Why Smart Contracts Reported as Vulnerable were not Exploited?

Tianyuan Hu, Jingyue Li, Bixin Li, André Storhaug

Abstract—Smart contract security is essential for blockchain applications. Studies show that few of the reported vulnerabilities are exploited. However, no follow-up study is performed to why the reported vulnerabilities are not exploited. We aim to understand the reasons for the low exploitation rate to help improve vulnerability detection practices. We first collect 136,969 unique real-world smart contracts and analyze them using seven vulnerability detectors. Then, we apply Strauss' grounded theory approach to understand if they are exploitable. In addition, we analyze the transaction logs of the exploitable vulnerabilities to understand their exploitations in history. Among the 4,364 smart contracts reported as vulnerable by the vulnerability detectors, 75.27% of them are unexploitable. Only 66 (0.015%) exploitable contracts have been exploited. We uncover 11 reasons for making the detectors misidentify unexploitable vulnerabilities and six reasons that may lower the possibility of exploitable contracts being exploited by attackers. We illustrate that: beyond treating the smart contracts as yet another Object Oriented (OO) application, it is essential to consider the Solidity programming language's design principle, smart contracts' application scenarios, and their execution environments. Based on the study's insights, we provide several suggestions to improve smart contract vulnerability detection, prioritization, and mitigation.

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 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 detectors, e.g., *Oyente* [2], *Securify* [3], *sFuzz* [4], *ContractFuzzer* [5], *ContraMaster* [6], *DefectChecker* [7], *EXGEN* [8] have been proposed to detect smart contract vulnerabilities. Ren et al. [9] point out that the detectors 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 detectors fairly. *SolidiFI* [10] is used to evaluate six detectors [2], [3], [11], [12], [13], [14]. The results show that none of the detectors detect all the injected bugs correctly, and all the evaluated detectors report several false positives. Durieux et al. [15] evaluated nine vulnerability detectors [2], [3], [11], [12], [13], [14], [16], [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 detectors, namely, *Oyente* [2], *ZEUS* [20], *MAIAN* [16], *Securify* [3], *TEETHER* [21], *Madmax* [22]. They classified the analyzed smart contracts as:

- **Reported as vulnerable:** A contract is reported as vulnerable if the vulnerability detector flags it. However, the flags could be false positives.
- **Exploitable:** A contract is exploitable if an attacker could exploit its vulnerability and cause security compromise.
- **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 detectors, only 463 contracts were exploited. They hypothesized that most reported vulnerabilities were either false positives or not exploitable. However, no follow-up study tried to confirm the hypothesis and understand the reasons for the low exploitation rate. "*Vulnerability deals with the theoretical, and exploitability deals with actuals. Understanding what is vulnerable and what remains exploitable can help companies prioritize and acknowledge where their security efforts can be improved* [23]." Ren et al. [9] pointed out that detectors with high precision might perform poorly in reality due to the ignorance of exploitability analysis. Thus, we are motivated to answer two research questions (RQs):

- **RQ1:** Are vulnerable smart contracts reported by vulnerability detectors exploitable? If the reported vulnerable contract is unexploitable, what are the possible reasons for that?
  If a smart contract is reported as vulnerable, they could be false positives, according to the definitions of each vulnerability type specified in the state-of-the-art books and literature, e.g., the Smart Contract Weakness Classification Registry [24] and the

---

- *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.

Ethereum book [25]. If the contract is a true positive according to the state-of-the-art vulnerability definitions, we want to know if it is exploitable. For the unexploitable ones, we want to know what makes them unexploitable.

- **RQ2:** Are exploitable smart contracts exploited? If not, what prevented attackers from exploiting them? If a smart contract is exploitable, we want to know if the vulnerability in this contract has been executed at least once, although we may not know the actual loss due to the execution. In addition, we want to identify commonalities between the exploitable smart contracts that have and have not been executed.

To answer RQ1, we collected 136,969 unique real-world smart contracts and used four efficient vulnerability detectors, namely, *Oyente* [2], *SmartCheck* [12], *Slither* [13], *SoliDetector* [26] to label their vulnerabilities. Then, we manually analyzed the source code of the vulnerable contracts to judge whether they are exploitable and adopted Strauss' grounded theory method [27] to identify the reasons for reporting unexploitable contracts by the detectors. The insights from the manual analysis are then cross-validated using automatic tools, which are mainly the dynamic detectors, namely, *Mythril* [11], *ConFuzzius* [28], and *Smartian* [29]. 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 also used Strauss' grounded theory method to understand the contracts' exploitations.

Results show that 4,364 contracts are labeled as vulnerable by at least two of the effective vulnerability detectors. Through open and axial coding, we identified 11 reasons causing the detectors to misidentify unexploitable vulnerabilities. The reasons can be grouped into three schemes.

Scheme 1: Weaknesses of the detector in adapting approaches to analyze OO-based applications result in reporting vulnerabilities that are not reachable and triggerable. This scheme contains three reasons, which are related to reporting false positives according to state-of-the-art vulnerability definitions.

- missing path feasibility analysis
- overlooking preventive execution condition
- insufficient data flow analysis

Scheme 2: Overlooking the characteristics of the Solidity programming language results in reporting unexploitable vulnerabilities. The reasons in this scheme provide novel insights into one main weakness of existing vulnerability detectors, i.e., they only treat smart contracts as typical OO language-based applications without considering the smart contract programming language characteristics.

- overlooking specific access control mechanism related to smart contract and solidity
- neglecting constraints caused by factory patterns
- neglecting characteristics related to contract inheritance
- assuming all fallback functions receive ether

Scheme 3: Smart contract application scenarios reduce exploitability. The reasons in this scheme provide novel insights to encourage smart contract vulnerability detectors to add smart contract application scenarios into consideration when reporting and prioritizing vulnerabilities.

- insufficient analysis of the values of the target contracts' addresses
- omitting the case that the ether transfer initiator is the ether's initial owner
- assuming critical operations after authorization
- assuming status inconsistency when the function call results are not checked

After analyzing the 4,106,134 transaction logs of the 4,364 vulnerable contracts, we found that only 66 exploitable contracts had been exploited. We have also found six reasons that may lower the possibility of exploitable contracts being exploited by attackers. The reasons are more fruitful and concrete than the state-of-the-art knowledge in [19].

- very little or no financial benefits for attackers
- insignificant impacts of the compromise
- high attack complexity because attackers must develop comprehensive attack contracts
- costly attack failure because attackers must deposit ether as a prerequisite of the attack
- attacker must be lucky in competition with randomness
- attacker must be a mining winner

Based on the novel insights and theories regarding the reasons for the low exploitation rate, we provided novel suggestions to improve smart contract vulnerability detection and ranking.

- enhancing the analysis of dependencies introduced by modifiers
- considering the specific characteristics and constraints of smart contract inheritances
- simulating the contract execution with real blockchain environments
- strengthening blockchain application scenario relevance in vulnerability detection.
- adding checks of the exploitation assumptions
- ranking the reported vulnerability according to smart contract exploitability risks

In addition, we 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 [9]. The dataset can help evaluate the vulnerability detectors' ability to detect vulnerabilities and their exploitability 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.

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

Year and ref.	Detector name	Vul. types covered	Inputs
Pattern Matching			
2018 [12]	SmartCheck	37 types	*SC*
2023 [26]	SoliDetector	20 types	*SC*
Symbolic Execution			
2016 [2]	Oyente	6 types	*SC*
2018 [20]	ZEUS	7 types	*SC*
2018 [17]	Osiris	Integer Vulnerability	*BC*
2018 [11]	Mythril	SWC Registry	*SC*
2018 [3]	Securify	37 types	*SC*/*BC*
2018 [21]	TEETHER	4 types	*BC*
2018 [16]	MAIAN	3 types	*SC*/*BC*
2019 [18]	HONEYBADGER	Honeypots	*BC*
2021 [7]	DefectChecker	8 types	*SC*
2022 [8]	EXGEN	4 types	*SC*
Data Flow Analysis			
2018 [22]	MadMax	3 types	*BC*
2019 [13]	Slither	71 types	*SC*
2020 [30]	Clairvoyance	Reentrancy	*SC*
2020 [31]	Ethainter	5 types	*BC*
Machine Learning			
2019 [32]	GNN-based	3 types	*SC*
2020 [33]	ContractWard	6 types	Opcode
2021 [34]	VSCL	6 types	*BC*
Fuzzing			
2018 [5]	ContractFuzzer	7 types	*BC*+ABI
2018 [35]	Reguard	Reentrancy	*SC*/*BC*
2020 [4]	sFuzz	9 types	*BC*
2020 [36]	Ethploit	3 types	*SC*
2021 [28]	ConFuzzius	10 types	SC
2021 [29]	Smartian	13 types	BC
2022 [6]	ContraMaster	5 types	SC

2.1 Detectors Using Pattern Matching Approaches

SmartCheck [12] translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns. *SoliDetector* [26] 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 [37] query to manipulate the knowledge graph and identify vulnerabilities.

