      dungeonTokenContract.addDungeonRewards(originDungeonId,
         requiredFee);
6      ......
7  }
8  contract DungeonToken {
9      function addDungeonRewards(uint _id, uint _Rewards) {
10         dungeons[_id].rewards += uint128(_Rewards);
11     }
12 }
```

Listing 20: A Contract Containing Arithmetic Overflow (Code from Contract: 0x141766882733cafa9033e8707548fdcac908db22). The contract is destroyed in Ethereum.

DelegateCall (DC). We did not find any exploitation of the 924 contracts with DC vulnerabilities.

Unchecked Call (UcC). We found three UcC exploitations, which contained failed calls and storage status changes after the call failures. An example of the exploited contracts is shown in Listing 21. In the contract's transaction, at a certain point in time, there was insufficient ether in the contract supporting the transfer in line 7. Therefore, the log shows that a transaction calling the function *sendTokensManager* did not call the *send()* function in line 7 successfully. However, the contract, e.g., *Exxcoin*, calling the function *sendTokensManager* still changed the storage variable *balances* in line 8 after the invocation of the *send()* function failed.

```
1  contract ExxStandart is ERC20 {
2    mapping (address => uint) balances;
3  }
4  contract Exxcoin is owned, ExxStandart {
5      function sendTokensManager(address _to, uint _tokens)
        onlyManager public{
6          require(manager != 0x0);
7          _to.send(_tokens);
8          balances[_to] = _tokens;
9          Transfer(msg.sender, _to, _tokens);
10     }
11 }
```

Listing 21: An example contract contains failed functions (Code from Contract: 0x11e44037a60da9b76bf928b33d3d6ded0a6730ab)

Reentrancy (RE). We found no exploitations of the 72 contracts with RE vulnerabilities.

Frozen Ether (FE) Among the 100 FE vulnerable contracts we choose to analyze, four of them have ether. We analyzed the transaction logs of these four contracts and found two contracts had never transferred ether to other accounts. Thus, we label these two contracts as exploited.

Nested Call (NC). 28 out of the 184 contracts with NC vulnerabilities were exploited when executing the functions containing a *for* loop. Listing 22 shows an example of the exploited contract, in which the *for* loop in the function *distribute* iterates over the input parameter *addresses* of the function. The total size of *addresses* in this transaction is 202 and the gas limit of this transaction is 2,417,107 as shown in Figure 10 in Appendix A. This transaction used up all the given gas and failed due to the gas shortage.

Another possible exploitation of NC is to set the transaction gas to be bigger than the block gas limit. The prerequisite of transaction execution is that the maximum gas

TABLE 6: Information about Vulnerability Exploitation

	UpS	TO	IOU	DC	UcC	RE	FE	NC	TD	TOD	Total
Number of Contracts Reported as Vulnerable	137	45	100	924	219	97	100	473	1,356	913	4,364
False Positives	25	21	37	802	178	26	86	291	1,106	717	3,289
Number of True Positives	112	24	63	122	41	71	14	182	250	196	1,075
Number of True Positives that are Exploited	19	0	15	0	2	0	2	28	-	-	67

limit for a transaction cannot exceed the gas limit for the block that packages the transaction [45]. The transaction will never succeed if the estimated gas consumption exceeds the block's gas limit. However, we did not find such an exploitation in our analyzed transactions.

```
function distribute(address[] calldata addresses, uint256[]
      calldata amounts) payable external {
    require(addresses.length > 0);
    require(amounts.length == addresses.length);
    for (uint256 i; i < addresses.length; i++) {
        uint256 value = amounts[i];
        address _to = addresses[i];
        address(uint160(_to)).transfer(value);
    }
}
```

Listing 22: An Example Contract Related to Out-of-Gas Transactions (Code from Contract: 0x469503159ddf6bfd0a9ec8eba8e97a84fd3eae5b)

5.3 Results of Analyzing Vulnerability Exploitation

As shown in Table 6, only 6% (67 out of 1,092) contracts truly vulnerable to various attacks were exploited. We have the following findings using Strauss' grounded theory approach to analyze the vulnerable contracts and their exploitation and non-exploitation.

