Empirical Evaluation of Smart Contract Testing: What Is the Best Choice?

Meng Ren
Tsinghua University
Beijing, China
rm19@mails.tsinghua.edu.cn

Zijing Yin
Tsinghua University
Beijing, China
Aurora@europe.com

Fuchen Ma
Tsinghua University
Beijing, China
mafc19@mails.tsinghua.edu.cn

Zhenyang Xu
University of Waterloo
Waterloo, Canada
zhenyang.xu@uwaterloo.ca

Yu Jiang*
Tsinghua University
Beijing, China
jiangyu198964@126.com

Chengnian Sun
University of Waterloo
Waterloo, Canada
cnsun@uwaterloo.ca

Huizhong Li
WeBank
Shenzhen, China
wheatli@webank.com

Yan Cai
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences

ABSTRACT

Security of smart contracts has attracted increasing attention in recent years. Many researchers have devoted themselves to devising testing tools for vulnerability detection. Each published tool has demonstrated its effectiveness through a series of evaluations on their own experimental scenarios. However, the inconsistency of evaluation settings such as different data sets or performance metrics, may result in biased conclusion.

In this paper, based on an empirical evaluation of widely used smart contract testing tools, we propose a unified standard to eliminate the bias in the assessment process. First, we collect 46,186 source-available smart contracts from four influential organizations. This comprehensive dataset is open to the public and involves different code characteristics, vulnerability patterns and application scenarios. Then we propose a 4-step evaluation process and summarize the difference among relevant work in these steps. We use nine representative tools to carry out extensive experiments. The results demonstrate that different choices of experimental settings could significantly affect tool performance and lead to misleading or even opposite conclusions. Finally, we generalize some problems of existing testing tools, and propose some possible directions for further improvement.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

*Yu Jiang is the corresponding author.

KEYWORDS

smart contract testing, evaluation, observations and solutions

ACM Reference Format:
Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. 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 (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464837

1 INTRODUCTION

Smart contracts are self-executing agreements that run on a blockchain. It permits trusted transactions to be carried out among disparate, anonymous parties. The original intention of using them is to realize complete decentralization, resolving trust problems through established rules and automated scripts. However, due to the lack of logical rigor and the defects of the underlying execution mechanism, various forms of vulnerabilities are easily hidden in smart contracts. These vulnerabilities are hard to notice but can be maliciously exploited by hackers.

To ensure the security of funds and privacy of all transaction participants, there has been a proliferation of work focusing on detecting vulnerabilities in smart contracts. Because of the different experiment settings and measure metrics, the performance of the same tool varies significantly across papers, and its real detection ability is difficult to judge. Similar to traditional software testing, in order to persuasively demonstrate that a new tool provides an advantage over existing work, authors need to follow a standard evaluation process [28], which generally consists of four parts:

- collecting target programs for testing – *benchmark suite*;
- choosing one or more representative tools to compare against – *baseline*;
- allocating identical and reasonable values for custom parameters before execution – *runtime parameters*;
- defining scientific and comprehensive metrics to measure each tool's performance – *performance metrics*.

The difference in any of these steps may lead to incomplete conclusions. Taking Slither [16] and Smartcheck [51] as an example, the authors of Slither said Smartcheck had a high false positive rate of 73.6% while theirs was only 10%. However, in the large experiment carried out by Asem Ghaleb [17], 90% of Slither's alarms were false warnings. On the contrary, Smartcheck has reported no false alarms. The main reason for this divergence is that the test suites used for evaluation are different.

We examine 27 recently published papers that aim at smart contract testing and have been accepted or cited in top-conference proceedings (see Table 1). Unfortunately, according to their descriptions in the paper, they did not share a unified setting within the four steps. Using nine representative tools, we perform a variety of tests from three aspects: test suites, runtime parameters and performance metrics. The results show that individual evaluations which treated these factors casually could easily lead to biased conclusions. The main reasons are as follows.

The performance of tools is closely related to the target test suite. More than $\frac{3}{4}$ of the examined papers only use a single category of dataset for evaluation, ignoring the biases caused by the distinctive vulnerability patterns. For example, in real-world contracts, the only form of expression of reentrancy is *call.value* operation without gas limit. But in contracts with artificially injected vulnerabilities, the form is more diverse, such as a *call.value* operation with an exorbitant gas limit. In this case, it is difficult to draw universal conclusions. Once the test suite is replaced, the result may be completely opposite. For instance, when using real-world contracts for testing, Slither could report 56 unique bugs, where only 18% of them were real. When switching to artificially constructed contracts, the precision rate increased to 78%.

The runtime parameters also have a significant influence on tools' performance, especially for fuzzers and concolic executors. However, nearly half of the dynamic tools even ignore to mention the setting of these parameters at all. During symbolic execution, parameters such as maximum recursion depth will affect the number of blocks that can be executed and the final execution state. For example, with a depth limit of 30, Mythril wins the first place with discovery of 94 real vulnerabilities. But when the depth limit is extended to 100, Mythril only finds 43 real vulnerabilities, which is less than half of Oyente's or Osiris's. Similarly, because of the randomness of fuzzing, parameters such as number of trials and execution duration also have a critical effect on fuzzing performance. A prime example is ContractFuzzer. At the best time, its precision rate can reach 100%; but at the worst time, both of its precision and recall are zero.

Different choices of performance metrics will affect the conclusion. In general, developers are most concerned about a tool's ability in finding "real" problems. Almost all of the examined papers have proved their effectiveness through the total number of reported bugs, precision rates or recall rates. But nearly half (13 out of 27) of them only assess in parts of metrics. There are also 11 papers use different forms of coverage to illustrate testing effectiveness. This is an intuitive metric, but in fact, there is no inevitable relationship between the raise of coverage and improvement of vulnerability detection performance. In sFuzz's coverage curve, only one out of 22 rises results in the discovery of new vulnerabilities. In addition, although its final coverage is 10% higher than ILF, it still lags behind ILF in practical terms with two missed vulnerabilities. Consequently, if we only use coverage to measure tool performance, we may get wrong conclusions contrary to the reality.

In this paper, we propose a series of standards to assist in conducting fair experimental evaluations. In order to comprehensively review each tool's ability in detecting different forms of vulnerabilities, we collect 46,186 smart contracts from four influential organizations. It not only contains real-world contracts used in large-scale distributed projects, but also includes labeled contracts that have been officially confirmed by experts, and artificially constructed contracts with injected vulnerabilities of specific patterns. Using this unified benchmark suite, we can examine the vulnerability detection ability of each tool in different scenarios, and eliminate the bias caused by the test suites. Following the idea of variable-controlling approach, we verify the impact of different runtime parameters on tool performance. Then, we demonstrate that depth, trials and timeout could have a significant influence on the number of discovered vulnerabilities. As to performance metrics, the fundamental purpose of testing tools is to find real and harmful vulnerabilities. In this work, we combine unique bugs, precision and recall as the evaluation metrics, and evaluated each tool comprehensively. We also prove that these metrics were complementary to each other. It is not accurate to show that a tool performs better than the previous work only through a part of the metrics.

Contributions We mainly make the following contributions:

- We construct a unified benchmark suite for smart contract security testing, which integrates unlabeled real-world contracts, artificially constructed contracts, and confirmed vulnerable contracts. It is available to the public[1].
- We conduct in-depth analysis from three aspects and propose a principled way to make evaluation for smart contract vulnerability detection tools.
- We carry out extensive experiments based on nine well-known smart contract testing tools and demonstrated the impact of improper settings on tool performance.

2 BACKGROUND

2.1 Smart Contracts

Smart contracts are self-executing, business automation applications that run on a decentralized network such as blockchain. Usually, they are a set of digital agreements with specific rules, and can be enforced without the involvement of a third party. Multiple users in the blockchain jointly participate in the formulation of smart contracts. Once deployed, the smart contract will be uploaded to the blockchain network and broadcast to all verification nodes.

However, as some researchers have pointed out [8, 14, 44, 47], due to the characteristics of EVM, writing secure smart contracts is far from trivial. Coupled with the restriction that a contract cannot be modified after deployment, any vulnerability hidden in it can cause huge financial losses once exploited.

