DogeFuzz: A Simple Yet Efficient Grey-box Fuzzer for Ethereum Smart Contracts

Ismael Medeiros[1], **Fausto Carvalho**[1], **Alexandre Ferreira**[1], **Rodrigo Bonifácio**[1], **Fabiano Cavalcanti Fernandes**[2]

[1]Computer Science Department, University of Brasilia (UnB), Brasilia, Brazil
[2]Federal Institute of Brasilia (IFB), Brasilia, Brazil

{ismael.medeiros96,faustocarva,alexandre.ps1123}@gmail.com,
rbonifacio@unb.br,fabiano.fernandes@ifb.edu.br

Abstract. *Ethereum is a distributed, peer-to-peer blockchain infrastructure that has attracted billions of dollars. Perhaps due to its success, Ethereum has become a target for various kinds of attacks, motivating researchers to explore different techniques to identify vulnerabilities in EVM bytecode (the language of the Ethereum Virtual Machine)—including formal verification, symbolic execution, and fuzz testing. Although recent studies empirically compare smart contract fuzzers, there is a lack of literature investigating how simpler grey-box fuzzers compare to more advanced ones. To fill this gap, in this paper, we present DogeFuzz, an extensible infrastructure for fuzzing Ethereum smart contracts, currently supporting black-box fuzzing and two grey-box fuzzing strategies: coverage-guided grey-box fuzzing (DogeFuzz-G) and directed grey-box fuzzing (DogeFuzz-DG). We conduct a series of experiments using benchmarks already available in the literature and compare the DogeFuzz strategies with state-of-the-art fuzzers for smart contracts. Surprisingly, although DogeFuzz does not leverage advanced techniques for improving input generation (such as symbolic execution or machine learning), DogeFuzz outperforms sFuzz and ILF, two state-of-the-art fuzzers. Nonetheless, the Smartian fuzzer shows higher code coverage and bug-finding capabilities than DogeFuzz.*

1. Introduction

The Ethereum platform was designed to provide a blockchain network enriched with a Turing-complete language that executes on top of a stack-based virtual machine. Ethereum has gained increased popularity, and developers use different programming languages to implement Ethereum smart contracts—small programs that benefit from a blockchain architecture to ease the implementation of distributed, decentralized, and consensus-based applications. Although developers can use other languages, Solidity is the most widely used language for implementing Ethereum smart contracts. While expected to support financial digital assets, Solidity and the Ethereum Virtual Machine (EVM) have several design flaws that have allowed bad-intentioned actors to attack the platform, resulting in significant financial losses.

For instance, the infamous DAO attack embezzled four million Ethereum coins (Ethers) [Atzei et al. 2017] by exploiting a vulnerability of a decentralized autonomous organization implemented as a smart contract. This incident ultimately forced the Ethereum community to implement a hard fork of the entire Ethereum blockchain.

The popular Parity Wallet was also a target of two attacks that led to significant losses [OpenZeppelin 2017]. In a more recent incident, attackers exploited Uniswap's liquidity pool using flash loans to manipulate imBTC coin prices [OpenZeppelin 2020]. By inducing a reentrancy attack, they drained a significant number of cryptocurrencies, showing that smart contracts remain vulnerable to well-known vulnerabilities such as reentrancy and unhandled exceptions, reflecting the need for more robust automated security tools. Common smart contract vulnerabilities (such as reentrancy) are recorded using the Smart Contract Weakness Classification (SWC).[1] The non-negligible financial impact of attacks targeting the Ethereum ecosystem, coupled with the characteristics of existing smart contracts, has prompted both practitioners and researchers to explore a wide range of techniques for identifying vulnerabilities in smart contracts.

These techniques include static analysis, symbolic execution, formal verification, and fuzz testing (fuzzing)—[Chu et al. 2023] present a survey of Ethereum smart contract vulnerabilities and the techniques used for vulnerability detection in smart contracts. ContractFuzzer [Jiang et al. 2018] was the first fuzzer designed specifically for smart contracts. Following its initial success, several other research groups began exploring optimization techniques and additional features to enhance the bug-finding capabilities of Ethereum fuzzers. In this context, we use the term "bug" to refer to issues that can eventually introduce a smart contract vulnerability. Consequently, recent fuzzers assert that ContractFuzzer is outdated and no longer incorporates it into their empirical studies [Nguyen et al. 2020, He et al. 2019, Wüstholz and Christakis 2020]. Advanced fuzzers targeting the Ethereum platform integrate different strategies for improving their performance, including symbolic execution (e.g., sFuzz [Nguyen et al. 2020]), machine learning (e.g., ILF [He et al. 2019]), and a combination of static and dynamic data-flow analysis (e.g., Smartian [Choi et al. 2021]). Section 2 presents an overview of smart contract vulnerabilities in Ethereum and fuzzing for smart contracts.

Although recent studies empirically compare smart contract fuzzers [Wu et al. 2024], there is a lack of literature investigating how simpler grey-box fuzzers compare to more advanced ones. To fill this gap, we present and evaluate DogeFuzz in this paper. DogeFuzz is an extensible grey-box fuzzer that currently supports two strategies for prioritizing fuzzer inputs. The first strategy, DogeFuzz-G, is a conventional grey-box fuzzer that prioritizes inputs likely to increase code coverage. The second strategy, DogeFuzz-DG, prioritizes input more likely to cover instructions close to the so-called critical instructions [Krupp and Rossow 2018]. We present some design decisions of DogeFuzz in Section 3. We evaluate DogeFuzz using a quasi-replication of a previous study [Choi et al. 2021] and two benchmarks that we detail in Section 4. The results of our empirical assessment (Section 5) revealed several findings, some of which were unexpected:

- Although the algorithms DogeFuzz-G and DogeFuzz-DG use quite different fuzzing strategies, their performance is quite similar. In the larger dataset we use in our experiment, both algorithms perform similarly when considering instruction coverage and bug-finding capabilities. In the smaller dataset, while DogeFuzz-G led to higher coverage than DogeFuzz-DG, their bug-finding capabilities are almost identical.