By performing the selective coding of the grounded theory analysis, we coded the reasons presented in the following Sections 5.3.1 and 5.3.2 for non-exploitation to the second theme, i.e., the smart contracts' application scenario and the execution environment and cost on blockchain demotivate or prevent the vulnerable contracts from being exploited.

5.3.1 Low motivation to exploit the vulnerabilities

We found that attackers may not be motivated to exploit the vulnerability because their gains or the attacks' impact are trivial.

Very little or no financial benefits. By exploiting the UpS, TO, RE, TD, and TOD vulnerabilities, the attacker may get profit. However, many vulnerable contracts contain no or very little ether. For example, 52 out of 57 contracts vulnerable to UpS and 71 out of 72 contracts vulnerable to RE have no ether, and only one contract vulnerable to RE holds 0.001307 ether. Many smart contracts vulnerable to TD are wallet contracts supporting users to purchase or withdraw tokens within a limited time. Users must deposit ether into the contract and exchange it for other digital assets. There are four game contracts from which users may win some rewards. However, three of them contain no balance, and the fourth one contains only 0.03 ether.

Insignificant impacts. Attackers may cause contract function failure or user losses by exploiting FE vulnerabilities to freeze ether or exploiting NC vulnerabilities to use up gas. Very few of the contracts vulnerable to FE have ether. Thus, locking a limited amount of ether will not bring significant impacts. Although the 28 NC vulnerability exploitations, few of them bring severe impacts. Similar to the example contact shown in Listing 22, the for loops in 27 contracts iterate over function input parameters, meaning the size of for iteration is controlled by the function caller. If the caller sets a small number for iterations or gives enough gas, the function will execute successfully. Therefore, most reported NC vulnerabilities will not lead to the permanent failure of the functions, and the failure of transfer functions caused by NC will lock ether in contracts.

5.3.2 Blockchain mechanisms provide extra defense

Some characteristics of Solidity language and Ethereum mechanisms demand attackers to putting in extra effort and investment and be lucky to exploit the vulnerabilities, which may demotivate their exploitation.

Needing to develop attack contracts. To exploit some vulnerabilities, e.g., RE, DC, UcC and, TO, attackers must develop attack contracts, which should be customized according to different vulnerabilities. For instance, designing an attacking contract containing a specified fallback function is vital to trigger the contracts vulnerable to RE. To exploit the contracts vulnerable to DC, the attacker needs an attacking contract that controls different global variables and modifies the storage values in vulnerable contracts.

Depositing ether is a prerequisite. Some vulnerable contracts are used for bidding, games, or wallets. These contracts usually require users to send ether to contracts first to get an authenticated identity. Therefore, sending ether is a prerequisite for an attacker to call the contract. In the example contract in Listing 23, line 11 has a TD vulnerability. Suppose an attacker wants to exploit the vulnerability and get all ether of this contract. In that case, the attacker must meet the condition in line 12, i.e., sending more than 0.001 ether (line 3) and getting a correct *randomNumber* (line 4) to start the exploitation.

Being lucky in random number competition. We found that setting a puzzle as a deciding condition for some critical operations is a popular defense method in our studied contracts. An attacker cannot get permission to run critical operations or get benefits unless the attacker is lucky enough to solve the puzzle successfully. As shown in Listing 23, the *randomNumber* is calculated by the *keccak256* function in line 4 based on the *block.number*, which cannot be controlled by the attacker.

Being mining winner. If attackers want to exploit TD and TOD vulnerabilities, they must monitor the transaction pool to capture critical transaction information, such as a puzzle answer. After that, the attacker can initiate a new transaction to compete with the old transaction. The attack will not succeed unless the attacker is a mining winner and the attacker's transaction is successfully packaged.

