SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Analyses

Jaeseung Choi*
KAIST
jschoi17@kaist.ac.kr

Doyeon Kim*†
LINE Plus Corporation
doyeon1017@linecorp.com

Soomin Kim
KAIST
soomink@kaist.ac.kr

Gustavo Grieco
Trail of Bits
gustavo.grieco@trailofbits.com

Alex Groce
Northern Arizona University
alex.groce@nau.edu

Sang Kil Cha
KAIST
sangkilc@kaist.ac.kr

***Abstract*—Unlike traditional software, smart contracts have the unique organization in which a sequence of transactions shares persistent states. Unfortunately, such a characteristic makes it difficult for existing fuzzers to find out critical transaction sequences. To tackle this challenge, we employ both static and dynamic analyses for fuzzing smart contracts. First, we statically analyze smart contract bytecodes to predict which transaction sequences will lead to effective testing, and figure out if there is a certain constraint that each transaction should satisfy. Such information is then passed to the fuzzing phase and used to construct an initial seed corpus. During a fuzzing campaign, we perform a lightweight dynamic data-flow analysis to collect data-flow-based feedback to effectively guide fuzzing. We implement our ideas on a practical open-source fuzzer, named SMARTIAN. SMARTIAN can discover bugs in real-world smart contracts without the need for the source code. Our experimental results show that SMARTIAN is more effective than existing state-of-the-art tools in finding known CVEs from real-world contracts. SMARTIAN also outperforms other tools in terms of code coverage.**

I. INTRODUCTION

Bugs in smart contracts can cause catastrophic failures because smart contracts often handle digital assets worth millions of dollars. In the notorious DAO attack in 2016 [23], for example, the attacker exploited a reentrancy bug in a smart contract to steal 3.6 million ether, which was worth 70 million USD at that time.

Understandably, there has been surging research interest in automatically finding bugs in smart contracts [24], [32], but to our knowledge, all the existing tools we found suffer from one or more of the following issues.

First, the tools neglect to emit test cases needed for coverage measurement. Many tools do not produce *replayable* test cases, and output incomplete information about transactions. Moreover, some tools focus only on bug-triggering test cases and ignore test cases that increase coverage. This makes it hard to quantitatively compare the coverage achievement of testing tools (see §II-C). Second, research papers in the field do not always provide their implementation or publish data sets used in the evaluation. This problem has been noted in another recent study [25]. Third, many of the tools focus only on a small set of bug classes, which significantly limits their usability. For instance, Echidna [31] can only detect assertion failures and check custom properties.

All these observations suggest a need for a practical testing tool that is (1) able to produce replayable test cases, (2) publicly available, and (3) able to find a set of various bug classes. Although fuzzing is a plausible technique to achieve these requirements, none of the current smart contract fuzzers satisfies them all.

Nevertheless, those are not the only requirements; there is a critical technical challenge in current fuzzers in handling stateful transactions. Smart contracts differ from traditional applications in that they take in a sequence of transactions as input while maintaining a *persistent state*. The main challenge in smart contract testing is to find a transaction sequence that can change the persistent state of the target contract in a critical way. Unfortunately, traditional code coverage feedback may not be effective enough for identifying such important transaction sequences. That is, two transaction sequences may achieve exactly the same branch coverage, although only one of them can change the persistent state in a meaningful way.

Previous fuzzers *partly* handle this problem either by randomly varying transaction orders [31], [53], [70] or by resorting to machine learning [35]. However, none of the approaches is deterministic, and thus, all of them are prone to potential failure in detecting crucial transaction sequences.

In this paper, we address this challenge by leveraging both static and dynamic analyses on EVM bytecode. The key intuition is that the significance of transaction sequences can be determined by the data dependencies between functions and persistent state variables. Therefore, analyzing data flows of persistent state variables can help in identifying critical transaction sequences.

In particular, we use both static and dynamic analyses for (1) generating an initial seed pool; and (2) evolving the seed pool at runtime. First, we statically analyze the target smart contracts (in the form of raw EVM bytecode) to figure out meaningful transaction orders, which can effectively modify the persistent states, as well as their sender constraints. Each

*Co-first authors.
†This work was done when the author was at KAIST.

transaction sequence obtained in this step is deemed to be a useful seed for fuzzing. Note this is a preprocessing step that runs only once per smart contract.

Next, we run fuzzing with the initial seeds obtained by the preprocessing step. However, we note that our fuzzer also needs to be able to discern useful transaction sequences at runtime to effectively update the seed pool. Thus, we introduce data-flow-based feedback, a novel feedback mechanism that carefully monitors dynamic data flows between state variables during a fuzzing campaign.

We design and implement SMARTIAN, an open-sourced smart contract fuzzer that can systematically generate critical transaction sequences for the smart contract under test with both static and dynamic analyses. We evaluated SMARTIAN on a benchmark including 500 real-world Ethereum smart contracts we collected based on their popularity and size. SMARTIAN outperformed existing tools in terms of both code coverage and bug-finding ability. Furthermore, we found 211 bugs in real-world smart contracts. All these results suggest that our analyses enable SMARTIAN to find bugs in smart contracts effectively.

In summary, we make the following contributions.

1) We propose a novel static analysis technique for generating initial seed pools, which is complementary to any existing smart contract fuzzers.
2) We present data-flow-based feedback, a novel and systematic feedback mechanism for fuzzing smart contracts.
3) We present SMARTIAN, a grey-box fuzzer for smart contracts, which is (1) able to generate replayable test cases; (2) open-sourced; (3) able to detect a superset of bug classes handled by existing fuzzers; and (4) able to systematically generate critical transaction sequences with the help of both static and dynamic analyses.
4) We make our benchmark public, which includes 500 non-trivial, real-world smart contracts.

II. BACKGROUND

In this section, we first introduce basic terms to understand the rest of the paper. Next, we summarize classes of well-known smart contract bugs. Finally, we present a comparative study on existing bug-finding tools for smart contracts.

A. Basic Terminologies

Ethereum [26] is the most popular blockchain-based distributed computing platform. A smart contract is essentially a collection of code and data that is located on the Ethereum blockchain. Ethereum Virtual Machine (EVM) is an execution environment for running smart contracts. Generally, contract code is first written in a high-level language like Solidity [10], but eventually, it must be compiled into bytecode to run on EVM. A smart contract maintains *storage*, which is essentially a key-value store for holding persistent *state variables*. Storage is different than *memory* or *stack* as its contents are non-volatile. To execute a function defined in the contract, a user needs to make a *transaction* to the contract. A transaction contains information about a function call, such as parameter values. Both a contract and a user are assigned a unique address, and can have a certain amount of digital cash called *ether*. A transaction is also used to transfer ether between contracts and users.

A deployer is a special user who initially publishes a smart contract on the blockchain network. Typically, the address of the deployer is saved in the *storage* during the initialization phase (see the example in Figure 1). The stored address can then be used to discern between the deployer and regular users. Although it is desirable for testing tools to be able to send transactions from both deployers and normal users, we are not aware of any existing fuzzer that can *systematically* select proper users during a fuzzing campaign.