Fig. 1 is a simplified version of DAO contract, which belongs to a popular decentralized investment fund project and was attacked in 2016. It uses a mapping named *balances* to store fund-raising of

[1] https://github.com/renardbebe/Smart-Contract-Benchmark-Suites

```
contract BasicDAO {
  mapping (address => uint) public balances;

  // transfer the entire balance of caller
  function withdrawBalance() public {
    uint money = balances[msg.sender];
    bool result = msg.sender.call.value(money)();
    if (!result) { throw; }
    // update balance of withdrawer
    balances[msg.sender] = 0;
  }
}

contract Attacker {
  address public owner;

  // Initiates the balance withdrawal.
  function callWithdraw(address addr) public {
    BasicDAO(addr).withdrawBalance();
  }

  // Fallback function for this contract.
  function () public payable {
    callWithdraw(msg.sender);
  }
}
```

Figure 1: A simplified DAO contract with reentrancy vulnerability inside *withdrawBalance* function and the attack contract for exploiting reentrancy in BasicDAO.

each project. When a funding project have received enough support from the community, the owner can call *withdrawBalance* function to withdraw ether from the DAO. Unfortunately, the transfer mechanism in DAO allowed to transfer the ether to the external address before updating the internal state. This gave the attacker a recipe for stealing more ether from the contract via reentrancy. The attacker could first ask for a legitimate withdrawal, and the *call.value* method inside *withdrawBalance* function would trigger caller's fallback function. Then another withdrawal happened. Since the stored value in balances array had not been updated, the contract would keep transferring ether to the attacker until the balance of contract became zero. This attack caused a loss of $60 million and also led to the hard fork of Ethereum, which brought many negative effects.

2.2 Testing Technologies and Tools

For ensuring the security of funds and privacy of all participants, a proliferation of work devoted themselves in detecting vulnerabilities hidden in smart contracts. Performing a literature review, we filtered 27 papers published between 2016 and 2020, which apply classic software testing technologies for smart contract security analysis. Table 1 lists them in chronological order. In this section, we will briefly summarize the principles of these tools, organized by the detection method they are mainly based on. Eventually, we will focus on how they *evaluate* their proposed methods, which will be discussed in detail in the next section.

Static Analysis. Static analysis refers to a method of analyzing the program without actually running it, which can be performed at both source code and bytecode level. Static analysis tools can scan the entire code base but also produce much false positives. SolidityCheck [61] retrieves each function at the source code level and checks the piece that matches the pre-defined vulnerability patterns. Normally, other tools will first obtain binary bytecode, and then use it to construct a custom intermediate representation, which have a series of forms, like SSA used in Slither [16], Datalog used in Securify [55] and MadMax [21], XML parsing tree used in SmartCheck [51] and XCFG used in ClairvOyance [60]. Based on this representation, vulnerability pattern definition and matching are performed to screen out suspected code snippets. As a static analysis method based on mathematical proof, formal verification is also widely used to verify the logical integrity of smart contracts, such as EtherTrust [22], VeriSmart [49] and Zeus [27]. Moreover, static analysis can also be used for feature extraction, which can be further used in training classifiers [31, 58].

Symbolic Execution. When using symbolic execution to analyze a program, it will use symbolic values as input instead of the specific values during the execution. When a fork is reached, the analyzer will collect the corresponding path constraints, and then use a constraint solver to obtain specific values that can trigger each branch. Symbolic execution can simultaneously explore multiple paths that the program can take under different inputs, but it also faces unavoidable problems such as path explosion. In most cases, the symbolic executor will first build a control flow graph based on Ethereum bytecode, then design corresponding constraints based on the characteristics of smart contract vulnerabilities, and finally use the constraint solver to generate satisfying test cases. For example, Oyente [32], Mythril [11] and their extensions [40, 53, 54, 62]. In recent years, there has been continuous research to optimize the process of symbolic execution. Manticore [37] adds the support of exotic execution environments, DefectChecker [7] extracts defect related features to help improve efficiency, sCompile [6] identifies critical paths which involve monetary transaction and VerX [45] focuses on verifying effectively external callback free contracts.

Dynamic Fuzzing. Fuzzing is a method of discovering software failures by constructing unexpected input data and monitoring the abnormal results of the target program during execution. It allows developers to ensure a uniform standard of quality through prepared tests, but does not narrow down the causes of detected bugs. When applied to smart contracts, a fuzzing engine will first try to generate initial seeds to form executable transactions. With reference to the feedback of test results, it will dynamically adjust the generated data to explore as much smart contract state space as possible. Finally, it will analyze the status of each transaction based on the finite state machine to detect whether there is an attackable threat. ContractFuzzer [26] is the first to apply fuzz testing to smart contracts. Later, other researchers start to study improvements to different parts of fuzzing. ReGuard [30] and Harvey [59] are dedicated to generating diverse inputs and transactions that are more likely to reveal vulnerabilities, ILF [23] and sFuzz [39] target at designing more effective generation or mutation strategy.

2.3 Related Work

In recent years, there have been many empirical surveys focus on smart contract test suites and test methods. Some of them summarized the traditional test methods, focusing on the limitations of the test method itself, such as the instability of fuzzing [29, 34, 57] and the state explosion problem of symbolic execution [2, 5]. Others were dedicated to constructing benchmarks that can be used for evaluation through web crawling [14], bug injection [17], patch enhancement [25], etc. They will also run some vulnerability detection tools to conduct preliminary experiments to demonstrate the quality of their benchmarks.

Different from above work, we strive to systematically, comprehensively analyze different factors that can bias the evaluations of smart contract testing tools, and have proposed a recommended evaluation methodology to control biases in future evaluations of similar tools. Different from empirical studies of fuzz testing and symbolic executions, we focus on the neglected details when these techniques are applied to smart contracts. More importantly, most of our findings and the solutions to eliminate biases are specific to smart contracts, such as the recommended setting range of search depth and the combination of multi-dimensional metrics. Different from the work which proposed benchmarks, we not only demonstrate that besides benchmarks there exist many other factors that can introduce biases, but also propose a bias-less evaluation methodology. Take SmartBugs [14] as representation, it provides two datasets and only discusses the consistency of the execution results, without analyzing the execution process or proposing a better evaluation methodology.

3 OVERVIEW AND EXPERIMENTAL SETUP

Smart contract security has been widely discussed in recent years, and related research papers have been continuously published in high-quality journals. We have investigated 5 highly cited comprehensive surveys [10, 14, 17, 24, 43], and chased citations to and from them to collect papers or tools related to smart contract testing. In all, there are 46 papers falling in this field, but only 27 of them focus on vulnerability detection. Others aimed at reverse engineering [50, 63], interpretation and visualization [4, 35, 41, 42, 48, 56], gas optimization [9, 33], etc., which are beyond the scope of this paper.

Our interest in this work is assessing the experimental evaluation process of smart contract testing tools. As mentioned before, to compare with previous work, the standard evaluation process should consist of four steps. Table 1 summarizes the implementation details of each work in these steps. For each tool, it indicates what kind of detection *method* was used; whether the tool is *available* to the public; which dataset was chosen as the *benchmark* suite; which tools were selected as the *baseline*; the *depth* limit set for each contract during execution; the number of *trials* performed per configuration; what *timeout* was used per trial; the size of *samples* used for bug verification; which indicators were used to group related *bugs* and what kind of code *coverage* was measured to judge the tool performance.

During the evaluation process, all the work on the list showed that they were superior to selected baselines or previous work. However, with the emergence of a large number of empirical studies on smart contract testing tools, the conclusions in some papers seem to be biased, or even completely opposite to the actual situation. The reason for this contradiction is the lack of unified benchmark suite, scientific evaluation process and general performance metrics. In this paper, we will demonstrate the importance of these factors through extensive experiments and put forward suggestions for follow-up research.