[1] https://swcregistry.io/

- Although DogeFuzz implements simpler fuzzing strategies, DogeFuzz-G and DogeFuzz-DG outperform state-of-the-art fuzzers (sFuzz and ILF). Even the black-box strategy we implemented achieved a performance on par with sFuzz and ILF. Also, our study confirms that the Smartian outperforms the other fuzzers we experiment with (including DogeFuzz).

Altogether, the contribution of this paper is twofold. First, we present the design and implementation of DogeFuzz, an open-source, extensible infrastructure for experimenting with fuzzers for smart contracts. Second, we provide an in-depth assessment of DogeFuzz, demonstrating that simple grey-box strategies might outperform sFuzz and ILF—smart contract fuzzers that rely on state-of-the-art features.

2. Background and Related Work

2.1. Smart Contracts Vulnerabilities

There are several Ethereum vulnerabilities registered in the Smart Contract Weakness Classification (SWC) repository. [Chu et al. 2023] and [Atzei et al. 2017] survey the field of smart contract vulnerabilities, serving as a good starting point to understand Ethereum vulnerabilities. In this section, we detail a few vulnerabilities that we analyzed and explored during our research, including Reentrancy, Dangerous Delegate Call, Gasless Send & Exception Disorder, and Number Dependency & Timestamp Dependency.

Reentrancy (SWC-107) is a design issue that arises from how the Ethereum Virtual Machine (EVM) handles communication between contracts. In Ethereum, smart contracts can be invoked using `CALL` instructions and can define a `receive()` method to handle incoming Ether (the native Ethereum cryptocurrency). Consequently, contracts can include some logic to manage incoming transactions. The vulnerability known as reentrancy occurs when a method can be repeatedly invoked within the same transaction by a malicious contract exploiting the `receive()` method. This vulnerability arises due to the misuse of `CALL` instructions without adequately updating the contract's internal state.

Listing 1 shows a simple contract that presents this vulnerability. In this contract, the method `withdraw()` sends the amount of ether to the caller before it updates its internal state (`balances` field). As the method `call` can be directed to another smart contract and the EVM resolves the calls sequentially, the method `withdraw()` can be called multiple times within a transaction by a malicious contract without the assignment `balances[msg.sender] = 0` being executed, draining all the *ether* from that vulnerable contract.

Dangerous Delegate Call (SWC-112) is caused by the use of a variant of the `CALL` instruction named `DELEGATECALL`. This method can be used to execute operations in other contracts using the context of the caller contract (e.g., the caller contract's balance and variables). This instruction should be used cautiously to avoid inadvertently calling malicious contracts. Listing 2 shows a contract with this vulnerability. In this case, the method `fwd()` can be called with any address as argument. As such, an attacker can pass a malicious contract that will be executed in the contract context (via `DELEGATECALL`). This makes the variable `owner` exposed to be changed by a malicious contract.

```
contract Reentrancy {
   mapping (address => uint) private balances;
   function withdraw() public {
        uint amount = balances[msg.sender];
        (bool success, _) =
            msg.sender.call.value(amount)("");
        require(success);
        balances[msg.sender] = 0;
   }
}
```

Listing 1. A contract with the Reentrancy vulnerability

```
contract DelegateCall {
 address owner;
 constructor() public { owner = msg.sender; }
 function fwd(address c, bytes data) public {
   require(c.delegatecall(data));
 }
}
```

Listing 2. A contract with the Dangerous Delegate Call vulnerability

Gasless Send & Exception Disorder (SWC-104) are vulnerabilities related to how an external call is handled. Each call returns the result of the execution, and when this execution fails, it will return a value to be handled by the caller contract. If this call is not handled in any way, it will lead to unexpected behaviors when a failure occurs.

Number Dependency & Timestamp Dependency (SWC-120) are related to introducing randomness into smart contracts. Intuitively, one might consider deriving a seed from the number of blocks or the time a block was mined to implement random computations within a contract method. However, relying on these data sources can lead to poor randomness computation.

2.2. Fuzzing Smart Contracts

Fuzz testing (or fuzzing for short) is a well-established dynamic approach for finding bugs and vulnerabilities [Zeller et al. 2024]. The central idea of fuzzing is to implement a program (P1) that (a) generates random inputs for another program (P2) and (b) executes P2 using those random inputs. Over 30 years ago, [Miller et al. 1990] conducted the first empirical study on "classic" black-box fuzz testing for Unix utilities—even though the research on fuzz testing is still active today, now focusing on newer technologies like mobile apps and smart contracts.

The first fuzzer designed specifically for Ethereum smart contracts, ContractFuzzer, was developed by [Jiang et al. 2018]. ContractFuzzer extends the official Go Ethereum VM implementation by incorporating seven bug oracles that monitor the execution of smart contracts and identify Ethereum vulnerabilities, such as Exception Disorder and Reentrancy—the latter being responsible for the infamous DAO attack. Additionally,

ContractFuzzer features a fuzzer that generates random inputs conforming to the smart contracts' interfaces, as their Application Binary Interfaces (ABIs) specify. Noteworthy, ContractFuzzer conducts fuzzing for each function declared in the smart contracts ABI. [Jiang et al. 2018] mined smart contracts from a public repository (Etherscan) and assessed ContractFuzzer's performance after deploying 6991 smart contracts onto their testnet. In their evaluation, ContractFuzzer accurately detected 459 vulnerabilities within these smart contracts [Jiang et al. 2018].

