Are We There Yet? Unraveling the State-of-the-Art Smart Contract Fuzzers

Shuohan Wu
The Hong Kong Polytechnic
University
Hong Kong, China
csswu@comp.polyu.edu.hk

Zihao Li
The Hong Kong Polytechnic
University
Hong Kong, China
cszhli@comp.polyu.edu.hk

Luyi Yan
The Hong Kong Polytechnic
University
Hong Kong, China
cslyan@comp.polyu.edu.hk

Weimin Chen
The Hong Kong Polytechnic
University
Hong Kong, China
cswchen@comp.polyu.edu.hk

Muhui Jiang
The Hong Kong Polytechnic
University
Hong Kong, China
jiangmuhui@gmail.com

Chenxu Wang
Xi'an Jiaotong University
Xi'an, China
cxwang@mail.xjtu.edu.cn

Xiapu Luo*
The Hong Kong Polytechnic
University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Hao Zhou
The Hong Kong Polytechnic
University
Hong Kong, China
cshaoz@comp.polyu.edu.hk

ABSTRACT

Given the growing importance of smart contracts in various applications, ensuring their security and reliability is critical. Fuzzing, an effective vulnerability detection technique, has recently been widely applied to smart contracts. Despite numerous studies, a systematic investigation of smart contract fuzzing techniques remains lacking. In this paper, we fill this gap by: 1) providing a comprehensive review of current research in contract fuzzing, and 2) conducting an in-depth empirical study to evaluate state-of-the-art contract fuzzers' usability. To guarantee a fair evaluation, we employ a carefully-labeled benchmark and introduce a set of pragmatic performance metrics, evaluating fuzzers from five complementary perspectives. Based on our findings, we provide direction for the future research and development of contract fuzzers.

KEYWORDS

Smart Contract, Fuzzing, Evaluation

---

*The corresponding author.

---

1 INTRODUCTION

Smart contracts have witnessed rapid growth and have been widely adopted for various financial purposes, such as trading, investing, and borrowing [5, 63]. As of February 2023, there are more than 44 million contracts deployed on Ethereum and the market cap of Ethereum has exceeded 210 billion USD [11, 79]. However, despite their widespread adoption, many smart contracts lack a thorough security audit, making them vulnerable to potential attacks [86]. In fact, a recent empirical study of 47,587 Ethereum smart contracts found potential vulnerabilities in a significant number of them [41]. This raises concerns, especially for contracts that handle financial assets of significant value, as they are a prime target for attackers [34, 39]. For instance, in June 2016, attackers exploited a reentrancy bug in Ethereum's decentralized autonomous organization (DAO), resulting in the theft of 3.6 million Ether [8]. Given that smart contracts cannot be modified after deployment [61], it is crucial to ensure their reliability before deployment.

Various approaches have been proposed to enhance the correctness and security of smart contracts, including formal verification [19, 24, 27], symbolic execution [49, 69, 73], and other static analysis methods [42, 45, 71, 90]. However, each method faces limitations. Formal verification relies on manual customized specification, which makes it difficult to automatically scale [111]. Symbolic execution is hindered by path explosion issues when exploring complex contracts [31, 91]. Static analysis approaches analyze the control and data flow in the contract, but they can exhibit high false positives because they over-approximate contract behavior [20, 37].

Fuzzing has been successful in discovering vulnerabilities in traditional programs over the years [26, 60, 64]. Owing to its dynamic nature, it produces few false positives and can detect unexpected

vulnerabilities without prior expert knowledge of vulnerable patterns [72]. Since 2018, fuzzing has been extensively employed in the realm of smart contracts [55]. With the emergence of various smart contract fuzzers [58], it is essential to systematically evaluate their usability and effectiveness. This can help pinpoint the strengths and weaknesses of each technique, guide researchers in selecting the most appropriate technique for their use cases, and inspire the development of new fuzzers based on the gained insights.

However, to our best knowledge, no study has systematically investigated existing contract fuzzers and comprehensively evaluated their effectiveness. Most related empirical works [32, 57, 78, 107] focus on comparing static analysis-based smart contract auditing tools, while two recent studies [47, 82] only evaluate a limited number of smart contract fuzzers (i.e., 3), which is far from comprehensive. Despite the fact that most studies that propose contract fuzzers experimentally validate their effectiveness, we identify a lack of uniform performance metrics among them (e.g., they employ different criteria like instruction and branch coverage to evaluate code coverage). Furthermore, the benchmarks used in these studies are not unified. These issues lead to inconsistent and biased results, making it difficult to accurately compare different fuzzers.

To fill this gap, in the paper, we first conduct a systematic review of all existing smart contract fuzzing techniques. Then, we perform a comparative study of state-of-the-art smart contract fuzzers. In order to perform a fair and accurate evaluation, we need to meet the following requirements:

• **(R1) Comprehensive performance metrics**: Existing contract fuzzing studies use limited performance metrics, leading to biased or inaccurate comparisons. To avoid this, we need to design and use comprehensive performance metrics that can be consistently applied across different fuzzers. Evaluating fuzzer's performance from multiple perspectives can provide a more complete picture of its strengths and weaknesses.

• **(R2) Unified benchmark suite**: Prior smart contract fuzzing studies use distinct benchmarks, making it hard to compare the effectiveness of different fuzzers. To ensure our reliability and reproducibility, we need to use a unified benchmark to test all fuzzers on the same set of contracts. This will create a level playing field for comparison, and ensure that our results are not skewed by the choice of benchmark contracts.

• **(R3) Correct labels**: While there are some available contract benchmarks, our experiments reveal that they are partially mislabeled. This results in false positives (non-existent vulnerabilities flagged as vulnerabilities) and false negatives (actual vulnerabilities not detected), making it difficult to draw valid conclusions about different fuzzers' performance. Hence, we need to use a correctly labeled benchmark to ensure the reliability of insights and findings from our evaluation.

Based on the requirements, we build a benchmark consisting of 2,000 carefully-labeled smart contracts and design five categories of performance metrics. Using this benchmark and metrics, we conduct extensive experiments to compare 11 state-of-the-art smart contract fuzzers. The experimental results show that state-of-the-art contract fuzzers are far from satisfactory in terms of vulnerability detection. We suggest that future fuzzers should consider enhancing their throughput, refining their test oracles, and optimizing the quality of initial seeds as potential areas of improvement. To further

grasp the real-world needs for contract fuzzers, we conduct surveys with 16 auditors from the industry. The survey result reveals that auditors favor fuzzers that provide convenience and flexibility in creating customized test oracles. This broadens the scope of vulnerability detection, enabling auditors to identify logic-related bugs within their contracts. Our study provides insights into the current state of smart contract fuzzing and suggests possible directions for future fuzzers. In summary, we conduct the first systematic study of smart contract fuzzers:

• We review the current research advances in smart contract fuzzing based on related literature published in recent years.
• We create a benchmark with 2,000 carefully-labeled contracts and utilize it to perform a comprehensive evaluation of 11 state-of-the-art contract fuzzers. Our codebase and benchmark are released at https://github.com/SE2023Test/SCFuzzers.
• We unveil problems in existing fuzzers and explore potential directions for the future design of fuzzers.

2 BACKGROUND

2.1 Smart Contract

Smart contracts are programs running on blockchains (e.g., Ethereum [9, 36]). Smart contracts begin with compiling the source code into bytecode, and deploying it onto blockchain [7, 38]. Once deployed, each contract is assigned a unique address. Users can interact with the contract by encoding the function signature and actual parameters [35, 106] into a transaction according to the Application Binary Interface (ABI). The ABI is JSON data generated in compilation, which describes methods to be invoked to execute smart contracts.

Smart contracts are stateful programs, and maintain a persistent storage for global states. The term "state variable" in contracts refers to the global variable of a contract. For instance, as shown in Line 2 of Figure 1, uBalance variable, which records the balance of user accounts, is a state variable to be permanently stored in the contract's storage. The contract execution can depend on the state variable. In the example provided, a user can withdraw funds only if the user has a sufficient balance (Line 8). The state variable can only be altered by transactions, such as sending a transaction to invoke the deposit() function to add funds to the account (Line 5).

2.2 Smart Contract Fuzzing

Fuzzing is a widely adopted testing technique to identify defects in traditional programs [1, 4, 28, 29, 33, 70]. It involves generating random or unexpected data as inputs and monitoring the effects during the program's execution [110]. Since 2018, fuzzing has been commonly applied to test smart contracts [55]. As shown in Figure 2,

```
1  contract Wallet{
2    mapping(address => uint) uBalance;
3
4    function deposit() public payable { //deposit money
5      uBalance[msg.sender] += msg.value; }
6
7    function withdraw(uint amount) public {
8      require(uBalance[msg.sender] >= amount, "Fail");
9      require(now > 30);
10     msg.sender.call.value(amount)();  //reentrancy bug
11     uBalance[msg.sender] -= amount;}}
```
Figure 1: A vulnerable contract.

a typical process of smart contract fuzzing begins by constructing an initial corpus, where each seed represents a sequence of transactions. Next, the fuzzing loop selects seeds in the corpus to perform mutations, generating new inputs that are executed by the execution engine (e.g., EVM [3]). During the execution, the engine collects runtime information, such as coverage and execution results, as feedback, which can be used to evaluate the fitness of the input. If the input achieves better fitness (e.g., higher coverage), it is retained as a seed.