2.2 Detectors Relying on Symbolic Execution

Oyente [2] is the first smart contract vulnerability detector based on symbolic execution. It symbolically executes the contract to identify vulnerabilities. *ZEUS* [20] combines abstract interpretation and symbolic execution to model smart contracts. *Osiris* [17] is a framework that combines symbolic execution and taint analysis to detect vulnerabilities related to arithmetic operations in Ethereum smart contracts. *Mythril* [11] uses symbolic execution and taint analysis to detect vulnerabilities. It produces execution traces using a dynamic symbolic execution engine called Laser-EVM (Ethereum Virtual Machine). *Securify* [3] combines abstract interpretation and symbolic execution and automatically classifies behaviors of a contract into three categories, i.e., compliance, violation, and warning. TEETHER [21] employs symbolic execution to create an exploit, but has difficulty solving hard constraints in execution paths and cannot simulate the blockchain behaviors. *MAIAN* [16] is a symbolic execution tool that classifies vulnerable contracts into three categories, namely, greedy, prodigal, and suicidal. *HONEYBADGER* [18] uses symbolic execution and predefined heuristics to expose honeypots. *DefectChecker* [7] symbolically executes the bytecode and uses eight rules to detect different vulnerabilities. However, the public version of *DefectChecker* supports only Solidity 0.4.24. *EXGEN* [8] generates multiple transactions as exploits to vulnerable Ethereum or EOS contracts and verifies the contracts' exploitability on a private chain.

2.3 Detectors Applying Data Flow Analysis

MadMax [22] is a gas-focused vulnerability detection tool consisting of a decompiler, which converts bytecode to code represented using an intermediate language. *Slither* [13] 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* [30], [38] presents a static analysis tool that models cross-function and cross-contract behavior to detect the reentrancy vulnerability. Brent et al. [31] present *Ethainter* to detect composite vulnerabilities that escalate a weakness through multiple transactions. Based on the Datalog language [39] and the Soufflé Datalog engine [40], *Ethainter* constructs graphs containing data flow and control flow dependencies to identify vulnerabilities.

2.4 Detectors Using Machine Learning Technologies

Zhuang et al. [32] use a graph neural network (GNN) to classify vulnerable smart contracts. *ContractWard* [33] is an ML-based vulnerability detection tool targeting six vulnerabilities. Their evaluations show that XGBoost is the best-performing classifier algorithm. *VSCL* [34] is a smart contract vulnerability detection framework that constructs a control flow graph (CFG) to understand program run time behavior. N-gram and Term Frequency–Inverse Document Frequency (TFIDF) techniques are used to generate vectors to present features of smart contracts.

2.5 Detectors Using Fuzz Testing

Fuzz testing [41] 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) and detects the vulnerabilities by matching predefined test oracles. *ReGuard* [35] is a fuzzing tool to detect reentrancy vulnerabilities. It first converts the input to smart contracts into a C++ program 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* [36] adopts a dynamic seed strategy and static taint analysis to generate exploit-targeted transaction sequences based on an instrumented EVM. *ContraMaster* [6] is an oracle-supported dynamic exploit generation framework. It uses the dynamic contract states to guide its mutations of the transaction sequences. *ConFuzzius* [28] combines evolutionary fuzzing, constraint solving, and dynamic data flow to generate test cases and detect vulnerabilities. *Smartian* [29] conducts static and dynamic analysis for fuzz testing smart contracts. It predicts the transaction sequences and uses the dynamic data flow to guide the test case generation.

2.6 Empirical Evaluations of Vulnerability Detectors

Although studies proposing new vulnerability detectors 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 detectors. Thus, other researchers performed empirical studies as shown in Table 5 in Appendix to evaluate and compare the smart contract vulnerability detectors.

Ghaleb et al. [10] proposed *SolidiFI* to evaluate six static vulnerability detectors [2], [3], [11], [12], [13], [14] using a dataset with injected vulnerabilities. Experiment results on a set of 50 contracts injected with 9,369 distinct vulnerabilities show that the evaluated detectors do not detect several instances of vulnerabilities despite their claims of being able to detect such vulnerabilities. Only one tool, i.e., *Slither* [13], detected all injected vulnerabilities. Ghaleb et al. [10] also found that all evaluated detectors have reported several false positives, ranging from 2 to 801 for different vulnerability types. However, they only manually analyze the vulnerabilities that are not reported by the majority of the detectors because manually inspecting all reported vulnerabilities involves a tremendous amount of effort and is, therefore, impractical. As a result, the number of false positives is underestimated.

Ferreira et al. [42] presented *SmartBugs*, an extensible framework that simplifies the execution of smart contract detectors. *SmartBugs* 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 [43]. However, the 47,518 real-world contracts are not manually labeled. By using *SmartBugs*, Durieux et al. [15] evaluated nine detectors [2], [3], [11], [12], [13], [14], [16], [17], [18]. The evaluation was based on 69 annotated vulnerable smart contracts and all the real-world smart contracts in *SmartBugs*. The evaluations showed that 97% of real-world contracts were labeled as vulnerable. Durieux et al. [15] questioned that many reported vulnerabilities are false positives. Ren et al. [9] evaluated six detectors [2], [4], [5], [11], [17], [44]. The experiment results demonstrated that different experimental settings could significantly affect performance and lead to misleading or even opposite conclusions.

Perez et al. [19] evaluated six detectors [2], [3], [16], [20], [21], [22] on real-world smart contracts and found many contradict results from different detectors [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 detectors [19]. In addition, they analyzed more than 20 million Ethereum 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 detectors were either false positives or not exploitable.

Although the aforementioned empirical studies hypothesized that existing vulnerability detectors report many false positives or that the reported vulnerabilities are not exploitable, especially for real-world contracts, no follow-up study was performed to confirm the hypothesis and to understand the reasons for this phenomenon. The insights could help security practitioners rank the reported vulnerabilities to allocate the security effort to the most urgent vulnerabilities to fix, i.e., those exploitable ones.

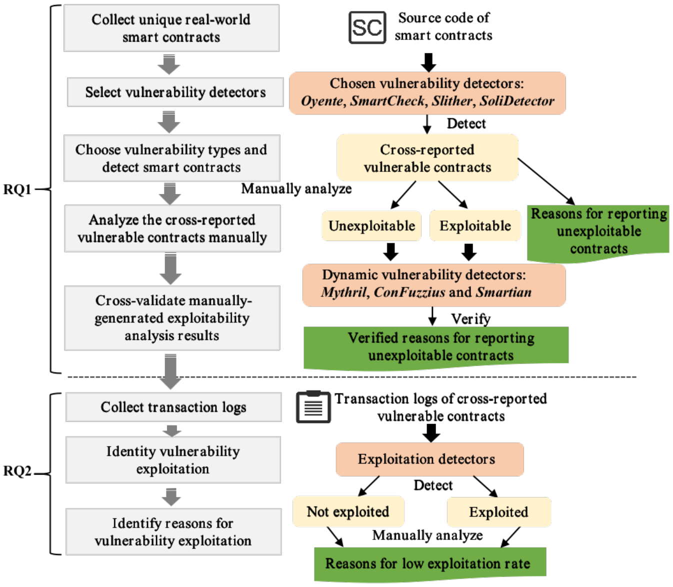

RQ1
Collect unique real-world smart contracts
Select vulnerability detectors
Choose vulnerability types and detect smart contracts
Analyze the cross-reported vulnerable contracts manually
Cross-validate manually-genenrated exploitability analysis results
SC
Source code of smart contracts
Chosen vulnerability detectors: Oyente, SmartCheck, Slither, SoliDetector
Detect
Cross-reported vulnerable contracts
Manually analyze
Unexploitable
Exploitable
Reasons for reporting unexploitable contracts
Dynamic vulnerability detectors: Mythril, ConFuzzius and Smartian
Verify
Verified reasons for reporting unexploitable contracts
RQ2
Collect transaction logs
Identity vulnerability exploitation
Identify reasons for vulnerability exploitation
Transaction logs of cross-reported vulnerable contracts
Exploitation detectors
Detect
Not exploited
Exploited
Manually analyze
Reasons for low exploitation rate

Fig. 1: Workflow of the study

3 RESEARCH DESIGN

3.1 Research Design to Answer RQ1

To answer RQ1, we designed the research flow as shown in Figure 1.

3.1.1 Step 1. Collect unique real-world smart contracts

We crawled all available smart contracts with at least one transaction from Etherscan [43] 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 [45]. 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

The criteria used to select the vulnerability detectors are listed below.

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

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

- *The detectors shall support multiple Solidity Versions (SV)*: We crawl real-world smart contracts from Ethereum. These smart contracts use various Solidity versions. If the detectors 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 detectors to support several versions.
- *The detectors shall be available to us (Available)*: Not all papers make their detectors publicly available. We exclude detectors we cannot access.
- *The detector shall not have high false negative risk (HighFN-Risk)*: The focus of our study is to understand the reasons for the low exploitability of the reported vulnerabilities. If the detectors have a high risk of reporting false negatives, we will miss many vulnerable smart contracts to analyze.