TABLE I
BUG CLASSES SUPPORTED BY SMARTIAN.

ID	Bug Name	Description
AF	Assertion Failure	The condition of an `assert` statement is not satisfied [2].
AW	Arbitrary Write	An attacker can overwrite arbitrary storage data by accessing a mismanaged array object [12].
BD	Block State Dependency	Block states (e.g. timestamp, number) decide ether transfer of a contract [36], [44].
CH	Control-flow Hijack	An attacker can arbitrarily control the destination of a `JUMP` or `DELEGATECALL` instruction [1], [36].
EL	Ether Leak	A contract allows an arbitrary user to freely retrieve ether from the contract [54].
FE	Freezing Ether†	A contract can receive ether but does not have any means to send out ether [36], [54].
IB	Integer Bug	Integer overflows or underflows occur, and the result becomes an unexpected value.
ME	Mishandled Exception	A contract does not check for an exception when calling external functions or sending ether [36], [44].
MS	Multiple Send	A contract sends out ether multiple times within one transaction. This is a specific case of DoS [5].
RE	Reentrancy	A function in a victim contract is re-entered and leads to a race condition on state variables [44].
RV	Requirement Violation‡	The condition of a `require` statement is not satisfied [8].
SC	Suicidal Contract	An arbitrary user can destroy a victim contract by running a `SELFDESTRUCT` instruction [54].
TO	Tranasaction Origin Use	A contract relies on the origin of a transaction (i.e. `tx.origin`) for user authorization [3].

† While other bugs deal with safety properties, **FE** concerns a liveness property. As it is unnatural to find the *absence* of behavior with testing, we make this oracle optional, and provide a command-line option to enable it.
‡ Since the official document [11] recommends to use `require` for validating program inputs, it is debatable whether this is a bug. Thus, we make this optional.

B. Smart Contract Bug Classes

Previous research defines their own bug classes with different terminologies, and there is no general consensus among them. Thus, we study and summarize them in Table I. SMARTIAN supports the detection of all these bugs (see §IV-C3).

First, we investigated bug classes handled by existing state-of-the-art fuzzers [31], [35], [36], [43], [53], [66], [70]. We included all the bugs from these fuzzers. In addition, we examined more previous work on smart contracts [4], [9], [44], [52], [54] and selected bugs that can be detected without excessive false positives.

As a result, we identified 13 types of bugs listed in Table I. While some papers make finer classifications of bugs, we tried

TABLE II
COMPARISON OF EXISTING BUG-FINDING TOOLS FOR SMART CONTRACTS.

Tool	Kind	Replayable Test Case[a]	Public Tool	Available Benchmark	Byte Code	Bug Oracle												
						AF	AW	BD	CH	EL	FE	IB	ME	MS	RE	RV	SC	TO
MadMax [30]	Static analyzer	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
Remix [57]	Static analyzer	✗	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓	✗	✓	✓
SASC [73]	Static analyzer	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Securify [64], [65]	Static analyzer	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓
Slither [27]	Static analyzer	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓
SmartCheck [62]	Static analyzer	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓	✓	✓	✗	✓	✗	✗	✓
Vandal [15]	Static analyzer	✗	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓	✗	✓	✗	✓	✓
VeriSmart [60]	Static analyzer	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗
Zeus [38]	Static analyzer	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	✗	✓	✗	✗	✓
Maian [54]	Symbolic executor	✗	✓	✗	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗
Manticore [50]	Symbolic executor	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✓
Mythril [52]	Symbolic executor	△[b]	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
Osiris [63]	Symbolic executor	✗	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗
Oyente [44]	Symbolic executor	✗	✓	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗
sCompile [16]	Symbolic executor	?[c]	✗	✗	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓	✗	✓	✗
teEther [39]	Symbolic executor	△	✓	✗	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗
ContractFuzzer [36]	Fuzzer	✗	✓	✓	✓	✗	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗
ContraMaster [66], [67]	Fuzzer	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗
Echidna [31]	Fuzzer	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Harvey [70], [71]	Fuzzer	?	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗
ILF [35]	Fuzzer	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗
Reguard [43]	Fuzzer	?	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
sFuzz [53]	Fuzzer	✗[d]	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗
★ SMARTIAN	Fuzzer	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

[a] Does the tool generate test cases that contain complete information needed to reproduce the transactions, enabling the measurement of coverage achievement?
[b] Prints to the terminal only about test cases that trigger bugs, and ignore test cases that increase coverage.
[c] We cannot identify the fact as the tool is not publicly available.
[d] Older version of sFuzz could generate replayable test cases, but this functionality disappeared in the latest version.

to merge closely related bug classes into one. For instance, ContractFuzzer [36] distinguishes *Timestamp Dependency* and *Block Number Dependency*, but we merge them into *Block State Dependency*. Also, we consider *Gasless Send* [36] as a specific case of *Mishandled Exception*.

We note that existing tools implement different bug oracles. Thus, in our evaluation, we carefully consider this difference to make fair comparisons (see §V-C).

C. Existing Tools

We studied 23 different tools for finding smart contract bugs, and summarized them in Table II.

The third column indicates whether a tool can generate test cases that contain complete information to reproduce the transactions. While it is natural for static analyzers to not generate test cases, we found that surprisingly many testing tools do not generate replayable test cases. Some tools, such as Oyente, simply emit terse information about inputs and do not output complete data needed to reproduce transactions. This makes it infeasible to reproduce bugs found or to measure coverage achievements. As a result, one can evaluate the tools only by looking at their textual reports, which are often prone to errors. For example, [53] states that Oyente reports infeasible paths as feasible.

The fourth and fifth columns respectively represent whether a tool and a benchmark are publicly available. Although it is crucial to publicize benchmarks for reproducing research [25], only a few of those tools make their benchmarks public.

The sixth column shows whether a tool runs at a bytecode level. As Ethereum deploys smart contracts in the form of bytecode, this enables testing smart contracts in the blockchain even if the source code is not available.

Finally, the rest of the columns present bug oracles employed by each tool. While all the other tools focus on a specific set of bug oracles, SMARTIAN handles everything as shown in the table.

```
1  contract C {
2    // State variables in the storage.
3    address owner = 0;
4    uint private stateA = 0;
5    uint private stateB = 0;
6    uint CONST = 32;
7
8    function C() { // Constructor
9      owner = msg.sender;
10   }
11   function f(uint x) {
12     if (msg.sender == owner) { stateA = x; }
13   }
14   function g(uint y) {
15     if (stateA % CONST == 1) {
16       stateB = y - 10;
17     }
18   }
19   function h() {
20     if (stateB == 62) { bug(); }
21   }
22 }
```

Fig. 1. Example smart contract.

III. OVERVIEW

In this section, we first present a motivating example to describe a unique challenge in smart contract fuzzing. We then briefly describe how SMARTIAN addresses this challenge by employing both static and dynamic analyses.

A. Motivating Example

Smart contracts impose a unique challenge to fuzzing due to their intrinsic structure where multiple transactions are interconnected to each other with persistent state variables.