Since then, fuzzing for smart contracts has attracted the interest of different research groups, and other fuzzers for smart contracts have been developed. Some of the open-sourced fuzzers for smart contracts include sFuzz [Nguyen et al. 2020], ILF Fuzzer [He et al. 2019], and Smartian [Choi et al. 2021]. sFuzz is an extensible, feedback-guided fuzzing engine for smart contracts. Like AFL (a well-known, industry-strength fuzzer for C programs) [AFL 2013], sFuzz models the *test generation problem* as an *optimization problem*, prioritizing inputs that cover additional statements in the contract. Unlike AFL, sFuzz also prioritizes inputs considering how far a seed is from covering missed branches [Nguyen et al. 2020]. [Nguyen et al. 2020] present a comparison between sFuzz and two other tools: ContractFuzzer and Oyente [Luu et al. 2016] (a symbolic execution tool). The authors report that sFuzz outperforms the other tools w.r.t the number of test cases generated within a given time interval. They highlight that since ContractFuzzer uses a standard EVM implementation, some computational effort is expended while running ContractFuzzer to obtain network consensus and to mine blocks. In contrast, sFuzz executes over an EVM that only simulates network operations relevant to identifying vulnerabilities in smart contracts. Their empirical evaluation also provides evidence that sFuzz outperforms ContractFuzzer and Oyente regarding branch coverage and the total number of revealed vulnerabilities [Nguyen et al. 2020].

The ILF fuzzer for smart contracts employs an *imitation learning* strategy [Hussein et al. 2017] to enable the test generation process to learn from symbolic execution outcomes—thus generating inputs capable of traversing deep program paths [He et al. 2019]. [He et al. 2019] investigated whether ILF achieves higher coverage and discovers more vulnerabilities compared to other security tools for Ethereum, including the fuzzers Echidna [Grieco et al. 2020] and ContractFuzzer [Jiang et al. 2018]. Their empirical study results indicate that ILF outperforms the other tools in terms of both code coverage and the number of vulnerabilities detected. Furthermore, the authors provide evidence that the *learning component* is essential for achieving higher performance. Differently, the design of Smartian aims to identify sequences of transactions that can critically modify the shared state of smart contracts [Choi et al. 2021], exploring a particular feature of smart contracts: sequences of transactions share a persistent state. To achieve this, Smartian combines static and dynamic analysis of the EVM code. [Choi et al. 2021] empirically evaluate Smartian using a curated benchmark (BENCH72) that comprises 72 bug-labeled smart contracts. The results show that Smartian outperforms other smart contract fuzzers (such as sFuzz and ILF) and symbolic execution tools (such as Oyente and teEther). [Choi et al. 2021] also explore Smartian in the wild, using a dataset (BENCH500) that contains 500 widely used and complex smart contracts.

Other techniques and fuzzers for detecting vulnerabilities in smart contracts do exist, and [Chu et al. 2023] present a survey on this subject. In this paper, we introduce our

smart contract fuzzing infrastructure (DogeFuzz), which currently supports three strategies for input generation: black-box fuzzing and grey-box fuzzing, utilizing both code coverage feedback and critical instruction-guided feedback. We present the main design decisions and an empirical assessment of DogeFuzz in Sections 3 and 4, respectively. Surprisingly, DogeFuzz outperforms sFuzz and ILF in BENCH72, even though its fuzzing strategies seem simpler compared to those employed by these tools. Similar to ContractFuzzer, DogeFuzz uses the Go Ethereum implementation of the EVM.

3. DogeFuzz Design Guidelines

This section outlines key decisions behind DogeFuzz's implementation. DogeFuzz aims to be a flexible, extensible tool for identifying vulnerabilities in Ethereum smart contracts via fuzzing. It currently supports classical black-box, coverage-guided, and a novel directed grey-box fuzzing strategy targeting critical smart contract instructions.

3.1. DogeFuzz Building Blocks

DogeFuzz's flexibility comes from its web-based architecture, which separates its components: a customized Go Ethereum Virtual Machine, the Vandal Ethereum static analysis tool, the DogeFuzz fuzzer, data collection, and report modules. The Go Ethereum [Ethereum 2019] implementation was chosen for its stability and strong development reputation in the Ethereum community.

Indeed, advanced fuzzing strategies require contextual data to generate new inputs, necessitating EVM customization for collecting execution data. DogeFuzz gathers dynamic call graphs, instruction execution, and local state changes. This data is used to (1) determine explored paths in a smart contract and (2) capture key EVM events that may indicate vulnerabilities. We customized Go Ethereum to collect this contextual data, inheriting changes from the ContractFuzzer project and adding instruction collection during transactions. This data is sent to the fuzzer module via an HTTP server.

Also, the grey-box fuzzing strategies of DogeFuzz rely on code coverage data obtained in real-time during the fuzzer execution. For that, DogeFuzz uses an abstract *call-graph* (CG) representation of a smart contract, which we compute using the Vandal tool [Brent et al. 2018]. Vandal is a static analysis tool tailored for Ethreum smart contracts security. We use the CG representation to compute path coverage for the grey-box strategy and to compute a distance map to critical instructions in the directed grey-box strategy. We believe that opting for a standard EVM implementation like Go Ethereum enables us to explore Ethereum vulnerabilities across various platform layers [Atzei et al. 2017]. This broader exploration wouldn't be feasible with lightweight EVM implementations, despite their significant performance improvements in fuzzer input generation. We want to better understand this tradeoff in the long term.

3.2. Data Gathering

DogeFuzz collects and processes metrics during smart contract fuzzing campaigns to generate new inputs. It uses an event-based processing architecture with Go routines for real-time execution metrics. Smart contract calls require parsing inputs into contract function parameters, achieved using a Go Ethereum library that maps Go types to Solidity types.