3.1.3 Step 3. Choose vulnerability types to focus on and use the selected detectors to detect the chosen types of smart contracts

The vulnerability detection results from a particular detector can be biased by the detectors' design flaws or bugs. As we want to identify generic reasons for the low exploitability, we choose to detect only the vulnerability types supported by at least two detectors to reduce the possible biases introduced by a single detector. The chosen smart contracts are, hereafter, called *cross-reported vulnerable contracts*. We get detection results containing vulnerability type names and locations after using the detectors to detect the chosen smart contracts on the chosen vulnerability types.

3.1.4 Step 4. Analyze the cross-reported vulnerable contracts manually

Our study aims to understand why reported vulnerabilities are not exploited. We believe that there must be reasons for the low exploitation rate. First, we read the source code of each smart contract reported as vulnerable and applied Strauss' grounded theory approach [27], often used to identify generic and unknown theories from data. Strauss' grounded theory approach [27] 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 classified the vulnerabilities into two categories, i.e., exploitable or unexploitable, in parallel with root cause analysis and open coding to categorize the reasons for reporting unexploitable contracts. 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 reporting unexploitable contracts across vulnerability types. The analysis ended when we could not derive more categories of reasons from the open codes. The axial coding resulted in 11 reasons for reporting unexploitable contracts explained in Section 4.5. After that, we performed selective coding to connect reasons identified through axial coding to generate coherent explanatory schemes, i.e., the theories.

3.1.5 Step 5. Cross-validate manually-generated exploitability analysis results

To verify our manually-generated findings of exploitability, for the reported vulnerabilities that are found to be exploitable, we want to use automatic tools, such as detectors based on dynamic code analysis approaches, to check if the identified vulnerable lines are reachable and triggerable. For the vulnerability found to be unexploitable, we also want to use the tools to cross-validate our manually identified reasons for the low exploitability.

3.2 Research Design to Answer RQ2

The steps to answer RQ2 in Figure 1 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 labels the reported vulnerable smart contracts as exploitable or unexploitable. For unexploitable contracts, we analyze their transaction logs to check if they are exploited. The purpose is to verify that our low exploitability analysis is correct. We expect that there shall have no exploitation in the transaction logs of the unexploitable contracts. We also analyze the exploitable 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 exploited contract on Ethereum's main network. Nevertheless, there are many exploitable smart contracts that are not exploited. We, again, use Strauss' grounded theory approach [27] 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. From the axial coding results, we performed selective coding to schemes, which will also be extensively recounted in Section 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*). Figure 7 in Appendix shows the transaction information of these contracts and indicates that 88.37% of contracts have more than one transaction. Figure 7 in Appendix 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

As shown in Table 7 in Appendix, we first excluded detectors due to their unavailability, inability to accept source code as input, or inability to localize the vulnerability based on the detector selection criteria in Section 3.1.2.

Mythril [11] and fuzzing-based detectors, e.g., *ConFuzzius* [28], *Smartian* [29], need to set a timeout to determine when to stop analyzing a smart contract. Setting a long timeout may make it too slow to analyze the 136,969 smart contracts. A short timeout can lead to high false negative rates. As there was no guideline on the timeout value to analyze the real smart contracts, we decided not to use these detectors to identify vulnerabilities to avoid missing many vulnerable smart contracts. However, the dynamic and fuzzing detectors are suitable to cross-validate our manual analysis results of the low exploitation rate, which is explained in Section 4.6. Finally, we choose to use four detectors, namely, *Oyente* [2], *Smartcheck* [12], *Slither* [13], *SoliDetector* [26] to label contracts. The chosen detectors cover pattern matching, symbolic execution, and data flow analysis approaches.

4.3 Chosen Vulnerability Types

To choose vulnerability types supported by at least two detectors, we did a mapping of the types between the detectors and decided to focus on ten types of vulnerability, as shown in Table 6 in Appendix. It is worth noting that the vulnerability names in Table 6 are extracted from the detection results of the detectors, which may be different from the names in the papers presenting the detectors because the authors of the detectors did not make the names consistent. According to state-of-the-art books and literature, e.g., [24] and [25], the definitions and characteristics of the ten chosen vulnerability types are as follows.

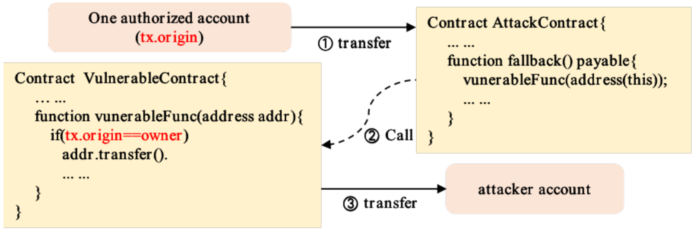
4.3.1 Unprotected Suicide (UpS)

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 [24], a contract vulnerable to the UpS attack has the following characteristics.

1) A function containing the *selfdestruct(address)* function.
2) No mechanism to prevent 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 [25]. 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. 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 and there is no receive ether function [46]. In Figure 2, 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 [25], a contract with TO vulnerability has the following characteristics.



```
Contract AttackContract{
  … …
  function fallback() payable{
    vunerableFunc(address(this));
    … …
  }
}
```

```
Contract VulnerableContract{
  … …
  function vunerableFunc(address addr){
    if(tx.origin==owner)
      addr.transfer().
      … …
  }
}
```

Fig. 2: An attack process exploiting TO

1) *tx.origin* is used for owner authorization in a function.

4.3.3 Arithmetic Overflow and Underflow

An arithmetic overflow or underflow [17], [47], 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. The arithmetic overflow or underflow may happen if a contract meets the following characteristic [24].

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

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 runs 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 [48], [49], because state-preserving of *delegatecall* refers to the storage slots rather than the variable name. According to [48], [49], [50], a contract vulnerable to the DC attack usually has the following characteristics.

1) A function containing the *delegatecall* function.
2) No method to prevent the attacker from specifying the calldata or changing the target contract address.

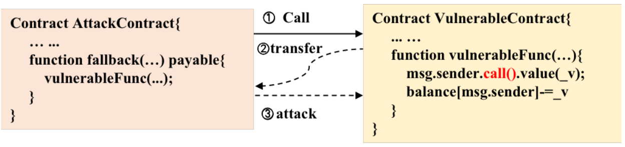
4.3.5 Unchecked Call (UcC)

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], [24], [51]. 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], [24], [51].

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.



```
Contract AttackContract{
    … …
    function fallback(…) payable{
       vulnerableFunc(...);
    }
}
```

```
Contract VulnerableContract{
    … …
    function vulnerableFunc(…){
       msg.sender.call().value(_v);
       balance[msg.sender]-=_v
    }
}
```

Fig. 3: An attack exploiting RE

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

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 9000 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 method to limit 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 [25]. As there are many variations of TOD attacks, finding a precise characteristic of smart contracts vulnerable to TOD is challenging. According to [25], a high-level characteristic of a smart contract vulnerable to TOD 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 Vulnerable Contracts Reported by Detectors

The results of analyzing the 136,969 smart contracts focusing on the ten vulnerability types are shown in Table 2. The data in the *Overlap* column of Table 2 show the *cross-reported vulnerable contracts* flagged as vulnerable by multiple detectors.

Results in Table 2 show that the detectors 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 2. In total, we analyzed 4,364 contracts reported as vulnerable.

TABLE 2: Detection Results of Chosen Detectors

Vul.	Oyente	SmartCheck	Slither	SoliDetector	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

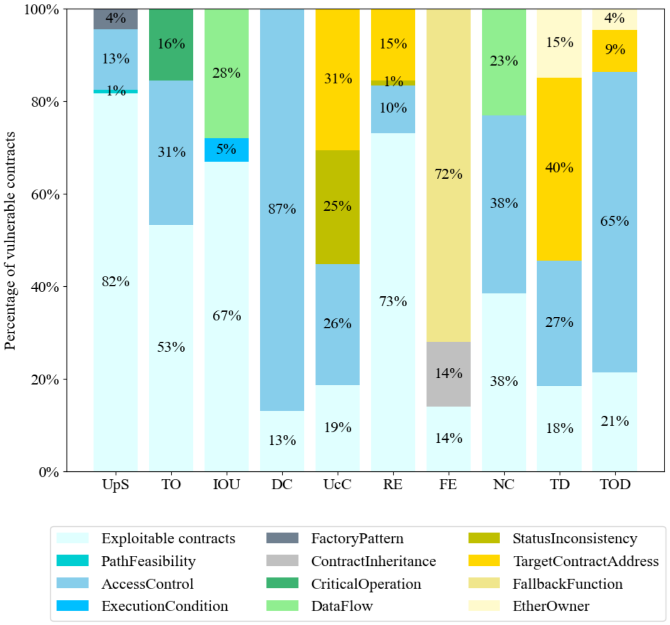


Fig. 4: The reasons for reporting unexploitable contracts

4.5 Results of Analyzing the Reported Vulnerabilities

We manually analyzed 4,364 *cross-reported vulnerable contracts* and found that 1,079 were exploitable and 3,285 were unexploitable. Using Strauss' grounded theory methodology, we identify 11 reasons shown in Figure 4 and further abstracted into three schemes.