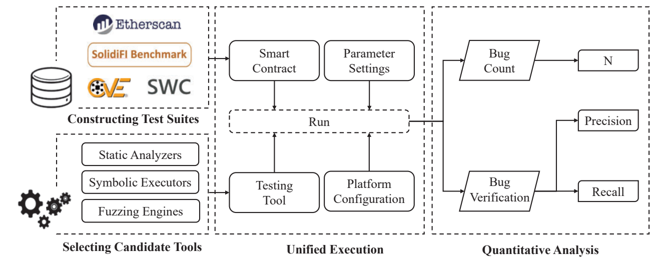


Figure 2: Overall experimental evaluation process.

The following part of this section will introduce the setup and procedure for our experiments. The overall process is described in Fig. 2. First, we build unified test suites and select candidate testing tools. Then, we filter and set runtime parameters based on the type of tools. Next, in a unified execution environment, each tool starts to analyze the contract and generates bug reports. Finally, we count the total number of alarms, manually confirm the authenticity of each of them, screen out the missed vulnerabilities and calculate the value of relevant metrics.

3.1 Constructing Test Suites

Due to the lack of standard benchmarks, such as LAVA-M [13] and Google fuzzer test suite [18] used in C/C++ programs, researchers have to crawl real-world smart contracts from Ethereum network, or use some vulnerable contracts posted on GitHub for experiment. This method mainly has two disadvantages: 1) Ethereum does not provide a direct interface access to all contracts, repetitive crawling wastes a lot of manpower and time. Meanwhile, the type and amount of vulnerabilities that may be contained in these contracts are unknown. Without clear labels, it is very difficult to determine the authenticity of detection results, and impossible to obtain the missing reports; 2) The vulnerable contracts fetched from GitHub have clear labels provided by experts, but they are often short and have no complex business logic.

Table 2: Statistical analysis of smart contract benchmarks. UR represents dataset of unlabeled real-world contracts; MI represents dataset of contracts with manually injected bugs; CV represents dataset of confirmed vulnerable contracts.

Category	# Contracts	# Bugs	# Bug Types	# Reentrancy
UR	45622	-	-	10
MI	350	$\geq$ 9369	7	119
CV	214	$\geq$ 214	35	10

To make up for the lack of a unified test set, we construct a benchmark suite with contracts crawled from Etherscan [15], SolidiFI repository [17], Common Vulnerabilities and Exposures library [36] and Smart Contract Weakness Classification and Test Cases library [38]. They can be classified into three categories: 1) unlabeled real-world contracts; 2) contracts with manually injected bugs; 3) confirmed vulnerable contracts, as shown in Table 2. We believe that the evaluation results will generalize due to the size and diversity of our benchmarks. For further analysis, with the assistance of two experts in the field of smart contract security, we expose and confirm the reentrant code pieces hidden in each contract in

Table 1: Summary of published smart contract testing tools' evaluations. Blank cell means the paper didn't mention this item; - means that this item is irrelevant; ? means that the paper mentioned this item, but there is no accurate description. Method: SA means static analysis, DF means fuzzing, SE means symbolic execution, DL means deep learning. Available: ✓ means the tool is open-source and freely available, ✗ means closed-source. Benchmark: E means Etherscan, B means blockchain (only contains bytecode), U means user-written contracts, C means Common Vulnerabilities and Exposures database. The number in brackets means the number of contracts used in evaluation. Baseline: O means Oyente, M means Mythril, S means Securify, H means Harvey, I means ILF, SL means Slither, SC means SmartCheck, SF means sFuzz, MA means MAIAN, MT means Manticore, CF means ContractFuzzer, RG means ReGuard, OS means Osiris, Z means Zeus. Depth: Maximum search depth. Trials: Number of runs. Timeout: Time reported in seconds (S), minutes (M) or hours (H). Samples: Number of samples used for manual verification. 10 means less than 10 samples, 100 means 10-100 samples, 500 means 100-500 samples, 1000 means 500-1000 samples, >1000 means more than 1000 samples. Bugs: Indicators for vulnerability confirmation. N means total number of alarms, TP means true positives, FP means false positives, FN means false negatives, TN means true negatives. Coverage: I, P, BL, BR respectively means instruction/path/basic-block/branch coverage.

tool	method	available	benchmark	baseline	depth	trials	timeout	samples	bugs	coverage
ClairvOyance[60]	SA	✓	E(17700)	O, S, SL, M, SF	-	-		>1000	N, TP, FP, FN	
CONFUZZIUS[52]	DF	✗	E(27)	H, I, O, M, MT, OS	-	24	1H	100	N, TP, FP	I
ContractFuzzer[26]	DF	✓	E(6991)	O	-	1	80H	500	N, TP, FP, FN	
ContractWard[58]	SA, DL	✗	E(14850)	O, S	-	-		>1000	N, TP, FP, FN, TN	
DefectChecker[7]	SE	✗	E(587), B(165621)	O, M, S		10		500	N, TP, FP, FN, TN	I
EtherTrust[22]	SA	✗		O	-	-				
Harvey[59]	DF	✗	E(27)		-	24	1H	100	N	P, I
HoneyBadger[54]	SE	✓	E(151935)		50		30M	500	N, TP, FP	P
ILF[23]	DF, DL	✓	E(18496)	MA, CF	-		3H	>1000	N, TP, FP	I
MadMax[21]	SA	✓	E(91800)		-	-	20S	100	N, TP, FP	
MAIAN[40]	SE	✓	B(970898)		3		300S	>1000	N, TP, FP	
Manticore[37]	SE	✓	B(100)				90M			P
Mythril[11]	SA, SE	✓			22		24H			
Osiris[53]	SE	✓	E(1383), B(1207335)	Z	50		30M	500	N, TP, FP	P
Oyente[32]	SE	✓	B(19366)		50		30M	500	N, TP, FP	P
ReGuard[30]	DF	✗	E(5)	O	-	?	20M	10	N, FP, FN	
SASC[62]	SA, SE	✗	E(2952)	O			10M		N	P, BL, I
sCompile[6]	SE	✗	E(36099)	O, MA	3		1M	>1000	N, TP, FP	P
Securify[55]	SA	✓	E(24594), U(100)	O, M	-	-		100	N, TP, FP, FN	
sFuzz[39]	DF	✓	E(4112)	O, CF	-	3	2M	500	N, TP, FP	BR
S-gram[31]	SA, DL	✗	E(1500)	RG	-	-			N, TP, FP	
Slither[16]	SA	✓	E(1000)	S, SC	-	-	2M	1000	N, TP, FP	
SmartCheck[51]	SA	✓	E(4603)	O, S, M	-	-		10	N, TP, FP, FN	
SolidityCheck[61]	SA	✓	E(1363)	M, O, S, SC, CF, OS	-	-		>1000	N, TP, FP, FN	
VeriSmart[49]	SA	✓	C(60), E(25)	O, M, OS, MT, Z	-	-	30M	100	N, TP, FP, FN, TN	
VerX[45]	SE	✓	E(138)	O, M, MT	5		1H	100	N, TP, FP, FN	
Zeus[27]	SA	✗	E(1524)	O	-	-	1M	>1000	N, TP, FP, FN	

advance. The number of labeled reentrancy vulnerabilities is 10, 119 and 10 respectively.

UR dataset collection and processing. The first category is collected from Etherscan, which is the largest decentralized platform for Ethereum smart contracts. All contracts stored on Etherscan are indexed by unique addresses, so we first retrieve the addresses of all contracts that have more than one transaction through Google BigQuery [12]. Using the following query request, we obtain 1,511,925 distinct contract addresses:

```
SELECT contracts.address AS tx_count
FROM `bigquery-public-data.ethereum_block-chain.
     contracts` AS contracts
JOIN `bigquery-public-data.ethereum_block-chain.
     transactions` AS transactions ON (transactions.
     to_address = contracts.address)
GROUP BY contracts.address
HAVING tx_count > 1
ORDER BY tx_count DESC
```

Then we write a script to obtain the bytecode and/or source code through Etherscan's API. Finally, we collect 765,928 contracts with only bytecode, 4,063 contracts with only source code, and 741,934 contracts with both source code and bytecode. For further analysis, we only retain source-available contracts[2]. Then we adopt the method used in [14] to remove the duplicates, that is checking whether the MD5 checksums of the two source files are identical after removing all the blank lines and comments. After deduplication, we get 45,622 unique contracts. In order to verify the authenticity of bugs reported by each tool, we add some labels to the UR dataset. Specifically, we use all tools to check these contracts, and obtain 8,200 suspected vulnerable ones. Together with two experts from our industry collaborator WeBank, we conduct manual inspection for two months, and filter out 10 real vulnerabilities, which are confirmed with exploitation.