Consider our motivating example in Figure 1, which has the constructor `C`, along with the three functions `f`, `g`, and `h`. While our system operates on EVM bytecode, the example code is written in Solidity for ease of explanation.

The constructor `C`, which simply stores the address of the deployer in the storage, runs once when the deployer instantiates the contract. This is indeed a commonly found

pattern as it provides means to distinguish the deployer from regular users. Note that `msg.sender` is an expression that evaluates to the current sender's address at runtime.

The contract has a bug in `h`, which can be triggered only if `stateB` is 62. For example, one can trigger the bug with a transaction sequence `[f(33),g(72),h()]`. Note that the three transactions should be made in the exact order to trigger the bug. Moreover, `f` should be sent by the deployer.

At a first glance, the conditions in each function may not seem so hard to satisfy. For example, one can penetrate the condition in Line 15 with the probability of $1/32$ by randomly mutating `stateA`. The condition in Line 20 is relatively harder to solve, but recent advances in grey-box fuzzing provide practical solutions to it [13], [18], [20], [53]. In our implementation, we adopt grey-box concolic testing technique from Eclipser [20].

However, finding this bug is still challenging as we need to generate the transactions in the correct order. For instance, let us assume that we have a sequence `[f(*),h(),g(*)]` as a seed[1], where `*` can be any value. Any mutation attempt on the function arguments will not trigger the bug because `h` cannot observe any difference for `stateB`. Therefore, our fuzzer needs to have a transaction sequence such as `[f(*),g(*),h()]` in the seed pool to find the bug.

One may argue that grey-box fuzzers are able to discover such a critical transaction sequence by randomly mutating transaction orders. However, it is *not* as trivial as it seems. Even if we manage to generate a sequence `[f(*),g(*),h()]` by randomly trying different transaction orders, we cannot realize that this is indeed a meaningful test case because traditional code coverage is *not* sensitive enough. For instance, consider two transaction sequences S_A = `[f(33),g(0),h()]` and S_B = `[f(33),h(),g(0)]`, which achieve the same branch coverage. If S_B was already in our test case pool, our fuzzer would have no chance to add S_A to the seed pool, even though it is the critical one.

Preliminary experiments: Despite the simplicity of the example contract in Figure 1, none of the existing fuzzers that we tested was able to find the bug. Specifically, we ran three open-sourced smart contract fuzzers, Echidna, ILF, and sFuzz, for one hour each. On the other hand, SMARTIAN was able to find the bug within just a few seconds.

B. Our Approach

To address the aforementioned challenges, SMARTIAN leverages both static and dynamic analyses. Figure 2 outlines the overall architecture of SMARTIAN. At a high level, our system runs in three major steps: (1) INFOGATHER, (2) SEEDPOOLINIT, and (3) DATAFLOWFUZZ.

1) INFOGATHER*:* First, SMARTIAN takes in the EVM bytecode under test as input and runs a static analysis to collect useful data-flow facts to guide both SEEDPOOLINIT and DATAFLOWFUZZ. Specifically, for each function in the contract, SMARTIAN figures out which state variables are

[1]In this paper, we interchangeably use the terms "transaction sequence", "test case", and "seed".

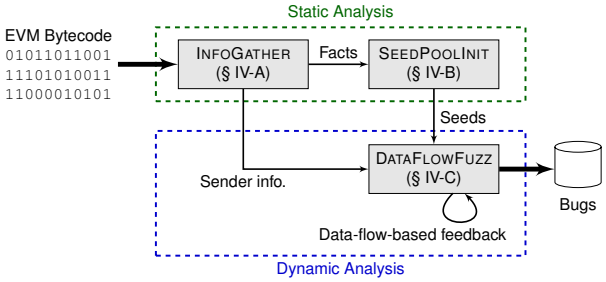

Fig. 2. SMARTIAN architecture.

defined and used by the function, and whether the function compares the transaction sender address against the deployer address. From our example contract, our static analysis will gather the following facts.

- `stateA` is defined by `f` and used by `g`.
- `stateB` is defined by `g` and used by `h`.
- `owner` is defined by `C` to be the deployer's address.
- `owner` is used by `f` to check the transaction sender.

Since our analysis directly runs on the low-level EVM bytecode, gathering such information is *not* trivial (see §IV-A).

2) SEEDPOOLINIT*:* Based on the gathered information, SMARTIAN predicts which transaction sequences are likely to lead to meaningful exploration, and consider them as initial seeds. Specifically, SMARTIAN realizes that `f` must precede `g` to explore execution paths affected by `stateA`. Similarly, it infers that `g` must be called before `h` to change the value of `stateB` and explore more paths in `h`. Finally, it figures out that `f` must be executed by the deployer in order to pass the sender check we identified in the first step. Consequently, SMARTIAN creates S_0 = `[f(0),g(0),h()]` as the initial seed for fuzzing, while making sure that the sender of `f` is the deployer. The transaction argument is set to 0 by default. See §IV-B for the detailed algorithm for seed initialization.

3) DATAFLOWFUZZ*:* While initializing the seed pool with meaningful transaction sequences can increase the probability to find the bug, this does not immediately solve the whole challenge. Ideally, we will first randomly mutate the given seed S_0 and obtain a new seed S_1 = `[f(33),g(0),h()]`, which can reach Line 16. If we can identify S_1 as a critical test case, we will add it to the test case pool and later apply the aforementioned grey-box concolic testing to figure out the proper argument value of `g` that triggers the bug (see §IV-C).

Unfortunately, as we emphasized in §III-A, we may fail to discern such critical intermediate seeds if we merely employ existing code coverage feedback. Assume that we accidentally generated S_2 = `[f(33),g(0)]` prior to S_1, and added S_2 to the test case pool. This is possible because our fuzzer can randomly add, remove, or reorder transaction(s) from a given seed. Once S_2 is added to the test case pool, S_1 will no longer be considered interesting because it provides no coverage gain over the existing seeds $\{S_0, S_2\}$.

To mitigate this problem, we employ a *dynamic* data-flow analysis to collect data-flow-based feedback. At a high level,

our approach considers the data-flow coverage as fuzzing feedback along with the branch coverage used by Eclipser [20] That is, we adopt dynamic instrumentation to observe data flows that occur in the given transaction sequence at runtime, and use them as feedback too. With S_1, there is a data flow from the definition of `stateB` in Line 16 to the use of `stateB` in Line 20. However, neither S_0 nor S_2 have this data flow. Based on this, we can conclude that S_1 discovers an interesting program behavior that is not observed in S_0 nor S_2. We detail our approach in §IV-C.

4) Impact of Data-flow Analyses: With the help of both static and dynamic analyses, SMARTIAN can find the bug from the above example within just five seconds on our machine. Meanwhile, when we disabled our analyses (see §V-B), SMARTIAN failed to find the bug in one hour.

C. Our Contribution over Previous Work

Our technical contribution is twofold: (1) we are the first in systematically generating seeds for smart contract fuzzing; and (2) we use data-flow-based coverage to effectively guide smart contract fuzzing.

1) Seed Generation: Previous fuzzers suffer from systematically generating proper transaction sequences. For instance, Echidna and sFuzz generate sequences at random.