In contrast to traditional fuzzing, contract fuzzing must consider transaction sequences for executing specific functions. This is because the execution of a function in a transaction may depend on a contract state, which is determined by the previous transactions executed. As illustrated in Figure 1, to trigger the vulnerable path (i.e., Line 10), we need first invoke deposit() function to deposit some funds, then call the withdraw() function. Another difference is that most vulnerabilities in smart contracts are related to business logic and will not crash the program (i.e., EVM) [92]. To detect vulnerabilities, contract fuzzers typically implement test oracles within the execution engine (e.g., EVM). Test oracles use pre-defined vulnerability patterns to match contract runtime behaviors.

2.3 Literature Search and Scope

This study primarily focuses on **Ethereum** smart contract fuzzers, given Ethereum's popularity as a go-to platform for developing and deploying contracts and the abundance of research centered on it. To identify relevant literature for our systematic survey, we set up the keywords as "smart contract fuzzing", and searched for related work published from 2016 to 2023 through seven academic databases, i.e., IEEE Explore, ACM Digital Library, Google Scholar, Springer Link, Web of Science, DBLP Bibliography, and EI Compendex. These databases maintain papers published in top conferences and journals in the field of computer science and engineering [51]. After the initial search, we retrieved a total of 75 papers. We then read the abstract of each paper to filter out those that do not propose a smart contract fuzzer. For example, we removed papers focusing on theoretical aspects of contract fuzzing. Eventually, we identified 28 papers meeting our inclusion criteria, with five focusing on other three platforms, and the remaining 23 on Ethereum. They are summarized and presented in Table 1.

3 SMART CONTRACT FUZZING APPROACH

Researchers employ various techniques to build effective fuzzers for triggering vulnerable code, including symbolic execution [40, 48, 91], machine learning [48, 65, 88, 102, 109], and static/dynamic data dependency analysis [40, 74, 93, 99, 100, 103]. This section offers a comprehensive review of the critical components in contract fuzzing approaches. Besides, we summarize the techniques employed by different fuzzers in Table 1, providing a clear overview
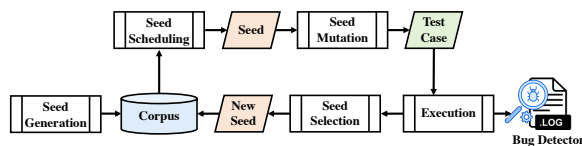

Figure 2: Overview of smart contract fuzzing.

of their methodologies and facilitating a thorough understanding of the current landscape.

3.1 Seed Generation

As highlighted by Herrera et al. [50], constructing the initial seed is a critical step that greatly impacts fuzzing effectiveness, as a good seed can generate potential mutations to reach deeper paths and uncover vulnerabilities. In smart contract fuzzing, seed generation involves two primary components: generating transaction arguments and determining transaction sequences.

• *Transaction arguments generation.* When generating transaction arguments, fuzzers [66, 76, 96] built on general fuzzers (e.g., AFL [1]) treat each argument as a byte stream and generate random bytes as inputs. Hence, they often require additional time to hit argument types (e.g., string). In contrast, Type-aware fuzzers (e.g., echidna [2]) extract parameter types from contracts' ABI specification, enabling them to generate values accordingly. For fixed-length arguments (e.g., uint256), they produce random values based on the argument's type. For non-fixed-length arguments (e.g., string), they typically generate a corresponding number of random elements with a random length.

In addition to randomly generating inputs, some fuzzers [55, 109] also utilize inputs from real-world transactions, as these inputs are more likely to trigger behaviors that mirror actual use cases. Moreover, ILF [48] utilizes values obtained from symbolic execution as its arguments, while ETHPLOIT [103] gathers inputs and outputs of complex functions (e.g., cryptography functions), along with magic numbers in function bodies. The rationale behind these approaches is that such constants may have the potential to satisfy certain branch conditions. However, inputs from unrelated contracts may not be effective in satisfying branch conditions in the target contract. To address this issue, Confuzzius [91] and Beak [104] leverage symbolic analysis to determine input values when fuzzing stalls in certain branches. Similarly, Smartian [40] leverages concolic testing, which combines concrete and symbolic execution to generate arguments that satisfy certain constraints.

• *Transaction Sequence Generation.* As previously mentioned, the execution of smart contracts often depends on specific states, which can only be achieved through a sequence of transactions. Therefore, it is crucial to create meaningful transaction sequences in seed [97]. Many fuzzers [46, 55, 76, 93, 96, 99, 108] simply generate random transaction sequences as inputs. To ensure all desired functions are tested, when generating sequences randomly, a common practice is to enumerate these functions in initial seeds, as done by SynTest-Solidity [75], ConFuzzius [91], and sFuzz [74]. Although the random strategy can produce diverse sequences and may yield unexpected results, it is not effective in finding critical sequences due to the vast search space of all possible sequences.

Several fuzzers employ machine learning (ML) techniques to generate high-quality transaction sequences, e.g., RLF [88], xFuzz [102], and ILF [48]. ILF learns sequences from coverage-guided symbolic execution, inheriting the ability to produce high-coverage inputs with reduced overhead compared to pure symbolic execution [25]. xFuzz utilizes an ML model trained on contracts, where vulnerable functions are labeled using static analyzers (e.g., Slither [42]). The trained model retains suspicious functions, effectively reducing

Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou

Table 1: Summary of technical details for existing smart contract fuzzers.

Name	year	OC	G/M	Type	Ran	Dict	Prev	ML	Sym	Ran	Dyn	Static	ML	Args	Env	Seq	Cov	Dis	Bug	SE
				Arguments Generation						**Sequence Generation**				**Seed Mutation**			**Seed Scheduling**			
ContractFuzzer [55]	2018	✓	G	✓	✓	✓				✓										
Reguard [66]	2018		M		✓					✓				✓			✓			
ILF [48]	2019	✓	G	✓	✓			✓					✓	✓	✓					
SoliAudit [65]	2019	✓	G	✓	✓					✓				✓	✓					
ContraMaster [93]	2019	✓	M	✓	✓		✓			✓	✓			✓	✓	✓	✓			
Harvey [99]	2020		M		✓		✓			✓				✓		✓	✓	✓		
ETHPlOIT [103]	2020	✓	M	✓	✓		✓					✓		✓	✓		✓			
sFuzz [74]	2020	✓	M	✓	✓					✓				✓	✓		✓	✓		
Echidna [46]	2020	✓	M	✓	✓	✓	✓			✓				✓			✓			
GasFuzzer [22]	2020		M	✓	✓	✓				✓				✓						
Targy [53]	2021		M	✓	✓					✓					✓		✓	✓		
Smartian [40]	2021	✓	M	✓	✓				✓		✓	✓		✓	✓	✓	✓			
SmartGift [109]	2021	✓	G	✓	✓			✓		✓										
ConFuzzius [91]	2021	✓	M	✓	✓	✓			✓	✓	✓			✓	✓	✓	✓			
Beak [104]	2022		M	✓	✓	✓			✓			✓		✓	✓	✓		✓		
xFuzz [102]	2022	✓	M	✓	✓							✓	✓	✓	✓		✓	✓		
EtherFuzz [95]	2022		M	✓	✓	✓				✓				✓	✓	✓	✓			
SynTest-S [75]	2022	✓	M	✓	✓					✓				✓		✓		✓		
RLF [88]	2022	✓	G	✓	✓								✓	✓	✓		✓		✓	
effuzz [54]	2023		M	✓	✓					✓				✓	✓		✓	✓		
IR-fuzz [67]	2023	✓	M	✓	✓							✓		✓	✓	✓	✓			
EF\CF [83]	2023	✓	M	✓		✓				✓				✓	✓	✓	✓			✓
ityfuzz [87]	2023	✓	M	✓	✓					✓				✓	✓	✓	✓	✓		✓

In the **Tool** column, OC: open-sourced; In the **G/M** column, G: generation-based, M: mutation-based; In the **Arguments Generation** column, Type: type-awareness, Ran: randomly generated arguments, Dict: arguments from a pre-prepared dictionary, Prev: values previously encountered or used, ML: arguments obtained through machine learning, Sym: arguments obtained via symbolic execution

In the **Sequence Generation** column, Ran: randomly generated transaction sequences, Dyn: sequences generated through analyzing dynamic data flow, Static: sequences generated by analyzing static data flow, ML: sequences generated through machine learning; In the **Seed Mutation** column, Args: mutating transaction arguments, Env: mutating environment properties, Seq: mutating transaction sequences; In the **Seed Scheduling** column, Cov: fitness by coverage, Dis: fitness by distance, Bug: fitness by vulnerability; The **SE** column indicates whether a standalone EVM is used.

the search space by excluding benign functions. RLF categorizes contract functions into clusters based on the functionalities they provide, simplifying the function space. To generate meaningful transaction sequences, it applies deep Q-learning [30], leveraging both previous experiences and current contract state to determine the appropriate cluster from which the next function is randomly selected. Although these fuzzers may generate effective sequences, their results can be non-deterministic due to the challenges of generalizing to unseen contracts [40].