```
function () public payable {
    require(msg.sender == tx.origin);
    require(msg.value >= 0.001 ether);
    uint256 randomNumber = uint256(keccak256(blockhash(
     block.number - 1)));
    if (randomNumber > highScore) {
        currentWinner = msg.sender;
        lastTimestamp = now;
    }
}
function claimWinnings() public {
    require(now > lastTimestamp + 1 days);
    require(msg.sender == currentWinner)
    msg.sender.transfer(address(this).balance);
}
```

Listing 23: A Contract Containing a Puzzle (Code from Contract: 0x954791f9a0f0ff7841cffea32c556ac71168eff8)

6 Discussion

Results of RQ1 and RQ2 bring novel insights to vulnerability detector development and evaluation.

6.1 Implication to Vulnerability Detector Development

Although we did not investigate all existing smart contract vulnerability detection approaches due to the selection criteria explained in Section 4.2, our study covered several of them, including pattern matching, symbolic execution, and data flow analysis.

Our analyses using the grounded theory approach revealed weaknesses of state-of-the-art smart contract vulnerability detectors, which usually customize classical static and dynamic approach to analyze OO applications to check smart contracts. The pattern-matching tool *SmartCheck* [13] identifies vulnerability mainly based on known vulnerability characteristics and search for vulnerability in small local code snippets. Although *SolDetector* [24] covers global relationships better, it overlooks several defense mechanisms, such as access control and benign target addresses. Tools adopt the symbolic execution, e.g., *Oyente* [2], stimulate the contract execution, and assume that attackers can execute all dangerous paths. Our results show that several implementations, such as extra access control or execution condition checking, can prevent attackers from reaching dangerous paths. Therefore, these tools can be improved with more path analysis to reduce false positives. Tools focus on data flow analysis, such as *Slither* [14], deal with false positives related to access control and complex relationships between functions and contracts better than other tools. To further reduce their false positives, the tools could be enhanced with semantic analysis to address issues like overlooking preventative execution conditions.

More importantly, beyond the general static and dynamic code analysis approaches, our results show that smart contract vulnerability detectors need to consider better the unique characteristics of blockchain and smart contract programming languages. Several issues that cause false positives, such as overlooking access control, neglecting the constraint caused by contract factory pattern, assuming critical operation after authorization, assuming status inconsistency when function calls are not checked, assuming all fallback functions receive ether, and mixing ether transfer initiator, are blockchain specific and require novel or unique detection technologies.

Results of RQ2 show that several factors could influence vulnerability exploitation possibilities and vulnerability criticality. When reporting and ranking the vulnerabilities, vulnerable contracts with no ether can be lower ranked. The extra effort and investments needed from attackers and the attackers' chance to execute the attack shall be considered in vulnerability criticality evaluation.

6.2 Implication to Vulnerability Detector Evaluation

Existing studies, e.g., [10], [11], [16], [19], applied three main methods to construct evaluation benchmarks, namely, collecting vulnerable contracts with manual labels, crawling real-world contracts, and injecting vulnerabilities into contracts. Durieux et al. [16] and Ren et al. [10] collected vulnerable contracts with clear labels to evaluate different tools. However, the number of vulnerable contracts is small, e.g., 69 contracts in [16] and 214 contracts in [10]. These vulnerable contracts are often short and have no complex business logic. Ghaleb et al. [11] constructed a dataset containing 50 contracts with 9,369 injected vulnerabilities. However, the vulnerability injection is limited to known characteristics of vulnerabilities. SolidiFi [11] provides 50 vulnerability patterns for each vulnerability, many of which share the same code logic and only differ in function or variable names. Although studies [10], [16], [19] construct the dataset using real-world contracts, the type and amount of vulnerabilities in these contracts are unknown. Durieux et al. [16] hypothesize that the tools they evaluate report a considerable number of false positives because the percent of vulnerable contracts (44,589/47,518, 93%) is high. However, the studies [10], [16], [19] do not further analyze the reasons for false positives. This study collected unique real-world smart contracts and labeled 4,364 of them. To our knowledge, our dataset is the largest one that can be used to evaluate smart contract vulnerability detectors.

6.3 Threats to Validity

In this study, we only focus on the types of vulnerabilities covered by at least two tools to avoid biases caused by a single tool. Such filtering excluded several types of vulnerabilities, which may reveal other reasons for false positives than we have discovered. The results of RQ1 and RQ2 are based on the vulnerability reported by the four analyzed tools. The tools filtered may already address the issues that cause false positives.