Scheme 1: Weaknesses of the detector in adapting approaches to analyze OO-based applications result in reporting vulnerabilities that are not reachable and triggerable.

Three reasons in Scheme 1 are explained in detail in Sections 4.5.1 to 4.5.3. The unexploitable smart contracts associated with these reasons can also be regarded as false positives, according to the vulnerability definitions in, e.g., [24] and [25]. We found one false positive for UpS, 33 false positives for IOU, and 109 false positives for NC.

4.5.1 Missing path feasibility analysis (PathFeasibility)

SoliDetector [26], and *Slither* [13] overlook the path feasibility analysis and assume all code paths are reachable. One contract reported as vulnerable to UpS is unexploitable because the vulnerable function *selfdestruct* locates in an infeasible path 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 (ExecutionCondition)

Vulnerability detectors reported the vulnerability that could not be triggered due to the preventive execution condition. For instance, in Listing 1, *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]*.

```
function transferFrom(address _from, address _to, uint256
     _value) {
    if (balances[_from] >= _value ) {
       balances[_from] -= _value;
       Transfer(_from, _to, _value);}
}
```

Listing 1: A code with IOU vulnerability but is unexploitable

4.5.3 Insufficient data flow analysis (DataFlow)

Data flow analysis is essential to detect variable-related vulnerabilities, such as IOU and NC. By analyzing the 100 contracts reported as vulnerable to IOU, we found cases where variables involved in arithmetic operations are fixed values or have a fixed range. The arithmetic operations on these variables will not trigger overflow or underflow due to data flow control within a function or across functions.

Listing 2 shows an example of a vulnerable code with IOU that is unexploitable caused by the data flow control within one function. In Listing 2, 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 2: A code with IOU vulnerability but is unexploitable

An example of data flow control preventing the reported NC vulnerability from being exploitable is shown in Listing 3. In Listing 3, 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 in line 1. Therefore, the total gas cost of the function *closeSession* will not be more than the given gas. Thus, the reported NC vulnerabilities will not lead to the permanent failure of the function *closeSession*.

```
1  uint public constant MaxInvestor = 20;
2  function closeSession (uint _priceClose) public onlyEscrow{
3      for (uint i = 0; i < session.investorCount; i++) {...}
4      session.investorCount = 0;
5  }
6  function invest (bool _choose) public payable{
7      require(msg.value >= minimunEth && session.investOpen);
8      require(session.investorCount < MaxInvestor);
9      session.investor[session.investorCount]=msg.sender;
10     session.amountInvest[session.investorCount]=msg.value;
11     session.investorCount+= 1;
12 }
```

Listing 3: A code with NC vulnerability but is unexploitable

Scheme 2: Overlooking the characteristics of the Solidity programming language results in reporting unexploitable vulnerabilities.

The smart contracts related to the reasons here are true positives, according to the vulnerability definitions. However, they are unexploitable. The reasons for low exploitability in Sections 4.5.4 to 4.5.7 can be coded to Scheme 2.

4.5.4 Overlooking specific access control mechanism related to smart contract and solidity (AccessControl)

Gavin Wood designed Solidity to support condition-oriented programming [52], a subdomain of contractorientated programming with 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 them from being exploitable. Thus, these reported vulnerabilities are unexploitable. This section presents several examples of methods in Solidity language that can defend attackers against exploiting the vulnerability.

Eight vulnerability types can be protected by access control, i.e., UpS, TO, DC, UcC, RE, NC, TD, and TOD. Examples of access control checking the identities of critical functions’ caller or controlling a critical variable are as follows.

① The *if/require* condition is set before critical operations, such as *require(msg.sender == owner)*. For example, the access control on the function containing the UpS vulnerability prevents the attacker from exploiting it. In Listing 4, 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 4: A code with UpS vulnerability but is unexploitable

Another example is that: a smart contract labeled vulnerable to TO if *tx.origin* is used for authorization. The TO vulnerability is unexploitable because multiple-level access control defends the vulnerability from being exploited. 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 reject external contracts 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 5, 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 [25]. In 13 RE vulnerabilities, the code of the modifier checks whether the *msg.sender* is *tx.origin* or *owner*, which makes the recursive call from another contract impossible.

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

Listing 5: A code with TO vulnerability but is unexploitable

③ Across control is performed across functions. Some NC vulnerabilities are unexploitable due to access control across multiple functions or modifiers. In Listing 6, the loop in line 6 is labeled with NC, and the max number of loop iterations 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 NC vulnerability in the function *buy*.

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

Listing 6: A code with NC vulnerability but is unexploitable

Although Solidity supports a few OO programming language principles, such as inheritance, its implementation of the OO features can differ, as explained in Sections 4.5.5 to 4.5.7.

4.5.5 Neglecting constraints caused by factory patterns (FactoryPattern)

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” [53]. Solidity supports the factory pattern, and smart contracts are the objects. A factory in Solidity is a contract (called **factory contract**) that can deploy multiple instances of other contracts (called **template contracts** in this paper) at runtime. In Listing 7, *SwapperFactory* is a factory contract that creates the template contract objects multiple times and destructs the objects by calling the function *destroy*.

```
contract SwapperFactory {//Factory contract
  function performSwap(address payable user){
     Swapper swapper = createClone(user, srcToken, dstToken
     , uniqueId);
     swapper.destroy(user); }
  function createClone( ) private onlyAdmin() returns (
     Swapper) {... ...}
}
contract Swapper {//Template contract
  function destroy(address payable user) external
  { selfdestruct(user); }
}
```

Listing 7: A code with UpS vulnerability but is unexploitable

When using the factor pattern, the *selfdestruct* in the instances created from the template contract cannot destruct the factory contract *SwapperFactory*. The vulnerability detectors report six contracts vulnerable to UpS with the factory pattern as shown in Listing 7. The *selfdestruct* in the reported vulnerable contract cannot be exploited by attackers.

4.5.6 Neglecting characteristics related to contract inheritance (ContractInheritance)

Solidity supports inheritance between smart contracts. A contract can inherit multiple contracts. The contract from which other contracts inherit is called 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 the methods, functionality, and variables of the base contract and can extend a base contract with additional functionality” [25].

If the contract can receive ether but cannot transfer ether by itself, the vulnerability detectors tag it vulnerable to FE. Fourteen contracts tagged as vulnerable to FE are base contracts containing no transfer operation. These base contracts do not have an account on Ethereum’s main network and do not own ether. Other contracts inherit these base contracts and implement ether transferring. Therefore, these base contracts will not lock the ether, meaning the FE vulnerability is unexploitable.

4.5.7 Assuming all fallback functions receive ether (FallbackFunction)

66 of the 100 contracts tagged as vulnerable to FE will never lock ether because their fallback functions cannot or refuse to accept ether. To receive ether, the fallback function must be marked *payable*, as shown in line 1 in Listing 8. 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 8. The approach to refuse ether is to insert the *revert()* into the fallback function as shown in lines 3 and 4 in Listing 8. 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. Therefore, the FE vulnerability cannot be exploited to cause the ether lock.

```
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 8: Fallback functions

Scheme 3: Smart contract application scenarios reduce exploitability.

The application scenarios of the smart contracts 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 is low [54], resulting in low exploitability of many true positive vulnerabilities. The reasons for low exploitability associated with Scheme 3 are presented in Sections 4.5.8 to 4.5.11.

4.5.8 Insufficient analysis of the values of the target contracts’ addresses (TargetContractAddress)

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. The specific target contracts’ addresses can prevent the attacker from exploiting the RE, TD, and TOD vulnerabilities.

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.

4.5.9 Omitting the case that the ether transfer initiator is the ether’s initial owner (EtherOwner)

To detect TOD vulnerability, *Oyente* and *SoliDetector* 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 *SoliDetector* label the contract as vulnerable to TD if the block *timestamp* is used as the condition to send ether. However, *Oyente*, *Slither*, and *SoliDetector* 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 9 shows an example, in which 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 in line 5. 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. As a result, the ether is returned to its initial owner.

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

Listing 9: A code with TD vulnerability but is unexploitable

4.5.10 Assuming critical operations after authorization (CriticalOperation)

One main characteristic of TO is using *tx.origin* for owner authorization in a function. If there is no critical operation following successful authentication, the TO vulnerability is not risky. Seven contracts vulnerable to TO are unexploitable because successfully bypassing the authentication will not bring security risks. The example vulnerable code is : *if(tx.origin == owner()) return;*.

4.5.11 Assuming status inconsistency when function call results are not checked (StatusInconsistency)

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 contracts vulnerable to UcC. Listing 10 shows an example of such vulnerable codes. 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 *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. The failed call in line 4 does not cause any compromise.

