Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.082577 Published Online: 02 July 2026

ARTICLE

# IConFuzz: Constraint-Aware Argument Mutation for Effective Smart Contract Fuzzing

**Hojin Choi and Jaeseung Choi***

Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea

*Corresponding Author: Jaeseung Choi. Email: jschoi22@sogang.ac.kr

Received: 18 March 2026; Accepted: 25 May 2026

**ABSTRACT:** Recently, extensive research has focused on addressing the unique challenges of smart contract fuzzing. Nevertheless, existing fuzzers still struggle to generate adequate function call arguments that can explore the deep smart contract states. In this paper, we introduce novel classes of argument constraints that capture the inter-argument relationships required to exercise meaningful contract logic. We propose a static analysis algorithm to extract these constraints from Solidity source code. In addition, we design a constraint-aware argument mutation strategy that leverages the identified constraints to guide test case generation for smart contract fuzzing. We implement our approach in a fuzzer named IConFuzz. Our evaluation on realistic benchmarks with integer overflow, suicidal contract, and ether leakage vulnerabilities demonstrates that IConFuzz outperforms state-of-the-art testing tools in both the number of bugs discovered and the speed of bug detection.

**KEYWORDS:** Software testing; static analysis; fuzzing; smart contract security

## 1 Introduction

The recent growth of smart contract usage has amplified the risks posed by software vulnerabilities in contract code. Since smart contracts often manage users' digital assets (e.g., tokens), a small flaw in the contract code can cause serious financial losses to users [1,2].

Among the various techniques for vulnerability detection, fuzz testing (fuzzing) [3,4] has been widely employed for smart contract testing due to its empirical effectiveness in exposing bugs. Unlike static analysis, which approximates program behavior without actual executions, fuzzing can generate concrete test cases that reproduce the bugs found. Accordingly, a series of smart contract fuzzers have been proposed to adapt recent advances in grey-box fuzzing to the smart contract domain [5–8].

Smart contracts have distinctive characteristics that set them apart from traditional software, and these properties introduce unique challenges for fuzzing. A smart contract is a stateful program that exposes functions that users can invoke. Therefore, a test case for a smart contract is a *sequence of function calls*. Depending on the order of this sequence, the contract may exhibit different behaviors and states. Consequently, previous research has mainly focused on developing techniques to generate transaction sequences that can explore deep contract states [7–11].

However, our observation reveals that merely ensuring the appropriate transaction order is insufficient for effectively detecting smart contract vulnerabilities. Due to common patterns in smart contract data usage and access control logic, it is also essential to consider the constraints that function call arguments must

satisfy. While various studies have explored methodologies for identifying and leveraging input constraints for fuzzing traditional software [12–14], this problem has been largely overlooked in the context of smart contract fuzzing.

To tackle this challenge, we introduce two types of constraints on function arguments that must be taken into account for effective smart contract fuzzing. First, a *mapping-key constraint* arises when two arguments are used as keys to read from and write to the same mapping-type variable. In this case, the arguments must hold the same value to induce a data-flow by accessing the same entry of the map. Second, a *privilege* constraint captures the relationship between two address-type arguments involved in authorization logic. If one address is designated as a privileged user of the contract and the other argument is checked against it, these argument values must properly align to pass the check.

We propose a *constraint-aware argument mutation* technique that leverages these two types of constraints during test input generation. Specifically, we first statically analyze the contract code to extract these constraints. Then, after applying random mutations, as in traditional fuzzing, we perform a post-processing step to adjust the argument values in accordance with the extracted constraints. By incorporating these constraints, the fuzzer can effectively generate transaction arguments that produce useful data-flows and pass authorization checks.

We implemented our approach in IConFuzz, which **i**dentifies and utilizes argument **con**straints to guide smart contract **fuzz**ing. We evaluated IConFuzz on two realistic benchmarks to demonstrate the effectiveness of the proposed technique. The first benchmark [8] includes 53 smart contracts containing 73 ether leakage vulnerabilities and 21 suicidal contract vulnerabilities. The second benchmark [11] consists of 77 smart contracts collected from CVE reports on integer overflow vulnerabilities. Our evaluation shows that IConFuzz outperforms state-of-the-art testing tools (Smartian, SmarTest, and RLF) in terms of both the number of detected bugs and detection speed.

To summarize, our contributions are as follows:

- We introduce mapping-key constraints and privilege between function arguments, which are key to exercising complex contract logic.
- We design a static analysis to automatically extract mapping-key and privilege constraints from Solidity source code.
- We propose constraint-aware argument mutation, which leverages the identified constraints to guide test case generation.
- We implement our approach in IConFuzz and demonstrate that it outperforms state-of-the-art testing tools.

## 2 Background

### *2.1 Smart Contract and Ethereum Platform*

A smart contract is a program that encodes business logic within its code. When a contract is uploaded to the blockchain, users can interact with the contract, for example by calling one of its functions. When such interaction occurs, the contract code executes according to the specified contractual conditions and updates its state stored in the blockchain.

Ethereum is one of the most popular smart contract platforms nowadays. Ethereum smart contracts are deployed and executed on a runtime environment called Ethereum Virtual Machine (EVM). Once deployed, they are primarily used to automate the exchange and management of digital assets, such as *ether*. Every entity on the blockchain, including the contracts themselves, is represented as an *account* and each account is assigned with a unique address.

To create an Ethereum smart contract, a developer must first write the source code of the contract. In the Ethereum ecosystem, Solidity is used as the primary programming language for writing the contract code. When a user deploys the contract on the blockchain, a contract account is created and assigned an address. Afterwards, other users can also interact with this contract. Here, the user who published the contract is referred to as the *deployer* of the contract.

A smart contract defines a set of functions that users can call. Each function can take arguments as input and use them in the contract's operations. When a user calls a function, the user can also transfer ether to the contract. To retain the data across function calls, Solidity programs can use *state variables*, which are conceptually similar to global variables in C.

Every action of an account, such as deploying a contract or invoking a contract function, is referred to as a *transaction*. In a transaction, all the relevant information about the action is included, such as the address of the transaction sender, function argument values, and the amount of ether being transferred.

### *2.2 Smart Contract Vulnerabilities*

In this paper, we focus on three common types of bugs that have been highlighted as critical vulnerabilities in previous work: integer overflows [6,7,15,16], ether leakage [8,11,17], and suicidal contracts [17,18].

**Integer Overflow.** Integer overflows occur when arithmetic operation results exceed the valid ranges of fixed-size integers. In smart contracts, such errors may cause serious malfunctions of a contract if the overflowed values are used in critical operations such as ether transfer. For conciseness, we will use the term *integer overflow* to refer to both overflow and underflow bugs.

**Ether Leakage.** In general, normal users (i.e., non-deployer) must not be allowed to freely withdraw the ether owned by the contract account. If there is a flaw in the contract code, such as the lack of proper permission checks, a normal user can make the contract transfer its ether to an unintended account.

**Suicidal Contract.** In Solidity, the `selfdestruct(addr)` statement allows a contract to destroy itself and send its remaining ether to the specified address. Under normal circumstances, only privileged users, such as the deployer, should be allowed to halt a contract's execution. Suicidal contract refers to a situation where a contract can be destroyed by a normal user.

We note that starting from Solidity 0.8.0, integer overflow checks are enforced at runtime by default. According to a previous study [19], approximately 10% of contracts available on Etherscan [20] were compiled with versions lower than 0.8.0 at the time of analysis. Despite their reduced prevalence in recent contracts, integer overflows can still occur in legacy contracts or in scenarios where developers explicitly disable overflow checks for gas optimization.

### *2.3 Smart Contract Testing*

Due to the serious security impact of smart contract vulnerabilities, there have been various studies to detect these bugs automatically. In this paper, we focus on testing-based approaches for effective detection of smart contract vulnerabilities.

One of the key challenges in smart contract testing is deciding the order of function calls in a transaction sequence, which corresponds to a test input in smart contract testing. Since a smart contract stores and maintains state variables, its functions may have dependencies with each other. For instance, one function may have to be called before another function in order to observe interesting behaviors of the contract.

Several previous studies have proposed approaches to generate effective transaction sequences for testing smart contracts. Smartian [7] statically analyzes data-flows to construct initial seeds with function call sequences that can write to and read from the same state variable. Then, it uses dynamic data-flow analysis

to evaluate the usefulness of generated test cases during the fuzzing phase. Meanwhile, ConFuzzius [10] directly relies on dynamic data dependency analysis to generate transaction sequences that may result in buggy contract states. SmarTest [11] employs dynamic symbolic execution guided by a statistical language model trained on known vulnerable contracts, which helps prioritize transaction sequences that are likely to trigger bugs. RLF [8] adopts a Markov decision process and reinforcement learning to generate transaction sequences that maximize a reward based on vulnerability detection. ILF [21] learns a machine learning model from symbolic execution results and uses the model to guide transaction sequence construction. xFuzz [22] also employs machine learning but focuses on generating transaction sequences that can trigger vulnerabilities arising from interactions between multiple contracts.