Harvey [70] partly addresses this challenge with runtime heuristics. First, Harvey forcefully mutates state variables to find which functions are affected by them. It then randomly prepends other transactions to those functions. However, this method is not scalable to complex contracts with a large number of functions. Moreover, Harvey may fail to distinguish constants, e.g., `CONST` in our motivating example, from variables, and spend its fuzzing budget to change such constants.

ILF [35] generates transaction sequences based on a machine-learning model obtained by symbolically executing smart contracts. While ILF can potentially find meaningful transaction sequences via learned models, the result is not deterministic as it is based on statistical reasoning. Moreover, our approach is complementary to ILF as we directly analyze the semantics of the contract to generate seeds.

2) Data-flow-based feedback: The use of data-flow graph coverage has been previously studied in data-flow testing [61]. However, none of the existing fuzzers except ContraMaster [67] had employed data-flow coverage as fuzzing feedback. While data-flow-based feedback shares the same key intuition with ContraMaster, ContraMaster uses data-flow coverage to decide whether to perform mutation on transaction orders. Meanwhile, we use the feedback to evaluate the generated seeds and decide whether to put them in the seed pool.

IV. DESIGN

This section presents the design details of SMARTIAN. Recall from §III, SMARTIAN operates in three major steps: INFOGATHER (§IV-A), SEEDPOOLINIT (§IV-B), and DATAFLOWFUZZ (§IV-C).

A. Information Gathering (INFOGATHER)

INFOGATHER analyzes the given EVM bytecode and returns a 4-tuple $\langle$`Funcs`, `Defs`, `Uses`, `SenderChecks`$\rangle$ where:

- `Funcs` is a set of identified functions.
- `Defs` is a map from each identified function to the state variables defined by the function.
- `Uses` is a map from each identified function to the state variables used by the function.
- `SenderChecks` is a set of functions that includes a sender-checking routine.

It starts by constructing a Control-Flow Graph (CFG) from the given EVM bytecode. It internally disassembles EVM instructions, and lifts them into an Intermediate Representation (IR). It then runs a constant propagation analysis on the IR to figure out the destinations of control-flow transfer instructions, e.g., `JUMP`, and identifies functions including constructors. We note that this step also includes resolving call edges within the contract, to enable an inter-procedural analysis. Finally, it runs the main analysis based on abstract interpretation [22].

Our main static analysis computes abstract values stored in the stack and the memory in a flow- and context-sensitive manner [49]. Tracking stack values is important because EVM is a stack-based machine that pushes instruction operands to the stack [26], [69]. Following memory values is also critical because these operands are often loaded from the memory, too. Our ultimate goal here is to figure out which state variables are defined and used by each function. To distinguish which state variable is used (or defined), we check which value is used as a key for `SLOAD` (or `SSTORE`) instruction.

To approximate values, we use a product domain [58] that entails three different domains. First, we employ the lifted integer domain [58] to trace constants. This is because smart contracts use a hard-coded unique constant as a key to access state variables of primitive data types, e.g., `uint`. Second, we use a variation of the lifted integer domain to abstract the output of hash instruction `SHA3`. This is because smart contracts access state variables of compound data types, e.g., `mapping`, through a computed hash value as a key. With both domains, we can track which specific state variable is accessed for every program point, and thus can update both `Defs` and `Uses` accordingly.

The last component of the product domain is the taint domain for tracking the flow of the deployer's address (recall from §III-B). With this domain, we compute `SenderChecks` by analyzing the following two conditions. First, we check if the constructor of a smart contract saves the deployer's address into the storage. Second, we see if a sender's address, which is returned by a `CALLER` instruction, flows into a conditional branch, and gets compared with the deployer's address. Note both flows can be easily tracked with traditional static taint analysis. If both conditions hold, then we put the function containing the conditional into `SenderChecks`.

Algorithm 1: Deriving Function Call Orders.

```
1  function GenSequences(Funcs, Defs, Uses)
2      seqs ← ∅
3      works ← InitWorks({[f] | f ∈ Funcs, Defs(f) ≠ ∅})
4      while works ≠ ∅ do
5          s ← works.pop()
6          nogain ← true
7          for f in Funcs do
8              if DataFlowGain(s || [f], Defs, Uses) then
9                  works.push(s || [f])
10                 nogain ← false
11         if nogain then
12             seqs ← seqs ∪ {s}
13     return seqs
```

B. Seed Pool Initialization (SEEDPOOLINIT)

To generate initial seeds, SMARTIAN first derives useful function call orders, based on the information gathered from INFOGATHER, and then generates concrete transaction sequences based on the orders.

1) Deriving Useful Call Orders: Algorithm 1 illustrates the decision of function call orders. It takes in as input `Funcs`, `Defs`, and `Uses` obtained from INFOGATHER, and outputs a set of function sequences.

In Line 3, we initialize the worklist (`works`) with singleton sequences containing each function in `Funcs`. We ignore functions that do not define any state variable as they cannot affect the persistent state. Next, we pull a sequence `s` out of the worklist (Line 5), and creates new sequences by appending each function in `Funcs` to `s`. We then examine each of the generated sequences with `DataFlowGain` to decide which sequence covers previously unseen data flows (Line 7–8). Line 9 pushes such sequences to the worklist, and internally removes redundant entries for greater efficiency.

Specifically, `DataFlowGain` statically approximates function-level data flows by collecting triples $\langle f_1, v, f_2 \rangle$ from a given sequence, where (1) f_1 and f_2 are functions that appear in the sequence, (2) f_1 defines v, and (3) f_2 uses that v. It returns true if a previously unseen triple is found from the sequence.

We repeatedly extend sequences in the worklist as long as a new data flow is observed. If a sequence produces no gain by extending it, we finalize the sequence by adding it to the output set (Line 11–12).

2) Generating Seeds: We now turn the generated function sequences into transaction sequences by concretizing their contents, which works mainly in two steps.

First, for every function in each transaction, we decide whether each function belongs to `SenderChecks`. If so, we set the sender of the transaction as the deployer. Otherwise, we randomly choose the sender (either deployer or a user).

Second, we need to initialize the function arguments of each transaction. Here, we consider the amount of ether to transfer as an additional argument, too. SMARTIAN internally represents each argument as a byte stream. When the target contract ships with its ABI specification, we leverage it to set the argument types as well as the length of the byte streams accordingly. When the ABI specification does not exist, SMARTIAN will simply set the length of each byte stream to a predefined maximum value. Correctly inferring data types of function arguments in the absence of ABI is beyond the scope of this paper, and we leave it as future work.

C. Data-Flow-Based Fuzzing (DATAFLOWFUZZ)

With the generated initial seed pool, SMARTIAN iteratively selects one and mutates it to generate new test cases (§IV-C1). SMARTIAN then evaluates the usefulness of the newly generated test cases by running the smart contract under test with each test case (§IV-C2). During each execution, our bug oracles check whether it is buggy (§IV-C3).

1) Mutation Methodologies: SMARTIAN employs two complementary strategies for mutating seeds. One is random mutation, and the other is grey-box concolic testing from [20]. SMARTIAN alternates between them to achieve synergy.