This library also facilitates communication with an Ethereum node for executing smart contract code. These features allow DogeFuzz to generate inputs, collect execution metrics, use these metrics during fuzzing campaigns, and identify possible vulnerabilities.

3.3. Fuzzing Strategies

DogeFuzz currently supports three fuzzing strategies: black-box, grey-box, and directed grey-box fuzzing. While the black-box fuzzing strategy only generates random inputs to a given smart contract, the grey-box fuzzing strategy uses coverage data to rank each generated input by the amount of code being explored. As such, the grey-box strategy uses a control flow graph (CFG) to compute the paths of the smart contract and uses the executed instructions to match each executed branch. DogeFuzz uses the coverage data of executed branches to compute the best past execution. With the inputs of these executions, small mutations are made to generate similar inputs that are more prone to better explore the smart contract.

The directed grey-box fuzzing uses a CFG to compute a separated distance map. This map contains the distances (according to the control-flow graph) to specific instructions classified as critical [Krupp and Rossow 2018]. We consider four EVM instructions as critical in this work: `CALL` (creates a regular transaction), `SELFDESTRUCT` (makes a contract unusable), `CALLCODE` (calls functions in other contracts using the original contract's context), and `DELEGATECALL` (calls functions in other contracts inside the target contract's context).

Those instructions were classified in that way because they are related to the studied vulnerabilities. With that, the directed grey-box fuzzing strategy matches the executed instructions to compute how close the fuzzer is getting to those specific statements of the smart contract. The DogeFuzz directed grey-box strategy uses that to rank the best inputs to be used as seeds to the next input generation cycle. These fuzzing strategies were implemented using a common *power schedule* component [Böhme et al. 2016], which is responsible for ordering the inputs according to each fuzzing strategy. The other important component is how DogeFuzz detects the vulnerabilities at runtime, using the data collected by the customization made in the EVM.

3.4. Bug Oracles

In order to spot vulnerabilities during fuzzing campaigns, DogeFuzz leverages the same approach as other fuzzers: it collects and analyzes event patterns from the instrumented EVM. Currently, DogeFuzz collects nine events: Delegate (D), GaslessSend (GS), SendOp (SO), ExceptionDisorder (ED), BlockNumber (BN), Timestamp (T), Reentrancy (R), StorageChanged (SC), and EtherTransfer (ET). One or more event combinations can indicate a vulnerability, which the bug oracles analyze. While the fuzzer explores possible smart contract paths, interpreting execution data is crucial for thorough security analysis. DogeFuzz uses event sequences to pattern match and identify vulnerabilities in a smart contract. These bug oracles follow a standard implementation, taking snapshots of transaction events to compute each combination.

4. Study Settings

The goal of our empirical assessment is threefold: (a) to estimate the improvements of DogeFuzz grey-box fuzzing strategies in comparison with the black-box fuzzing strategy

we borrow from ContractFuzzer, (b) to investigate how DogeFuzz performs compared to state-of-the-art fuzzers for smart contracts, and (c) to understand how DogeFuzz performs "in the wild"—considering popular and non-trivial smart contracts. To this end, we conduct two experiments that aim to answer the following research questions:

(RQ1) How does the DogeFuzz bug-finding effectiveness compare to state-of-the-art fuzzers for Ethereum?
(RQ2) How efficient is DogeFuzz in fuzzing large-scale, real-world Ethereum smart contracts?

(RQ1) deals with the first and second goals of this empirical assessment, while (RQ2) deals with our third goal. We use a quasi-replication research method, replicating an empirical assessment conducted in a previous study [Choi et al. 2021] to address our research questions. Throughout our paper, whenever we refer to the "original study," we are referring to [Choi et al. 2021]. Our research validates and extends the findings of the original study by evaluating two metrics often used by the fuzzing research community: code coverage and bug detection effectiveness. We compare the DogeFuzz strategies with three modern fuzzers: sFuzz, ILF, and Smartian.

Benchmarks. We use two distinct benchmarks in our experiments: BENCH72 and BENCH500. BENCH72 consists of 72 smart contracts and was curated by the Smartian team [Choi et al. 2021], who gathered a subset of samples from the SmartBug benchmark [Durieux et al. 2020]. We used BENCH72 to answer (RQ1). BENCH72 exhibits different classes of vulnerabilities, totaling 82 labeled bugs across multiple contracts (a single contract might bear more than one bug). The types of vulnerabilities in BENCH72 are: Block State Dependency (**BD**) with 13 instances, Mishandled Exception (**ME**) with 50 instances, and Reentrancy (**RE**) with 19 instances. Table 1 shows a mapping between DogeFuzz bug oracles, SWC weakness IDs and the BENCH72 bug oracle names. DogeFuzz distinguishes between Timestamp Dependency and Block Number Dependency, but we merge them into Block State Dependency for compatibility with BENCH72. Additionally, Smartian considers Gasless Send, Dangerous Delegate Call, and Exception Disorder as specific cases of Mishandled Exception.

Table 1. Bug Oracles mapping.

DogeFuzz Bug Oracle	SWC	BENCH72 Taxonomy
Reentrancy	SWC-107	RE
Dangerous Delegate Call	SWC-112	ME
Gasless Send	SWC-104	ME
Exception Disorder	SWC-104	ME
Number Dependency	SWC-120	BD
Timestamp Dependency	SWC-120	BD

To address our second research question (RQ2), we utilize a second benchmark called BENCH500. This dataset includes 500 widely-used smart contracts from the Ethereum mainnet, each with over 30,000 transactions and no labeled bugs (meaning they have not been explicitly identified or marked as containing known vulnerabilities). Nevertheless, some contracts in this benchmark did not compile in our environment due

to missing pragma directives and even syntactic errors. As a result, we excluded 91 samples from our assessment, resulting in our second dataset with 409 smart contracts. The Smartian team was also responsible for curating BENCH500 [Choi et al. 2021], selecting smart contracts from Etherscan [Etherscan 2013].