Real contracts usually have more lines of code and more complex internal logic. In addition, due to the existence of mechanisms such as interface calls and inheritance, the interaction behaviors between contracts are also more complicated. The possible vulnerabilities contained in these contracts are unknown, and they are often hidden in deeper branches, making them difficult to detect. Also, the pattern of vulnerabilities contained in the UR category is single, and tools can design targeted strategies to get better performance. Hence, we need supplementary datasets with more diverse forms of bugs to prevent overfitting.

MI dataset collection and processing. The second category is a manually constructed benchmark designed to assist academic research, named SolidiFI. It is a large library which is constructed by injecting bugs into different potential locations to produce vulnerable contracts with specific security vulnerabilities. SolidiFI supports 7 different bug types, namely, reentrancy, timestamp dependency, unhandled exceptions, unchecked send, transaction order dependency, integer overflow/underflow, and use of tx.origin. For each category, there are 50 original bug-free contracts. After injection, 9,369 distinct bugs are scattered across all contracts.

These vulnerabilities are independent of each other and have no interaction with other internal functions. Their code logic is relatively simple, involving only a few obvious vulnerability patterns.

[2]In the future, we will add the support for analysis of contracts with only bytecode through the Gigahorse decompiler [20].

In addition to the dataset of the modified contracts, the repository also contains the injection logs that can be used to refer the location and type of each bug.

CV dataset collection and processing. The last category consists of vulnerable contracts that have been confirmed by CVE reviewers or SWC registry. First, we use "smart contract" as the keyword to query the CVE list and find 513 entries that matched the search. We download all source code files and classify them into six categories: arithmetic overflow/underflow, bad randomness, access control, unsafe input dependency and others, as classified in [49]. After removing duplicates, we obtain 124 distinct contracts. Then, we visit the official website of SWC registry and crawl 90 contracts related with 37 identifiers. Finally, we merge these vulnerable contracts to form a dataset with the size of 214.

All vulnerabilities contained in each contract are professionally confirmed, and clearly classified and labeled. Moreover, the average number of code lines per contract in this dataset is under 200, and there are no interface calls or interactions between contracts.

3.2 Selecting Candidate Tools

Our experiment used nine most frequently compared open-source tools, three for each detection method (see Table 3). For static analysis, we select Securify, SmartCheck and Slither; for symbolic execution, we select Oyente, Mythril and Osiris; and for dynamic fuzzing, we select ContractFuzzer, sFuzz and ILF.

Table 3: List of selected smart contract testing tools, where #Baseline indicates the number of times the tool is used as a baseline, #Citation indicates the number of citations.

Tool	Method	#Baseline	#Citation	Publication
Securify	SA	7	267	CCS'18
SmartCheck	SA	2	147	WETSEB'18
Slither	SA	1	43	WETSEB'19
Oyente	SE	16	1008	CCS'16
Mythril	SE	10	1500	White Paper
Osiris	SE	3	56	ACSAC'18
ContractFuzzer	DF	3	140	ASE'18
sFuzz	DF	1	8	ICSE'20
ILF	DF	1	22	CCS'19

Because each tool targets specific issues, the types of vulnerabilities it supports vary greatly. For a fair comparison, we investigate the types of vulnerabilities supported by each candidate tool. Then we map them to a list of the most common vulnerability types that have been reported for EVM-based smart contracts [14]. Lastly, we find reentrancy is the only one that is supported by all of them. The summary is shown in Table 4.

3.3 Unified Execution

A uniform execution environment is extremely important for dynamic executors, including the unified platform and consistency of runtime parameters. In the experiment of each set of parameters, we provide the same value for the corresponding parameter of each tool and keep other environment variables consistent. Considering that each tool has an arbitrary number of self-defined parameters, for tools based on the same implementation method, we choose the general parameters that they all support to carry out experiments. In addition, in view of the randomness of the fuzzing procedure, we

Table 4: A summary of vulnerability types supported by candidate tools. For each vulnerability type: AC means access control; IO means integer overflow; DS means denial of service; TO means transaction order dependency; RE means reentrancy; TM means time manipulation; UE means unhandled exception and LE means locked ether.

Tool	Vulnerability Types							
	AC	IO	DS	TO	RE	TM	UE	LE
Securify	✓			✓	✓		✓	✓
SmartCheck	✓	✓	✓		✓	✓	✓	✓
Slither	✓		✓		✓	✓	✓	✓
Oyente	✓	✓	✓	✓	✓	✓		
Mythril	✓	✓		✓	✓	✓	✓	✓
Osiris		✓	✓		✓	✓		
ContractFuzzer	✓				✓	✓	✓	✓
sFuzz	✓	✓			✓	✓	✓	✓
ILF	✓				✓	✓	✓	✓

also add the number of trials as a parameter. The final parameter list used by this paper is: depth limit, trials and timeout.

For each of the parameters, we prepare (at least) 20 values for change. The minimum is 1, and the maximum is 2X of the mode value[3] that most tools use in the setting of this parameter. The first two parameters only aim at evaluating symbolic executors or dynamic fuzzers, so the interval between every two values is fixed. When it comes to timeout, it is a parameter that shared by all tools. In consideration of the divergence in execution speed among different tools, we set increasing intervals in different ranges. For example, in the first 5 seconds, the time interval is 2 seconds; from the 5th to the 30th second, the interval increases to 5 seconds; from the 30th second to the 3rd minute, the interval is 30 seconds; when it comes to the 30th minute, the interval is 5 minutes; and after the 30th minute, the interval is 30 minutes. Table 5 shows the setting of each runtime parameter in detail.

Table 5: Setting of each parameter in our experiment.

Parameter	Method	Mode	Maximum	Interval
Depth limit	SE	50	100	5
Trials	DF	24	48	-
Timeout (s)	SA, SE, DF	1800	3600	2, 5, 30, 300, 1800

All experiments were performed atop three machines. Each of them is equipped with 8 cores (Intel i7-7700HQ @3.6GHz), 24GB of memory, and uses Ubuntu 16.04.6 as the host operating system. In order to avoid possible variations among these machines, each candidate tool is always tested on the same computer.

3.4 Quantitative Analysis

We select unique bugs, precision and recall as metrics to measure performance. When the execution is finished, each tool will generate an output file that contains the results of the analysis in json format. Then we use a python script to parse it, building a mapping from vulnerability type to line numbers. By matching it with manually tagged labels, we can calculate the total number of reported bugs (Ns), true positives (TPs), false positives (FPs), and missed samples (FNs) reported by each tool on each contract. Using these data, we can calculate each metric with the methods below:

Unique bugs. It refers to the number of bugs reported by each tool on each contract after deduplication. We regard the same type of vulnerability reported in the same line as a duplicate, and keep only one count. The total number of bugs after de-duplication is the value of unique bugs (N). Using the same method, we can get unique true positives (TP), false positives (FP) and missed samples (FN), which are used in following formulas.

Precision. It is defined as a ratio of true positives and the total number of positives predicted by a model. A higher precision value indicates a higher percentage of correct alarms and fewer false alarms, making it easier for bug verification and code modification.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

Recall. It is the fraction of total amount of relevant instances that were actually retrieved. A higher recall value indicates that the more real vulnerabilities can be found, the fewer hidden vulnerabilities are missed, and the risk of unknown attacks is lower. Specially, precision and recall are also irrelevant to magnitude, which graduates that the conclusions in our paper will not be biased by the absolute number of seeded bugs.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

4 RESULTS AND ANALYSIS

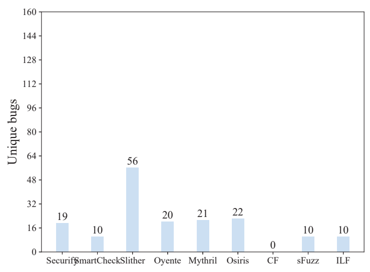
In this section, we will present the firsthand experimental results and analyze the variation of tools' performance on different test suites, runtime parameters and evaluation metrics. Observing the trends under different experiment settings, we have discovered some problems that have been neglected in previous evaluations. Then we will propose some guidelines to which future work should adhere while comparing with other baselines.