## 3 Motivation and Overview

### *3.1 Motivating Example*

Fig. 1 presents a simplified smart contract that implements a token minting mechanism with access controls enforced through modifiers. The contract tracks the amount of tokens owned by each user with a mapping-type variable, `balances`. Also, `owner` and `mintAgent` store the address of deployer and the address with the minting privilege, respectively. Upon deployment, the constructor initializes `owner` with the sender of the deployment transaction. Accesses to the `setMintAgent` and `mint` functions are protected by the `onlyOwner` and `onlyMintAgent` modifiers, ensuring that only authorized entities can successfully execute them.

```
1  contract Example {
2      address public owner;
3      address public mintAgent; // Initialized to default value, zero
4      mapping (address => uint32) public balances;
5
6      modifier onlyOwner() { require(msg.sender == owner); }
7      modifier onlyMintAgent() {
8          require(mintAgent != address(0) && msg.sender == mintAgent);
9      }
10
11     constructor(){
12         owner = msg.sender;
13         balances[owner] = 4294967295; // 2^32 - 1
14     }
15
16     function setMintAgent(address agent) public onlyOwner {
17         mintAgent = agent;
18     }
19
20     function transfer(address recipient, uint amount) {
21         require(balances[msg.sender] > amount);
22         require(balances[recipient] + amount > balances[recipient]);
23         balances[recipient] = balances[recipient] + amount;
24         balances[msg.sender] = balances[msg.sender] - amount;
25     }
26
27     function mint(address addr) public onlyMintAgent {
28         require(addr != owner);
29         balances[addr] = balances[addr] + 10000;
30     }
31 }
```

**Figure 1:** Motivating example.

The contract has two functions that can modify user balances: `transfer()` and `mint()`. The `transfer` function accepts a recipient and the amount of tokens to transfer as arguments. Meanwhile, the `mint` function allows a designated minting agent to create new tokens and distribute them to the specified user.

This contract contains an integer overflow bug at Line 29, where newly minted tokens are added to the balance of a specified user. While the `transfer` function includes checks to prevent integer overflows, the `mint` function lacks such checks. As long as the minted tokens are not sent to the `owner`, `mint()` increments the balance of `addr`, which may cause an integer overflow.

Despite the simplicity of this example, detecting this vulnerability with testing is not straightforward. First, the dependencies between functions must be considered to generate a transaction sequence that can trigger this bug. As `mint()` uses `mintAgent` for privilege checking, `setMintAgent()` must be called beforehand to set this variable. Also, `transfer()` must be invoked before `mint()` to increase a user's balance to a large value prone to integer overflow.

As discussed in Section 2.3, prior research on smart contract testing [7,8,11,21,22] has addressed this problem by focusing on generating function call sequences that can test smart contracts more effectively. Smartian identifies these function-level interdependencies by analyzing the def-use chains across functions. Meanwhile, SmarTest and RLF rely on machine learning to decide the next function to call during transaction sequence generation.

Unfortunately, however, focusing solely on the sequence of function calls is insufficient for detecting smart contract bugs promptly. In this example, there are two additional constraints that must be satisfied to trigger the integer overflow bug in Line 29. First, the buggy function `mint` must be called from an address that is designated as the minting agent by a prior call to `setMintAgent()`. Second, the `addr` argument of the `mint` function must refer to an address whose balance has been previously updated by calling `transfer()`.

Previous research on determining the transaction order has largely overlooked the importance of considering such *constraints on argument values*. Without careful handling of these constraints, smart contract testing tools may fail to discover the bug in this example effectively. The random mutation strategies employed by existing smart contract fuzzers [7,8] often consume significant time to generate arguments that satisfy these constraints. Tools based on dynamic symbolic execution [11] also struggle to generate a proper argument for `mint()`. Since this constraint does not arise from an explicit conditional statement, path conditions collected during symbolic execution cannot capture it. Note that the conditional statement in `mint()` only compares its argument against `owner`; it does not check whether it is equal to the `recipient` argument value of a previous call to `transfer()`. Therefore, the path condition collected in dynamic symbolic execution cannot encode the constraint that `addr` must satisfy.

### *3.2 Overview of IConFuzz*

In this paper, we present a novel smart contract fuzzing technique that utilizes constraints on transaction arguments during input mutation. Here, we use the term *argument* in a broad sense, treating the transaction sender as a special case of an argument. Fig. 2 illustrates a concrete transaction sequence that can trigger the integer overflow bug in the previous motivating example. The figure also shows the key constraints that must be satisfied between the transaction arguments. The red arrow indicates that the sender of the transaction invoking `mint()` must be equal to the first argument of the preceding call to `setMintAgent()`. We refer to this kind of constraint as a *privilege constraint*, as it is related to conditional statements that check the privilege of the transaction sender. Meanwhile, the blue arrow denotes the constraint that the first argument

of `transfer` and the argument of `mint` should be identical. We refer to this second kind of constraint as a *mapping-key constraint*, since these two arguments are used as keys to access the same mapping-type storage variable.


**Figure 2:** Transaction sequence for the motivating example.

IConFuzz performs static analysis on contract code to identify these two types of constraints by observing how function arguments are used in state variable accesses (see Section 4.2 for details). At a high level, if one argument is used to update an ADDRESS-type state variable and another argument is compared against this updated variable, we conclude that the two arguments are related through a privilege constraint. On the other hand, if two arguments are used as keys for the same mapping-type state variable during a read and a write operation respectively, these arguments are linked by a mapping-key constraint.

The identified constraints are then employed for constraint-aware argument mutation during the fuzzing phase (see Section 4.3 for details). After applying random mutations to argument values, IConFuzz performs post-processing to make the arguments satisfy the identified constraints. By incorporating these constraints, the fuzzer can effectively produce semantically meaningful transaction arguments that are more likely to uncover vulnerabilities in the contract.

Our work makes a unique contribution compared to previous work focusing on data dependencies in smart contracts. Existing studies that leverage dynamic data-flow as fuzzing feedback [7,10] are only able to recognize data-flows in test cases *after* they are generated. In other words, while these techniques can identify data-flows at runtime, they rely on random chance to generate actual argument values that can trigger such data-flows. In contrast, we statically analyze argument constraints and use them *during* test case generation to effectively induce data-flows at runtime. Note that while Smartian [7] also performs static data-flow analysis, its application is limited to deciding transaction orders.

## 4 Methodology

### *4.1 System Architecture*

Fig. 3 shows the system architecture of IConFuzz, which consists of two main phases: static analysis and fuzzing. The static analyzer examines the semantics of smart contract code to obtain information that can enhance the effectiveness of fuzzing. Then, this information is passed to the fuzzing module and used during the generation of new test inputs.

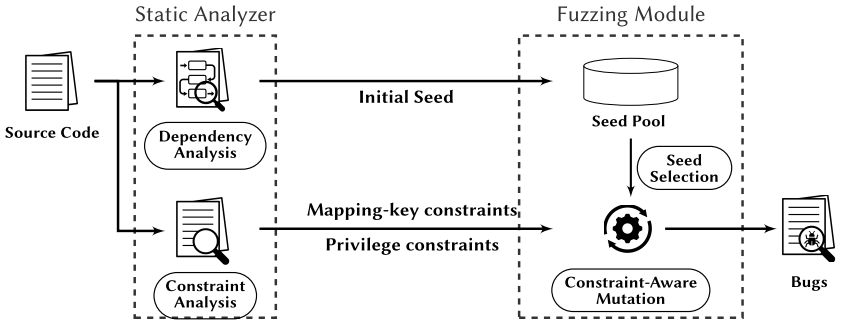


**Figure 3:** IConFuzz architecture.

The static analyzer extracts two types of information from the contract code. First, it identifies mapping-key constraints and privilege constraints, which capture crucial relationships between function arguments. These constraints are later used to guide constraint-aware argument mutation of the fuzzing module. It also performs function-level dependency analysis to decide the transaction order of initial seeds. For this analysis, we adopt the approach used in Smartian [7] to collect def-use chains across functions.

The fuzzing module receives two inputs derived from the static analysis phase. First, the initial seed corpus generated based on function-level dependencies is provided to the fuzzer. Second, the constraints extracted by the static analyzer are also used by the fuzzer to enable constraint-aware argument mutation during input generation.

### *4.2 Static Analysis for Constraint Identification*

Algorithm 1 describes the key functionality of our static analysis module for extracting mapping-key constraints and privilege constraints from the contract code. First, `CollectConstraints()` iterates through all functions in the contract to analyze their accesses to state variables. For each function, it invokes `AnalyzeDefUse()` to examine how arguments are used during accesses to mapping-type or address-type variables (Line 3–4).