Baselines. In our experiments, we conduct a comparative analysis between DogeFuzz and three advanced open-source fuzzers: sFuzz, ILF, and Smartian as the baselines. ContractFuzzer was omitted from our evaluation because we also integrated the black-box strategy of ContractFuzzer into DogeFuzz. Our selection of ILF [He et al. 2019] was motivated by its demonstrated superiority over existing smart contract fuzzers. Additionally, ILF incorporates modern symbolic execution techniques in conjunction with imitation learning, enhancing its effectiveness. sFuzz [Nguyen et al. 2020] was chosen for its foundation on AFL [AFL 2013], a widely recognized fuzzing tool, and its role as a base for numerous derivative fuzzers [Ji et al. 2023, Xue et al. 2024]

Experimental Procedures. We conduct different runs of the first benchmark (BENCH72) for one hour per contract using each tool. We repeat each experiment five times to minimize statistical errors and then record the average results. In evaluating all target tools (ILF, sFuzz, Smartian), we leverage the same scripts, programs, and Docker images from the Smartian team [Choi et al. 2021] artifact repository.[2] To ensure compatibility in the DogeFuzz fuzzing campaigns, we integrate custom scripts and helpers for comparison purposes. For the second benchmark (BENCH500), we conduct a 15-minute run for each contract utilizing all three of our fuzzing strategies (black-box, grey-box, and directed grey-box). Since this dataset serves as the basis for evaluating coverage across different fuzzing modes, we deem one fuzzing campaign adequate. We conduct all our experiments using an Ubuntu 20.04 server featuring 42 Intel Xeon CPUs clocked at 2.2 GHz and 432 GB of RAM. We also benefit from Docker version 25.0 in our experiments, with docker-compose being used for orchestrating DogeFuzz and a single Docker image used for running the external tools.

5. Results

In this section, we present the results of our empirical assessment and answer our two research questions. The outcomes of the experiments and all the scripts we use to analyze the results are available in a replication package at zenodo.

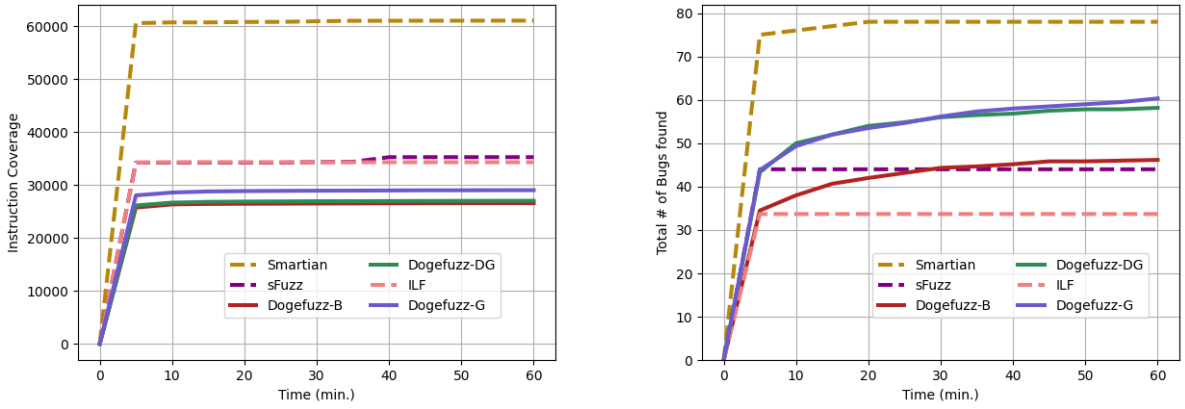
5.1. RQ1: DogeFuzz Comparison with Other Fuzzers

To answer our first research question, we compare the different variants of DogeFuzz with state-of-the-art fuzzers using BENCH72. Figures 1a and 1b summarize the results of the comparison. In the figures, DogeFuzz black-box fuzzer is denoted as **DogeFuzz-B**, the grey-box coverage feedback-guided fuzzer is named **DogeFuzz-G**, and the critical instruction directed grey-box fuzzer is named **DogeFuzz-DG**. Considering the DogeFuzz strategies, we can realize in the figure that, initially, all variants exhibit consistent performance during the first few minutes of testing. Subsequently, the DogeFuzz-G variant surpasses the others, primarily due to its strategy of seed prioritization that relies on coverage feedback.

[2]https://github.com/SoftSec-KAIST/Smartian-Artifact

On average, DogeFuzz-G increases the coverage on this dataset by 9.73% over DogeFuzz-B. DogeFuzz-DG leads to a slight increase in coverage, showing a 2.2% improvement over the black-box variant. This suggests that our directed grey-box implementation, which prioritizes seeds likely to execute code close to critical instructions, may also result in achieving higher coverage compared to black-box techniques. Note that after 30 minutes of execution, the coverage barely increases. This observation is likely due to the DogeFuzz lack of more advanced techniques to overcome hard-to-solve conditions (such as using a symbolic execution tool to enrich the fuzzing strategies).

DogeFuzz presents the smallest instruction coverage compared to other tools, partly because we filtered out functions with modifiers like view or pure. The reasoning for this decision lies in the fact that these functions, which have no influence on the contract states and are unable to alter state variables, are unlikely targets for exploitation by attackers [Li et al. 2022]. DogeFuzz also does not call fallback functions on a non-fallback contract or send ethers to functions that are not marked with the payable keyword. We argue that, despite not achieving higher code coverage compared to other tools, DogeFuzz successfully altered the persistent state of contracts, leading to the discovery of more bugs than those identified by sFuzz and ILF.