4.1 Variation on Different Test Suites

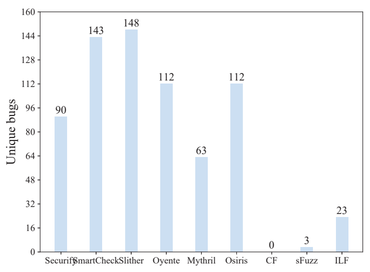
Recent published work chose a wide variety of datasets as the test suites, such as real-world contracts crawled from Etherscan or vulnerable contracts confirmed by CVE organization. However, none of the existing benchmark choices met the demands of comprehensiveness. As shown in the fourth column in Table 1, 78% of the tools (21 out of 27) only use a single category of dataset for evaluation. In this situation, the code characteristics are limited, and the tool can easily formulate some targeted detection rules to achieve better performance in an over-fitting manner. Therefore, conclusions drawn by these experiments may be one-sided and cannot represent the capabilities of the tool in all scenarios.

Fig. 3 shows the variation of each metric under different test suites. For each dataset, we repeatedly run each tool for 48 times to collect final stable results. Most evaluated tools (6/9) are deterministic, and each run produces the same result. The other tools run for 12 hours to converge. From the horizontal direction, we can see that the performance of each tool varies greatly across datasets. After statistical analysis, we further illustrate that most differences are statistically significant with $p<0.021$ in Wilcoxon test. Different choices of datasets may lead to different experimental conclusions. Taking SmartCheck and Slither as an example, it is reported in Slither that SmartCheck had a large number of false positives due

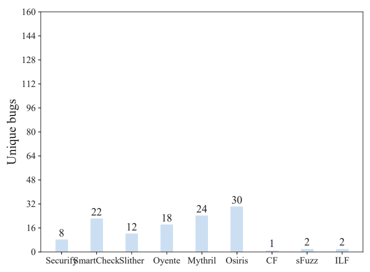
[3]The mode is the value that appears most often in a set of data values.



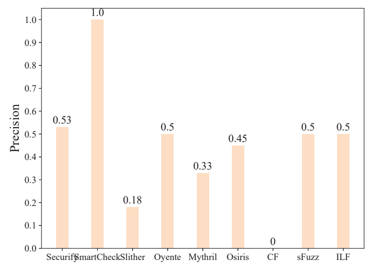
(a) Variation in unique bugs on UR dataset



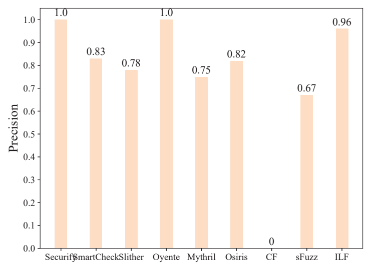
(b) Variation in unique bugs on MI dataset



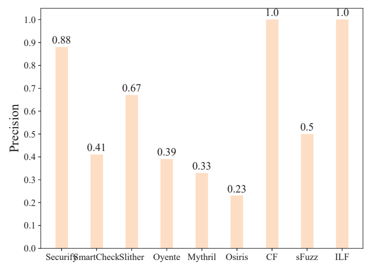
(c) Variation in unique bugs on CV dataset



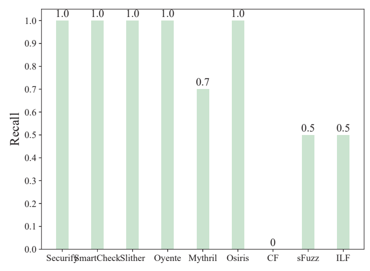
(d) Variation in precision on UR dataset



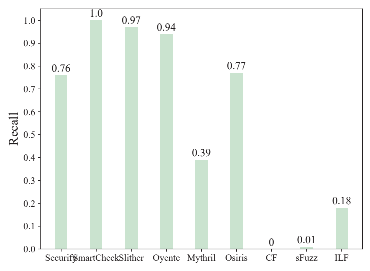
(e) Variation in precision on MI dataset



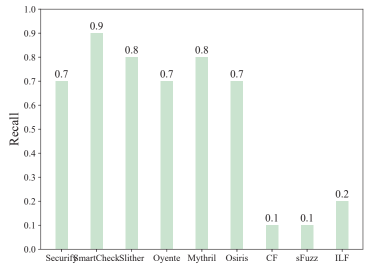
(f) Variation in precision on CV dataset



(g) Variation in recall on UR dataset



(h) Variation in recall on MI dataset



(i) Variation in recall on CV dataset

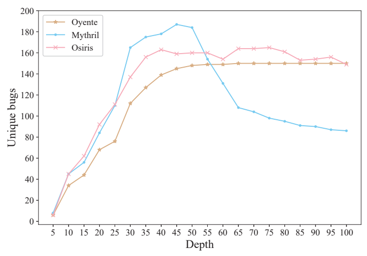
Figure 3: Variation in unique bugs, precision and recall on different test suites of all tools.

to a lack of in-depth understanding of Solidity [16]. However, our experiment shows that this conclusion is not always valid. On UR dataset, SmartCheck exactly finds 10 existing vulnerabilities and gets full marks in precision rate, which is 5X higher than Slither. When it comes to MI dataset, they are roughly the same. Only on the last dataset, Slither has a 1.6X higher precision than SmartCheck.

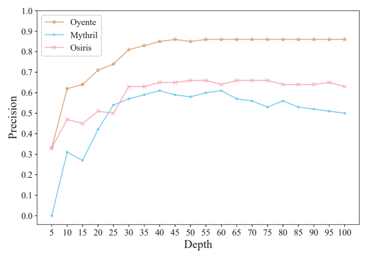
The same situation can also be observed in Mythril and Osiris. Either on real-world contracts or buggy contracts injected with artificial bugs, Osiris is able to find more hidden problems more accurately, which manifests through a 10% increase in precision and 30% higher in recall. But surprisingly, on CV dataset where code structures are far more simple, the observation is completely reversed. Although Osiris finds 6 more bugs in total, it falls behind Mythril by 10 percentage points in both precision and recall.

Deeper Analysis. Targeted rule designation for a specific vulnerability pattern and insufficient processing capabilities for deep calls are the main reasons for the unstable performance of tools. In addition, different judgment standards for real vulnerabilities also lead to differences in tool performance evaluation. In this paper, only vulnerabilities that can be attacked in practice are identified as positive samples, which is more practical than the judgment conditions in previous papers, but also leads to a decrease in precision. Take Slither as an example, it will mark *addr.transfer()*, which has a built-in gas limit of 2300 to prevent re-entering problems, as a reentrancy bug. This kind of false positive samples were marked as true bugs during the original evaluation of the published tools, so most of them (22/27) had a high precision rate in their papers, while performs poorly in reality.

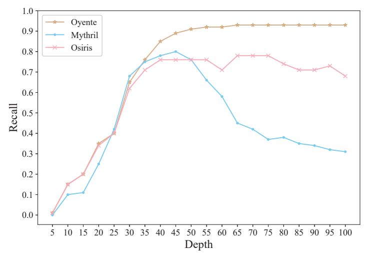
Finding: The code size and vulnerability pattern of the test suite have a great impact on tool performance. So it is better to carry out the evaluation on a multi-type integrated benchmark suite to eliminate the biases.