Specifically, for mapping-type variable accesses, writes to $m[a_i]$ are recorded in the $\texttt{Def}_k$ set, whereas reads from $m[a_j]$ are recorded in the $\texttt{Use}_k$ set (Line 11–14). Similarly, for address-type variable accesses, assignments to address-type state variables are recorded in the $\texttt{Def}_a$ set, while comparisons against them are recorded in the $\texttt{Use}_a$ set.

After analyzing all functions, `CollectConstraints()` derives mapping-key constraints and privilege constraints by iterating over the collected sets (Line 5–6). For mapping-key constraints, a constraint $\langle f_1, a_1\rangle \approx \langle f_2, a_2\rangle$ is added to the $C_{map}$ if there exists a state variable $m$ such that $\langle f_1, a_1, m\rangle$ is in $\texttt{Def}_k$ and $\langle f_2, a_2, m\rangle$ is in $\texttt{Use}_k$ (Line 5). Here, the notation $\approx$ denotes a constraint that argument $a_1$ of $f_1$ and argument $a_2$ of $f_2$ must be equal. For privilege constraints, a constraint $\langle f_1, a_1\rangle \approx \langle f_2, a_2\rangle$ is added to the $C_{priv}$ if entries in $\texttt{Def}_a$ and $\texttt{Use}_k$ share the same $v$ (Line 6).

While Algorithm 1 presents the high-level concept of our approach, our actual implementation handles a wider range of cases. For mapping-key constraints, our analyzer properly identifies accesses to multi-dimensional (nested) mappings as well, although Line 11 and 13 illustrate this in a simplified form. During the extraction of privilege constraints, our static analyzer also captures the inequality constraints between arguments. While we omit this detail in Algorithm 1 for brevity, extending the algorithm to support inequality constraints is straightforward.

**Algorithm 1:** Collect constraints from functions

```
1  function Collect Constraints(prog)
2      Def_k ← ∅, Use_k ← ∅, Def_a ← ∅, Use_a ← ∅
3      for f in GetFuncs(prog) do
4          AnalyzeDefUse(f, Def_k, Use_k, Def_a, Use_a)
5      C_map ← {⟨f1, a1⟩ ≈ ⟨f2, a2⟩ | ∃m, ⟨f1, a1, m⟩ ∈ Def_k ∧ ⟨f2, a2, m⟩ ∈ Use_k}
6      C_priv ← {⟨f1, a1⟩ ≈ ⟨f2, a2⟩ | ∃v, ⟨f1, a1, v⟩ ∈ Def_a ∧ ⟨f2, a2, v⟩ ∈ Use_a}
7      return C_map ∪ C_priv
8  function AnalyzeDefUse (f, Def_k, Use_k, Def_a, Use_a)
9      a1, a2, ..., an ← GetArgs(f)
10     for stmt in GetStatements(f) do
11         if stmt writes to a mapping entry m[a_i] then
12             Def_k.Add(⟨f, a_i, m⟩)
13         if stmt reads from a mapping entry m[a_j] then
14             Use_k.Add(⟨f, a_j, m⟩)
15         if stmt assigns a_i to an address state variable v then
16             Def_a.Add(⟨f, a_i, v⟩)
17         if stmt compares v to a_j then // e.g., require (v == a_j)
18             Use_a.Add(⟨f, a_j, v⟩)
```

Note that our algorithm does not perform alias analysis on reference types or trace the mapping keys derived from transformed arguments (e.g., via arithmetic operations). This is because the purpose of our constraint extraction is to guide argument mutation in the fuzzing phase. To serve this purpose, our analysis focuses on generating precise constraints, at the cost of soundness. For instance, if a reference is used in a function, it implies that a state variable or mapping to be accessed is dynamically decided at runtime. If our analyzer were to over-approximate such accesses using pointer analysis, the resulting constraints would be prone to false positives, which would misguide argument mutation during fuzzing. In Section 5.3, we also provide an empirical justification for this design choice.

### *4.3 Constraint-Aware Argument Mutation*

Algorithm 2 details how the extracted constraints are incorporated into the fuzzing process. The top-level function `Fuzz()` first statically analyzes the contract and collects mapping-key constraints and privilege constraints (Line 2). After collecting the constraints, `Fuzz()` repeatedly selects a seed from the queue and generates a new seed with `ConstrAwareMutate()` until the timeout is reached. Following standard grey-box fuzzing [7], the fuzzer executes the mutated seed and saves it to the queue if any coverage gain is achieved (Line 6).

`ConstrAwareMutate()` illustrates the high-level idea of our constraint-aware argument mutation technique. It begins by selecting a target function ($f_1$) and an argument ($arg_1$) on which to perform an initial random mutation (Line 9–10). Next, the algorithm determines whether to enforce constraints on the selected target. If no applicable constraint exists or `rand()` returns a value less than parameter $\alpha$, the seed is returned with only the initial random mutation. Otherwise, `SelectConstraint()` randomly selects a relevant constraint $\langle f_1, arg_1 \rangle \approx \langle f_2, arg_2 \rangle$. For brevity, Algorithm 2 depicts only the handling of the equality constraint ($\approx$), where the dependent argument $arg_2$ of function $f_2$ is set equal to the mutated value of $arg_1$ (Line 12–13). In practice, inequality constraints ($\not\approx$) can also be selected and handled similarly by ensuring that $arg_2$ takes a different value from $arg_1$ (not shown in the algorithm for brevity).

**Algorithm 2:** Constraint-aware argument mutation

```
1  function Fuzz (prog, seedQueue)
2      constrs ← CollectConstraints (prog)
3      while timeout not reached do
4          seed ← seedQueue.Choose()
5          seed′ ← ConstrAwareMutate (seed, constrs)
6          SaveIfCoverageGained (prog, seed′, seedQueue)
7      return seedQueue
8  function ConstrAwareMutate (seed, constrs)
9      f1, arg1 ← ChooseMutateTarget (seed, constrs)
10     seed.TX[f1].Args[arg1] ← Rand()
11     if constraint exists and () >= α then
12         ⟨f1, arg1⟩ ≈ ⟨f2, arg2⟩ ← SelectConstraint (constrs, f1, arg1)
13         seed.TX[f2].Args[arg2] ← seed.TX[f1].Args[arg1]
14     return seed
```

Note that although IConFuzz guides the argument mutation with the extracted constraints, it also deliberately introduces randomness to preserve the input diversity. That is, (1) identified constraints can be ignored with probability $\alpha$ (currently set to 0.1 and we evaluate its impact in Section 5.4), and (2) only one of the constraints is selected and applied to adjust the seed after random mutation. The intuition behind this design choice is that strictly enforcing all constraints may overly restrict the input space that fuzzing can explore. Our evaluation results empirically show that this strategy effectively improves the bug-detection capability of smart contract fuzzing.

### *4.4 Implementation*

We implemented IConFuzz by extending the latest version of Smartian (commit 8b45b6f). To support open science, we release IConFuzz as an open-source tool available on GitHub at https://github.com/infosec-sogang/IConFuzz.

For the analysis of Solidity source code, we reused and modified the front-end of SmarTest [11]. Also, we wrote 866 lines of F# code to implement the static analysis module for constraint extraction. Our analysis is performed on an AST preprocessed by the Solidity compiler, where class inheritance is properly resolved and modifier code is embedded into functions. Therefore, our semantic analysis can capture constraints from base class functions or modifier blocks as well. Our analyzer currently focuses on the general semantics of Solidity statements, but it is straightforward to extend it to support library-specific semantics. For instance, by encoding the semantics of `grantRole()` and `hasRole()` into the analyzer, we can extract privilege constraints from contracts using the AccessControl [23] library for role-based access control.

For the fuzzing component, we added 838 lines of code to implement constraint-aware argument mutation during test case generation. Alongside these extensions, IConFuzz inherits basic fuzzer components from Smartian, such as test oracles, seed initialization algorithm, random mutation operations, and a grey-box concolic testing strategy [14]. This allows us to evaluate the effectiveness of our proposed techniques using Smartian as a sound baseline in Section 5.

## 5 Evaluation

### *5.1 Experimental Setup*

#### *5.1.1 Machine Environment*

All experiments were conducted on an Ubuntu 22.04 server machine with two Intel Xeon Gold 6426Y (2.5 GHz) CPUs. We used Docker version 25.0.0 for our experiments and spawned 60 containers in parallel. Each container was executed to test a single target contract with one CPU core and 6 GB of main memory.