(a) Instruction coverage comparison between DogeFuzz and other tools on BENCH72.

(b) Bug detection comparison against state-of-the-art tools on BENCH72.

Figure 1. Comparison of the fuzzers in BENCH72

So, regarding bug detection comparison, Figure 1b illustrates the overall performance of each DogeFuzz variant. The figure illustrates the results of our bug-finding comparison over an one-hour period. DogeFuzz-G and DogeFuzz-DG significantly outperform the black-box variant. Interestingly, DogeFuzz-DG shows an improvement in bug-finding capability of more than 25% compared to DogeFuzz-B, despite only a marginal coverage improvement. DogeFuzz-G demonstrates a bug-finding capability slightly higher than DogeFuzz-DG, though. The results indicate that DogeFuzz-DG achieves performance levels comparable to DogeFuzz-G, despite covering fewer instructions. Although previous studies have examined the application of directed grey-box fuzzing to cover recently changed code [Böhme et al. 2017], to the best of our knowledge, we are the first to demonstrate the feasibility of guiding a fuzzer strategy for input generation that prioritizes seeds that are more likely to execute code close to EVM critical

instructions. In our replication of the original study [Choi et al. 2021], we were able to closely reproduce the results for sFuzz and Smartian. However, regarding ILF, we encountered an issue within the original study's replication package, which compromised its ability to detect bugs associated with the Block State Dependency bug oracle. By addressing this issue, we observed a $\approx 32\%$ improvement in the ILF performance compared to what was reported in the original study. We have submitted a pull request with the fix to the authors of the original study for resolution, but as of now, we have not received a response.

We can see in Figure 1b that sFuzz and ILF reach a stable point around 15 minutes, after which no new bugs are found. In contrast, DogeFuzz-G and DogeFuzz-DG continue to discover bugs until the end of the fuzzing campaigns. This is a promising result for DogeFuzz, assuming that sFuzz and ILF are still considered state-of-the-art fuzzers for smart contracts that employ advanced strategies for input generation. As discussed in the original study [Choi et al. 2021], Smartian detects substantially more bugs than sFuzz and ILF. Figure 1b shows that Smartian also outperforms the DogeFuzz strategies, possibly because it relies on a novel strategy whose fuzzing target is sequences of transactions—instead of fuzzing individual transactions which is the strategy the other three fuzzers employ.

Table 2. Accuracy by tool for each bug class in Bench72

	TP	FP	FN	Precision	Recall	F_1 score
BlockDependency						
ILF	5	0	8	1	0.38	0.56
sFuzz	10	0	3	1	0.77	0.87
Smartian	11	0	2	1	0.85	0.92
Dogefuzz-G	9	1	4	0.90	0.69	0.78
Dogefuzz-DG	9	1	4	0.90	0.69	0.78
Dogefuzz-B	8	0	5	1	0.62	0.76
MishandledException						
ILF	11	0	39	1	0.22	0.36
sFuzz	29	6	21	0.83	0.58	0.68
Smartian	48	0	2	1	0.96	0.98
Dogefuzz-G	39	9	11	0.81	0.78	0.80
Dogefuzz-DG	35	7	15	0.83	0.70	0.76
Dogefuzz-B	31	4	19	0.89	0.62	0.73
Reentrancy						
ILF	18	2	1	0.90	0.94	0.92
sFuzz	5	20	14	0.20	0.26	0.26
Smartian	19	0	0	1	1	1
Dogefuzz-G	16	4	3	0.80	0.84	0.82
Dogefuzz-DG	14	4	5	0.78	0.74	0.76
Dogefuzz-B	7	4	12	0.64	0.37	0.47

Table 2 summarizes our findings for vulnerabilities detected by each tool for each category of bugs we evaluated. Each column gives the number of true positives (TP), false positives (FP), false negatives (FN), precision (P), recall (R), and F_1 score (F1) for every tool and bug class. For instance, in the case of Reentrancy (RE), sFuzz identified 5 true positives out of a total of 19 actual vulnerabilities, with 20 false positives reported. Overall, DogeFuzz strategies correctly detect more bugs than sFuzz ($\approx$ 38,6%) and ILF ($\approx$ 84,8%). DogeFuzz-G and DogeFuzz-DG achieve an accuracy (F_1 score) above 76%, contrasting with sFuzz and ILF that achieve a F_1 score close to 60%. Smartian leads to higher accuracy, with F_1 score of 92%. Note that DogeFuzz did not achieve 100% precision, even though we reused the same bug oracles as ContractFuzzer, along with a few fixes we implemented. [Jiang et al. 2018] claim that the ContractFuzzer bug oracles are highly precise and should not generate a higher number of false positives. However, in our assessment, DogeFuzz exhibited lower precision compared to a previous assessment of ContractFuzzer. We believe that this difference can be mostly attributed to an improved understanding of the requirements needed to identify a bug that could potentially lead to a vulnerability, as well as the use of a more accurate dataset that correctly labels the bugs in the smart contracts.

We manually reviewed the contracts labeled with Block Dependency to investigate the reasons behind the low DogeFuzz recall rate. During the inspection, we discovered four contracts that had not been identified as having a Timestamp or Number dependency bug. These contracts contain conditions that use the Solidity expression `require` with specific numbers (magic numbers), which primarily represented ether values or `if` conditions with identical arguments. In complex contracts with rigid constraints, it becomes challenging to satisfy these "magic" numbers. Even fuzzers that incorporate symbolic execution struggle with these constraints due to path explosion. Other tools in our benchmark performed better than DogeFuzz by using advanced strategies to handle this issue. For instance, sFuzz uses an adaptive strategy that selects seeds based on a quantitative measure of how close a test case is to cover any recently missed branch. Similarly, Smartian addresses this challenge through concolic testing to determine the appropriate argument values [Choi et al. 2019].