When answering RQ1, we found that more than one reason caused some false positives of a contract. We give only one reason for a false positive to each smart contract because we focus on understanding the reasons rather than counting their numbers. The percentage numbers in Figure 4 are calculated based on this strategy. However, such a strategy will not impact the main findings of RQ1, i.e., the eleven reasons. For RQ2, there are probably false negatives due to unknown attacks and exploitations because our log analysis is limited to the rules presented in Table 7.

7 Conclusion and Future Work

As smart contracts' security is critical, many vulnerability detection tools have been proposed. Several empirical studies show that the tools report many vulnerabilities and,

therefore, hypothesize many reported vulnerabilities are false positives. In this study, we have analyzed the real-world smart contracts reported as vulnerable by four tools and confirm the hypothesis that there is a significant gap between the number of contracts reported as vulnerable and the number of exploited contracts. Our analysis also reveals eleven reasons causing false positives. In addition, we discover six aspects that may demotivate attackers to exploit vulnerable contracts. The results of this study delight the need to consider more the characteristics of smart contract programming languages and smart contract application scenarios and execution environments to analyze, report, and rank the smart contract vulnerabilities.

Besides minimizing false positives, another critical aspect of improving vulnerability detectors is reducing false negatives. Our future work will focus on analyzing the false negative results of the vulnerability detectors and give more suggestions to improve the detectors.

ACKNOWLEDGMENTS

This work is jointly supported by the National Key Research and Development Program of China (No. 2019YFE0105500) and the Research Council of Norway (No. 309494) and the Key Research and Development Program of Jiangsu Province (No. BE2021002-3). TY.H thanks the Chinese Scholarship Council (CSC) for financial support (202106090057).

REFERENCES

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

[2] 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.

[3] 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*, 2018, pp. 67–82.

[4] 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.

[5] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, 2018, pp. 259–269.

[6] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[7] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[8] L. Jin, Y. Cao, Y. Chen, D. Zhang, and S. Campanoni, "Exgen: Cross-platform, automated exploit generation for smart contract vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2022.

[9] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1591–1607.

[10] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: What is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 566–579. [Online]. Available: https://doi.org/10.1145/3460319.3464837

[11] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[12] (2018) Mythril: an open-source security analysis tool for ethereum smart contracts. [Online]. Available: https://github.com/ConsenSys/mythril

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

[14] 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*, 2019, pp. 8–15.

[15] (2019) Manticore: a symbolic execution tool for analysis of smart contracts and binaries. [Online]. Available: https://github.com/trailofbits/manticore

[16] T. Durieux, J. a. 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*, ser. ICSE'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 530–541. [Online]. Available: https://doi.org/10.1145/3377811.3380364

[17] I. Nikolic, 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*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 653–663.

[18] 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*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 664–676. [Online]. Available: https://doi.org/10.1145/3274694.3274737

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

[20] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Network and Distributed System Security Symposium*, 2018, pp. 18–33.

[21] J. Krupp and C. Rossow, "Teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium*, 2018, pp. 1317–1333.

[22] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[23] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 332–343. [Online]. Available: https://doi.org/10.1145/2970276.2970347

[24] H. Tianyuan, P. zhenyu, and L. Bixin, "Soldetector: Detect defects based on knowledge graph of solidity smart contract," in *Proceedings of the 32rd International Conference on Software Engineering and Knowledge Engineeringy*, 2021, pp. 423–429.

[25] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 120–131.

[26] 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)*, 2020, pp. 1029–1040.

[27] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[28] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural networks," ser. IJCAI'20, 2021.

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

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

[31] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. Association for Computing Machinery, 2018, pp. 65–68.

[32] Q. Zhang, Y. Wang, J. Li, and S. Ma, "Ethploit: From fuzzing to efficient exploit generation against smart contracts," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 116–126.