Moreover, some fuzzers incorporate data flow analysis to predict feasible transaction sequences. For instance, ETHPLOIT employs static taint analysis. It labels the read state variables and function arguments as taint sources, while written state variables and external calls as taint sinks. When generating transaction sequences, it selects candidate functions that can extend the taint propagation. This approach allows it to generate the seeds that reflect the data flow of real-world attack scenarios, potentially exploitable by attackers. Similarly, IR-Fuzz [67] and Beak examine read-after-write (RAW) dependencies of global variables between functions through static analysis, while ConFuzzius tracks dynamic data flow on state variables to identify RAW dependencies for arranging transaction orders. Besides RAW dependencies, Smartian uncovers use-def relationships of state variables using static analysis, helping explore relevant execution paths in contracts.

3.2 Seed Scheduling

Effective seed scheduling is essential for smart contract fuzzing, as it determines which inputs will be used to guide the exploration of the contract's state space. To assess the quality of seeds, fitness metrics such as code coverage are employed. Seeds with higher fitness values are deemed more valuable, warranting a higher allocation of fuzzing budget. Next, we will delve into the different fitness metrics employed by smart contract fuzzers to optimize seed scheduling.

• *Fitness by Coverage.* Code coverage has long been a cornerstone metric in traditional fuzzing, and many smart contract fuzzers [40, 66, 74, 76, 88, 93, 95, 96, 102] have adopted it to prioritize seeds that can potentially uncover new code paths. Typically, these fuzzers calculate code coverage based on executed instructions and covered basic blocks, while some [40, 91] adopt a more fine-grained approach by calculating the covered branches. Covered branches provide a deeper understanding of a contract's behavior by examining its execution paths [17]. In addition to code coverage, Smartian [40] also incorporates data-flow coverage. This metric evaluates the dynamic data flows between state variables during the fuzzing process, providing insight into how effectively the seed exercises the contract's state variables. Ityfuzz [87] tracks the value of state variables, identifying them as "interesting" if they are written with a previously unseen value. Both Smartian and ityfuzz adopt these strategies as they deem a smart contract's unique state vital for future exploration.

• *Fitness by Distance.* Distance is also a popular fitness metric, which helps identify seeds that are more likely to reach unexplored areas. Branch distance [53, 67, 74, 75, 87, 99, 104] and code distance [104] are two commonly used distance metrics in smart contract fuzzing, Branch distance evaluates how close a seed is to satisfy a missed branch by assessing the proximity of the seed to the branch condition. For example, in the code below, if the value of msg.value in a seed is 5, its branch distance for the then-branch is 95 (100-5).

```
1  function test() public {
2    if (msg.value >= 100) { /**then*/}}
```

This example highlights how branch distance provides more fine-grained feedback. When using code coverage, the seeds with msg.value = 99 and msg.value = 5 are considered equivalent. However, with branch distance, fuzzers prioritize the seed with msg.value = 99, as it is more likely to satisfy the conditions after mutation. Instead of branch distance, Beak [104] evaluates seeds

using code distance, which calculates the distance of seed to uncovered code in the control-flow graph. Hence, a seed with a shorter code distance indicates that new paths can be reached more easily through mutation.

Once distance metrics are obtained, fuzzers need to select the next seed for mutation based on distance fitness. Some fuzzers like IR-Fuzz [67] prioritize seeds by taking the minimum distances, while others use more sophisticated methods that consider different objectives or optimization algorithms. For example, SynTest-Solidity [75] employs a many-objective optimization algorithm, DynaMOSA [77], to promote inputs that get closer to the yet-uncovered branches and lines. Similarly, sFuzz [74] combines two complementary strategies: keeping seeds that increase code coverage and selecting the best seed for each just-missed branch with the closest distance. Beak [104] sorts seeds by their distances and allocates energy using a simulated annealing algorithm [84].

• *Fitness by Vulnerability.* RLF [88] prioritizes seeds based on their potential to uncover vulnerabilities. It counts the number of detected vulnerabilities to prioritize the seed that contains vulnerable function-call sequences. However, relying solely on vulnerability fitness might lead to local optima traps due to the sparsity of bugs. Therefore, RLF combines vulnerability fitness with code coverage when scheduling seeds.

• *Other Fitness.* Smart contract fuzzers have adopted various other fitness metrics beyond those previously discussed. For example, Harvey [99] employs the Markov Chain-based scheduling strategy, which assigns more fuzzing budget to seeds that traverse the rare paths. The rationale is that rare paths require more resources because they are harder to reach than easily explored paths. In ConFuzzius, the number of storage writes performed by the seed is also taken into account when calculating its fitness. The intuition is that in the subsequent crossover, a seed with more write operations may exhibit a higher probability of forming useful transaction sequences (i.e. sequences with RAW dependencies).

3.3 Seed Mutation

Seed mutation also plays a crucial role in fuzzing, enabling the generation of extensive test inputs by modifying existing seeds randomly or heuristically. Smart contract fuzzers can perform seed mutation at three levels: transaction argument level, environment properties level, transaction sequence level.

• *Transaction argument mutation.* This level of seed mutation focuses on modifying the arguments of each transaction (i.e., the invoked function). Generation-based fuzzers [2, 48, 55, 65, 85, 89, 109] have no seed mutation process as they do not use feedback to improve their seeds. Some fuzzers [22, 74, 76, 93, 95, 108] utilize mutation operators found in general fuzzers (e.g., AFL), such as bit flips and additions, to generate new test inputs (For variable-length arguments, these fuzzers mutate them by pruning or padding bits). However, many fuzzers mutate all arguments in each mutation, which may lose previously satisfied conditions [68, 81, 101]. For example, the function below has multiple conditional statements.

```
1  function test(int256 a, int256 b, int256 c) public {
2    if (a < 5) {
3      require (b > 10);
4      /**bug;**/}}
```

If the current input is (4,5,0), the execution gets stuck at Line 3. Mutating all three arguments (e.g., to (11,22,0)) causes previously

satisfied conditions (e.g., Line 2) to be unsatisfied, preventing fuzzer from exploring deeper branches. Moreover, mutating all arguments at once makes it hard to identify which argument change led to a new path. To address this issue, Harvey [99] chooses to mutate only a single argument at a time. Similarly, Targy [53] and effuzz [54] use taint analysis to find the arguments that are relevant to the target conditional branch and mutate only those arguments (e.g., argument b in example).

Traditional mutation operators lack insight into the input's semantics, resulting in syntactically correct but semantically invalid mutations [80, 94].Hence, some fuzzers [2, 93, 103] opportunistically replace some arguments with values from a mutation pool. The pool contains interesting or effective inputs that have been discovered during the fuzzing process, which can be reused to enhance the effectiveness of fuzzing.

• *Environment Properties Mutation.* The execution of smart contract can also be affected by environmental properties. For example, in Line 9 of Figure 1, the bug path can only be exercised when the timestamp is greater than 30. Therefore, many fuzzers mutate environmental properties to trigger more contract behaviors. The sender address is commonly mutated by fuzzers, as contract usage rights can vary according to account permissions. Smartian [40] employs bit-flipping to mutate the sender address. Most fuzzers [65, 74, 83], select the sender address from a predefined account set, including roles like creator, administrator, user, and attacker. Gas limits and call return values are also targeted for mutation by some fuzzers [91, 93, 103]. Modifying gas limits enables the exploration of out-of-gas exception-related behaviors, while changing contract call return values simulates diverse outcomes of called contracts, which helps identify unhandled exceptions within the contract under test. Overall, ConFuzzius [91] stands out by mutating a more comprehensive set of environmental properties compared to others. This includes sender address, amounts, gas limits, timestamps, block numbers, size of external code and contract call return values. Similar to transaction argument mutation, the mutation of environment properties is mainly achieved by replacing them with a random value or reusing a previously observed value. Some tools such as sFuzz, treat them as byte streams and modify the individual bits.