First, our random mutation strategy runs at both the sequence level and the transaction level. Sequence-level mutation consists of the following operations: (1) inserting a new transaction for a random function; (2) removing a random transaction; and (3) swapping two random transactions. When inserting a transaction, we refer to `SenderChecks` gathered from the static analysis and use it to decide the sender, as in §IV-B. Transaction-level mutation mainly modifies arguments of each transaction. We leverage classic mutation operators widely used in grey-box fuzzers, such as bit-flipping mutation [72]. Besides, we randomly mutate the sender of the transaction, too.

However, it is well known that random mutation can easily get stuck on conditional branches such as magic value checks [13], [18], [41]. Eclipser [20] addresses this challenge by introducing the *grey-box concolic testing* technique, which operates similarly to traditional concolic testing [29], [59], but without SMT solving [51] or expensive extra instrumentation.

2) Data-flow-based Feedback: Recall from §III-A, previous code coverage feedback is not enough to discern interesting seeds during a fuzzing campaign. To overcome this, SMARTIAN introduces data-flow-based feedback in addition to the traditional code coverage feedback. That is, SMARTIAN considers a seed as interesting when it exhibits a previously unseen data flow or covers previously unvisited code.

To collect data flows, we dynamically instrument the EVM bytecode by modifying an EVM emulator in order to monitor the storage accesses during the execution. Particularly, we capture a dynamic data flow with a *def-use chain*. Let $p_1 \xrightarrow{v} p_2$ be a def-use chain over a state variable v defined in a program point p_1, and used in a program point p_2. We can then represent def-use chains from the example in Figure 1 as follows. S_A yields def-use chains $12 \xrightarrow{\text{stateA}} 15$ and $16 \xrightarrow{\text{stateB}} 20$, while S_B only yields $12 \xrightarrow{\text{stateA}} 15$. Therefore, SMARTIAN can recognize that S_A exhibits an interesting program behavior not presented by S_B. In the actual implementation, we use an

instruction address as a program point p_i, and use the key of the storage as a state variable v.

Recall that we also check for data flows during the seed pool initialization in §IV-B. Note that in Algorithm 1 we statically approximated the data flows at the function level. Meanwhile, in fuzzing, we trace data flows that actually take place, and calculate them at instruction-level granularity. That is, we first statically analyze data flows to decide promising transaction sequences that are likely to reveal more dynamic data flows during the fuzzing. Then, we employ concrete and more fine-grained data flows as feedback at the fuzzing phase.

3) Bug Oracles: Here, we summarize our bug oracle implementation for 13 classes of bugs supported by SMARTIAN. Again, we modify the EVM emulator to implement these bug oracles. Thus, our runtime instrumentation is responsible for both collecting data-flow-based feedback and detecting bugs during the execution.

AF At the bytecode level, an assertion failure corresponds to the execution of an `INVALID` instruction. Therefore, we can precisely detect **AF** by checking if an `INVALID` instruction is executed. We note that compilers also automatically insert `assert` statements to prevent errors such as division by zero. We consider the failures from these compiler-inserted assertions as **AF**, too.

BD We leverage dynamic taint analysis to check if a block state can affect an ether transfer. We trace both direct and indirect taint flows for this. We first taint the returns of instructions that acquire the state of a block (e.g. `TIMESTAMP`, `NUMBER`). Then, we monitor if the tainted value flows into the operands of a `CALL` or `JUMPI`.

CH First, we raise an alarm if a normal user can set the destination contract of a `DELEGATECALL` into an arbitrary user contract. Second, we also report an alarm if the destination of a `JUMP` instruction is manipulatable.

EL We employ an oracle similar to that of Mythril [52], which checks if a normal user can gain ether by sending transactions to the contract. However, this is prone to false positives, because some contracts allow the deployer to hand over the ownership of the contract to another user. In such contracts, it is an intended behavior that a user can withdraw the contract's balance when the deployer allows to. To avoid such false positives, we report alarms only when the transaction sequence does not have any preceding transaction from the deployer.

IB We monitor `ADD`, `SUB`, `MUL` instructions to check if they cause an integer over/underflow. If so, we taint the resulting value, and perform a dynamic taint analysis to check whether the tainted value is used to determine the amount of ether to transfer, or is used to update the state variables. This is to avoid raising alarms on benign integer over/underflows. For example, without this taint analysis, we will raise an alarm on a safe code snippet '`if(x + y < x) revert();`'.

ME We run a taint analysis to make sure that the return value of a `CALL` instruction flows into a predicate of a `JUMPI` instruction. If there is a return value that is not used by a `JUMPI`, we report an alarm.

MS We detect multiple ether transfers taking place in a single transaction.

RE We first monitor if there is a cyclic call chain during an ether transfer, as ContractFuzzer [36] or sFuzz [53] does. In addition, we use taint analysis to identify state variables that affect this ether transfer, similarly to as we did for **BD**. Then, we report **RE** if such variables are updated *after* the transfer takes place.

RV We check for the execution of a `REVERT` instruction, which corresponds to a requirement violation.

SC We check if a normal user can execute `SELFDESTRUCT` instruction and destroy the contract. Similarly to **EL**, we reduce false positives by filtering out test cases that have any preceding transaction from the deployer in the sequence.

TO We taint the return value of `ORIGIN` instruction, and check if it flows into the predicate of a `JUMPI` instruction.

For the rest of the bug classes, we implemented the same bug oracle as ContractFuzzer [36] (**FE**) and Harvey [70] (**AW**).

D. Implementation

To implement our static analyzer, we used B2R2 [37] as a front-end to parse and disassemble EVM bytecode. The main logic of static analysis (§IV-A) is written in 1,053 source lines of F# code. The fuzzing component of SMARTIAN (§IV-C1) is implemented by extending Eclipser [19] to operate with EVM and transaction sequences, and is composed of 3,112 source lines of F# code. We used Nethermind EVM [7] for deploying contracts and emulating transactions with dynamic instrumentation. Specifically, we added 979 lines of C# code to Nethermind to implement data-flow-based feedback (§IV-C2) as well as our bug oracles (§IV-C3). We make all our source code and benchmarks public at: https://github.com/SoftSec-KAIST/Smartian.

V. EVALUATION

In this section, we answer the following research questions.

RQ1. Can our analyses improve the fuzzing effectiveness of SMARTIAN? (§V-B)

RQ2. Can SMARTIAN find known bugs more effectively compared to existing state-of-the-art tools? (§V-C)

RQ3. How does SMARTIAN perform on a large-scale benchmark? (§V-D)

A. Experimental Setup

*1) **Our Environment**:* We ran all our experiments on an Ubuntu 18.04 server machine equipped with two Intel E5-2699 v4 (2.2 GHz) CPUs and 512 GB of main memory. We used Docker 20.10.3 for our experiments, and used one container to run a tool on a single contract. We spawned at most 72 containers in parallel, and assigned a single CPU core and 6 GB of memory to each container. To compile contracts, we used `solc-0.4.25`.

TABLE III
BENCHMARKS USED.