#### *5.1.2 Tools for Comparison*

We used three smart contract testing tools as our comparison targets. We chose Smartian (commit 8b45b6f) [7] and RLF (commit 840ff37) [8] as representative fuzzers, and SmarTest (commit 36d191e) [11] as a symbolic execution tool. Since each tool supports the detection of different types of vulnerabilities, we constructed and used two benchmarks, each targeting distinct bug classes.

#### *5.1.3 Benchmark*

Our primary goal was to construct a benchmark consisting of realistic smart contracts that contain non-trivial vulnerabilities, which are challenging to trigger. To this end, we selected the real-world benchmark used in previous research [8,11], instead of using a benchmark that contains toy examples [24]. In addition, we applied a filtering process to exclude trivial bugs that all testing tools could detect quickly. We argue that such easily triggered bugs do not exhibit meaningful differentiation among the tested tools, and hence provide a limited basis for evaluating the effectiveness of the proposed technique.

For ether leakage and suicidal contract vulnerabilities, we adopted the benchmark from RLF [8], which contains 85 contracts labeled with 154 bugs distributed across 126 functions (108 functions with ether leakage and 46 functions with suicidal contract, with an overlap of 28 functions. These two kinds of vulnerabilities often co-exist because both are typically caused by incorrect authorization logic. Therefore, we evaluated these two vulnerabilities in the same benchmark. Then, we filtered out trivial bugs that were detected by all tools within one minute. This resulted in a benchmark consisting of 53 contracts with 94 bugs distributed across 84 functions (73 functions with ether leakage, 21 functions with suicidal contract bugs, and 10 functions containing both). We refer to this final benchmark as **B-ELSC**. To summarize, out of the 154 bugs in the original dataset, 60 bugs were filtered out and the remaining 94 bugs constitute **B-ELSC**.

For integer overflow vulnerabilities, we employed the benchmark used in the evaluation of SmarTest [11]. This benchmark contains 249 real-world contracts, each associated with a single CVE ID and labeled with the buggy program point. From this benchmark, we removed 3 contracts that raised parsing errors in Smartian, to ensure a fair comparison. We then filtered out trivial bugs that all tools that support integer overflow detection (Smartian, SmarTest, and IConFuzz) could trigger within one minute. The resulting benchmark contained 77 contracts, which we refer to as **B-IO** in this paper. To sum up, out of the 249 bugs in the original benchmark, 172 bugs were filtered out and the remaining 77 bugs constitute **B-IO**.

#### *5.1.4 Metric*

To evaluate the effectiveness of each testing tool, we first compared the number of labeled bugs that each tool detects within the given time budget. In addition, we leveraged *Time-to-Exposure (TTE)* as an evaluation metric. TTE represents the time that a testing tool consumed to trigger the specified bug from the given program. This metric can be used when the benchmark contains known vulnerabilities which can serve as *ground truth* [25–27]. Shorter TTE values indicate that the testing tool can detect the labeled bugs more promptly. For Smartian and IConFuzz, the TTE value also includes the time spent on static analysis.

To mitigate the randomness of testing tools, we repeated each experiment 40 times for each contract in the benchmark.

Note that we do not report precision metrics in bug detection experiments. Unlike static analyzers, testing tools generate concrete inputs that can reproduce the found bugs. Therefore, testing is theoretically free from false positives as long as there is a clear agreement on test oracle logic. In practice, however, smart contract tools may define the same bug differently and implement different test oracles. We confirmed that the test oracles implemented in the evaluated tools are consistent with each other and are designed to precisely detect the bugs in our benchmark. One exception was RLF, which checks for a weaker condition for suicidal contract. This permissive oracle gives RLF a slight advantage in the experiment.

### *5.2 Comparison with State-of-the-Art Tools*

We evaluated the effectiveness of IConFuzz against three state-of-the-art tools, Smartian [7], RLF [8], and SmarTest [11] on our two benchmarks, **B-ELSC** and **B-IO**, running each tool for two hours per contract.

On **B-ELSC**, IConFuzz detects 1.11×, 1.80×, and 1.73× more bugs than Smartian, SmarTest, and RLF, respectively (Fig. 4a). Furthermore, IConFuzz discovered a more comprehensive set of bugs than the other tools, as illustrated in Fig. 4b. Specifically, IConFuzz found 89 bugs, which is a strict superset of the 68 bugs found by Smartian. Moreover, IConFuzz identified 24 unique bugs that were not detected by SmarTest or RLF, while failing to detect only three bugs they found. To further validate our results, we conducted a statistical analysis, which is summarized on the left side of Table 1. We performed the Mann–Whitney U test on the number of bugs detected by each tool. Based on $p$-values lower than 0.001 and effect sizes ($\hat{A}_{12}$) greater than than 0.99, we can conclude that IConFuzz finds significantly more bugs than the other tools in **B-ELSC**.

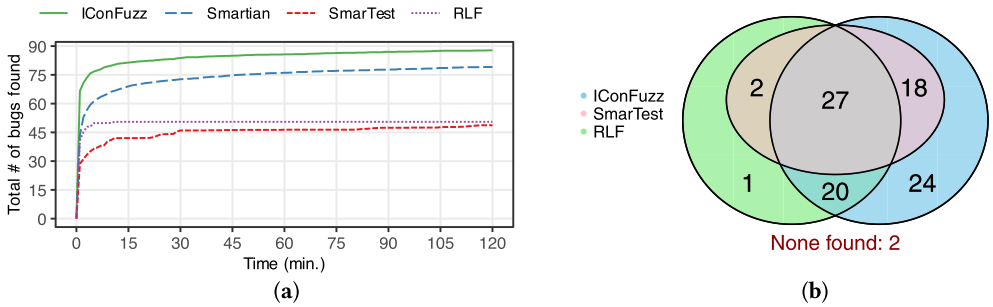


**Figure 4:** Comparison of IConFuzz with state-of-the-art tools on **B-ELSC**. (**a**) Comparison of the total number of bugs found over time; (**b**) venn diagram of the bugs found by each tool.

**Table 1:** Statistical analysis results on **B-ELSC** and **B-IO** benchmarks. Mean ± Std represents the average and standard deviation of the total number of bugs found. $\hat{A}_{12}$ and $p$-value are computed with the Mann-Whitney U test against IConFuzz.

| Tool | B-ELSC | | | B-IO | | |
|---|---|---|---|---|---|---|
| | **Mean ± Std** | $\hat{A}_{12}$ | $p$-value | **Mean ± Std** | $\hat{A}_{12}$ | $p$-value |
| IConFuzz | 92.35 ± 1.44 | — | — | 71.85 ± 0.36 | — | — |
| Smartian | 86.12 ± 2.09 | 0.9931 | $1.52 \times 10^{-14}$ | 70.72 ± 0.85 | 0.8569 | $1.71 \times 10^{-9}$ |
| SmarTest | 61.88 ± 1.02 | 1.0000 | $4.35 \times 10^{-15}$ | 64.33 ± 0.57 | 1.0000 | $3.86 \times 10^{-16}$ |
| RLF | 55.75 ± 0.67 | 1.0000 | $6.00 \times 10^{-16}$ | N/A | N/A | N/A |

We further compared IConFuzz against the second-best fuzzer, Smartian, in terms of TTE for the bugs in **B-ELSC**. We computed the median TTEs for each of the 89 bugs detected by IConFuzz and then summed

these values. For bugs that Smartian failed to detect in more than half of the 40 iterations, we assigned a median TTE of two hours (the timeout used in our experiment) to provide a favorable bias toward Smartian. Note that there were no bugs that Smartian detected while IConFuzz did not. Overall, the sum of median TTEs of IConFuzz was 2.49× smaller than that of Smartian. We also conducted the Mann–Whitney U test on the sum of TTEs for each iteration, while handling timeouts in the same way. The result showed that IConFuzz detects the same set of bugs significantly faster than Smartian, with a $p$-value lower than 0.001.

On **B-IO**, IConFuzz detected 72 bugs—a superset of the 71 bugs found by Smartian and the 64 bugs found by SmarTest (Fig. 5a); RLF was excluded from this comparison as it does not support integer overflow bug detection. While the number of bugs found by IConFuzz and Smartian was similar, statistical analysis confirms that the observed differences between IConFuzz and the other tools are not due to random chance, but reflect a consistent performance advantage. As indicated on the right side of Table 1, the $p$-values are smaller than 0.001 and the effect sizes ($\hat{A}_{12}$) are greater than 0.85 in all cases.