• *Transaction sequence mutation.* This mutation generates more diverse transaction sequences to change contract's states, potentially unlocking branches guarded by them. Smartian defines three mutation operations: adding a random function, pruning a function, and swapping two functions. In addition, fuzzers like [53, 74, 75] also employ a crossover operation to evolve transaction sequences. Crossover merges parts of two existing inputs to produce a new input, allowing the new input to inherit "good genes".

Beak and ConFuzzius append transactions that read from a specific storage slot to transactions that write to the same slot. This approach preserves read-after-write (RAW) dependencies and generates more reliable sequences. Similarly, ContractMaster switches the order of two functions if they operate on the same state variable, potentially satisfying RAW dependencies. Harvey proposes a unique approach. It first fuzzes the persistent states to find the functions whose coverage is affected by them. Then, it prepends transactions that can modify the persistent state to those functions.

By focusing on functions that are most sensitive to changes of contract's states, Harvey enables a more thorough exploration. Pivoting from these methods, ityfuzz notes the high cost of re-executing previous transactions (in seed) to build up previous states. To address this, it directly snapshots the state and then explores it using random transactions, thus enhancing the efficiency.

3.4 Engineering Optimization

Smart contract fuzzers have made various engineering efforts to improve overall fuzzing performance. One approach is contract transformation, where fuzzers [54, 66, 76, 96] transform smart contract into native C++/Go code. This offers several optimizations: First, the C++ compiler inlines opcode handlers in the produced native code, eliminating the interpreter loop and minimizing call overhead. Second, the transformation reduces EVM stack operations, and it reduces the overhead caused by bounds and stack-overflow checking. Furthermore, by converting the contracts to C++, fuzzers can leverage the mature ecosystem of fuzzing and program analysis tools. This not only saves engineering effort but also allows for the reuse of these well-refined and efficient tools (e.g., AFL, LibFuzzer [4]). However, it is important to note that, except for EF\CF [83], which pairs the translated C++ program with a C++ variant of EVM, these fuzzers do not use the EVM runtime as the execution environment. Consequently, they may not accurately reflect contract behavior, potentially leading to FPs or FNs.

Moreover, the majority of fuzzers [40, 74, 91] simulate only necessary components of the blockchain environment that are relevant to running smart contracts. This choice reduces the cost of expensive operations, such as mining new blocks and validating transactions. Taking it a step further, some fuzzers [83, 87] adopt standalone EVMs, which focus solely on executing contract code and don't need to handle other tasks. Another optimization is the shift towards offline program analysis or vulnerability detection, e.g., sFuzz [74], which conducts vulnerability detection offline once every 500 test cases. Finally, some fuzzers employ more efficient programming languages like C/C++ to improve their execution speed [54, 74, 76, 102].

4 VULNERABILITY DETECTION

4.1 Test Oracles

Test oracles are critical components in detecting vulnerabilities during smart contracts fuzzing. As smart contracts do not crash in case of vulnerabilities and many vulnerabilities are related to business logic, their detection can be challenging. Existing work relies on test oracles to detect smart contract vulnerabilities. There are two main types of test oracles used in smart contract fuzzing: rule-based oracles and general-purpose oracles.

• **Rule-based oracle.** Rule-based oracles are widely used by smart contract fuzzers [40, 55, 75, 89, 91, 93, 102, 103]. These oracles use pre-defined rules and patterns to analyze the execution trace of contract (e.g., opcode sequences), to identify potential vulnerabilities. For example, to detect "Integer Overflow," rule-based oracles analyze the trace to look for the data flow from the result of an arithmetic overflow to the CALL instruction.

To implement rule-based oracles, most fuzzers hook into EVM to log necessary information, like executed opcodes, values of the

stack, and program counter. This collected data is used to check the relevant rules. Echidna and Harvey require developers to write the oracles into contract's source code. Therefore, their detection capability largely relies on the quality of oracles written by developers.
• **General-purpose oracle.** General-purpose oracles rely on high-level invariants to check the correctness of contract operations. These invariants are properties that hold true for all valid executions of the contract. For example, ContraMaster [93] establishes invariants over the contract balance and the sum of user balances to check the correctness of transfer-related operations. If a transaction that transfers ether fails but the exception is not handled, an inconsistency will emerge between the recorded amount and the actual amount in contract, thus detecting the exception disorder bug. However, Zhang et al. [105] point out that many modern DeFi projects use complex business models that go beyond the proposed invariants in ContraMaster. For instance, many lending projects typically involve multiple assets and the total balances of a single asset can be volatile.

Table 2: Common smart contract vulnerabilities.

Bug Name	Description
Dangerous Delegatecall (**DD**)	Contract uses delegatecall() to execute an untrusted code.
Block State Dependency (**BD**)	Contract uses Block states (e.g., timestamp, number) to decide a critical operation (e.g., ether transfer).
Freezing Ether (**FE**)	Contract has no function for sending Ether, or it allows unauthorized use of contract self destruction.
Ether Leak (**EL**)	Contract allows arbitrary users to retrieve ether from the contract.
Gasless Send (**GS**)	Contract mishandles out-of-gas exceptions when transferring the ether, the attackers may keep the untransferred assets.
Unhandled Exception (**UE**)	Contract doesn't check for exception after calling external functions.
Reentrancy (**RE**)	Contract doesn't update states (e.g., balance) before making an external call, the malicious callee reenters it and leads to a race condition on the state.
Suicidal (**SC**)	Contract can be destroyed by the arbitrary user through selfdestruct interface because of missing access controls.
Integer Bug (**IB**)	Integer operation exceeds the integer range, it can be harmful when modifying the contract's state variables.

4.2 Vulnerability Types

Our study targets contract layer vulnerabilities, which are the focus of most fuzzers [40, 55]. In this context, we identify nine common vulnerability types, which represent the range of vulnerabilities that current fuzzers (in Table 1) can detect. Table 2 illustrates each of them.

5 EVALUATION

In this section, we present a thorough evaluation of state-of-the-art smart contract fuzzers. Given the distinct design of each fuzzer, it is challenging to evaluate their respective techniques and components in a fine-grained manner. Therefore, we turn to use widely accepted metrics to assess their performance across key dimensions. To provide a comprehensive assessment, we first study the metrics adopted by previous traditional fuzzing [56, 62] and smart contract fuzzing studies [91], and then design five performance metrics that are tailored to smart contract fuzzers, ensuring our metrics encompass all dimensions assessed in existing studies. The five performance metrics we propose for evaluation are as follows:
• **Throughput**: measures the number of transactions a fuzzer can generate and execute per second, reflecting its speed and efficiency in generating tests cases.

Table 3: The number of vulnerabilities detected by fuzzers.

Flaw	ContractFuzzer	ILF	RLF	ConFuzzius	sFuzz	xFuzz	Smartian	SmartGift
DD (29)	TP:8, FP: 8	TP:18, FP: 1	TP:20, FP: 1	TP: 22, FP: 0	TP:**25**, FP: 2	TP:**25**, FP: 2	-	TP:6, FP: 8
BD (317)	TP:39, FP: 28	TP:81, FP: 45	TP: 93, FP: 57	TP: 175, FP: 27	TP: **223**, FP: 32	TP: 184, FP: 27	TP: 102, FP: 9	TP:38, FP: 31
FE (80)	TP:12, FP: 4	TP:48, FP: 0	TP:47, FP: 0	TP:**48**, FP: 0	TP:2, FP: 20	TP:1, FP: 14	-	TP:12, FP: 4
EL (43)	-	TP:32, FP: 67	TP:**38**, FP: 79	TP:37, FP: 56	-	-	TP:26, FP: 84	
GS (122)	TP:17, FP: 5	-	-	-	TP:48, FP: 227	TP:**55**, FP: 247	-	TP:13, FP: 4
UE (188)	TP: 14,FP: 10	TP: 13,FP: 3	TP: 13,FP: 3	TP: **76**,FP: 27	TP: 41,FP: 19	TP: 38,FP: 19	TP: **76**,FP: 23	TP: 13,FP: 6
RE (121)	TP: 6, FP: 11	TP: 48, FP: 9	TP: **54**, FP: 11	TP: 42, FP: 21	TP: 8, FP: 8	TP: 8, FP: 7	TP: 12, FP: 3	TP: 6, FP: 9
SC (22)	-	TP: 18, FP: 20	TP: **20**, FP: 22	TP: 10 , FP: 4	-	-	TP: 8, FP: 0	-
IB (581)	-	-	-	TP: **566**, FP: 169	TP: 129, FP: 106	TP: 112, FP: 80	TP: 413, FP: 182	-

In **Flaw** column, the number (e.g., 29) indicates the count of contracts with each vulnerability type in our benchmark.

-: such bug oracle is not supported by corresponding fuzzers. **TP**: true positives, **FP**: false positives.