```
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;}
```

Listing 10: A code with UcC vulnerability but is unexploitable

4.6 Results of cross-validating manually-generated exploitability analysis results

To verify exploitation analysis results, dynamic and fuzzing vulnerability detectors are suitable because they are good at covering complex paths to check whether the exploitable smart contracts are reachable and triggerable.

To choose the detectors, we first excluded unavailable fuzzing detectors, i.e., *Reguard* [35], *ContraMaster* [6], and *Ethploit* [36]. Then, we excluded the *ContractFuzzer* [5] and *sFuzz* [4] because the empirical evaluation [9] on nine detectors [2], [3], [4], [5], [11], [12], [13], [17], [44] demonstrates that *Mythril* [11] outperforms other fuzzing detectors. In addition to using *Mythril* [11], we choose to use two latest fuzzing detectors, i.e., *ConFuzzius* [28] and *Smartian* [29] for cross-validations.

For the exploitable smart contracts resulting from our manual analysis, we run these detectors to see if the exploitable vulnerabilities are reachable and triggerable. Results are in Table 3 and show that 56.72% of the 1,079 exploitable contracts are executable by *Mythril* [11], *ConFuzzius* [28], or *Smartian* [29]. Specifically for the RE vulnerability, 59 vulnerable contracts reported by Smartian are all exploitable. Such results give support to the conclusions of our manual analysis. However, *Mythril* [11], *ConFuzzius* [28], and *Smartian* [29] cannot guarantee to explore all paths of the code. In addition, they may encounter execution errors due to missing external contract dependencies we cannot resolve. Therefore, they cannot verify all exploitable contracts we identify.

As shown in Section 4.5, our manual exploitability analyses illustrate that *Oyente* [2], *SmartCheck* [12], *Slither* [13], and *SoliDetector* [26] are weak at analyzing infeasible path, preventative execution condition, and data flow control to report false positives. *Mythril* [11], *ConFuzzius* [28], and *Smartian* use more dynamic approaches to better cover complex paths. For the vulnerabilities that are labeled as unexploitable, we apply *Mythril* [11], *ConFuzzius* [28], and *Smartian* [29] to see if they can exploit them. The results are in Table 3 and show that the three detectors can help reduce false positives related to the reasons coded to Scheme 1 when detecting UpS and IOU. For example, a contract vulnerable to UpS with an infeasible path is not reported by three detectors. However, these detectors are also weak at differentiating unexploitable vulnerabilities related to other reasons coded to Schemes 2 and 3. For instance, they all ignore the access control when detecting the UpS vulnerability. Listing 11 shows the code reported as vulnerable by *Mythril*, *ConFuzzius*, and *Smartian*. The *selfdestruct* in line 6 is labeled as vulnerable to UpS. However, an execution condition of *selfdestruct* is “tx.origin == O”. The address *O* is the creator of this contract. Only the creator of the contract can destroy the contract, and all balances of the contract will be sent to the creator. The attacker can neither destroy the contract nor get ether. Therefore, the UpS vulnerability in line 6 is unexploitable.

TABLE 3: Detection Results of *Mythril*, *ConFuzzius*, and *Smartian* on Exploitable Smart Contracts (ExSC) and Unexploitable Smart Contracts (UnExSC)

Vul.	Nr. of ExSC /UnExSC	Nr. of Reported ExSC/UnExSC Mythril	ConFuzzius	Smartian	Total	P(%)
UpS	112/25	50/7	49/3	32/1	72	64.29
TO	24/21	8/3	-	20/6	20	83.33
IOU	63/37	1/0	-	17/0	18	28.57
DC	122/802	-	-	-	-	-
UcC	41/178	15/43	21/62	33/103	37	90.24
RE	71/26	54/3	22/3	59/0	67	94.36
FE	14/86	-	1/0	0/0	1	7.14
NC	182/291	45/33	-	85/73	94	51.65
TD	250/1106	84/390	94/336	118/562	187	74.80
TOD	196/717	-	116/72	-	116	59.18
Total	1079	257	303	364	612	56.72

Note: The ”*Total*” column shows the number of unique **exploitable** smart contracts detected by *Mythril*, *ConFuzzius*, and *Smartian*. The ”-” indicates that the tool does not support detection for the vulnerability type; the ”*P(%)*” indicates the percentage of contracts detected out of all **exploitable** smart contacts.

```
1 contract GrungeTuesday{
2     address O = tx.origin;
3     function() public payable {}
4     function multi_x() public payable {
5         if (msg.value >= this.balance || tx.origin == O)
6            { selfdestruct(tx.origin);}
7     }
8 }
```

Listing 11: A code with UpS vulnerability but is unexploitable

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. The transaction log consists of several 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 of 1024.
- *stack* is an internal place where temporary variables, such as local variables, intermediate calculation results, and return addresses, are stored.
- *memory* is a temporary place to store data, of which a contract obtains a freshly cleared instance for each message call [46].

- *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*.

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 have 5,450,975 transactions, too many for us to replay and perform the ground theory analysis. We randomly selected 100 UcC contracts reported as vulnerable, which have 145,469 transactions. Of the 219 contracts, one contract had the largest number of transactions at 4,911,428, accounting for ninety percent of all transactions (5,450,975). Because we selected 100 contracts randomly, the contract that had the largest transaction number was excluded. Thus, the selected 100 UcC contracts only have 145,469 transactions. The numbers of transactions analyzed for the eight vulnerability types are as follows: UpS (33,920), TO (77,934), IOU (131,643), DC (1,683,694), UcC (145,469), RE (5,362), FE (396,127), and NC (1,631,985). We designed analysis rules as shown in Figure 5 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 [25]. 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 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 [28]. Thus, we only trace the overflow or underflow that has updated the blockchain state. If the error result flows into an 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 exactly one bigger than the 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)

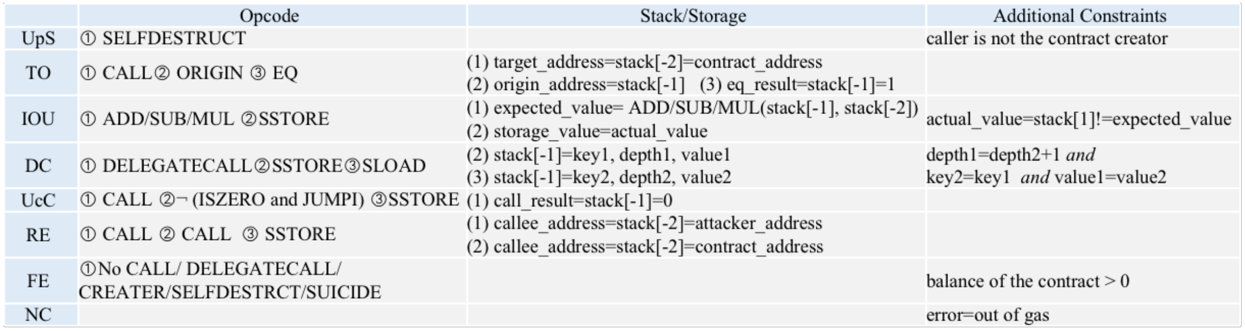
The *send()* and *call()* functions are used to send ether and are compiled into the EVM CALL instructions. The CALL instruction results are pushed onto the stack, where 0 means failure and 1 means success. The CALL result is stored in the log block where the depth of the trace turns back to the value 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,



	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 ③ SSTORE	(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. 5: Analysis rules of transaction logs

the vulnerable contracts are labeled by detectors that 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 transaction on Ethereum's main network matches our analysis rules, we count the contract as exploited. The numbers of identified exploitations are shown in Table 4. We do not find any exploitation of contracts that we labeled as unexploitable, which confirms our manual analysis in answering RQ1. For the identified exploitable contracts, their exploitations are explained below.

TABLE 4: Information about Vulnerability Exploitation

	UpS	TO	IOU	DC	UcC	RE	FE	NC	TD	TOD	Total
VC	137	45	100	924	219	97	100	473	1,356	913	4,364
UnExploitableSC	25	21	33	802	178	26	86	291	1,106	717	3,285
ExploitableSC	112	24	67	122	41	71	14	182	250	196	1,079
ExploitedSC	19	0	15	0	2	0	2	28	-	-	66

VC: The number of contracts reported as vulnerable.
UnExploitableSC: The number of contracts labeled as unexploitable.
ExploitableSC: The number of contracts labeled as exploitable.
ExploitedSC: The number of contracts that were exploited.

Unprotected Suicide (UpS). We found 19 contracts vulnerable to UpS that have been exploited, meaning 19 contracts were destructed but not by their owners. Listing 12 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 inputted by the user 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 }
```

Listing 12: A code with UpS vulnerability that was exploited

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 the transactions of a self-destructed contract, in which 0.033 ether are locked.

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

IOU. We found 15 occurrences of arithmetic overflows. Listing 13 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.

```
DungeonToken public dungeonTokenContract;
function transport() external payable {
    // ** STORAGE UPDATE **
    // Increment the accumulated rewards for the dungeon.
    dungeonTokenContract.addDungeonRewards(originDungeonId,
      requiredFee);   ......
}
contract DungeonToken {
    function addDungeonRewards(uint _id, uint _Rewards){
        dungeons[_id].rewards += uint128(_Rewards);
    }
}
```

Listing 13: A code with arithmetic overflow

DC. We did not find any exploitation of the 924 contracts reported as vulnerable to DC.