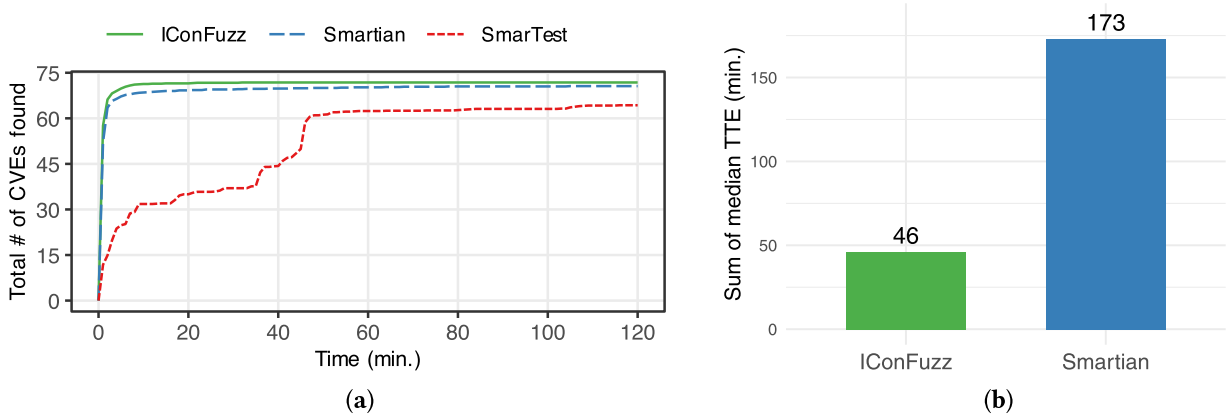


**Figure 5:** Comparison of IConFuzz with state-of-the-art tools on **B-IO**. (**a**) Comparison of the total number of bugs found over time; (**b**) comparison of IConFuzz and Smartian in terms of TTE.

Again, we performed a deeper comparison between IConFuzz and Smartian, the second-best performer in Fig. 5a, by summing the median TTEs in the same manner as for **B-ELSC**. Fig. 5b shows that the sum of median TTEs of IConFuzz was 3.76× smaller than that of Smartian, indicating that IConFuzz can detect the same set of bugs significantly faster than Smartian. Notably, for particularly challenging bugs, such as CVE-2018-13722, IConFuzz detected the vulnerability with a median TTE of only 232 s, compared to 6877 s for Smartian, highlighting the effectiveness of our constraint-aware argument mutation. The Mann–Whitney U test on the sum of TTEs for each iteration also confirms that IConFuzz requires significantly shorter time than Smartian to detect the bugs in **B-IO**, with a $p$-value lower than 0.001.

### *5.3 Efficiency and Accuracy of Constraint Analysis*

We evaluated the efficiency and accuracy of our static analysis for identifying argument constraints. We measured the static analysis time taken to extract constraints from the contracts in our benchmark. On average, it took 1.0 s per contract in both **B-ELSC** and **B-IO**. This negligible overhead confirms that the analysis is suitable for practical use in smart contract fuzzing. In terms of the number of identified constraints, our analysis identified a total of 2108 constraints, with 559 constraints from **B-ELSC** and 1549 constraints from **B-IO**.

To assess the precision and recall of our static analysis, we performed a manual verification of the identified constraints. For manual investigation, we randomly sampled 10% of the contracts from **B-ELSC**

and **B-IO** respectively, to avoid an imbalanced distribution across the benchmarks. To verify that the sampled contracts are representative of the original benchmarks, we examined the distributions of contract sizes in terms of lines of code (LoC), and the ratio of extracted constraint types. For **B-ELSC**, the complete contract set has a mean LoC of 239, while the sampled contracts have a mean LoC of 194. Similarly, for **B-IO**, the full benchmark and the sampled set have mean LoC of 161 and 192, respectively. In terms of constraint types, 74.48% of the extracted constraints were mapping-key constraints in the full dataset and 65.0% of the extracted constraints were mapping-key constraints in the sampled contracts.

Table 2 presents the results of our manual investigation. We inspected the 12 sampled contracts and identified 202 constraints (132 mapping-key constraints and 70 privilege constraints). According to our investigation, all constraints identified by our static analyzer correctly reflected the inter-procedural dependencies between function arguments, without any false positives. Meanwhile, we observed two false negatives in the `GAWToken` contract, where a function argument is passed to a modifier and then used to determine the mapping key through a complex data-flow that is not captured by our implementation. Overall, the manual investigation confirms that our static analysis achieves over 99% precision and recall on the sampled contracts, demonstrating its reliability in identifying mapping-key constraints and privilege constraints from smart contracts.

**Table 2:** Manual verification result of the sampled constraints. **GT**, **TP**, **FP**, and **FN** denote ground truth, true positives, false positives, and false negatives, respectively.

| Benchmark | Contract | Mapping-Key Constraints | | | | Privilege Constraints | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | GT | TP | FP | FN | GT | TP | FP | FN |
| **B-ELSC** | BitherumToken | 8 | 8 | 0 | 0 | 3 | 3 | 0 | 0 |
| | CNEXToken | 10 | 10 | 0 | 0 | 8 | 8 | 0 | 0 |
| | GAWToken | 4 | 2 | 0 | 2 | 7 | 7 | 0 | 0 |
| | MyAdvancedToken | 14 | 14 | 0 | 0 | 4 | 4 | 0 | 0 |
| | VoipToken | 3 | 3 | 0 | 0 | 8 | 8 | 0 | 0 |
| **B-IO** | ALUXToken | 2 | 2 | 0 | 0 | 7 | 7 | 0 | 0 |
| | HYIPToken | 16 | 16 | 0 | 0 | 2 | 2 | 0 | 0 |
| | MonkeyTreeToken | 19 | 19 | 0 | 0 | 4 | 4 | 0 | 0 |
| | PMHToken | 9 | 9 | 0 | 0 | 9 | 9 | 0 | 0 |
| | PylonToken | 12 | 12 | 0 | 0 | 8 | 8 | 0 | 0 |
| | SMT | 26 | 26 | 0 | 0 | 8 | 8 | 0 | 0 |
| | SPXToken | 9 | 9 | 0 | 0 | 2 | 2 | 0 | 0 |
| | Total | 132 | 130 | 0 | 2 | 70 | 70 | 0 | 0 |

### *5.4 Impact of Random Mutation Probability Parameter*

To evaluate the impact of the randomness parameter $\alpha$ introduced in Section 4.3, we conducted a sensitivity analysis by comparing the sum of median TTEs with varying $\alpha$ values. We tested four different settings: $\alpha$ = 5%, 10%, 20%, and 40%.

Fig. 6 presents the experimental results. On both benchmarks, IConFuzz achieved the best performance when $\alpha$ was set to 10%. Specifically, on **B-ELSC**, the sum of median TTEs with $\alpha$ = 10% was minimized to 471 min, whereas it increased to 603 min at $\alpha$ = 5% and 752 min at $\alpha$ = 40%. A similar trend was observed on **B-IO**, where the sum of median TTEs was minimized to 62 min at $\alpha$ = 10%.

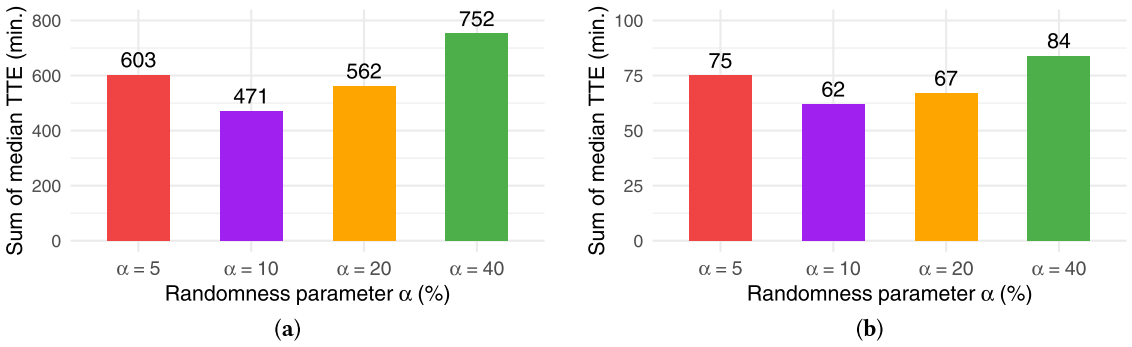


**Figure 6:** Impact of random mutation probability on fuzzing effectiveness. (**a**) Impact of parameter $\alpha$ on **B-ELSC**; (**b**) impact of parameter $\alpha$ on **B-IO**.

These results empirically validate our design choice of introducing randomness into constraint-aware argument mutation. When $\alpha$ is too small, the fuzzer may overly enforce constraints, restricting the exploration of the input space. Conversely, a large $\alpha$ diminishes the influence of constraints, reducing their effectiveness in guiding the fuzzing process. Therefore, we conclude that $\alpha = 10\%$ provides an optimal balance between constraint guidance and input diversity for bug detection.