5.2. RQ2: DogeFuzz Assessment in the Wild

To answer our second research question, we execute DogeFuzz variants DogeFuzz-B, DogeFuzz-G, and DogeFuzz-DG using BENCH500—which comprises 500 popular Ethereum smart contracts [Choi et al. 2021]. Our goal here is to understand how DogeFuzz scales with more complex smart contracts. Unfortunately, there is missing information in the BENCH500 contracts, preventing DogeFuzz from compiling or executing 91 smart contracts. Ultimately, we address RQ2 using 409 contracts from BENCH500. Here, we execute each fuzzing campaign for 15 minutes. Figure 2 summarizes the code coverage achieved using each DogeFuzz variant. Executing the fuzzers DogeFuzz-G and DogeFuzz-DG for 15 minutes led to a median code coverage of about $\approx 48\%$, while the black-box strategy achieved a median code coverage of close to $\approx 40\%$.

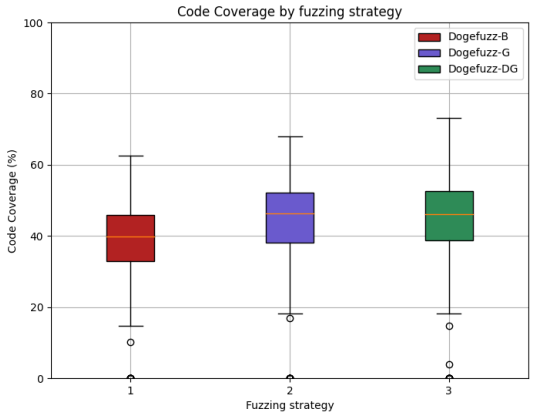


Figure 2. Code coverage comparison between DogeFuzz fuzzing strategies on BENCH500.

This time, DogeFuzz-DG achieves a code coverage quite similar to DogeFuzz-G. Nonetheless, when considering the bug-finding capabilities, DogeFuzz-G found more bugs (see Table 3): three more Reentrancy bugs and five more Gasless Send bugs. Since the authors of BENCH500 do not label the bugs in each contract, we cannot investigate the precision and recall of the DogeFuzz variants in this experiment. However, we identified

Table 3. Bugs Reported in BENCH500.

DogeFuzz Bug	DogeFuzz-G	DogeFuzz-DG
Reentrancy	16	13
Dangerous Delegate Call	0	0
Gasless Send	29	24
Exception Disorder	2	2
Number Dependency	7	8
Timestamp Dependency	78	78
Totals	132	125

that many of the bugs DogeFuzz-G found, DogeFuzz-DG also found. As we mentioned, this is an unexpected result since DogeFuzz-G and DogeFuzz-DG use pretty different strategies for seed prioritization. A possible explanation for this unexpected result comes from the fact that smart contracts often contain a small number of lines of code and conditional statements. As such, both strategies achieve almost the same set of executed instructions. Additionally, we found that critical instructions often appear in the contracts we analyzed (see Table 4), which might actually cause our guided grey-box strategy to yield similar results to the coverage-guided grey-box strategy.

Table 4. Usage of critical instructions in BENCH72

	Instruction	Total	Average (per contract)
1	`CALL`	171	2.41
2	`CALLCODE`	4	1.33
3	`DELEGATECALL`	2	1.00
4	`SELFDESTRUCT`	33	5.50

6. Discussion and Threats To Validity

The results of our empirical study demonstrate the impact of employing grey-box fuzzing strategies in the realm of smart contracts. Although previous research has already presented the benefits of using directed grey-box [Böhme et al. 2017] that use as target recently changed code, our strategy is novel and, intriguingly, yields a performance almost identical to the classical coverage-guided strategy. Since we evaluated the DogeFuzz strategies using only two datasets (already published in the literature), we cannot generalize our results. Still, we consider our current findings promising, particularly akin to the design of other fuzzers; the extensibility of DogeFuzz will enable us (and hopefully other groups) to experiment with new fuzzing strategies and their combinations.

The findings of our study have some implications for the field of smart contract fuzzing. Some of them were unexpected. For instance, our comparative analysis, using well-known accuracy metrics (i.e., precision, recall, and F_1 score), highlights the efficacy of grey-box fuzzing techniques, particularly DogeFuzz-G and DogeFuzz-DG, performing better than more advanced techniques used by sFuzz and ILF. Surprisingly, even employing the black-box strategy of DogeFuzz, borrowed from ContractFuzzer, yielded performance comparable to that of sFuzz and ILF. This finding is intriguing because other fuzzers often regard the ContractFuzzer black-box approach as

outdated [Nguyen et al. 2020, He et al. 2019, Wüstholz and Christakis 2020], resulting in previous research excluding black-box strategies from their empirical studies. In this paper, we address this gap in the literature. Our research results also provide evidence that Smartian performs better than DogeFuzz. This result might be partially due to Smartian fuzzing sequences of transactions, instead of individual transactions that is the approach DogeFuzz, sFuzz, and ILF follows.

As mentioned before, our choice of benchmarks might threaten the validity of our study. We conducted experiments on two benchmarks derived from Smartian, namely BENCH500 and BENCH72. These benchmarks contain known bugs and consist of large, popular real-world contracts. Although we evaluated our technique across many contracts, we cannot generalize the results of this paper to contracts available in other benchmarks. Another limitation of our study arises from our dependence on Vandal for the static analysis phase of DogeFuzz. This tool has not received updates since 2020, and thus our choice to use Vandal might have slightly compromised the performance of DogeFuzz. To address this limitation, future work involves developing or utilizing an updated version of DogeFuzz that employs more advanced tools to construct the call and control flow graphs. Also, a potential issue arises from the possibility of introducing errors during the setup and configuration of the empirical study. However, we carefully and systematically mapped bugs from Smartian to our bug oracles and tailored scripts and files accordingly. Indeed, our procedures even revealed a bug in the setup of previous studies, which we fixed and reviewed the performance measurements of ILF—which is slightly better than reported in previous studies [Choi et al. 2021].