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 14. In the contract's transaction, at a certain point in time, there was insufficient ether in the contract supporting the transfer. Therefore, the log shows that a transaction calling the function *sendTokensManager* did not call the *send()* function in line 6 successfully. However, the contract, e.g., *Exxcoin*, calling the function *sendTokensManager* still changed the storage variable *balances* in line 6 after the invocation of the *send()* function failed.

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

Listing 14: A code containing failed functions

RE. We found no exploitations of the 71 contracts with RE vulnerabilities.

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.

NC. 28 out of the 182 contracts with NC vulnerabilities were exploited when executing the functions containing a *for* loop. Listing 15 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.

```
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 15: A code related to out-of-gas transactions

5.3 Results of Analyzing Vulnerability Exploitation

As shown in Table 4, only 6% (66 out of 1,079) exploitable contracts were exploited. We compared actual exploitations and unexploited contracts to determine the causes of the low exploitability rate. We categorize the reasons for unexploitation through open and axial coding. Figure 6 shows the number of unexploited exploitable contracts of each vulnerability type and the associated reasons for low exploitations.

5.3.1 Application scenarios demotivate exploitations

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

Very little or no financial benefits for attackers (LowBenefit). One of the main motivations for exploiting smart contracts' vulnerabilities is to get profit. Perez et al. [19] noted that a small number of contracts hold the majority of the ether. As shown in Figure 6, the unexploited exploitable contracts associated with the ten vulnerability types are all linked to the LowBenefit reason, meaning the contracts retain no or very little ether. For example, 70 out of 71 contracts vulnerable to RE have no ether, and the remaining one holds only 0.001307 ether.

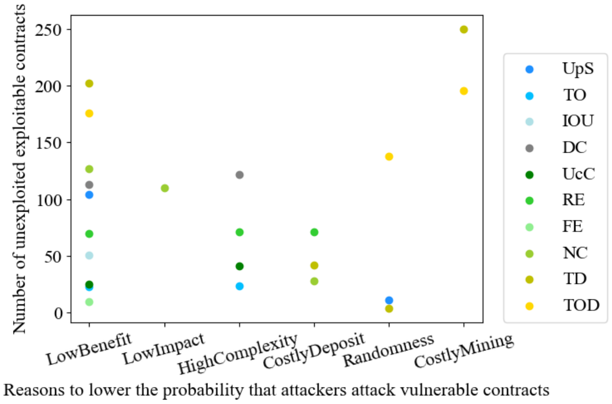


Fig. 6: The number of exploitable contracts associated with reasons to lower the probability that attackers attack

Insignificant impacts of the compromise (LowImpact). Although 28 exploitable contracts vulnerable to NC were exploited, 27 exploitations had little impact. The NC vulnerabilities were triggered by the caller's one-time input parameter, resulting in a temporary failed transaction rather than a permanent function failure. After reviewing the NC vulnerabilities in 182 exploitable contracts, we discovered the NC vulnerabilities in 110 unexploited exploitable contracts were identical to the 27 contracts, meaning the NC vulnerability exploitations would not cause permanent function failure. Therefore, Figure 6 shows that 110 unexploited exploitable contracts with NC are linked to the LowImpact reason.

5.3.2 Execution environment and cost lower the attack probability

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.

High attack complexity because attackers must develop comprehensive attack contracts (HighComplexity). Attackers must create attack contracts that are tailored to each vulnerability in order to exploit some vulnerabilities, such as RE, DC, UcC, and TO. For instance, designing an attacking contract containing a specified fallback function is vital to trigger the contracts vulnerable to RE, UcC, or TO. To exploit the contracts vulnerable to DC, the attacker needs an attack contract that controls different global variables and modifies the storage values in vulnerable contracts. Atzei et al. [1] present various attack strategies on six well-known real-world smart contracts and show that attack contracts are required to attack smart contracts. The attacker must comprehend the semantics of the target smart contract to create the attack contracts, which might have reduced the possibilities of successful attacks.

Costly attack failure because attackers must deposit ether as a prerequisite of the attack (CostlyDeposit). In 28 exploited contracts vulnerable to NC, attackers did not need to send ether into the contract, meaning they would not lose money if the attacks failed. In comparison, 28 unexploited contracts vulnerable to NC can only be exploited by calling the vulnerable contract and sending ether. Attackers did not exploit these contracts, probably due to the potential financial loss if the attacks failed. Some contracts vulnerable to TD are used for bidding, games, or wallets. For instance, 42 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. 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 16, line 9 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 10, i.e., sending more than 0.001 ether (line 3) and getting a correct *randomNumber* (line 4) to start the exploitation.