- **Detected Bugs**: measures the average number of vulnerabilities detected by the fuzzers, reflecting their ability to identify vulnerabilities in the smart contract.
- **Effectiveness**: measures the speed of fuzzers in finding bugs, indicating how quickly a fuzzer can detect vulnerabilities.
- **Coverage**: measures the contract's code executed during fuzzing, indicating its thoroughness in exploring the contract.
- **Overhead**: measures the system resources consumed by the fuzzer during fuzzing, indicating its resource efficiency. This metric is instructive when users have limited resources.

5.1 Experiment Setup

Tool Selection. We select the tools in Table 1 based on the availability of their source code. Five open-source tools are excluded from our selection. Echidna is excluded due to its property-based nature, which requires testers to manually write property tests within the contracts, demanding a good understanding of contract logic. Additionally, we are unable to execute four tools in our environment. SoliAudit [65] does not offer instructions for execution. IR-fuzz can not be installed using the provided script. SynTest-S [75] and ContraMaster [93] have runtime exceptions. Notably, the same exception persists for SynTest-S even after running the Docker image provided by the authors. Despite our efforts to report these exceptions to authors via email, we have unfortunately not received their responses. Therefore, our experiments focus on the remaining **11** fuzzers (see tools in Figure 3).

Environments. All experiments were executed on a server equipped with an AMD EPYC 7H12 CPU (64 cores and 128 threads) running at 2.60 GHz, 512 GB memory, and operating on Ubuntu 16.04 LTS. To ensure a controlled and reproducible environment, each fuzzer was deployed and executed within an individual Docker container. The containers were allocated with 8 dedicated CPU cores, 16 GB RAM, and 2 GB swap space.

Benchmark. We assembled a benchmark of 2,000 smart contracts from previous research datasets [40, 44, 55, 82, 91] and third-party repositories [43], ensuring a diverse and representative sample. The average number of lines of code was 224 lines, while the average number of functions was 22. To achieve a reliable and impartial evaluation, each fuzzer was subjected to 20 repetitions on the benchmark. Each smart contract was subjected to 30 minutes of testing by each fuzzer.

While building our benchmark, we observed that some contracts in previous datasets were incorrectly labeled. For example, the buggy_41 contract in SolidiFI Dataset [6] was labeled as "Integer Bug", but was found to have a "Block State Dependency" bug in Line 76. This issue affects not only the accuracy of our evaluation but also the validity of conclusions. To ensure a precise evaluation, it was necessary to relabel the contracts in our benchmark. First, we preserved their original labels. Then, we applied the 11 fuzzers to the benchmark. We perform manual inspection and relabel two groups of contracts: those reported with a vulnerability different from their original label by two or more fuzzers, and those whose originally labeled vulnerability type was not identified by any of fuzzers in our experiment. This approach was informed by the work of Durieux et al. [41], which suggests aggregating the results of multiple analysis tools can produce more accurate outcomes.

We recruited three volunteers to manually inspect contracts requiring re-labeling. Each volunteer inspected the contracts independently and documented their findings. In cases where inconsistencies arose in the volunteers' results, they engaged in a discussion to reconcile their findings and reach a consensus. This collaborative approach ensured that the relabeled benchmark accurately reflected the ground truth.

5.2 Throughput

We first evaluate the throughput of the fuzzers by measuring the average number of transactions executed per second, as shown in Figure 3. ContractFuzzer and SmartGift exhibit low throughput, generating and executing only 0.09 and 0.07 transactions per second, respectively. This is due to their simulation of the entire blockchain network, a process that continuously appends newly mined blocks (including verification, execution of each transaction, etc.). SmartGift uses an NLP-model for transaction arguments generation, which further lowers its throughput. ETHPLOIT also shows a limited throughput, primarily due to its reliance on runtime program analysis, inducing heavy overhead. RLF, which is built upon ILF, is more efficient in test generation since it produces transaction arguments randomly, while ILF uses complex neural networks to generate the arguments. sFuzz and xFuzz can generate and execute tests at a more rapid pace, achieved through offline
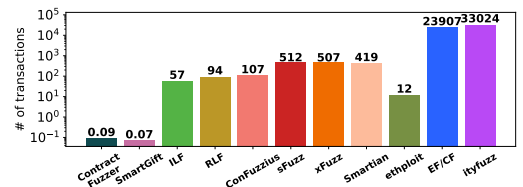
Figure 3: Fuzzers' throughput.

vulnerability detection (i.e., in batches, once every 500 transactions) and their avoidance of program analysis to assist fuzzing. Additionally, their integration of the well-optimized traditional fuzzer, AFL, may also help. Both EF\CF and ityfuzz demonstrate impressive throughput. This is largely due to their adoption of standalone EVMs (e.g., revm [15]), which provide a lightweight and simplified execution environment. Furthermore, they are coded in more efficient languages such as Rust and C++, and their superior code quality may also contribute to this performance.

> **Finding:** This experiment shows that using standalone EVMs can significantly improve the fuzzers' throughput.

5.3 Vulnerabilities Detection Comparison

Table 3 displays the average number of vulnerabilities detected by each tool, each running for 30 minutes per contract, over 20 repetitions. In this experiment, we exclude **three** tools: ETHPLOIT, ityfuzz, and EF\CF. ETHPLOIT offers only vague oracles, such as "Bad Access Control", which cover several vulnerability types in Table 2. Ityfuzz requires testers to manually write test oracles, while EF\CF only has a general oracle that checks if the sender's balance has increased, without specifying the exact vulnerability type. We compare the output of each tool against the labeled dataset (§5.1) to measure their performance. From Table 3, we observe that most fuzzers generate both high false positives and high false negatives, indicating that the state-of-the-art fuzzers are far from satisfactory in vulnerability detection.

False negative: False negatives occur due to several reasons. Firstly, some vulnerable paths remain unexplored, protected by hard-to-satisfy branch constraints. Second, many fuzzers' test oracles are problematic (see Table 4). For instance, when detecting "Freezing Ether," ConFuzzius doesn't filter out CALL instructions introduced by swarm source contract bytecode [10], causing it to miss vulnerabilities in contracts that have no ether transfer function. Some fuzzers' oracles are overly strict and fail to cover all vulnerable contract behaviors. One example is Smartian, which only reports "Freezing Ether" if a contract has no ether transfer instructions but contains a delegatecall that can destroy it. Moreover, we find some contracts cannot be executed by fuzzers due to various issues: 1) coding errors in tools (e.g., xFuzz), 2) compatibility problems with outdated Truffle (which is used to deploy contracts), 3) outdated EVM versions that don't support new Solidity instructions, and 4) outdated Solc compilers unable to process contracts written in newer Solidity versions.

False positive: The main cause of false positives is problematic test oracles (see Table 4). Some test oracles are too broad, capturing non-vulnerable cases. For instance, when detecting "Reentrancy", ILF and RLF only examine the existence of an external call followed by a storage operation in the traces, neglecting the data dependency between them. "Integer Under/Overflow" is often falsely reported by most fuzzers, as their oracles only syntactically check for data flow between arithmetic operations and store instructions, ignoring the actual effects on contracts (as there can be some arithmetic operations introduced by the masking operations from the compiler or compiler's optimization). Furthermore, all these fuzzers fail to

consider the context of contracts (i.e., their actual semantics), such as existing access control mechanisms in place. This may produce false positives since they might flag a behavior as vulnerable even it is not exploitable in practice. Additionally, lack of context prevents fuzzers from understanding contracts' higher-level intentions or purpose. Consequently, they struggle to differentiate between the contract's intended behavior and actual malicious one. For example, fuzzers may incorrectly label lottery contracts [23] as "Ether Leak".

> **Finding:** Existing contract fuzzers are far from satisfactory in terms of vulnerability detection. Their adoption of overly generalized or overly specific test oracles contributes to high false positives and negatives.
>
> **Call to action: Refine test oracles.** To improve vulnerability detection, fuzzers should implement precise, comprehensive rules in oracles, rooted in the nature of vulnerabilities. This involves discerning vulnerability indicators and impacts. The incorporation of data flow analysis and machine learning (trained on vulnerability datasets) can further enhance oracles. Furthermore, ongoing effort is needed to expand understanding of vulnerabilities, uncover new attack vectors, and advance test oracles.

5.4 Speed of Vulnerability Detection

We evaluate the average time taken by each tool to detect a vulnerability (true positive). Figure 4 (box plot on the left Y-axis) presents the results, for most fuzzers, they can detect a true vulnerability within 1 second. ConFuzzius, with its smaller upper outliers, benefits from symbolic taint analysis and data dependency analysis, expediting the discovery of vulnerable paths in certain cases.