### *5.5 Ablation Study*

From Section 5.3, we confirmed that our static analysis can accurately identify constraints from contracts. We now evaluate the extent to which these constraints can enhance the fuzzing performance. To this end, we conducted an ablation study with four different modes of IConFuzz: (1) a baseline that does not consider any constraints, (2) a mode using only mapping-key constraints, (3) a mode using only privilege constraints, and (4) a mode using both types of constraints.

Fig. 7 presents the results of our ablation study on both benchmarks, **B-ELSC** and **B-IO**. We compared the sum of median TTE for each contract in the benchmark, which is defined in the same manner as in Section 5.2. The results indicate that all three constraint-guided modes of IConFuzz significantly outperform the baseline, confirming that constraint-aware argument mutation significantly enhances the fuzzing effectiveness of IConFuzz. The performance improvements from incorporating mapping-key constraints and privilege constraints are comparable to each other. We can also observe that the integration of the two constraint types achieves a slight synergy that further improves the fuzzing performance.

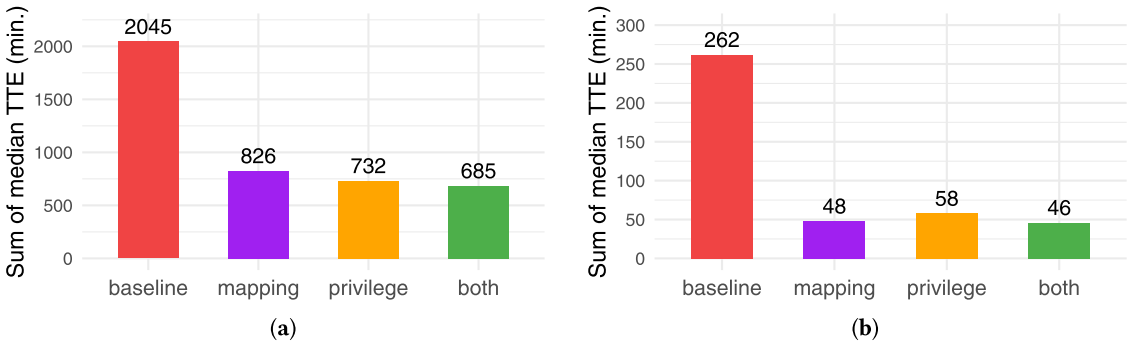


**Figure 7:** Ablation study of IConFuzz. (**a**) Impact of constraint-aware argument mutation on **B-ELSC**; (**b**) impact of constraint-aware argument mutation on **B-IO**.

## 6 Discussion

### 6.1 Internal Threats to Validity

First, the accuracy of our static analysis for constraint collection was only partially evaluated. As described in Section 5.3, we manually verified the analysis results from randomly sampled contracts. Nevertheless, the evaluation result provides empirical evidence that our analysis is accurate enough to identify meaningful constraints that significantly enhance the effectiveness of fuzzing, as shown in Section 5.5.

Second, differences in test oracles of the tools can lead to either overestimation or underestimation of each tool's effectiveness. In particular, RLF adopts a more permissive test oracle for ether leakage and suicidal contract bugs compared to other tools, which may provide a more favorable evaluation setting for RLF. Nevertheless, despite this potential bias, our experimental results demonstrate that IConFuzz outperforms RLF.

Next, filtering out trivial bugs during the benchmark construction in Section 5.1.3 could have introduced bias toward specific tools. For the 60 bugs filtered out during the construction of **B-ELSC**, the average median TTEs reported by the tools were all below 3 s, except for Smartian (11 s). Similarly, for the 172 bugs filtered out during the construction of **B-IO**, the average median TTEs were all below 10 s, except for Smartian (12 s). Therefore, we conclude that the bias introduced by the filtering process is negligible.

Finally, since fuzzing inherently involves randomness, the evaluation results may vary across different runs. To mitigate this threat, we repeated each experiment 40 times to ensure statistical reliability.

### 6.2 External Threats to Validity

We note that our technique analyzes the general semantics of smart contracts, (e.g., mapping access), rather than targeting behaviors specific to certain bug classes. In this sense, our approach is applicable to other bug classes as well. However, its impact on testing effectiveness may vary depending on bug classes. Specifically, bug classes that do not rely on contract state or require interactions between functions may benefit less from our approach. In this work, our evaluation focused on three classes of bugs (ether leakage, suicidal contract, and integer overflow), which have been actively studied and evaluated in the smart contract testing literature.

In Section 5.3, we showed that our static analyzer performs efficiently on the dataset used for the evaluation. We now discuss the scalability of our analysis in general. In Algorithm 1, the main semantic analysis is performed during the iterative invocation of `AnalyzeDefUse()`, which consumes time proportional to the total number of statements in the preprocessed AST. The efficiency of our analysis stems from the fact that it analyzes each function individually, and then combines the results to derive inter-procedural constraints between arguments.

Lastly, our work focuses on extracting two types of semantic constraints that can effectively guide smart contract fuzzing—privilege constraints and mapping-key constraints—rather than covering all types of constraints that can arise between function arguments. According to our observation, these constraints are common patterns in smart contracts that hinder effective fuzzing in our benchmark. However, the representativeness of these constraint types in a broader population of contracts remains an open question, and there may exist other useful constraints that can be explored to further improve fuzzing effectiveness.

### 6.3 Limitation and Future Work

First, our current implementation requires the availability of source code to perform static analysis. As a non-negligible portion of smart contracts on Ethereum do not make their source code publicly available, this limits the applicability of our approach. To address this limitation in practical deployments, we plan to

extend our analysis to support EVM bytecode in future work, based on existing studies on bytecode-level analysis [7,28].

During the evaluation, we observed an engineering issue that IConFuzz cannot properly handle contracts using hard-coded addresses as owners. For example, the contract associated with CVE-2018-13325 in **B-IO** sets its owner to a fixed address embedded in the code, rather than to the actual deployer address. Therefore, to test this contract properly, it must be deployed by an EOA with the designated address. We believe this issue can be addressed by introducing a preprocessing phase that identifies hard-coded owner addresses from contracts.

Beyond addressing these limitations, our methodology that extracts and leverages semantic constraints in function code can be applied to domains beyond smart contract fuzzing. In particular, it can be extended to other languages or environments where function arguments play a crucial role in exploring program paths, such as system call fuzzing [29] or API fuzzing [30]. We leave the exploration of these broader applications as future work.

## 7 Related Work

In the field of smart contract security, various studies [5,16,31] have systematically investigated and categorized different types of vulnerabilities. Atzei et al. [31] and Luu et al. [16] provided comprehensive taxonomies of smart contract vulnerabilities, including integer overflows, reentrancy, gasless sends, timestamp dependency, and mishandled exceptions. Jiang et al. [5] further refined the classification of smart contract bugs and presented ContractFuzzer, a pioneering smart contract fuzzer designed to detect these vulnerabilities. In Maian [17], additional bug categories, such as ether leakage, freezing ether, and suicidal contract, were introduced.

Static analysis offers valuable insights into smart contract security by examining the structure and semantics of code. Various static analysis tools [15,32–34] have been developed to identify vulnerabilities in smart contracts. Slither [32] is a static analysis framework that performs data-flow and taint analysis on the intermediate representation of smart contract code. Zeus [33] verifies smart contracts by leveraging abstract interpretation and symbolic model checking. VeriSmart [15] statically detects integer overflow bugs while inferring invariants among state variables, which are leveraged to increase the precision of the analysis. Securify [34] employs Datalog-based analysis to detect smart contract vulnerabilities at the bytecode level. Although static analysis can effectively identify vulnerabilities in smart contracts, it cannot provide concrete test cases that can reproduce the found vulnerabilities. In contrast, our work leverages static analysis to collect useful information for guiding fuzzing, instead of using it to directly detect vulnerabilities.

Software testing techniques, which are capable of generating concrete program inputs, have also been actively explored for smart contract security. Among them, fuzzing is widely used for testing smart contracts because it is simple yet effective. ContractFuzzer [5], the first smart contract fuzzer proposed in academia, utilizes a basic black-box fuzzing strategy. Subsequent research led to the development of coverage-guided grey-box fuzzers for smart contracts, such as sFuzz [6]. In sFuzz, the genetic algorithm from AFL [35] was integrated with search-based testing techniques [36] to generate inputs that maximize code coverage. Wüstholz and Christakis [37] proposed a directed fuzzer with online static analysis to guide exploration toward target locations by considering the reachability of each input. ItyFuzz [38] adopts snapshot-based fuzzing to avoid the inefficiency of traditional fuzzers that must re-execute transaction sequences from the initial state for each test input. While all these approaches aim to enhance the effectiveness of smart contract fuzzing, they mainly focus on improving the prioritization strategy of generated inputs [6,37] or boosting the efficiency of input execution [38]. In contrast, IConFuzz focuses on generating meaningful function arguments by considering program semantics during the mutation phase of fuzzing.