ID	Source	Used For	Avg. SLoC	SD† SLoC	Num. of Contracts
B1	Verismart [60]	**RQ1**, **RQ2**	136	48	58
B2	SmartBug [25]	**RQ2**	51	75	72
B3	Etherscan	**RQ3**	331	277	500

† Standard Deviation.

*2) **Comparison Targets**:* We selected two fuzzers and two symbolic executors as our comparison targets. To select fuzzers, we first filtered open-sourced fuzzers that are published in top conferences, and obtained ContractFuzzer, ILF, and sFuzz. We chose ILF and sFuzz over ContractFuzzer, since the two tools respectively outperformed ContractFuzzer in their experiments [35], [53]. To select symbolic executors, we initially applied the same criteria and obtained Oyente and teEther as a result. However, we found that Oyente reports unfeasible paths as executable, according to [53]. Also, teEther supports only a small set of bug classes, making the comparison against sFuzz difficult. Thus, we chose Mythril and Manticore instead, as they support various bug classes.

For each of the selected tools, we added functionality to emit replayable test cases if the tool does not already have it. Also, we modified their code to save all the test cases that increase code coverage. We publicize the modified versions of the tools in GitHub.

*3) **Benchmarks**:* We used three distinct benchmarks for our experiments. Table III summarizes them. We make these benchmarks public as well, in order to support open science.

B1. First, we used the benchmark from VeriSmart [60], which consists of 58 real-world contracts. Each of the contracts is assigned a CVE for an integer bug (**IB**). Note that the authors of VeriSmart originally collected 60 contracts, but they confirmed that two of the CVEs were not real bugs.

B2. While **B1** is a realistic benchmark with known vulnerabilities, all the assigned CVEs are for **IB**. Thus, we constructed **B2** that contains more bug classes, by extracting contracts from SmartBug [25]. In particular, we imported contracts that have block state dependency (**BD**), mishandled exception (**ME**), or reentrancy (**RE**), as these bug classes are supported by all of our comparison targets (§V-A2). Then, we performed preprocessing such as filtering out contracts that have any argument in the constructor. This is because some of the tools we selected for comparison did not run properly on such contracts. As a result, we obtained a total of 72 contracts that contain 13 **BD**, 50 **ME**, and 19 **RE**. Note that a single contract can have multiple classes of bugs here.

B3. This benchmark comprises 500 popular and complex smart contracts obtained from Etherscan [6], which is an online platform that provides code and statistics of Ethereum smart contracts. We first downloaded contracts that have more than 30,000 transactions, and filtered out contracts that do not compile with the `solc` version we used. Then, we filtered out contracts that have any constructor argument, to make the benchmark usable by as many tools as possible. Finally, we sorted the remaining contracts based on their bytecode size, and selected the 500 largest contracts in order to gather both popular and complex contracts.

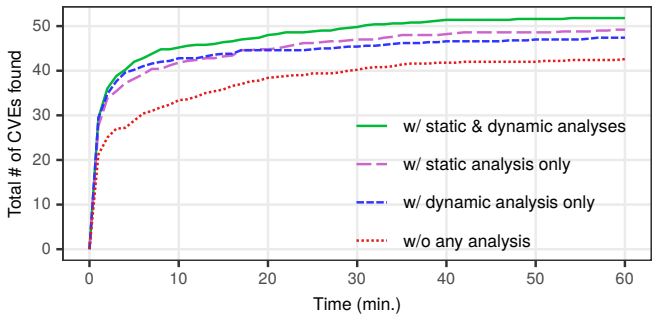

Fig. 3. Impact of our data-flow analyses on **B1**.

B. Impact of Our Analyses

Do our analyses help SMARTIAN find bugs more effectively? How much overhead do they introduce? We answer these questions by comparing the effectiveness of SMARTIAN on **B1** with and without our analyses. We used **B1** here because each contract in the benchmark contains a previously known bug, which serves as ground truth for our evaluation. Also, all the contracts in **B1** are real-world contracts, whereas some of the contracts in **B2** are artificially created toy programs.

To assess the impact of both static (§IV-A) and dynamic (§IV-C) analyses, we ran SMARTIAN in four different modes: (1) with both of the analyses, (2) only with the static analysis, (3) only with the dynamic analysis, and (4) without any analysis. We ran with each mode for one hour on **B1**, and repeated the experiment for five times.

1) Impact on Bug Finding: We first measured the impact of our analyses in terms of bug finding. We say our tool found the ground truth bug in each contract if it can pinpoint the exact program point for the assigned CVE. This is important because each contract may also have integer overflows or underflows that are irrelevant to the assigned CVE. Moreover, some of them can be benign, as we discussed in §IV-C3.

Figure 3 compares the number of unique bugs found over time with the four different modes. We note that there is a significant difference in finding deep bugs at the later stage of fuzzing. On average, we were able to find about 22% more unique bugs with our analyses than without them (p-value $<$ 0.05 from Mann-Whitney U-Test). This result confirms that there are real-world smart contract bugs, which can only be found by considering stateful transaction sequences, and our analyses indeed help in finding them.

2) Impact on Code Coverage: We also measured the number of executed instructions with and without using our analyses. As a result, we found that turning on both static and dynamic analyses helped in covering 1% more instructions

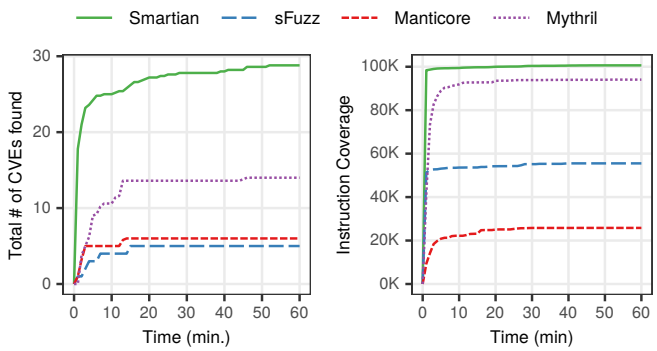

Fig. 4. Comparison against state-of-the-art tools on the subset of **B1**.

on average. While the coverage gap is not significant, recall that there was a significant difference in the number of bugs found. This observation aligns with the key motivation of our work: while the traditional code coverage feedback can be effective in guiding fuzzers to increase coverage, it may not be enough to trigger bugs when a stateful transaction sequence is required.

3) Overhead of Our Analyses: We also evaluated how much overhead is imposed by applying our analyses. For the contracts in **B1**, the static analysis took less than two seconds on average, and took five seconds in the worst case. This is indeed a negligible overhead for fuzzing. We also measured the overhead of our dynamic data-flow analysis, i.e., data-flow-based feedback. Specifically, we collected test cases generated from our previous experiments, and replayed them with and without the data-flow-based feedback computation. As a result, we observed that computing data-flow-based feedback incurred only 2.7% overhead in terms of execution time. Thus, we conclude that our analyses incur reasonably small overhead, and despite the overhead, they allow SMARTIAN to find more bugs and to achieve more coverage.

Answer to RQ1. Both static and dynamic analyses can effectively guide fuzzing to find more bugs. When used together, the analyses enable SMARTIAN to find 22% more bugs in our benchmark.