To fairly compare fuzzing strategies of different fuzzers, irrespective of their throughput, we compute the number of transactions each fuzzer requires to find a vulnerability. This is achieved by multiplying the median time from the box plot in Figure 4 by the throughput in Figure 3. We use the median rather than average, as it is typically more representative, reducing the impact of extreme outliers. The result is shown by the line graph in Figure 4 (on the right Y-axis). The last three, sFuzz, xFuzz, and Smartian, require more transactions. Because sFuzz and xFuzz batch the vulnerability detection (roughly 500 transactions, this count may vary due to sensitive instructions present). Similarly, Smartian also experiences a detection delay. For example, in the case of "Reentrancy", the oracle only triggers to check for CALL instructions in previous transactions after the SSTORE (in current transaction) is executed. For the first five fuzzers, ContractFuzzer and SmartGift which require notably longer times to detect a vulnerability, maintain transaction counts for actual detection comparable to other three. This suggests that the speed of vulnerability detection is largely dependent on fuzzers' throughput.

> **Finding:** In our experiment, the speed of vulnerability detection is largely influenced by fuzzers' throughput.
>
> **Call to action: Improve throughput.** To speed up vulnerability detection, fuzzers should take actions to increase throughput. This could be achieved by adopting lightweight, standalone EVM frameworks, conducting code optimizations, using efficient languages such as C, incorporating optimized libraries like libAFL.

Table 4: Problematic test oracles in smart contract fuzzers.

Oracle	Issue Description	Fuzzers	Fault
RE	Only check for the presence of external call followed by storage operation, ignoring data dependency.	ILF, RLF	FP
	Only validate if a method can be re-entered but ignore the storage access.	ContractFuzzer, SmartGift, sFuzz, xFuzz	FP
	Taint analysis over-approximates the possible execution paths.	ConFuzzius	FP
	Check for the use of any state variables in cyclic calls, regardless of whether there are any write operations performed on them.	Smartian	FP
BD	Only check for ether transfer and block state related instructions (e.g., TIMESTAMP), ignoring the dependency between them.	ContractFuzzer,SmartGift, sFuzz, xFuzz	FP
	Consider all block-related information that flows into condition statements as vulnerable.	ConFuzzius	FP
	Fail to consider indirect taint flow propagation, only look for direct taint flow.	ILF, RLF	FN
	Fail to check if the block-related information affects the conditional branch.	Smartian	FN
GS	Roughly classify all external calls with 2300 gaslimit as vulnerable, without checking the occurrence of out-of-gas exception.	sFuzz, xFuzz	FP
	Fail to recognize that the transfer() automatically reverts the program state when there is not enough gas.	ContractFuzzer, SmartGift	FP
DD	Not check if the delegatecall parameters can be controlled by user input.	ContractFuzzer, SmartGift	FP
IB	Only check the flow of arithmetic operations, ignoring their actual effects on the contract.	Smartian, ConFuzzius, sFuzz, xFuzz	FP
	Fail to consider MUL instruction.	sFuzz, xFuzz	FN
EL	Fail to consider the case where the ether transfer can be reverted when the account has no balance.	ConFuzzius	FP
UE	Only check for 'INVALID' in the execution trace, without considering thrown exception with a REVERT instruction.	sFuzz, xFuzz	FN
	Only consider the case where the exception is handled immediately.	ILF, RLF	FP
	Z3 [18] inaccurately solves tainted stack items, which symbolically represent exception-related variables.	ConFuzzius	FP
	Roughly checks for all unused return values, even though not every return value indicates a failed call.	Smartian	FP
FE	Fail to consider cases where a contract can receive ether but has no functions to send ether.	sFuzz, xFuzz	FN
	Fail to consider cases where a contract with a self-destruct operation can lock the ether.	ILF, RLF	FN
	Fail to filter out CALL instructions introduced by the swarm source of bytecode, which are wrongly treated as ether transfer.	ConFuzzius	FN
SC	Incorrectly treats the owner account as an attacker account.	ILF, RLF	FP
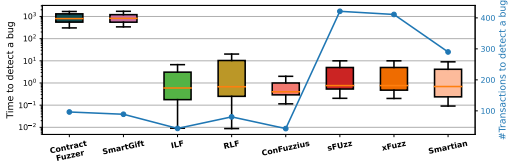
Figure 4: Time to find vulnerability.

5.5 Code Coverage

We next evaluate the code coverage achieved by **11** contract fuzzers. To ensure a consistent comparison, we instrument the EVM uniformly to measure instruction coverage. Figure 5 presents the results. Notably, ityfuzz, Smartian, ILF, and ConFuzzius achieve higher code coverage compared to other tools. This is largely due to their use of coverage-guided approaches in test case generation. In addition, they all mutate the environmental properties during seed mutation, which crucially simulates diverse conditions that trigger different contract behaviors. ILF shows the fastest coverage increase in initial stages, albeit with a limited increment in later stages. This is likely due to its high-quality initial seed. It pre-trains a model from coverage-guided symbolic execution expert for generating transaction sequences and arguments. ityfuzz also exhibits rapid initial coverage growth, due to its impressive throughput. EF\CF starts with a slow increase in coverage, as it begins with a colorization stage [21]. In contrast, ContractFuzzer and SmartGift exhibit the lowest coverage due to two main factors. First, their throughput is markedly slow (§5.2). Second, as black-box fuzzers, they do not use feedback to guide test generation. Although SmartGift utilizes practical inputs from similar functions in the real world, its ML approach suffers from generalization issues (i.e., many test functions are not well-represented in the training set).

The code coverage in some fuzzers is lowered due to their focus only on potentially vulnerable functions during test generation. For example, sFuzz, ETHPLOIT, and RLF exclude 'View' and 'Pure' functions in contracts that do not modify state variables. The rationale behind this decision is that these functions are unlikely to be exploited by attackers since they have no impact on the contract states [59]. Similarly, xFuzz filters out benign functions using an ML model to reduce the search space of function call combinations.

In some cases, all these fuzzers struggle to achieve effective coverage. Hard constraints in complex contracts, like those relying on a 'magic' number, are difficult to satisfy. Even fuzzers incorporating symbolic execution cannot solve these constraints due to the path explosion. Additionally, reaching contract states that require long transaction sequences remains challenging. While some fuzzers analyze data flow between transactions (e.g., RAW) to arrange their sequences, they have difficulty determining transaction dependencies due to potential circular dependencies between state variables.

Finding: The quality of initial seeds and fuzzers' throughput are pivotal for quickly reaching high coverage.
Call to action:
● **Enhancing initial seeds.** To enhance coverage effectiveness, fuzzers should use techniques such as program analysis and machine learning to build high-quality initial seeds, which in turn, generate more effective and meaningful test inputs.
● **Optimize mutation scheduling.** Currently, all existing fuzzers randomly choose mutations from a list of mutational operators, which may waste fuzzing iterations on ineffective mutations. Recent research in traditional fuzzing [52, 98] highlights the pivotal role mutation scheduling plays in enhancing coverage effectiveness. Therefore, future fuzzers should consider adopting advanced methods such as genetic algorithms or differential evolution algorithms to improve mutation scheduling.

5.6 Overhead

We next assess the overhead of each fuzzer. We use the "docker stats" command to monitor the resource usage in the Docker containers. Figure 6 reports the result. Overall, ConFuzzius exhibits the lowest overhead, as it uses PyEVM, which is a lightweight implementation
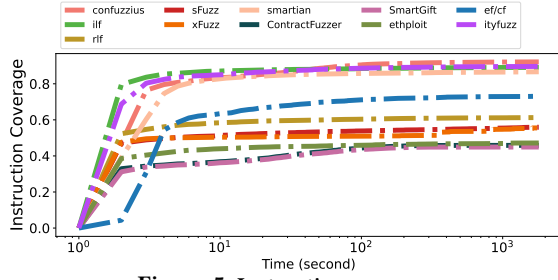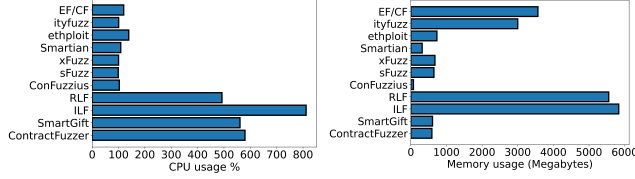
Figure 5: Instruction coverage.

(a) CPU Consumption.   **(b) Memory Consumption.**

Figure 6: Overhead.

of EVM. SmartGift and ContractFuzzer exhibit high CPU usage, primarily because they simulate the entire blockchain network, which can be very CPU-intensive. ILF employs complex neural network models for both transaction argument and sequence generation, which requires significant computational resources. Since the neural network used in ILF is more complex than the DNN used in RLF, more processing power is needed. The use of neural networks also results in both RLF and ILF consuming more memory, with an average memory consumption of 5.83 GB. This is due to the large number of parameters involved in neural networks that need to be stored in memory.

> **Conclusion:** ConFuzzius stands out in most key metrics. It identifies the most vulnerabilities and does so at the highest speed, achieving top-tier coverage and maintaining manageable overhead. However, there's still room for ConFuzzius to improve. For example, using a standalone EVM could boost its throughput, and refining its test oracle could elevate the detection accuracy.