(a) Unique bugs vs. depth limit

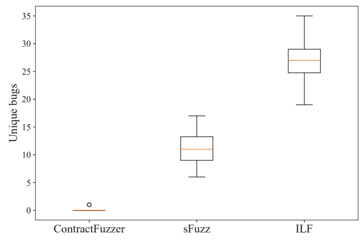


(b) Precision vs. depth limit

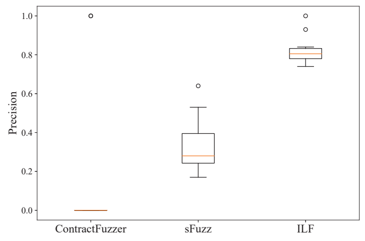


(c) Recall vs. depth limit

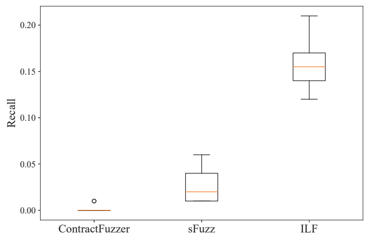
Figure 4: Variation in unique bugs, precision and recall with different depth limit of Oyente, Mythril and Osiris.



(a) Summary statistics of unique bugs



(b) Summary statistics of precision



(c) Summary statistics of recall

Figure 5: Variation in unique bugs, precision and recall throughout 48 trials of ContractFuzzer, sFuzz and ILF.

4.2 Variation on Runtime Parameters

This section will experimentally demonstrate the impact of different parameters on the performances of tools[4]. The execution of a tool depends on the execution environment, which consists of the generic platform and proper runtime parameters. For static tools, they often construct an intermediate expression based on the bytecode, and then perform pattern matching or data structure analysis based on it. Therefore, after installing the correct dependency packages, user only needs to provide the source code or bytecode of the contract and the maximum execution time. Dynamic tools are more complicated, requiring the user to additionally set appropriate values for runtime parameters, such as maximum recursion depth. Most people use the same parameter values for all tools in the evaluation process. However, this may not be the most appropriate choice. Experiments based on values that are not suitable for them may lead to inconsistent conclusions. Details are listed below.

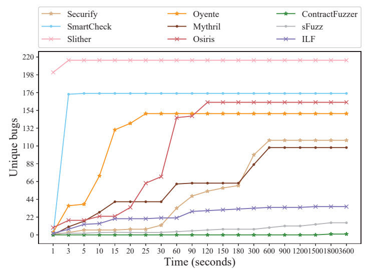
Adjust Maximum Search Depth: Depth limit refers to the maximum depth that can be accessed along a single path. Most symbolic executors adopt a depth-first strategy to visit nodes in the control flow graph. Starting from the entry node, each instruction inside a basic block will be executed symbolically. When finishing the last instruction, the executor will enter the next block according to the jump condition. This process will keep repeating until the termination block is reached or the search depth reaches the limit.

Depth limit plays a critical role in detecting deeper vulnerabilities. However, when it is set too large, blocks on other execution paths will be inaccessible within a specified period of time, which will lead to the omission of vulnerabilities. The depth settings of different symbolic execution tools vary greatly. Among 10 recent papers (see Table 1), MAIAN and sCompile set a depth limit of 3; VerX and Mythril respectively increase it to 5 and 22; Oyente and its derivatives, Osiris and HoneyBadger, set a large threshold of 50; while others did not mention the depth limit in the paper.

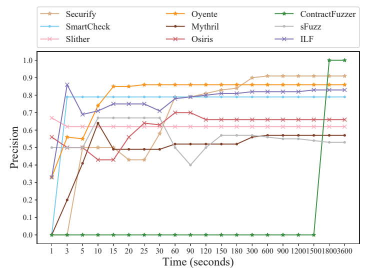
We tested Oyente, Mythril and Osiris with different depth limits, and the performance variations are shown in Fig. 4. As the search depth increases, the path that each tool can access becomes deeper, and the number of reported bugs, precision and recall continue to rise. But when the search depth exceeds 50, it starts to hinder the normal execution of the tool. Since too much time is wasted on deep blocks of a single path, the performance of each tool experiences varying degrees of decline. The choice of different depth limits for evaluation may lead to different conclusions. As an example, when the maximum search depth is set to 25, both Mythril and Osiris report 110 unique bugs, but Mythril outperforms Osiris with 4% higher precision rate and 4 more real vulnerabilities. However, when the depth limit is liberalized to 70, Mythril's performance is far behind Osiris's. This time, Osiris reports 164 unique bugs, with a precision rate of 66% and recall rate of 78%. While Mythril is only able to find 104 suspected bugs, with a precision rate of 56% and recall rate of 42%.

Increase Test Trials: As is known to all, the impact of randomness on dynamic fuzzers cannot be ignored [64]. During evaluation, it is insufficient to simply run each fuzzer only once and compare their performances. One suggested solution is to carry out multiple rounds of experiments to make statistically sound comparison [1].

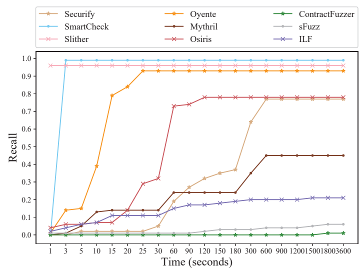
[4]The experimental results in this section are displayed based on the mix collections, and the detailed statistics of each dataset can be accessed at: https://github.com/renardbebe/Smart-Contract-Benchmark-Suites/tree/master/experiments-config.



(a) Unique bugs vs. execution timeout



(b) Precision vs. execution timeout



(c) Recall vs. execution timeout

Figure 6: Variation in unique bugs, precision and recall with different execution timeout of all tools.

As to other testing tools based on fuzzing, this criterion should also hold true. Unfortunately, Table 1 indicates that only 3 out of all fuzzing tools perform experiments for more than one time.

We separately ran ContractFuzzer, sFuzz and ILF to verify whether the performance of each run is stable. Fig. 5 describes the overall distribution of each tool's performance throughout 48 separate runs. For each tool, the data distribution is divided into four parts equally and a box is used to indicate the positions of the upper and lower quartiles. The orange line inside it represents the median value and lines outside the box respectively indicates the minimum and maximum value. Outliers, which are generally extremely good or bad points, are represented individually by small dots. It can be found that under the same execution environment, the performance can vary dramatically from run to run. Take ILF as the example. It can find up to 35 unique bugs, which is nearly 2X more than the lowest point. Its best precision rate is 100%, and it can find up to one-fifth of real vulnerabilities. Meanwhile, in some rounds, its precision rate drops to about 70%, and the recall rate is less than 10%. The result obtained by a single run has a certain degree of contingency, and the dominance relation reflected by it may be biased. In practice, it is suggested to perform multiple trials and evaluate based on the averages or the most frequently occurring results.

Extend Execution Timeout: Execution timeout refers to the longest time a tool can take to analyze a contract. For each tool, it is a parameter that can be customized by the user. A commonly accepted view is a longer time may illuminate a more stable performance trend [46]. Meanwhile, it is also generally believed that more execution time beyond a certain threshold is unnecessary and only results in a waste of resources [28]. The reason is that they can only cause small fluctuations in performance and will not change the dominance relation between tools. These two assumptions leave a large space for the setting of execution timeout. The column **timeout** in Table 1 shows the dramatic difference in timeout settings across prior evaluations, which ranges from 20 seconds to more than 3 days. The most common choice is 30 minutes, which was used by 4 papers. Nearly 70% of the rest set the timeout with an arbitrary value between 2 minutes and 1 hour. MadMax, sCompile and Zeus use a short timeout of less than 2 minutes, where most of them are static tools. The remaining 4 papers run their tools for more than 1 hour, where all of them are dynamic tools.

To identify the impact of execution timeout on tool's performance, we set up 20 different termination times and conducted experiments on all selected tools. The trends are shown in Fig. 6. Due to the difference in execution speed, dynamic tools often take longer to report bugs and reach stability. For example, ContractFuzzer did not enter the branch where the bug was located until the 1800th second. Since it's the only discovered bug, the precision jumps from 0% to 100% and stays. In contrast, SmartCheck finished the analysis of the entire code within two seconds, so the precision fails to increase after that. Before reaching the upper bound, the improvement in performance is non-linear. This will result in the varying relative performance between tools over time. In this condition, using a short timeout for evaluation may yield an incomplete conclusion. When running Mythril and Osiris with a timeout of 10 seconds, the total number of bugs reported by Mythril is 28, which is 5 more than Osiris's. In terms of precision and recall, Mythril is also better than Osiris, with 21% and 6% higher respectively. But running them longer seems to tell a different story. When the timeout is set to 120 seconds, Osiris reports 164 unique bugs in all, 109 of which are real, accounting for 78% of all hidden vulnerabilities. In contrast, Mythril only reports 64 unique bugs, half of which are real, accounting for less than a quarter of vulnerabilities.