```
1  function () public payable {
2      require(msg.sender == tx.origin);
3      require(msg.value >= 0.001 ether);
4      uint256 randomNumber = uint256(keccak256(blockhash(
        block.number - 1)));
5      if (randomNumber > highScore)
6          {currentWinner = msg.sender; lastTimestamp = now;}
7  }
8  function claimWinnings() public {
9      require(now > lastTimestamp + 1 days);
10     require(msg.sender == currentWinner)
11     msg.sender.transfer(address(this).balance);
12 }
```

Listing 16: A code containing a puzzle

Attacker must be lucky in competition with randomness (Randomness). We found that setting a puzzle as a deciding condition for some critical operations is a popular defense method in our studied contracts, especially for the UpS, TD, and TOD vulnerabilities. In 19 exploited contracts due to UpS, there is no puzzle condition before the critical operation *selfdestruct*. The 11 unexploited exploitable UpS contracts, however, set a puzzle, such as ”*require(sha256(bytes(geheimnis)) == hash);*” before *selfdestruct*. An attacker cannot get permission to run critical operation *selfdestruct* or get benefits unless the attacker is lucky enough to solve the puzzle successfully by obtaining the right random number or string. Therefore, we infer that the puzzle makes attacking the vulnerability more challenging. Although the 138 contracts vulnerable to TOD also contain puzzles, we could not compare the characteristics of exploited and unexploited contracts to provide more evidence of the effectiveness of puzzles because we did not detect TD and TOD exploitation, as explained in Section 5.1.

Attacker must be a mining winner (CostlyMining). 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. Because we did not detect TD and TOD exploitation, as explained in Section 5.1, we could not compare the exploited and unexploited contracts to give evidence of the mining process's impact on vulnerability exploitation.

6 DISCUSSION

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

6.1 Implication to Vulnerability Detector Development

Results of RQ1, particularly the reasons in Schemes 2 and 3, reveal that several issues cause low exploitability. Accordingly, we provide six suggestions for improving the state-of-the-art vulnerability detectors at the theoretical level.

Suggestion 1. Enhancing the analysis of dependencies introduced by modifiers. As discussed in Scheme 2, overlooking the specific Solidity programming language characteristics will result in reporting unexploitable vulnerabilities. Unlike traditional OO languages, Solidity uses a unique language-level keyword called *modifier* to restrict other functions. The modifier may define access control policies or other business functions, leading to complicated relationships between variables, modifiers, and functions. Thus, it is vital to analyze the control and data dependencies between modifiers and other code elements to assess the feasibility of the attacking path to identify exploitable vulnerabilities.

Suggestion 2. Considering the specific characteristics and constraints of smart contract inheritances. As explained in Section 4.5.5, Solidity uses the factory contract to deploy and manage multiple template contract objects. The code in the template contract does not affect the factory contract, meaning we should distinguish between the factory and template contracts when detecting vulnerabilities. Solidity supports smart contract inheritances. Some functions are implemented in the base contracts, and others are in the derived contracts. As illustrated in Section 4.5.6, we should strengthen the analysis of the smart contract inheritance relationships to avoid overlooking functions implemented in the derived contracts.

Suggestion 3. Simulating the contract execution with real blockchain environments. Although the results in Section 4.6 show that dynamic detectors [11], [28], [29] are better at checking if codes are reachable and triggerable than the static ones, e.g., [13], [26], the state-of-the-art dynamic detectors often use EVM blockchain emulators to simulate blockchain behaviors and contract execution. The environment settings of the EVM emulators are usually insufficient to reflect the contract's actual execution status. The contract address, caller's account, and timestamp value setups on the EVM emulators can be biased. For instance, the setup may default all callers can break through the access control and hold sufficient balances. Furthermore, many complex conditions in smart contracts depend on blockchain environment parameters, such as *block.timestamp*, *block.number*, etc. Simulating the contract execution with the real blockchain environment, such as in a private chain, could help avoid reporting unexploitable vulnerabilities.

Suggestion 4. Strengthening blockchain application scenario relevance in vulnerability detection. Our findings associated with Scheme 3 point to the necessity of combining vulnerability detection with blockchain application scenarios. As presented in Sections 4.5.8 to 4.5.11, ether flow, ether's owner, function criticality, and status change will all affect the vulnerability exploitability. For example, if we detect smart contracts used as a wallet, e.g., the decentralized financial (DeFi) smart contracts, analyzing the values in the wallet and value flows would create a better assessment of the exploitation threat [55].

Suggestion 5. Adding checks of the exploitation assumptions. Some exploitations can only bring consequences if their assumptions, e.g., all fallback functions receive ether, there is a critical operation following user authorization, and there is a status change after calling the functions *send()* and *call()*, are satisfied. Missing checks of the assumptions could lead to reporting unexploitable vulnerabilities, as discussed in Sections 4.5.7, 4.5.10, and 4.5.11. Thus, beyond identifying vulnerabilities, it is vital to add exploitation assumptions checks to measure the exploitation hazards.

Suggestion 6. Ranking the reported vulnerability according to smart contract exploitability risks. 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., [9], [10], [15], [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. [15] and Ren et al. [9] collected vulnerable contracts with clear labels to evaluate different detectors. However, the number of vulnerable contracts is small, e.g., 69 contracts in [15] and 214 contracts in [9]. These vulnerable contracts are often short and have no complex business logic. Ghaleb et al. [10] constructed a dataset containing 50 contracts with 9,369 injected vulnerabilities. However, the vulnerability injection is limited to known characteristics of vulnerabilities. SolidiFi [10] 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 [9], [15], [19] construct the dataset using real-world contracts, the type and amount of vulnerabilities in these contracts are unknown. This study collected unique 4,364 real-world smart contracts which are cross-labeled by at least two tools as vulnerable. In addition, we categorized the contracts into exploitable and unexploitable. Thus, our dataset can be used as a novel benchmark to evaluate the vulnerability detectors' capability to identify exploitable smart contracts.

6.3 Comparison with Related Work

Durieux et al. [15] hypothesize that the detectors they evaluate report a considerable number of false positives because the percent of vulnerable contracts (44,589/47,518, 93%) is high. Perez and Livshits [19] also hypothesized that most reported vulnerabilities are either false positives or unexploitable. They identified one factor affecting the actual exploitation of smart contracts, e.g., ether distribution. However, no follow-up study tried to identify and understand other possible reasons for the low exploitation rate.

Different from [19], we uncover that vulnerabilities reported by the detectors are possibly unexploitable, and there are 11 reasons for that. These reasons are not limited to the three reasons related to faulty vulnerability detector implementation resulting in false positives. The other eight reasons provide new insights that overlooking the characteristics of the Solidity programming language and smart contract application scenarios causes the state-of-the-art detectors to report many unexploitable vulnerabilities. The identified reasons can guide researchers to improve the detectors to identify vulnerabilities more accurately.

For the exploitable vulnerabilities, beyond ether distribution mentioned in [19] that may demotivate exploitations, our results of RQ2 uncover five other reasons, which are related to financial aspects and blockchain mechanisms, that might have also helped lower the attackers' possibility to compromise the exploitable smart contracts. Some of the reasons, e.g., setting puzzles, can possibly help guide developers to mitigate the security risks of smart contracts.

6.4 Threats to Validity

In this study, we only focus on the types of vulnerabilities covered by at least two fast vulnerability detectors to avoid bias caused by a single tool. This filtering excluded several types of vulnerabilities that may have different reasons for the low exploitability than those we identified. We found that more than one reason caused the low exploitability of a contract. We give only one reason for 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 11 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 5. For RQ2, another possible reason for not exploiting the exploitable contracts is that the attackers are unaware of the vulnerabilities. However, we do not have data to analyze such a reason.