6 INDUSTRIAL PERSPECTIVES

6.1 Fuzzing Tools Used by Auditing Companies

Despite recent research [32, 105] suggesting that academic fuzzers fall short in detecting real-world contract vulnerabilities, we find that the industry continues to rely on fuzzing as an important component of contract auditing [13]. To better understand the practical usage of existing smart contract fuzzers in the industry, we explore their use by various auditing companies. Specifically, we collect public audit reports from their respective GitHub repositories. The companies considered are drawn from a list of notable Blockchain Security auditing companies [14]. By scanning these reports, we identify the fuzzers employed in their auditing process. We also manually read the auditing methodology sections on their official websites to extend our knowledge of the employed fuzzers. The result is reported in Table 5. Echidna (a product of Trail of Bits [16]) is the most favored tool among these companies. Conversely, most of the fuzzers listed in Table 1 from academic research are not actively employed in real-world security audits. This highlights the gap

between theoretical research advancements and the actual needs for security auditing in the industry.

Table 5: Fuzzing tools used by different auditing companies.

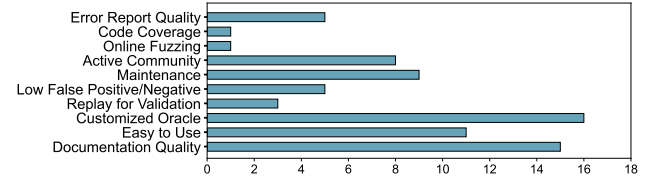| Tools	Auditing Company
Echidna [2]	ImmuneBytes, Halborn, Trail of Bits, QuillAudits, Solidified, Pessimistic, ChainSafe, yAcademy, yAudit, Truscova, Zellic, Zokyo, Cyfrin, ABDK.
sfuzz [74]	ImmuneBytes.

Figure 7: Essential factors in fuzzer selection by auditors; x-axis represents the number of participants.

6.2 Survey on Auditors

To further understand the practical needs for fuzzers, we conducted an anonymous online survey. We recruited 16 participants via our professional contacts within the community, who averaged 1.68 years of experience in smart contract review. Of these, 11 were solo auditors, with the rest working in three auditing companies. Our survey primarily questioned the participants about the fuzzers they employ during auditing, their reason for these choices, and the essential factors they consider when using fuzzers. Due to space constraints, the complete survey questionnaire is accessible in our GitHub repository. The survey result echoed our previous findings: 14 participants use Echidna to aid their audits, two use Foundry [12] which supports fuzzing in its unit tests, and one also uses ityfuzz.

Regarding their preferences, all Echidna users praise its highly conducive support for custom oracles. As Echidna uses property-based testing [46], auditors can write invariants (constants that should perpetually hold true, see 4.1) as customized oracles. This allows them to examine the business logic in their contracts, thereby making Echidna applicable to logic-related vulnerabilities. Upon reviewing the fuzzers listed in Table 1, it's evident that, besides Echidna, only ityfuzz offers substantial support for custom oracles. In contrast, the rest require developers to have a deep understanding of the source code and the ability to modify it. In addition, three participants underscore Echidna's straightforward integration into the audit workflow. This implies that auditors can use it with other tools in their toolkit to extract crucial information prior to and during the fuzzing campaign. For instance, it can be readily paired with Slither [42], a highly recognized static analysis tool for Solidity contracts (a product from the same company). One participant, a former Echidna user, shifts to ityFuzz due to its support for online fuzzing – a feature currently exclusive to it. This feature is important in detecting real-world vulnerabilities, given its capacity to test smart contracts in a realistic setting compared to offline fuzzing, which starts from a blank contract state. This participant

also notes another advantage of online fuzzing - the ability to detect cross-contract vulnerabilities, as it involves interacting with other deployed contracts, a complexity that offline fuzzing fails to mimic. As for the essential factors in selecting fuzzers, most auditors value ease of use, quality of documentation, oracle customization flexibility, and continuous code maintenance. Detailed results can be seen in Figure 7.

> **Call to action:**
> • **Promoting Custom Oracle Creation.** Fuzzers should facilitate users in creating customized test oracles in a straightforward and flexible manner. This feature allows for the capture of complex business rules, broadening the fuzzer's applicability and leading to more accurate audits.
> • **Support online-fuzzing.** Fuzzers should include online fuzzing features for testing in real settings, facilitating cross-contract vulnerabilities detection and increasing chances of discovering real exploits.
> • **Emphasize Usability.** Fuzzers should uphold the quality of their documentation and provide clear tutorials for quick user onboarding. Furthermore, an intuitive interface that provides comprehensive information is also crucial. It can aid users monitoring the fuzzing process and better interpret the results.
> • **Build Strong Community.** Establishing a strong community can create a valuable space for users to ask questions, share knowledge, and address encountered issues.
> • **Implement Replay Features.** Fuzzer should incorporate exploits replay, as seen in ityFuzz, the only tool supporting replay currently. The replay feature can reproduce issues, eliminate false positives, and validate fixes post bug repairs.

7 THREAT TO VALIDITY

The main threat to external validity in this study is related to the benchmark. Firstly, the process of re-labeling contracts involves manual effort, which may introduce potential subjectivity and bias. Additionally, in the re-labeling process, some bugs in contracts may not be identified by all fuzzers and thus, could be missed by our volunteers during manual inspection. Second, the benchmark we used is based on contracts collected from previous works, which may not reflect the diversity and complexity of real-world contracts. Therefore, the generalizability of our findings may be constrained. However, the results can help researchers gain a good understanding of the limitations of existing works and motivate researchers to develop more advanced tools. In our future work, we will expand our benchmark to include contracts used in production. Additionally, we will use more diverse sources to collect contracts, such as bug bounty programs, security audits.

Another threat to external validity is that, while different fuzzers adopt different techniques, we only evaluated their overall effectiveness, without evaluating the importance of their main components at a more fine-grained level. This could lead to a lack of understanding of how the unique features of each fuzzer contribute to their overall performance. In our future work, we will conduct ablation experiments to demonstrate the effects of different unique features of the fuzzers.

Lastly, we recognize that our literature review may not be exhaustive. However, our study represents the most comprehensive investigation of smart contract fuzzers to date. Our future work will systematically compare newly released fuzzers on different blockchain platforms or contract languages.

8 CONCLUSION

In this paper, we conduct a thorough investigation of existing smart contract fuzzing techniques through a literature review and empirical evaluation. We assess the usability of 11 state-of-the-art smart contract fuzzers using a carefully-labeled benchmark and comprehensive performance metrics. Our evaluation identifies some notable issues and suggests possible directions for future fuzzers. We believe our research offers the community valuable insights, inspiring future breakthroughs in this field.

ACKNOWLEDGMENT

REFERENCES

[1] 2022. American fuzzy lop. https://lcamtuf.coredump.cx/afl/
[2] 2022. Ethereum smart contract fuzzer. https://github.com/crytic/echidna
[3] 2022. Go Ethereum. https://geth.ethereum.org/
[4] 2022. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html
[5] 2022. Real-World Use Cases for Smart Contracts and dApps. https://www.gemini.com/cryptopedia/smart-contract-examples-smart-contract-use-cases/
[6] 2022. SolidiFI Benchmark. https://github.com/DependableSystemsLab/SolidiFI-benchmark
[7] 2022. Solidity Programming Language. https://soliditylang.org/.
[8] 2022. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. https://www.coindesk.com/markets/2016/06/17/the-dao-attacked-code-issue-leads-to-60-million-ether-theft/.
[9] 2022. Welcome to Ethereum. https://ethereum.org/en/.
[10] 2022. What is the cryptic part at the end of a solidity contract bytecode? https://ethereum.stackexchange.com/questions/23525/what-is-the-cryptic-part-at-the-end-of-a-solidity-contract-bytecode.
[11] 2023. Ethereum Market Cap. https://ycharts.com/indicators/ethereum_market_cap
[12] 2023. Foundry is a blazing fast, portable and modular toolkit for Ethereum application development written in Rust. https://github.com/foundry-rs/foundry
[13] 2023. Fuzz it. https://consensys.io/diligence/fuzzing/
[14] 2023. A list of notable Blockchain Security audit companies and location of public audits. https://github.com/0xNazgul/Blockchain-Security-Audit-List
[15] 2023. Rust Ethereum virtual machine (revm) Is EVM written in rust that is focused on speed and simplicity. https://github.com/bluealloy/revm
[16] 2023. Trail of Bits. https://www.trailofbits.com/
[17] 2023. What Is Branch Coverage and What Does It Really Tell You? https://linearb.io/blog/what-is-branch-coverage/
[18] 2023. Z3 Theorem Prover. https://en.wikipedia.org/wiki/Z3_Theorem_Prover
[19] Tesnim Abdellatif and Kei-Léo Brousmiche. 2018. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5.
[20] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*.
[21] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.
[22] Imran Ashraf, Xiaoxue Ma, Bo Jiang, and Wing Kwong Chan. 2020. GasFuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. *IEEE Access* 8 (2020), 99552–99564.