C. Comparison against Existing Tools

Next, we compare SMARTIAN against state-of-the-art tools that we selected in §V-A2. For this, we measured the bug-finding effectiveness as well as coverage achievement of each tool. To measure coverage achievement, we replayed the test cases generated by each tool. Recall from §V-A2 that we modified the tools to emit all the test cases that can increase coverage, in a replayable format.

*1) Comparison on **B1**:* We first compare SMARTIAN with other testing tools on **B1**. As we mentioned in §V-A, some of our comparison targets did not properly operate on contracts with constructor arguments. Therefore, we excluded such contracts and obtained a subset of **B1**, which comprises 32 contracts. We ran the tools for one hour on each contract, and repeated the experiment five times to compute the average. To measure the bug-finding effectiveness, we counted the number of ground truth bugs found by each tool. As in §V-B, we checked whether the tool can report the exact program point that corresponds to the CVE assigned for an integer bug (**IB**).

Figure 4 shows the comparison result between SMARTIAN and other tools. The left-hand-side plot presents the number of CVEs found over time, whereas the right-hand-side plot presents the instruction coverage over time. Note that ILF is not included in this comparison, because it does not support the detection of **IB** (see Table II).

As the figure indicates, SMARTIAN constantly outperformed other tools in terms of bug-finding effectiveness. SMARTIAN found 5.8×, 4.8×, and 2.1× more ground truth bugs (CVEs) than sFuzz, Manticore, and Mythril, respectively. This result was consistent over the five times of repeated experiments (p-value < 0.01 from Mann-Whitney U-Test for all the tools). Moreover, there was only one bug that SMARTIAN missed but one of the other tools could find.

Also, SMARTIAN covered more instructions than other tools throughout the whole fuzzing campaign. SMARTIAN covered 1.8×, 3.9×, and 1.1× more instructions than sFuzz, Manticore, and Mythril, respectively. While Mythril was the closest to SMARTIAN in terms of code coverage, it still found significantly fewer bugs than SMARTIAN. This implies that Mythril failed to modify the state variables in a critical way while it was able to cover enough code.

Difference in Bug Oracles. As we discussed in §II-B, each tool implements its own oracle for the same bug class. This may affect the bug-finding effectiveness of each tool. For instance, sFuzz only monitors additions and subtractions to detect integer overflows, and ignores multiplications. This makes sFuzz prone to false negatives.

To tackle this problem, we ran additional experiments by modifying SMARTIAN to have the same oracle logic with the comparison target. For example, we replaced our **IB** oracle with the oracle of sFuzz, and then compared the modified SMARTIAN against sFuzz. This way, we can compare the bug-finding effectiveness of each tool without being affected by the inconsistency of oracles. It turned out that SMARTIAN outperforms other tools even after aligning the **IB** oracle with them. SMARTIAN still found 4.0×, 3.8×, and 2.1× more CVEs than sFuzz, Manticore, and Mythril, respectively.

We further investigated the result to compare the bug oracles. First, when we replaced our bug oracle with the oracle of sFuzz and Manticore, SMARTIAN found 31% and 21% fewer CVEs. When we modified SMARTIAN's oracle to match with that of Mythril, SMARTIAN found 1% more CVEs, but it raised 46% more alarms instead. Thus, we conclude that our **IB** oracle in §IV-C3 is most appropriate for this benchmark.

*2) Comparison on **B2**:* We now compare our system against other tools on **B2**, which we constructed from SmartBug benchmark [25]. We selected contracts that were labeled with block state dependency (**BD**), mishandled exception (**ME**), or reentrancy (**RE**). However, we found that the labels were incomplete for some of the contracts. For instance, a contract

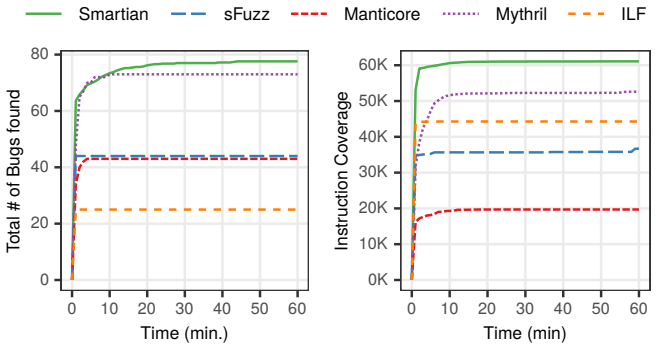

Fig. 5. Comparison against state-of-the-art tools on **B2**.

TABLE IV
NUMBER OF TP AND FP ALARMS RAISED BY EACH TOOL ON **B2**.

Bug ID	SMARTIAN		ILF		sFuzz		Manticore		Mythril	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
BD	11	0	0	0	10	0	6	5	8	0
ME	48	0	10	0	29	6	18	0	46	0
RE	19	0	15	2	5	20	19	3	19	38

that was classified to have **ME** often contained **RE**, as well. Thus, we manually investigated the contracts and labeled them again. Then, we ran each tool for one hour on each of the contracts in **B2**. To measure the bug-finding effectiveness, we checked whether the tool can report the bugs labeled on each contract.

Figure 5 illustrates the result of this comparison. We present the number of bugs found over time on the left side, and instruction coverage over time on the right side. The results are averaged over five times of repeated experiments.

The figure shows that SMARTIAN is able to find more bugs than other tools. When compared to fuzzers, SMARTIAN found 3.1× and 1.8× more bugs than ILF and sFuzz, respectively. SMARTIAN also outperformed symbolic executors, by finding 1.8× and 1.1× more bugs than Manticore and Mythril, respectively. The result was consistent over the five repetitions (p-value < 0.01 from Mann-Whitney U-Test for all the tools). In addition, there were only two bugs that were missed by SMARTIAN but found by any other tool.

Moreover, SMARTIAN constantly covered more instructions than other tools, too. In particular, SMARTIAN covered respectively 1.4× and 1.7× more instructions than ILF and sFuzz. Also, it covered 3.1× more instructions than Manticore and 1.2× more instructions than Mythril.

We also count the number of false positive alarms raised by each tool and summarize them in Table IV, along with the number of found bugs (i.e. true positives). The table shows that SMARTIAN not only has the highest recall, but it also has the highest precision for this benchmark. Especially, other tools suffered from a high false positive rate for **RE** bugs. This is mainly because these tools do not properly consider the storage access pattern of the target contract, and simply check whether there is a call to an external contract.

TABLE V
NUMBER OF BUGS FOUND BY SMARTIAN ON **B3**.

Bug ID	Description	# of Bugs Reported	# TP	# FP
AW	Arbitrary Write	0	0	0
BD	Block State Dependency	26	20	6
CH	Control Hijack	0	0	0
EL	Ether Leak	5	3	2
IB	Integer Bug	170	170	0
ME	Mishandled Exception	2	2	0
MS	Multiple Send	5	5	0
RE	Reentrancy	0	0	0
SC	Suicidal Contract	0	0	0
TO	Transaction Origin Use	11	11	0
Total	-	219	211	8

Difference in Bug Oracles. Again, we reimplemented the bug oracles of SMARTIAN to match with other tools' oracles, to compare the bug-finding effectiveness under the same condition. In particular, we modified our **BD**, **ME**, and **RE** oracles to align with those of other tools. SMARTIAN still prevailed other tools after the alignment of the oracles. The modified SMARTIAN found 2.8× more bugs than ILF and 1.7× more bugs than sFuzz. Also, it found 1.9× more bugs than Manticore and 1.1× more bugs than Mythril.