When manually analyzing the source code of the vulnerable smart contract to label their exploitability, a possible risk is mislabelling. However, we believe that such a risk is low because there are often easy-to-distinguish code features associated with the reasons for the low exploitability, as shown below:

Solidity access control: There is the modifier, i.e., *onlyOwner, onlyAdmin, onlyManager* or *if/require* statement.

Constraints caused by factory patterns: The contracts using the factory pattern use the same template in Listing 7.

Characteristics related to contract inheritance: The contract inheritance is declared with the contract name.

***fallback* functions refuse ether**: If the *revert* statement is in the *fallback* function, the contract refuses to receive ether.

Ether transfer initiator: The amount of ether is *msg.value* that comes from the caller.

Critical operation: Critical operations include function calls or assignment statements, etc.

Status inconsistency: There is a statement after the *call* function that may cause inconsistency between the transfer and the state variable change.

Target contract's address: The address is hard-coded, fixed, or under the control of the contract owner.

7 CONCLUSION AND FUTURE WORK

As smart contracts' security is critical, many vulnerability detectors have been proposed. Several empirical studies show that the detectors report many vulnerabilities and the exploitation rate is low, and, therefore, hypothesize many reported vulnerabilities are either false positives or unexploitable. This study analyzed the exploitability of 4,364 unique real-world smart contracts reported as vulnerable by multiple detectors. We identified 11 reasons causing low exploitability and six aspects that may have lowered the possibility of exploiting exploitable contracts and provided six suggestions for improving vulnerability detection approaches. This paper gives smart contract security researchers a moment of introspection. They can take a step back and review the vulnerability detection techniques, study the security risks associated with smart contracts, examine the shortcomings of the current detectors, decide how to more effectively improve the traditional approaches to detect smart contract vulnerabilities, and then plan to put those improvements into practice.

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. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 03, pp. 1795–1809, may 2022.

[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] 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

[10] 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.

[11] (2018) Mythril: an open-source security analysis tool for ethereum smart contracts. [Online]. Available: https://github.com/ConsenSys/mythril

[12] 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.

[13] 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.

[14] (2019) Manticore: a symbolic execution tool for analysis of smart contracts and binaries. [Online]. Available: https://github.com/trailofbits/manticore

[15] 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

[16] 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.

[17] 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

[18] 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.

[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] (2017) Vulnerability vs. exploitability: Why they're different. [Online]. Available: https://cloudtweaks.com/2017/07/vulnerability-vs-exploitability/

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

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

[26] T. Hu, B. Li, Z. Pan, and C. Qian, "Detect defects of solidity smart contract based on the knowledge graph," *IEEE Transactions on Reliability*, pp. 1–17, 2023.

[27] 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.

[28] 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.

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

Conference on Automated Software Engineering (ASE), 2021, pp. 227–239.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] (2013) World wide web consortium, sparql 1.1 update. [Online]. Available: https://www.w3.org/TR/sparql11-update

[38] 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.

[39] (2021) Wikipedia, datalog: a declarative logic programming language. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Datalog oldid=1053711548

[40] 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.

[41] 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.

[42] 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

[43] (2019) Ethereum (eth) blockchain explorer. [Online]. Available: https://etherscan.io

[44] 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.

[45] (2022) Jaccard index: a statistic used for gauging the similarity and diversity of sample sets. [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index

[46] "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.

[47] 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.

[48] (2020) Delegatecall to untrusted callee. [Online]. Available: https://swcregistry.io/docs/SWC-112

[49] 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.

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

[51] 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.

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

[53] 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.

[54] Y. Xue, J. Ye, W. Zhang, J. Sun, L. Ma, H. Wang, and J. Zhao, "xfuzz: Machine learning guided cross-contract fuzzing," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–14, 2022.

[55] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, and X. Luo, "Security threat mitigation for smart contracts: A comprehensive survey," *ACM Comput. Surv.*, apr 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3593293

Tianyuan Hu is currently working toward the Ph.D. degree at 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.

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.

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.

APPENDIX

TABLE 5: Empirical Studies of Vulnerability Detectors

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

TABLE 6: Mapping of the Different Vulnerabilities Analyzed

Vul.	Oyente	SmartCheck	Slither	SoliDetector
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

TABLE 7: Reasons for Excluding Vulnerability Detectors

Year and ref.	Tool Name	SCI	SV	VL	Availability	HighFN-Risk
Pattern Matching						
2018 [12]	Smartcheck					
2021 [26]	SoliDetector					
Symbolic Execution						
2016 [2]	Oyente					
2018 [20]	ZEUS				•	
2018 [17]	Osiris	•				
2018 [11]	Mythril					•
2018 [3]	Securify		•			
2018 [21]	TEETHER	•				
2018 [16]	MAIAN			•		
2019 [18]	HONEYBADGER	•				
2021 [7]	DefectChecker		•			
2022 [8]	EXGEN				•	
Data Flow Analysis						
2018 [22]	MadMax	•				
2019 [13]	Slither					
2020 [30]	Clairvoyance				•	
2020 [31]	Ethainter	•				
Machine Learning						
2019 [32]	GNN-based			•		
2020 [33]	ContractWard			•		
2021 [34]	VSCL	•		•		
Fuzzing						
2018 [5]	ContractFuzzer					•
2018 [35]	Reguard				•	
2020 [4]	sFuzz					•
2020 [36]	Ethploit				•	
2021 [28]	ConFuzzius					•
2021 [29]	Smartian					•
2022 [6]	ContraMaster				•	

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

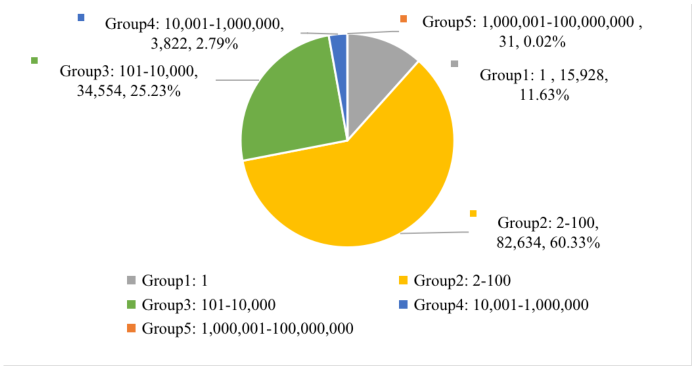
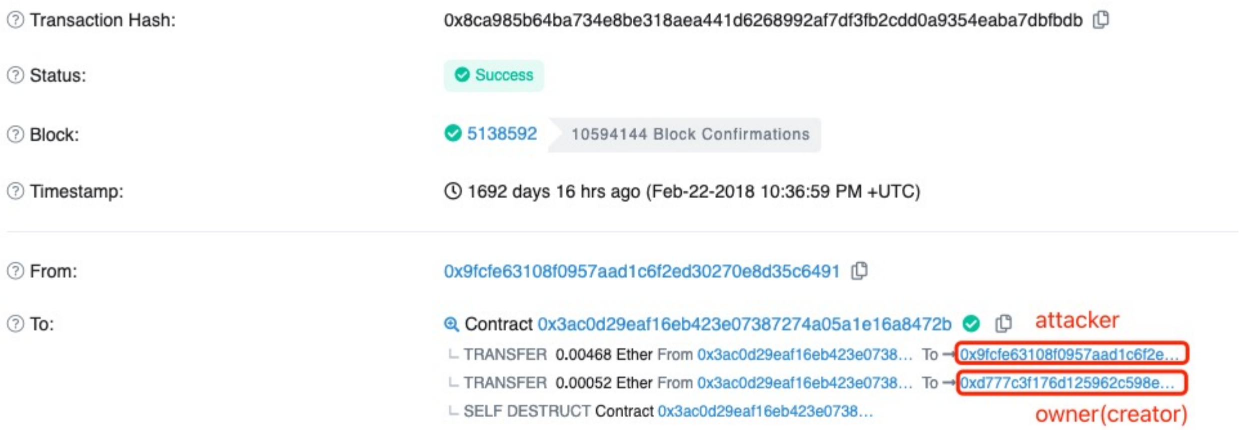


Fig. 7: Transactions information of the chosen unique smart contracts



Transaction Hash: 0x8ca985b64ba734e8be318aea441d6268992af7df3fb2cdd0a9354eaba7dbfbdb

Status: Success

Block: 5138592 10594144 Block Confirmations

Timestamp: 1692 days 16 hrs ago (Feb-22-2018 10:36:59 PM +UTC)

From: 0x9fcfe63108f0957aad1c6f2ed30270e8d35c6491

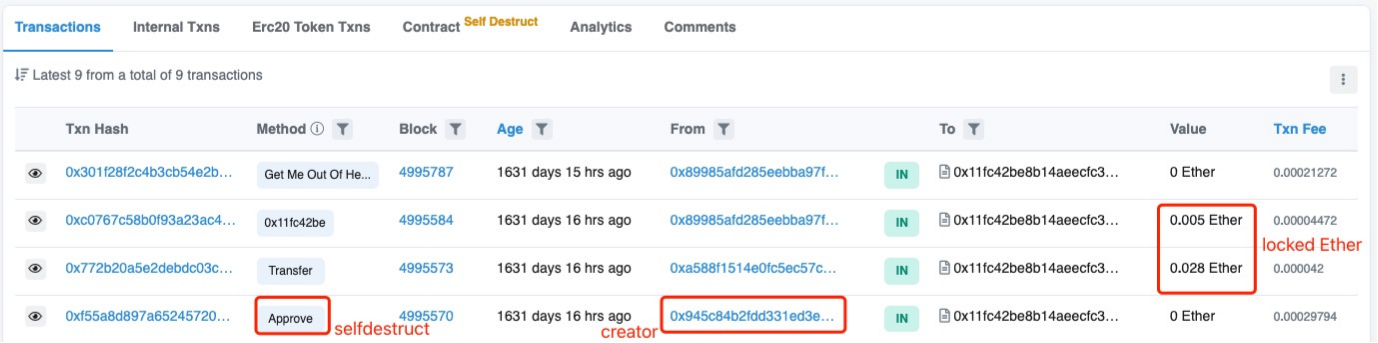
To: Contract 0x3ac0d29eaf16eb423e07387274a05a1e16a8472b

- TRANSFER 0.00468 Ether From 0x3ac0d29eaf16eb423e0738… To → 0x9fcfe63108f0957aad1c6f2e…
- TRANSFER 0.00052 Ether From 0x3ac0d29eaf16eb423e0738… To → 0xd777c3f176d125962c598e…
- SELF DESTRUCT Contract 0x3ac0d29eaf16eb423e0738…

attacker

owner(creator)

Fig. 8: The transaction details of the exploited UpS



Latest 9 from a total of 9 transactions

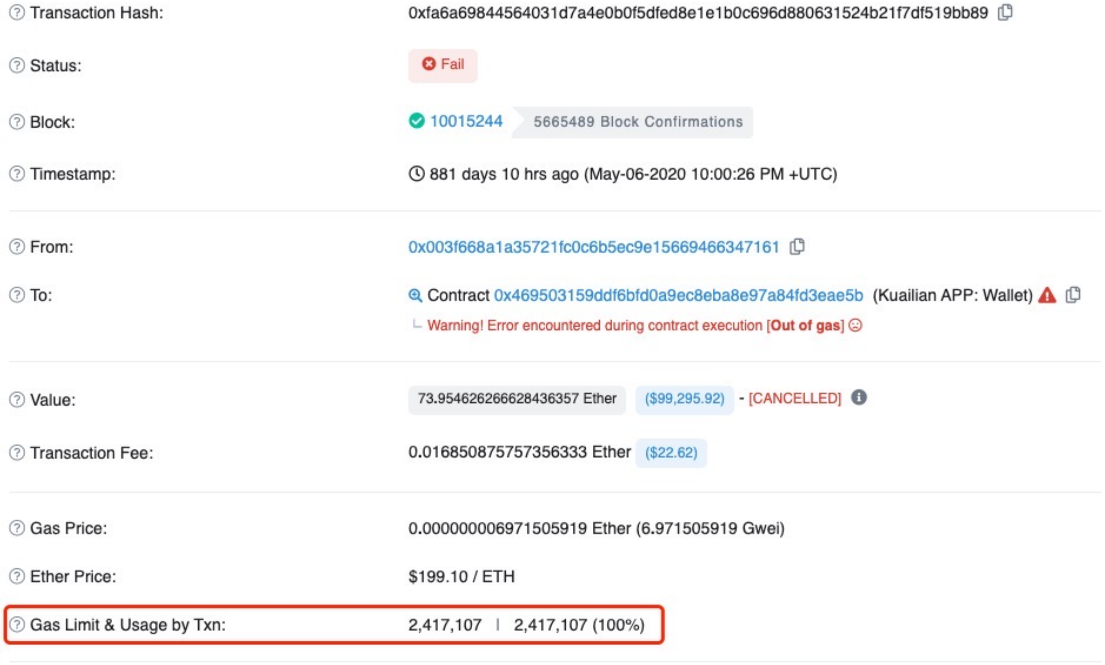
	Txn Hash	Method	Block	Age	From		To	Value	Txn Fee
	0x301f28f2c4b3cb54e2b…	Get Me Out Of He...	4995787	1631 days 15 hrs ago	0x89985afd285eebba97f…	IN	0x11fc42be8b14aeecfc3…	0 Ether	0.00021272
	0xc0767c58b0f93a23ac4…	0x11fc42be	4995584	1631 days 16 hrs ago	0x89985afd285eebba97f…	IN	0x11fc42be8b14aeecfc3…	0.005 Ether	0.00004472
	0x772b20a5e2debdc03c…	Transfer	4995573	1631 days 16 hrs ago	0xa588f1514e0fc5ec57c…	IN	0x11fc42be8b14aeecfc3…	0.028 Ether	0.000042
	0xf55a8d897a65245720…	Approve	4995570	1631 days 16 hrs ago	0x945c84b2fdd331ed3e…	IN	0x11fc42be8b14aeecfc3…	0 Ether	0.00029794

locked Ether

selfdestruct

creator

Fig. 9: Locked ether in a self-destructed contract



Transaction Hash:	0xfa6a69844564031d7a4e0b0f5dfed8e1e1b0c696d880631524b21f7df519bb89
Status:	Fail
Block:	10015244 5665489 Block Confirmations
Timestamp:	881 days 10 hrs ago (May-06-2020 10:00:26 PM +UTC)
From:	0x003f668a1a35721fc0c6b5ec9e15669466347161
To:	Contract 0x469503159ddf6bfd0a9ec8eba8e97a84fd3eae5b (Kuailian APP: Wallet) Warning! Error encountered during contract execution [**Out of gas**]
Value:	73.954626266628436357 Ether ($99,295.92) - [CANCELLED]
Transaction Fee:	0.016850875757356333 Ether ($22.62)
Gas Price:	0.000000006971505919 Ether (6.971505919 Gwei)
Ether Price:	$199.10 / ETH
Gas Limit & Usage by Txn:	2,417,107 | 2,417,107 (100%)

Fig. 10: Transaction information on Etherscan