[23] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer, 164–186.

[24] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. 2018. Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th international conference on software and computer applications*. 322–326.

[25] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. [n. d.]. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* ([n. d.]).

[26] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2011. Finding software vulnerabilities by smart fuzzing. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*.

[27] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*. 91–96.

[28] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[29] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *TSE* 45, 5 (2017), 489–506.

[30] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. [n. d.]. Deep reinforcement fuzzing. In *IEEE SPW*.

[31] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *PLDI*.

[32] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. 2023. Smart contract and defi security: Insights from tool evaluations and practitioner surveys. *arXiv preprint arXiv:2304.02981* (2023).

[33] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[34] 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.

[35] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, et al. 2021. SigRec: Automatic recovery of function signatures in smart contracts. *TSE* (2021).

[36] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, et al. 2019. Dataether: Data exploration framework for ethereum. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1369–1380.

[37] 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 *ESEM*.

[38] Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, and Xiaosong Zhang. 2020. Understanding ethereum via graph analysis. *ACM Transactions on Internet Technology (TOIT)* 20, 2 (2020), 1–32.

[39] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *ACM CCS*.

[40] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.

[41] 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.

[42] Josselin Feist, Gustavo Grieco, and Alex Groce. [n. d.]. Slither: a static analysis framework for smart contracts. In *Proc. IEEE/ACM WETSEB*.

[43] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1349–1352.

[44] 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*. 415–427.

[45] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *OOPSLA* (2018).

[46] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[47] Xiang Guo. 2022. Analysis between different types of smart contract fuzzing. In *2022 3rd International Conference on Computer Vision, Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA)*. IEEE, 882–886.

[48] 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.

[49] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. {EOSAFE}: security analysis of {EOSIO} smart contracts. In *30th USENIX Security Symposium (USENIX Security 21)*. 1271–1288.

[50] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 230–243.

[51] Tian-Yuan HU, ZeCheng Li, BiXin Li, and QiHao BAO. [n. d.]. Contractual Security and Privacy Security of Smart Contract: A System Mapping Study. In *CHINESE JOURNAL OF COMPUTERS*.

[52] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. 2022. DARWIN: Survival of the Fittest Fuzzing Mutators. *arXiv preprint arXiv:2210.11783* (2022).

[53] Songyan Ji, Jian Dong, Junfu Qiu, Bowen Gu, Ye Wang, and Tongqi Wang. 2021. Increasing fuzz testing coverage for smart contracts with dynamic taint analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 243–247.

[54] Songyan Ji, Jin Wu, Junfu Qiu, and Jian Dong. 2023. Effuzz: Efficient fuzzing by directed search for smart contracts. *Information and Software Technology* (2023), 107213.

[55] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 259–269.

[56] 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.

[57] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Ethereum smart contract analysis tools: A systematic review. *IEEE Access* (2022).

[58] Chhagan Lal and Dusica Marijan. 2021. Blockchain testing: Challenges, techniques, and research directions. *arXiv preprint arXiv:2103.10074* (2021).

[59] Bixin Li, Zhenyu Pan, and Tianyuan Hu. 2022. ReDefender: Detecting Reentrancy Vulnerabilities in Smart Contracts Automatically. *IEEE Transactions on Reliability* (2022).

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

[61] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. 2020. A survey on the security of blockchain systems. *Future generation computer systems* 107 (2020), 841–853.

[62] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers.. In *USENIX Security Symposium*. 2777–2794.

[63] Zihao Li, Jianfeng Li, Zheyuan He, Xiapu Luo, Ting Wang, Xiaoze Ni, Wenwu Yang, Chen Xi, and Ting Chen. 2023. Demystifying DeFi MEV Activities in Flashbots Bundle. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[64] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[65] Jian-Wei Liao, Tsung-Ta Tsai, Chia-Kang He, and Chin-Wei Tien. 2019. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 458–465.

[66] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 65–68.

[67] Zhenguang Liu, Peng Qian, Jiaxu Yang, Lingfeng Liu, Xiaojun Xu, Qinming He, and Xiaosong Zhang. 2023. Rethinking Smart Contract Fuzzing: Fuzzing With Invocation Ordering and Important Branch Revisiting. *arXiv preprint arXiv:2301.03943* (2023).

[68] Han-Lin Lu, Guan Ming Lin, and Shih-Kun Huang. 2022. Guan-fuzz: Argument Selection With Mean Shift Clustering for Multi-argument Fuzzing. In *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 421–430.

[69] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. [n. d.]. Making smart contracts smarter. In *ACM CCS*.

[70] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[71] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2021. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4380–4396.

[72] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. [n. d.]. The art, science, and engineering of fuzzing: A survey. *TSE* ([n. d.]).

[73] 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 *ASE*.

[74] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[75] Mitchell Olsthoorn, Dimitri Stallenberg, Arie Van Deursen, and Annibale Panichella. 2022. Syntest-solidity: Automated test case generation and fuzzing for smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 202–206.

[76] Siddhasagar Pani, Harshita Vani Nallagonda, Saumya Prakash, R Vigneswaran, Raveendra Kumar Medicherla, and MA Rajan. 2022. Smart contract fuzzing for enterprises: the language agnostic way. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, 1–6.

[77] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. [n. d.]. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *TSE* ([n. d.]).

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

[79] Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson. 2023. Smart Contract Upgradeability on the Ethereum Blockchain Platform: An Exploratory Study. *arXiv preprint arXiv:2304.06568* (2023).

[80] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596* (2017).

[81] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[82] 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*. 566–579.

[83] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. 2023. EF/CF: High Performance Smart Contract Fuzzing for Exploit Generation. *arXiv preprint arXiv:2304.06341* (2023).

[84] Rob A Rutenbar. [n. d.]. Simulated annealing algorithms: An overview. *Circuits and Devices magazine* ([n. d.]).

[85] Noama Fatima Samreen and Manar H Alalfi. 2020. Reentrancy vulnerability identification in ethereum smart contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 22–29.

[86] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: Attacks and protections. *IEEE Access* 8 (2020), 24416–24427.

[87] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333.

[88] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[89] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[90] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*. 9–16.

[91] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, et al. 2021. CONFUZZIUS: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts.(2021). (2021).

[92] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. Vultron: catching vulnerable smart contracts once and for all. In 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). IEEE, 1ś4 (2019).

[93] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2020. Oracle-supported dynamic exploit generation for smart contracts. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2020), 1795–1809.

[94] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[95] Xiaoyin Wang, Jiaze Sun, Chunyang Hu, Panpan Yu, Bin Zhang, and Donghai Hou. 2022. EtherFuzz: Mutation Fuzzing Smart Contracts for TOD Vulnerability Detection. *Wireless Communications and Mobile Computing* 2022 (2022).

[96] Scott Wesley, Maria Christakis, Jorge A Navas, Richard Trefler, Valentin Wüstholz, and Arie Gurfinkel. 2022. Verifying solidity smart contracts via communication abstraction in SmartACE. In *Verification, Model Checking, and Abstract Interpretation: 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16–18, 2022, Proceedings 23*. Springer, 425–449.

[97] Canghai Wu, Jie Xiong, Huanliang Xiong, Yingding Zhao, and Wenlong Yi. 2022. A Review on Recent Progress of Smart Contract in Blockchain. *IEEE Access* 10 (2022), 50839–50863.

[98] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*. 1634–1645.

[99] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1398–1409.

[100] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).

[101] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.

[102] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2022).

[103] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. 2020. Ethploit: From fuzzing to efficient exploit generation against smart contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 116–126.

[104] Wei Zhang. 2022. Beak: A Directed Hybrid Fuzzer for Smart Contracts. *International Core Journal of Engineering* 8, 4 (2022), 480–498.

[105] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. ICSE.

[106] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. DeepInfer: Deep Type Inference from Smart Contract Bytecode. In *ESEC/FSE*.

[107] Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023. Turn the Rudder: A Beacon of Reentrancy Detection for Smart Contracts on Ethereum. *arXiv preprint arXiv:2303.13770* (2023).

[108] Jianfei Zhou, Tianxing Jiang, Shuwei Song, and Ting Chen. 2022. AntFuzzer: A Grey-Box Fuzzing Framework for EOSIO Smart Contracts. *arXiv preprint arXiv:2211.02652* (2022).

[109] Teng Zhou, Kui Liu, Li Li, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. [n. d.]. SmartGift: Learning to generate practical inputs for testing smart contracts. In *Proc. IEEE ICSME*.

[110] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *CSUR* (2022).

[111] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network.. In *IJCAI*. 3283–3290.