Dynamic symbolic execution (a.k.a. concolic testing) [39] is another important software testing technique used for detecting smart contract vulnerabilities. Dynamic symbolic execution can systematically explore execution paths by treating inputs as symbolic variables. Manticore [18] and Mythril [40] are two well-known dynamic symbolic execution tools that support the detection of various smart contract vulnerabilities. TeEther [41] and Maian [17], by contrast, are dynamic symbolic execution tools specialized in detecting security flaws related to ether transfer, such as ether leakage. These tools statically analyze EVM bytecode to identify vulnerable paths that could result in ether leakage or suicidal contracts. They then use symbolic execution to derive path constraints and generate concrete inputs that exercise specific instruction patterns. On the other hand, SymX [42] focuses on validating the exploitability of detected vulnerabilities by replaying the attack transaction sequence and checking for concrete outcomes, such as balance changes. Its goal is to verify the real-world impact of a vulnerability and eliminate false positives where the detected flaw does not result in any meaningful loss of ether.

One of the key challenges in smart contract testing stems from the stateful property of contracts, where a function's behavior depends on the sequence of preceding transactions. This state-dependency between functions makes the generation of meaningful transaction sequences a crucial aspect of effective testing. To address this challenge, various techniques have been proposed in smart contract testing. Harvey [9] uses a heuristic approach that mutates state variables to identify stateful functions. Then, other function calls are prepended to such stateful functions to construct transaction sequences. Smartian [7] leverages static and dynamic data-flow analysis to identify function pairs with data dependencies over state variables. This information is used to decide transaction ordering during seed initialization or to evaluate the usefulness of generated inputs. ConFuzzius [10], which employs hybrid fuzzing [43,44] to test smart contracts, utilizes dynamic analysis to track RAW (read-after-write) data dependencies during runtime. Then, the genetic algorithm employed in the fuzzing phase selects test inputs and combines their transaction sequences based on these dependencies. Currently, IConFuzz adopts the strategy of Smartian when deciding the order of transaction sequences.

In other lines of work, machine learning has been adopted to effectively generate transaction sequences that can reveal interesting behaviors of the contract under test. On one hand, there have been studies that integrate machine learning techniques with smart contract fuzzing to guide transaction sequence generation [8,21,22]. ILF [21] first collects transaction sequences that achieve high code coverage through symbolic execution. It then trains a neural network on these sequences and uses the trained model during fuzzing to predict transaction sequences that are likely to yield high code coverage. RLF [8] employs reinforcement learning to generate transaction sequences that maximize a reward defined based on vulnerability detection and code coverage. In xFuzz [22], a machine learning model is trained with functions labeled as vulnerable by static analysis. During the fuzzing phase, it uses this model to prioritize transaction sequences that are more likely to reach vulnerable code regions. On the other hand, So et al. [11] proposed SmarTest, which integrates machine learning with dynamic symbolic execution. When the symbolic execution engine must decide the next function to call, SmarTest uses a language model to predict the function that is most likely to trigger a vulnerability. A common characteristic of these learning-based approaches is that they learn the code patterns from past executions or pre-labeled data to guide transaction sequence generation. However, these approaches require sufficiently large datasets of vulnerable contracts for training and may struggle to identify vulnerabilities whose patterns are not present in the training data.

While these studies significantly enhanced the effectiveness of smart contract testing by addressing the problem of transaction sequence ordering, they largely overlooked the challenge of generating meaningful function arguments. Smartian partially tackled this challenge using grey-box concolic testing [14], but it primarily focused on generating arguments that satisfy branch conditions involving integer comparisons.

Similarly, ILF reuses the argument values that its trained model deems meaningful, but this strategy is also limited to integer-type arguments. Although dynamic symbolic execution can systematically generate argument values that explore various program paths, the path constraints used in SMT solving often miss implicit constraints on mapping index variables, as discussed in Section 3.1. LLAMA [45] introduces a hierarchical prompting strategy that instructs LLMs to synthesize semantically valid transaction arguments that are difficult to generate through random mutation. Although LLAMA leverages the reasoning capabilities of LLMs to overcome the lack of semantic information in initial seeds, such model-driven approaches often involve non-deterministic inference and significant computational overhead during testing. In contrast, IConFuzz employs a systematic argument generation strategy that considers both the context of the contract and implicit constraints between function arguments, thereby enabling effective generation of meaningful test cases.

Recent studies have further advanced the state-of-the-art in smart contract testing by focusing on scalability or search efficiency. ConFuzz [28] addresses the scalability challenge by enabling test case generation from EVM bytecode alone, even when both the source code and ABI specification are unavailable. To improve search efficiency, Vulseye [46] introduces a stateful directed grey-box fuzzing approach. By analyzing both Solidity source code and EVM bytecode, it identifies specific vulnerability patterns and defines target program states required to trigger them, thereby steering the fuzzing process toward highly vulnerable code regions. FunFuzz [47] takes a similar approach by arguing that maximizing code coverage may not always lead to effective vulnerability detection. Instead, it employs a function-oriented fuzzing strategy that limits the exploration scope by analyzing contracts to identify specific functions that are more likely to contain vulnerabilities.

Additionally, there have been studies that focus on specific vulnerability classes. JACKAL [48] aims to detect a new class of vulnerability termed *confused contracts*, which arise from cross-contract interactions. This vulnerability occurs when a contract includes a function call that can be redirected to a different contract, while the function arguments can be controlled by an adversary. This allows the attacker to manipulate the execution flow to call an unintended function in another contract, potentially leading to the loss of funds. CPMMX [49] targets CPMM (Constant Product Market Maker) composability bugs by identifying invariant-breaking transaction sequences through a shallow-then-deep search strategy. It effectively synthesizes profitable transaction sequences by detecting invariant violations and then exploring deeper state changes through the repetition of critical transactions.

## 8 Conclusion

In this paper, we presented IConFuzz, a smart contract fuzzer designed to address the limitations of existing smart contract testing tools in performing effective argument mutation. By introducing lightweight static analysis to identify constraints that reflect the complex execution contexts of smart contracts, IConFuzz employs constraint-aware argument mutation to generate semantically meaningful inputs for deeper exploration of contract states. Our evaluation shows that IConFuzz significantly outperforms existing state-of-the-art tools, detecting up to 1.80× more vulnerabilities and finding bugs up to 52.13× faster.

**Acknowledgement:** None.

**Funding Statement:** This research was supported by the Korea government (MSIT) through (1) a National Research Foundation (NRF) research grant (RS-2025-24442977), and (2) the Graduate School of Metaverse Convergence support program (IITP-2023-RS-2022-00156318) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP).

**Author Contributions:** The authors confirm contribution to the paper as follows: conceptualization, Hojin Choi and Jaeseung Choi; methodology, Hojin Choi and Jaeseung Choi; software, Hojin Choi; writing—original draft preparation, Hojin Choi and Jaeseung Choi; writing—review and editing, Jaeseung Choi; visualization, Hojin Choi; supervision, Jaeseung Choi. All authors reviewed and approved the final version of the manuscript.

**Availability of Data and Materials:** The implementation of IConFuzz is publicly available at https://github.com/infosec-sogang/IConFuzz. We also provide the experimental scripts, benchmarks, and docker images at https://github.com/infosec-sogang/IConFuzz-Artifact.

**Ethics Approval:** Not applicable.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References


1. Finley K. A $50 million hack just showed that the DAO was all too human. 2016 [cited 2026 Mar 1]. Available from: https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/.
2. Banisadr E. How $800k evaporated from the PoWH coin Ponzi scheme overnight. 2018 [cited 2026 Mar 1]. Available from: https://medium.com/@ebanisadr/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530/.
3. Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Commun ACM. 1990;33(12):32–44. doi:10.1145/96267.96279.
4. Manès VJM, Han H, Han C, Cha SK, Egele M, Schwartz EJ, et al. The art, science, and engineering of fuzzing: a survey. IEEE Trans Softw Eng. 2021;47(11):2312–31. doi:10.1109/tse.2019.2946563.
5. Jiang B, Liu Y, Chan WK. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering; 2018 Sep 3–7; Montpellier, France. p. 259–69.
6. Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the International Conference on Software Engineering; 2020 Jul 6–11; Seoul, Republic of Korea. p. 778–88.
7. Choi J, Kim D, Kim S, Grieco G, Groce A, Cha SK. SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: Proceedings of the International Conference on Automated Software Engineering; 2021 Nov 15–19; Melbourne, Australia. p. 227–39.
8. Su J, Dai HN, Zhao L, Zibin Zheng XL. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In: Proceedings of the International Conference on Automated Software Engineering; 2022 Oct 10–14; Rochester, MI, USA. p. 1–12.
9. Wüstholz V, Christakis M. Harvey: a greybox fuzzer for smart contracts. In: International Symposium on the Foundations of Software Engineering: Industry Papers; 2020 Nov 8–13; Virtual Event. p. 1398–409.
10. Torres CF, Iannillo AK, Gervais A, State R. ConFuzzius: a data dependency-aware hybrid fuzzer for smart contracts. In: Proceedings of the IEEE European Symposium on Security and Privacy; 2021 Sep 6–10; Vienna, Austria. p. 103–19.
11. So S, Hong S, Oh H. SmarTest: effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution. In: Proceedings of the USENIX Security Symposium; 2021 Aug 11–13; Seattle, WA, USA. p. 1361–78.
12. Chen P, Chen H. Angora: efficient fuzzing by principled search. In: Proceedings of the IEEE Symposium on Security and Privacy; 2018 May 21–23; San Francisco, CA, USA. p. 855–69.
13. You W, Liu X, Ma S, Perry D, Zhang X, Liang B. SLF: fuzzing without valid seed inputs. In: Proceedings of the International Conference on Software Engineering; 2019 May 25–26; Montreal, QC, Canada. p. 712–23.
14. Choi J, Jang J, Han C, Cha SK. Grey-box concolic testing on binary code. In: Proceedings of the International Conference on Software Engineering; 2019 May 25–26; Montreal, QC, Canada. p. 736–47.