[33] (2013) World wide web consortium, sparql 1.1 update. [Online]. Available: https://www.w3.org/TR/sparql11-update

[34] J. Ye, M. Ma, Y. Lin, Y. Sui, and Y. Xue, "Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2020, pp. 274–275.

[35] (2021) Wikipedia, datalog: a declarative logic programming language. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Datalog oldid=1053711548

[36] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 422–430.

[37] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[38] J. a. 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*, ser. ASE'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1349–1352. [Online]. Available: https://doi.org/10.1145/3324884.3415298

[39] (2022) Solhint: an open source project for linting solidity code. [Online]. Available: https://www.npmjs.com/package/solhint/

[40] (2019) Ethereum (eth) blockchain explorer. [Online]. Available: https://etherscan.io

[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] (2022) Jaccard index: a statistic used for gauging the similarity and diversity of sample sets. [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index

[43] (2021) Securify 2.0: a security scanner for ethereum smart contracts supported by the ethereum foundation and chainsecurity. [Online]. Available: https://github.com/eth-sri/securify2

[44] "Smart contract weakness classification and test cases," https://swcregistry.io/, 2020, accessed 28 May 2022.

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

[46] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "Easyflow: Keep ethereum away from overflow," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 23–26.

[47] (2020) Delegatecall to untrusted callee. [Online]. Available: https://swcregistry.io/docs/SWC-112

[48] P. Praitheeshan, L. Pan, X. Zheng, A. Jolfaei, and R. Doss, "Solguard: Preventing external call issues in smart contract-based multi-agent robotic systems," *Information Sciences*, vol. 579, pp. 150–166, 2021.

[49] C. F. Torres, H. Jonker, and R. State, "Elysium: Automagically healing vulnerable smart contracts using context-aware patching," *arXiv preprint arXiv:2108.10071*, 2021.

[50] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[51] G. Wood. (2016, Jun.) Condition-orientated programming. [Online]. Available: https://gavofyork.medium.com/condition-orientated-programming-969f6ba0161a

[52] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides *et al.*, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[53] "Solidity: a statically-typed curly-braces programming language designed for developing smart contracts that run on ethereum," https://soliditylang.org/, 2022, accessed 25 July 2022.

[54] C. Ferreira Torres, A. K. Iannillo, A. Gervais *et al.*, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *European Symposium on Security and Privacy, Vienna 7-11 September 2021*, 2021.

Tianyuan Hu is currently working toward the Ph.D. degree with the School of Computer Science and Engineering, Southeast University under the supervision of Dr. Bixin Li. Her research interests include program analysis, vulnerability detection, blockchain security, and software engineering.

Jingyue Li is a Professor at the Computer Science Department, Norwegian University of Science and Technology (NTNU). He received his Ph.D. degree in software engineering from the Department of Computer Science, NTNU, in 2006. His research interests include software engineering, software security, and blockchain technologies.

André Storhaug is a Ph.D. student in the Department of Computer Science at the Norwegian University of Science and Technology (NTNU). His research interests include machine learning, software engineering, software security, and blockchain technologies.

Bixin Li received his bachelor's degree and master's degree both in mathematics from Anhui University in 1991 and 1994, respectively, and received his doctor' degree in software engineering from Nanjing University in 2001. He is a full Professor of School of Computer Science and Engineering of Southeast University, he is the chairman of Technology Committee of Software Engineering Standards of Jiangsu Province, and he is also the header of Software Engineering Institute of Southeast University in that he is working hard together with more than 50 young people on software architecture and blockchain security projects etc. His main research interests include program slicing and its application, software evolution and maintenance, software testing and verification, software safety and security techniques etc. He has published over 180 research papers and patented more than 80 inventions of china up to now.