7. Final Remarks

In this paper, we presented DogeFuzz, an extensible fuzzer for Ethereum smart contracts that currently supports three strategies: black-box fuzzing, coverage-guided grey-box fuzzing, and a novel directed grey-box, which aims to prioritize inputs that approximate the fuzzing campaigns to critical Ethereum VM instructions. We compared the performance of DogeFuzz using two benchmarks available in the literature (BENCH72 and BENCH500) [Choi et al. 2021], and our results show that DogeFuzz grey-box strategies present pretty similar results in terms of bug-finding capabilities. Moreover, DogeFuzz grey-box strategies outperform two cutting-edge fuzzers (sFuzz and ILF) in BENCH72, even though these fuzzers integrate advanced techniques for input generation (such as symbolic execution and imitation learning). Nonetheless, we found that Smartian outperforms the DogeFuzz fuzzing strategies.

Artefact Availability

We make our data, scripts and source code publicly available for further investigations in Github at dogefuzz_sbseg_artifact and dogefuzz

References

AFL (2013). American fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: April 02, 2024.

Atzei, N., Bartoletti, M., and Cimoli, T. (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, pages 164–186, Berlin, Heidelberg. Springer, Springer-Verlag.

Böhme, M., Pham, V., Nguyen, M., and Roychoudhury, A. (2017). Directed greybox fuzzing. In Thuraisingham, B., Evans, D., Malkin, T., and Xu, D., editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344, New York, NY, USA. ACM.

Böhme, M., Pham, V.-T., and Roychoudhury, A. (2016). Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA. Association for Computing Machinery.

Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., and Scholz, B. (2018). Vandal: A scalable security analysis framework for smart contracts.

Choi, J., Jang, J., Han, C., and Cha, S. K. (2019). Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 736–747, Montreal, Quebec, Canada. IEEE Press.

Choi, J., Kim, D., Kim, S., Grieco, G., Groce, A., and Cha, S. K. (2021). SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 227–239, Melbourne, Australia. IEEE.

Chu, H., Zhang, P., Dong, H., Xiao, Y., Ji, S., and Li, W. (2023). A survey on smart contract vulnerabilities: Data sources, detection and repair. *Inf. Softw. Technol.*, 159:107221.

Durieux, T., Ferreira, J. a. F., Abreu, R., and Cruz, P. (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*, ICSE '20, page 530–541, New York, NY, USA. Association for Computing Machinery.

Ethereum, G. (2019). go-ethereum. https://geth.ethereum.org/. Accessed: April 09, 2024.

Etherscan (2013). Etherscan. https://etherscan.io/. Accessed: April 02, 2024.

Grieco, G., Song, W., Cygan, A., Feist, J., and Groce, A. (2020). Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 557–560, New York, NY, USA. Association for Computing Machinery.

He, J., Balunovic, M., Ambroladze, N., Tsankov, P., and Vechev, M. T. (2019). Learning to fuzz from symbolic execution with application to smart contracts. In Cavallaro, L., Kinder, J., Wang, X., and Katz, J., editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 531–548, New York, NY, USA. ACM.

Hussein, A., Gaber, M. M., Elyan, E., and Jayne, C. (2017). Imitation learning: A survey of learning methods. *ACM Comput. Surv.*, 50(2):21:1–21:35.

Ji, S., Wu, J., Qiu, J., and Dong, J. (2023). Effuzz: Efficient fuzzing by directed search for smart contracts. *Information and Software Technology*, 159:107213.

Jiang, B., Liu, Y., and Chan, W. K. (2018). ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, New York, NY, USA. ACM.

Krupp, J. and Rossow, C. (2018). teEther: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, Baltimore, MD. USENIX Association.

Li, B., Pan, Z., and Hu, T. (2022). Redefender: detecting reentrancy vulnerabilities in smart contracts automatically. *IEEE Transactions on Reliability*, 71(2):984–999.

Luu, L., Chu, D., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In Weippl, E. R., Katzenbeisser, S., Kruegel, C., Myers, A. C., and Halevi, S., editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269, New York, NY, USA. ACM.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44.

Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., and Minh, Q. T. (2020). sfuzz: an efficient adaptive fuzzer for solidity smart contracts. In Rothermel, G. and Bae, D., editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 778–788, New York, NY, USA. ACM.

OpenZeppelin (2017). On the parity wallet multisig hack. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7. Accessed: April 09, 2024.

OpenZeppelin (2020). Exploiting uniswap: From reentrancy to actual profit. https://blog.openzeppelin.com/exploiting-uniswap-from-reentrancy-to-actual-profit. Accessed: April 09, 2024.

Wu, S., Li, Z., Yan, L., Chen, W., Jiang, M., Wang, C., Luo, X., and Zhou, H. (2024). Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 127:1–127:13. ACM.

Wüstholz, V. and Christakis, M. (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*, ESEC/FSE 2020, page 1398–1409, New York, NY, USA. Association for Computing Machinery.

Xue, Y., Ye, J., Zhang, W., Sun, J., Ma, L., Wang, H., and Zhao, J. (2024). xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 21(2):515–529.

Zeller, A., Gopinath, R., Böhme, M., Fraser, G., and Holler, C. (2024). Fuzzing: Breaking things with random inputs. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. Retrieved 2024-01-18 18:11:45+01:00.