15. So S, Lee M, Park J, Lee H, Oh H. VeriSmart: a highly precise safety verifier for ethereum smart contracts. In: Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP); 2020 May 18–21; San Francisco, CA, USA. p. 1678–94.
16. Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the ACM Conference on Computer and Communications Security; 2016 Oct 24–28; Vienna, Austria. p. 254–69.
17. Nikolić I, Kolluri A, Sergey I, Saxena P, Hobor A. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the Annual Computer Security Applications Conference; 2018 Dec 3–7; San Juan, PR, USA. p. 653–63.
18. Mossberg M, Manzano F, Hennenfent E, Groce A, Grieco G, Feist J, et al. Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019); 2019 Nov 11–15; San Diego, CA, USA. p. 1186–9.
19. Kado C, Yanai N, Cruz JP, Yamashita K, Okamura S. Empirical study of impact of solidity compiler updates on vulnerabilities in ethereum smart contracts. Distrib Ledger Technol Res Pract. 2025;4(2):1–14. doi:10.1145/3688812.
20. Etherscan. [cited 2026 Mar 1]. Available from: https://etherscan.io/.
21. He J, Balunović M, Ambroladze N, Tsankov P, Vechev M. Learning to fuzz from symbolic execution with application to smart contracts. In: Proceedings of the ACM Conference on Computer and Communications Security; 2019 Nov 11–15; London, UK. p. 531–48.
22. Xue Y, Ye J, Zhang W, Sun J, Ma L, Wang H, et al. xFuzz: machine learning guided cross-contract fuzzing. IEEE Trans Dependable Secur Comput. 2022;21(2):515–29.
23. AccessControl Library. [cited 2026 Mar 1]. Available from: https://docs.openzeppelin.com/contracts/5.x/access-control.
24. Durieux T, Ferreira JF, Abreu R, Cruz P. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering; 2020 Jul 6–11; Seoul, Republic of Korea. p. 530–41.
25. Böhme M, Pham VT, Nguyen MD, Roychoudhury A. Directed greybox fuzzing. In: Proceedings of the ACM Conference on Computer and Communications Security; 2017 Oct 30–Nov 3; Dallas, TX, USA. p. 2329–44.
26. Du Z, Li Y, Liu Y, Mao B. Windranger: a directed greybox fuzzer driven by deviation basic blocks. In: Proceedings of the International Conference on Software Engineering; 2022 May 8–27; Pittsburgh, PA, USA. p. 2440–51.
27. Luo C, Meng W, Li P. SELECTFUZZ: efficient directed fuzzing with selective path exploration. In: Proceedings of the IEEE Symposium on Security and Privacy; 2023 May 22–24; San Francisco, CA, USA. p. 1050–64.
28. Wong T, Zhang C, Ni Y, Luo M, Chen H, Yu Y, et al. Confuzz: towards large scale fuzz testing of smart contracts in ethereum. In: Proceedings of the IEEE International Conference on Computer Communications. IEEE; 2024 May 20–23; Vancouver, BC, Canada. p. 1691–700.
29. Chen W, Hao Y, Zhang Z, Zou X, Kirat D, Mishra S, et al. SyzGen++: dependency inference for augmenting kernel driver fuzzing. In: Proceedings of the IEEE Symposium on Security and Privacy; 2024 May 19–23; San Francisco, CA, USA. p. 4661–77.
30. Yandrapally R, Sinha S, Tzoref-Brill R, Mesbah A. Carving UI tests to generate API tests and API specification. In: Proceedings of the International Conference on Software Engineering; 2023 May 14–20; Melbourne, Australia. p. 1971–82.
31. Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts SoK. In: International Conference on Principles of Security and Trust. Berlin/Heidelberg, Germany: Springer; 2017.
32. Feist J, Grieco G, Groce A. Slither: a static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain; 2019 May 27; Montreal, QC, Canada. p. 8–15.
33. Kalra S, Goel S, Dhawan M, Sharma S. Zeus: analyzing safety of smart contracts. In: Proceedings of the Network and Distributed System Security Symposium; 2018 Feb 18–21; San Diego, CA, USA.
34. Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Bünzli F, Vechev M. Securify: practical security analysis of smart contracts. In: Proceedings of the ACM Conference on Computer and Communications Security; 2018 Oct 15–19; Toronto, ON, Canada. p. 67–82.

35. Zalewski M. American fuzzy lop. [cited 2026 Mar 1]. Available from: http://lcamtuf.coredump.cx/afl/.
36. McMinn P. Search-based software testing: past, present and future. In: Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops; 2011 Mar 21–25; Berlin, Germany. p. 153–63.
37. Wüstholz V, Christakis M. Targeted greybox fuzzing with static lookahead analysis. In: Proceedings of the International Conference on Software Engineering; 2020 Jul 19–27; Seoul, Republic of Korea. p. 789–800.
38. Shou C, Tan S, Sen K. ItyFuzz: snapshot-based fuzzer for smart contract. In: Proceedings of the International Symposium on Software Testing and Analysis; 2023 Jul 17–21; Washington, DC, USA. p. 322–33.
39. Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C. In: Proceedings of the International Symposium on the Foundations of Software Engineering; 2005 Nov 5–11; Portland, OR, USA. p. 263–72.
40. Mueller B. Smashing ethereum smart contracts for fun and actual profit. In: Proceedings of the HITB Security Conference; 2018 Apr 9–13; Amsterdam, The Netherlands.
41. Krupp J, Rossow C. teEther: gnawing at ethereum to automatically exploit smart contracts. In: Proceedings of the USENIX Security Symposium; 2018 Aug 15–17; Baltimore, MD, USA. p. 1317–33.
42. Li Z, Zhao Z, Li W, Zhang R, Xue R, Lu S, et al. Enhancing smart contract security comprehensively through dynamic symbolic execution. In: Proceedings of the ACM Conference on Computer and Communications Security; 2024 Oct 14–18; Salt Lake City, UT, USA. p. 5072–4.
43. Yun I, Lee S, Xu M, Jang Y, Kim T. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: Proceedings of the USENIX Security Symposium; 2018 Aug 15–17; Baltimore, MD, USA. p. 745–62.
44. Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, et al. Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium; 2016 Feb 21–24; San Diego, CA, USA. p. 1–16.
45. Gai K, Liang H, Yu J, Zhu L, Niyato D. LLAMA: multi-feedback smart contract fuzzing framework with LLM-guided seed generation. IEEE Trans Inf Forensics Secur. 2026;21:3281–96.
46. Liang R, Chen J, Wu C, He K, Wu Y, Cao R, et al. Vulseye: detect smart contract vulnerabilities via stateful directed graybox fuzzing. IEEE Trans Inf Forensics Secur. 2025;20:2157–70.
47. Ye M, Nan Y, Dai HN, Yang S, Luo X, Zheng Z. FunFuzz: a function-oriented fuzzer for smart contract vulnerability detection with high effectiveness and efficiency. ACM Trans Softw Eng Methodol. 2024;33(191):1–20.
48. Gritti F, Ruaro N, McLaughlin R, Bose P, Das D, Grishchenko I, et al. Confusum contractum: confused deputy vulnerabilities in ethereum smart contracts. In: Proceedings of the USENIX Security Symposium; 2023 Aug 9–11; Anaheim, CA, USA. p. 1793–810.
49. Han S, Kim J, Lee SJ, Yun I. Automated attack synthesis for constant product market makers. In: Proceedings of the International Symposium on Software Testing and Analysis; 2025 Jun 25–28; Trondheim, Norway. p. 47–68.