APPENDIX

TABLE 7: Reasons for excluding Vulnerability Detection Tools

Year and ref.	Tool Name	SCI	SV	VL	Eff	Availability
Pattern Matching						
2018 [13]	Smartcheck					
2021 [24]	SolDetector					
Symbolic Execution						
2016 [2]	Oyente					
2018 [20]	ZEUS					•
2018 [12]	Mythril				•	
2018 [3]	Securify		•			
2018 [21]	TEETHER	•				
2018 [17]	MAIAN			•		
2019 [9]	HONEYBADGER	•				
2021 [7]	DefectChecker		•			
2022 [8]	EXGEN					•
Data Flow Analysis						
2018 [18]	Osiris	•				
2018 [22]	MadMax	•				
2019 [14]	Slither					
2020 [26]	Clairvoyance					•
2020 [27]	Ethainter	•				
Machine Learning						
2019 [28]	GNN-based			•		
2020 [29]	ContractWard			•		
2021 [30]	VSCL	•		•		
Fuzzing						
2018 [5]	ContractFuzzer	•				
2018 [31]	Reguard				•	
2020 [4]	sFuzz	•				
2020 [32]	Ethploit			•		
2021 [54]	ConFuzzius					

Note: • means that the tool does no meet the criterion.

TABLE 8: Experiment information of existing tools

Year and ref.	Tool Name	Dataset	FDR=FP/(TP+FP)
2016 [2]	Oyente	175 real-world contracts [2]	6.4%