Deeper Analysis. The maximum benefit that each tool's detection strategy can bring depends on the setting of a series of execution parameters. Even tools based on the same implemented method may show different sensitivity to parameters. For example, the optimal search depth limit of tools based on breadth first search may be greater than that of tools based on depth first search, since the larger depth limit will not prevent breadth-first tools from discovering shallow vulnerabilities in other branches, but will prevent depth-first tools from entering other branches.

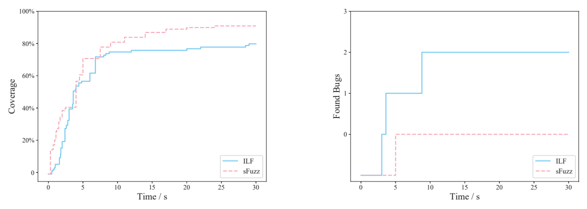
Finding: The configuration parameters at runtime have a significant impact on tool's performance. To make the evaluation more fair and convincing, during the experiment, reasonable and suitable values should be provided for custom parameters of each tool.

4.3 Variation on Performance Metrics

In addition to benchmark suite and runtime parameters, the selection of performance metrics also needs to be persuasive and reasonable. Looking back on published work in recent years, there are two commonly used performance metrics, namely the discovery of real vulnerabilities and code coverage. The former can be further subdivided into three metrics, total number of bugs, precision and recall; the latter generally has three manifestations. As shown in Table 1, the most commonly used is path coverage, which was used by 6 papers, referring to the coverage of all feasible paths in the code. Then comes instruction coverage used by 5 papers, which counts the number of executed instructions. The remaining two types are basic-block and branch coverage, which are calculated as the percentage of the covered basic-blocks or branches.

It seems to make sense that coverage can represent the ability of a tool to discover vulnerabilities: the more code a tool executes, the more likely it is to find vulnerabilities [19, 28]. However, the relationship between them does not seem to be strong. Directly using it to prove the bug finding ability of the tool (such as Manticore) is also inadvisable and inadequate.

Take a real-world contract named *ItemToken*[5] as an example. We separately ran ILF and sFuzz on it, and recorded the variation in instruction coverage[6] and total number of reported vulnerabilities over time. The results are shown in Fig. 7. After 30 seconds, neither ILF nor sFuzz was able to find new vulnerabilities, and there was no significant improvement in coverage. Therefore, we drew the curves of the first 30 seconds for explanation.



(a) Instruction coverage vs. time (b) Number of vulnerabilities vs. time

Figure 7: Variation in coverage and number of vulnerabilities reported by sFuzz and ILF over time.

Although sometimes, the increase in coverage is accompanied with the growth in vulnerabilities. For example, at the fifth second, sFuzz succeeded in entering a new branch, and the coverage increased from 61% to 71%, at the same time a new vulnerability was discovered. But comparing the number of steps in (a) and (b), it can be found that, in most of the time, while the coverage increases, the number of vulnerabilities remains unchanged. In sFuzz's coverage curve, only one out of 22 rises results in the discovery of new vulnerabilities. Same thing goes for ILF, the coverage curve has raised 29 times, while the vulnerability discovery curve has only changed three times. What's more, higher coverage does not guarantee a stronger ability in finding vulnerabilities. As shown in Fig. 7 (a), the coverage rate of sFuzz reaches 91%, and ILF only reaches 80%. However, as can be seen from (b), ILF finally finds three unique bugs while sFuzz can only find one. In this case, it is biased to conclude that ILF performs better than sFuzz according to a higher coverage.

Deeper Analysis. The most fundamental metric to measure tool performance is the number and percentage of real vulnerabilities it finds. As an indirect indicator, a higher coverage cannot stand for a better performance of the tool. The increase in coverage may be caused by some meaningless code, such as test code, which is not directly helpful to vulnerability detection.

Finding: Different tools have different performance on different metrics. To reach a rigorous conclusion, we need to evaluate on more comprehensive metrics, including the number of unique bugs, precision, recall and coverage.

5 OBSERVATIONS AND SOLUTIONS

Though current testing tools can detect a significant amount of potential security vulnerabilities in actual projects, we still find out the following points worthy to be discussed:

It is necessary to have a unified evaluation standard. In this paper, we have investigated 27 related papers and analyzed their experimental methodologies. The description in paper showed that they did not share a unified setting. Using nine representative tools, we illustrated that individual evaluations with different experiment settings could lead to misleading conclusions. We find that

- More than 70% of papers only used unlabeled real-world contracts as the benchmark, where the distribution of vulnerabilities are unknown and the vulnerability pattern is simple. This makes it impossible to comprehensively evaluate the tool's ability in detecting each type of vulnerability and is easy to overfit, such as the performance of SmartCheck in the UR dataset. At the same time, without clear labels, it is also impossible to accurately tally false negative samples. Our experiments showed that different choice of test suites have a great impact on tool performance.
- Papers varied widely on the setting of runtime parameters, such as maximum search depth. Our experiments showed that too large threshold will hinder the normal execution of the tool.
- Most papers failed to take varying performance caused by runs into consideration. Our experiments showed that it is necessary to perform multiple trials with statistical tests to balance the impact of randomness.
- Among all papers, the choice of execution timeout ranges from 20 seconds to 80 hours. Our experiments showed that a longer timeout is needed to draw a full picture of a tool's performance, especially for the dynamic tools.
- Fewer than half of the papers evaluated tool's performance from all perspectives: unique bugs, precision and recall. Others only selected one or two of them to illustrate their tools were the best, or even only evaluated the performance based on the coverage. Our experiments showed that using part of them is not sufficient.

To produce trustworthy and comparable conclusions, it is necessary to standardize from the following three aspects: test suites, runtime parameters and performance metrics. First, a solid, diverse benchmark suite is needed to assess the vulnerability detection ability of each tool in different applicable scenarios. It should not

[5]Source code available at: https://etherscan.io/address/0x0a8b758bbc4a5791c5647ca80351e008f1e3bca1#code.

[6]We extended sFuzz to support instruction coverage with additional instrumentation, which did not affect the vulnerability mining ability of the tool.

only be close to the contracts used in actual projects, but also have annotations to help developers confirm false and missing reports. Second, while comparing to other tools, the author should provide the custom parameters with suitable values to ensure the best performance of each tool. For example, during symbolic execution, it is better to set the depth limit to around 50. Too deep or too shallow will have a negative impact on the normal execution of the tool. As for fuzzers, a longer timeout and multiple runs are essential. Finally, the evaluation metrics should aim at finding as many real vulnerabilities as possible, and should give attention to both quantity and quality.

It is important to reduce the false positives reported by static tools, although they can scan the code completely. According to direct pattern matching, some harmless operations will be wrongly identified as vulnerabilities. For example, the false positive rate of Slither can be up to 82% on real-world contracts. To eliminate the false alarms which can't be reproduced in reality, some work adopts the test method of combining dynamic and static analysis. That is, the code snippets that may contain vulnerabilities are screened out through static analysis first, and then the dynamic tools are used to execute the suspected target code. If it can be triggered successfully, it will be marked as a real bug.

To evaluate the performance of this method, we combine the static tool with the highest false positives (Slither) and the dynamic tool with best overall performance (ILF) for test. As shown in Table 6, ILF filters out 74 code snippets that are mistakenly classified by Slither, improving the precision by nearly 30%. Another method is critical path filtering. In addition to control flow and data flow, it is also meaningful to introduce the recognition and analysis of money flow, filter out paths related to sensitive operations such as transfers, and then perform rule satisfiability judgments based on these money-flow sensitive paths, thereby reducing false alarms.

Table 6: Gains of dynamic and static combination.

Tool	TP	FP	FN	N	Precision	Recall
Slither	134	82	5	216	0.62	0.96
Slither + ILF	62	8	77	70	0.89	0.45

Furthermore, although in general, static tools have a high false positive rate, some targeted static tools perform well in the detection of a certain sub-set of vulnerabilities. For example, MadMax [21] for out-of-gas errors and Ethainter [3] for composite vulnerabilities. Integrating these tools can also increase the number of supported vulnerability types while maintaining high detection accuracy.