2018 [13]	SmartCheck	3 real-world contracts [13]	68.9%
2019 [14]	Slither	1000 real-world contracts [14]	10.9%
2021 [24]	SolDetector	179 curated contracts [24]	3.1%
		1,562 real-world contracts [24]	17.4%

Results for RQ2.

TABLE 9: Results of Mythril and ConFuzzius on overlapping contracts

Vulnerability		Total Number	Mythril			ConFuzzius		
			Reported	P(%)	FDR(%)	Reported	P(%)	FDR(%)
RE	exploitable	71	54	96.43	3.57	22	95.65	30.99
	unexploitable	26	2			0		
UcC	exploitable	41	15	25.86	74.14	21	25.30	51.22
	unexploitable	178	43			62		
TO	exploitable	24	8	72.72	27.27	-	-	-
	unexploitable	21	3			-	-	-
TD	...							
...	...							

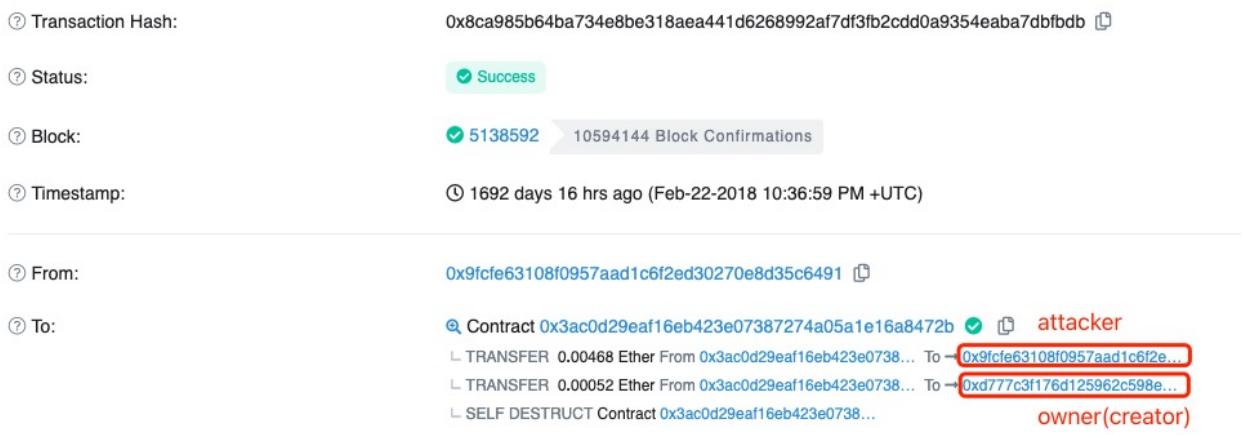

Fig. 8: The Transaction Details of the Exploited UpS

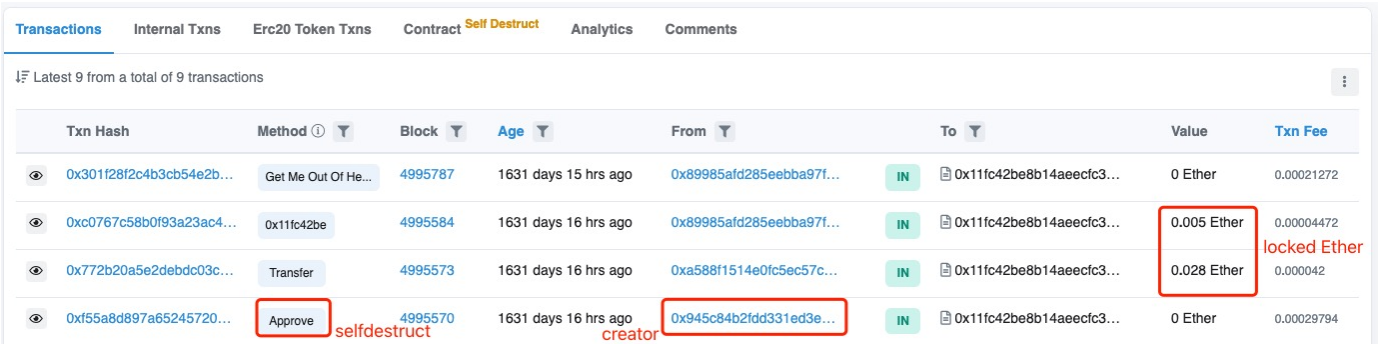

Fig. 9: Locked Ether in a Self-Destructed Contract (Contract Address: 0x11fC42Be8B14aEeCfc371Af217c4648e6423fA60)

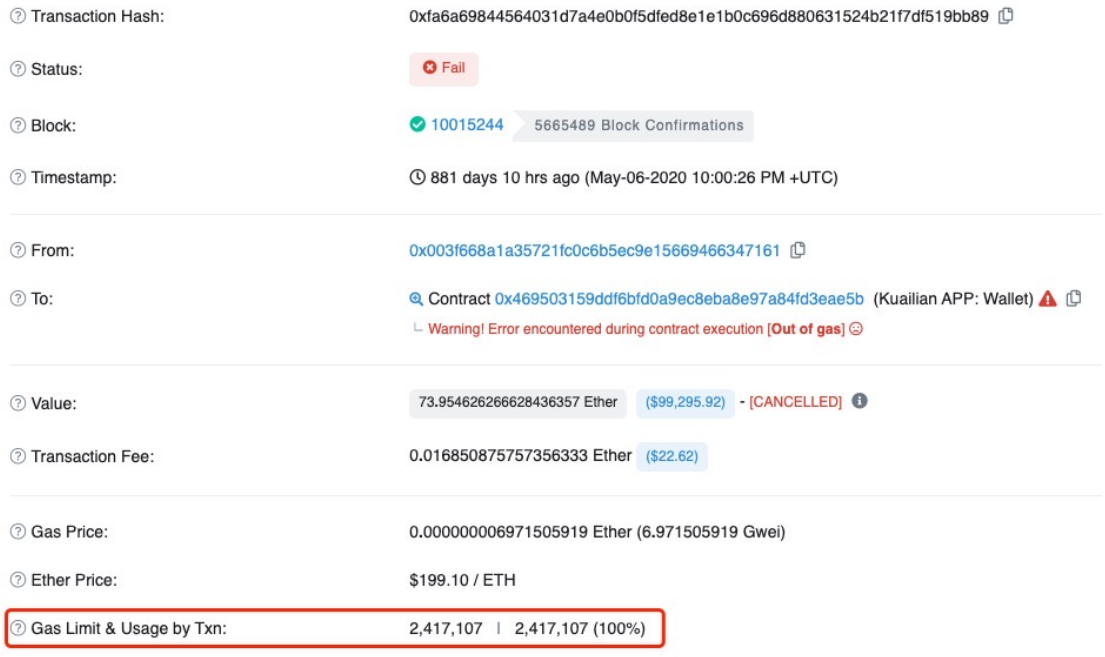

Fig. 10: Transaction Information on Etherscan (Hash:0xfa6a69844564031d7a4e0b0f5dfed8e1e1b0c696d880631524b21f7df519bb89)