More work is needed to further improve execution efficiency of dynamic tools, though they are more accurate. They usually have a higher probability of finding real vulnerabilities, which benefits from the simulation of real execution process. However, because of the randomness of execution path and the finiteness of guidance information, these dynamic tools are often inefficient. Within a limited time, they can only cover a small part of reachable paths, and have a poor ability in handling complex logic. As an example, both sFuzz and ILF reach a high false negative rate of 90% on confirmed vulnerable contracts. In order to help dynamic tools detect more vulnerabilities in a short time, the most commonly used method is integration. The simplest way is to execute different tools in parallel and take the union of their output results [14].

In order to verify the benefits of integration testing, we execute three dynamic fuzzers in parallel, consolidate and de-duplicate the reported warnings. Then we compare the integrated results with those of a single tool. The differences are shown in Table 7. We surprisingly find that, just after simple integration of output results, false negative rate is reduced by an average of 10%. On this basis, further seed synchronization and tester scheduling can be performed. For example, establishing a shared seed pool, and synchronizing the seeds with new coverage to each tester during the execution process; or selecting applicable seed generation strategies for different code blocks and calling different testers to check.

Table 7: Gains of integrated fuzz testing.

Tool	TP	FP	FN	N	Precision	Recall
Only ContractFuzzer	1	0	138	1	1.00	0.10
Only sFuzz	8	7	131	15	0.53	0.06
Only ILF	29	6	110	35	0.83	0.21
Integrate three fuzzers	30	8	109	38	0.79	0.22

In addition, most fuzzers cannot realize the automation of whole process. Different from static analyzers and symbolic executors, fuzzers need to deploy the contract to be tested on a local private chain first, and then interact with the contract by sending transactions in order. Taking ContractFuzzer as an example, it provides the deployment script but requires a series of config files, like the abi files and pairs of signatures. These files are regarded as the preparation work and need users to imitate the example and generate by themselves, which is user-unfriendly and error-prone in practice.

6 CONCLUSION

Smart contracts have a wide range of applications, involving a large amount of digital assets. For developers and transaction participants, security is critical. Though various testing tools for smart contracts have emerged endlessly, the lack of a unified evaluation standard makes it difficult to evaluate their performance in a scientific, fair and comprehensive way. For assisting follow-up researchers in carrying out comparative experiments more reasonably, we collect 46,186 diversified contracts, propose a systematic evaluation process and perform extensive experiments. We recommend that the evaluation process should take the following factors into consideration: (a) a set of diverse test suites; (b) a unified execution environment with suitable runtime parameters; (c) more quantitative and multi-dimensional performance metrics.

During experiments, we also find some notable problems and point out potential directions for future work. The first is to better combine dynamic and static methods to reduce false positives. The second is to develop more advanced integration models that can give full play to the strengths of different tools. We hope this paper can provide ideas and references to follow-up researchers, and help them develop more reliable, powerful and friendly testing tools.

ACKNOWLEDGEMENT

This research is sponsored in part by the NSFC Program (No. 62022046, U1911401, 61802223), National Key Research and Development Project (Grant No. 2019YFB1706200), the Huawei-Tsinghua Trustworthy Research Project (No. 20192000794).

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1–10.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[3] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[4] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

[5] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1066–1071.

[6] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. 2019. sCompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods*. Springer, 286–304.

[7] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. arXiv:2009.02663 [cs.SE]

[8] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* (2020).

[9] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[10] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ting Wang, Teng Hu, Xiuzhuo Xiao, Dong Wang, Jin Huang, and Xiaosong Zhang. 2019. A large-scale empirical study on control flow identification of smart contracts. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

[11] ConsenSys. 2018. Mythril. https://github.com/ConsenSys/mythril-classic.

[12] A Day and E Medvedev. 2019. Ethereum in BigQuery: a public dataset for smart contract analytics.

[13] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121.

[14] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 530–541.

[15] Etherscan. 2019. Etherscan. https://etherscan.io/.

[16] J. Feist, G. Grieco, and A. Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15.

[17] Asem Ghaleb and Karthik Pattabiraman. 2020. How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 415–427. https://doi.org/10.1145/3395363.3397385

[18] Google. 2018. Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite.

[19] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. 72–82.

[20] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1176–1186.

[21] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[22] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[23] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 531–548.

[24] Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. 2020. A Comprehensive Survey on Smart Contract Construction and Execution: Paradigms, Tools and Systems. *arXiv preprint arXiv:2008.13413* (2020).

[25] Sungjae Hwang and Sukyoung Ryu. 2020. Gap between theory and practice: An empirical study of security patches in solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 542–553.

[26] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018* (2018). https://doi.org/10.1145/3238147.3238177

[27] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts.. In *NDSS*.

[28] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[29] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 1–13.

[30] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[31] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 814–819.

[32] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. *IACR Cryptology ePrint Archive* (2016), 633.

[33] Fuchen Ma, Ying Fu, Meng Ren, Wanting Sun, Zhe Liu, Yu Jiang, Jun Sun, and Jiaguang Sun. 2019. Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *arXiv preprint arXiv:1910.02945* (2019).

[34] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[35] Anastasia Mavridou and Aron Laszka. 2018. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 270–277.

[36] MITRE. 2018. Common vulnerabilities and exposures. https://cve.mitre.org/.

[37] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[38] MythX. 2019. Smart Contract Weakness Classification and Test Cases. https://swcregistry.io/. Accessed November 4, 2019.

[39] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. *arXiv preprint arXiv:2004.08563* (2020).

[40] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663.

[41] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, and Andrea Cullen. 2018. Visual emulation for Ethereum's virtual machine. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–4.

[42] Trail of Bits. 2018. ethersplay. https://github.com/crytic/ethersplay.

[43] Reza M Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. *arXiv preprint arXiv:1809.02702* (2018).

[44] Daniel Perez and Benjamin Livshits. 2019. Smart contract vulnerabilities: Does anyone care? *arXiv preprint arXiv:1902.06710* (2019).

[45] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*. 18–20.

[46] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2155–2168.

[47] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. 2019. Security analysis methods on Ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605* (2019).

[48] Raine Revere. 2018. solgraph. https://github.com/raineorshine/solgraph.

[49] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1678–1694.

[50] Matt Suiche. 2017. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF con* 25 (2017), 11.

[51] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, and Yaroslav Alexandrov. 2018. SmartCheck: static analysis of ethereum smart contracts. In *the 1st International Workshop*.

[52] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2020. Towards Smart Hybrid Fuzzing for Smart Contracts. *arXiv preprint arXiv:2005.12156* (2020).

[53] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 664–676.

[54] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1591–1607.

[55] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *ACM Conference on Computer and Communications Security*.

[56] Patrick Ventuzelo. 2018. Octopus. https://github.com/pventuzelo/octopus.

[57] Mingzhe Wang, Jie Liang, Chijin Zhou, Yuanliang Chen, Zhiyong Wu, and Yu Jiang. [n.d.]. Industrial Oriented Evaluation of Fuzzing Techniques. ([n. d.]).

[58] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su. 2020. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Transactions on Network Science and Engineering* (2020), 1–1.

[59] Valentin Wüstholz and Maria Christakis. 2019. Harvey: A greybox fuzzer for smart contracts. *arXiv preprint arXiv:1905.06944* (2019).

[60] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. 2020. Clairvoyance: Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 274–275. https://doi.org/10.1145/3377812.3390908

[61] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2019. SolidityCheck: Quickly Detecting Smart Contract Problems Through Regular Expressions. *arXiv preprint arXiv:1911.09425* (2019).

[62] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5.

[63] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: reverse engineering ethereum's opaque smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1371–1385.

[64] Xiaogang Zhu, Xiaotao Feng, Tengyun Jiao, Sheng Wen, Yang Xiang, Seyit Camtepe, and Jingling Xue. 2019. A feature-oriented corpus for understanding, evaluating and improving fuzz testing. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 658–663.