Answer to RQ2. SMARTIAN is more effective in finding bugs compared to existing state-of-the-art tools. SMARTIAN finds 1.1–5.8× more bugs than our comparison targets.

D. Large-Scale Study

Now that we have evaluated the comparative performance of SMARTIAN, we now turn our attention to the scalability of our system. Specifically, we ran SMARTIAN on **B3**, which consists of 500 popular and large contracts we gathered from Etherscan (see §V-A3). We ran SMARTIAN on each contract for one hour, and manually investigated the reported alarms to classify them into true and false alarms. We did not include other tools in this experiment because implementing other tools' bug oracles in SMARTIAN as we did in §V-C, requires significant engineering effort. Instead, we focus on the scalability of SMARTIAN here.

Table V summarizes the result. Note that we do not report **FE** and **RV** here, for the reasons we discussed in Table I. In addition, we also omit **AF** found by SMARTIAN. While SMARTIAN found hundreds of true positive **AF**, it is debatable whether they can be considered as serious bugs, so we do not include **AF** in the table. After excluding these bug classes, SMARTIAN reported a total of 219 bug alarms. Out of the 500 contracts, 72 contracts were flagged to have at least one of these alarms. We manually inspected the alarms and confirmed that 96.3% of them were true positives. Recall from §IV-C3, this was possible because SMARTIAN employs precise bug oracles to reduce false alarms.

We confirmed that some of the bugs found by SMARTIAN had similar patterns to the CVEs in **B1**. Some of the bugs could even cause contract users or owners to unexpectedly lose their assets. We also found that most of the bugs were caused by poor software engineering practices. Thus, we conclude

that SMARTIAN can indeed find meaningful bugs in real-world smart contracts.

Answer to RQ3. SMARTIAN is effective in finding various kinds of bugs from a large-scale benchmark. SMARTIAN could find 211 bugs from 500 complex real-world contracts we collected.

E. Threats to Validity

First, we performed our experiments on a limited set of benchmarks. We used two benchmarks (**B1** and **B2**) containing known bugs, and another benchmark (**B3**) that consists of large and popular real-world contracts. We showed the effectiveness of our system on these benchmarks, but other benchmarks may yield different results. We open-source our code so that SMARTIAN can be further evaluated in other work.

Another threat to validity is related to the manual processes included in our evaluation. For instance, we manually labeled bugs to each contract in **B2**, and manually classified the alarms that SMARTIAN reported from the contracts in **B3**. Although we tried our best to carefully inspect the contract code, we might have erroneously concluded whether the bug indeed exists or not. We also make our dataset public, to enable cross-checks from other researchers.

VI. DISCUSSION

Due to the over-approximating nature of static analysis, data-flow facts gathered from our INFOGATHER step (§IV-A) may contain spurious data-flows that cannot actually occur in runtime. If such false positives are prevalent in the analysis result, our static analysis may even degrade the fuzzing performance. However, the evaluation in §V-B empirically shows that our analysis is precise enough to guide fuzzing effectively. In the future, we may further improve the static analysis precision and study how it affects the fuzzing effectiveness.

SMARTIAN currently inherits the limitations of Eclipser, such as lack of handling non-linear branch conditions. Adopting other grey-box technologies and recent advances in the field, such as [17], [42], [46], [47], to complement SMARTIAN would be a promising direction for future research.

Although SMARTIAN is specifically designed for fuzzing smart contracts, we believe the idea of leveraging data-flow analyses in fuzzing can be applied to other areas as well. For instance, generating the sequence of system calls is a critical problem in kernel fuzzing [21], [33], [55]. We leave it as future work to apply our idea to other domains.

VII. RELATED WORK

Fuzzing has become a *de facto* standard technique for finding security bugs [14], [34], [40], [45], and there has been significant research effort on adopting fuzzing in the domain of smart contracts, too. Recall from §III-C, our contribution is unique in that we are the first in adopting both static and dynamic analyses to systematically deal with stateful transactions of smart contracts for fuzzing.

ContractFuzzer [36] is the first academically developed fuzzer for smart contracts. Since it is a black-box fuzzer, it has difficulties achieving high code coverage. Echidna [31] checks a set of user-defined invariant rules to detect bugs. An analyst should embed these rules within the contract source code itself. On the other hand, SMARTIAN does not require any human intervention.

Harvey [70] is a commercial (closed-source) fuzzer. It employs a heuristic referred to as the aggressive mode, which directly mutates state variables to figure out the dependencies between functions. In contrast, SMARTIAN systematically addresses this by statically analyzing the semantics of code. The same authors recently enhanced Harvey by employing a static analyzer called Bran [71], which can guide grey-box fuzzing towards target locations. Bran is orthogonal to our work, and SMARTIAN can also benefit from it.

sFuzz [53] incorporates AFL [72] with the idea of branch distance feedback used in search-based testing [48] in order to explore hard-to-reach branches. However, sFuzz does not directly handle the stateful transaction problem we address in this paper. ILF leverages machine learning to effectively generate transaction sequences. It is orthogonal to our technique, and our analysis can complement ILF, too.

There are several fuzzers outside the domain of smart contracts, which utilize data flow analysis [18], [28], [56], [68] to figure out which input bytes need to be mutated or which values should be used for the mutation. Our work is orthogonal to them as we are using data-flow information to find meaningful transaction sequence orders.

VIII. CONCLUSION

We studied the current limitation of smart contract testing tools, and identified several design and technical issues. Specifically, we tackled the problem of effectively handling multiple stateful transactions of smart contracts, which leads to the introduction of combined static and dynamic analysis techniques for generating seeds and updating the seed pool during a fuzzing campaign. Our study showed that the proposed techniques incur negligible overhead while enabling effective fuzzing in terms of both code coverage and bug finding. We also compared SMARTIAN against the various state-of-the-art testing tools on a carefully designed benchmark, and confirmed the effectiveness of it. We publicize both our tool and benchmarks to boost future research.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments and suggestions. We also thank Josselin Feist and Felipe Manzano for their helpful advice on smart contract testing. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01332, Developing Next-Generation Binary Decompiler).

REFERENCES

[1] "Arbitrary jump with function type variable," https://swcregistry.io/docs/SWC-127.
[2] "Assertion failure," https://swcregistry.io/docs/SWC-110.
[3] "Authorization through tx.origin," https://swcregistry.io/docs/SWC-115.

[4] "Decentralized application security project," https://dasp.co/.

[5] "Dos with failed call," https://swcregistry.io/docs/SWC-113.

[6] "Etherscan," https://etherscan.io/.

[7] "Nethermind," https://github.com/NethermindEth/nethermind.

[8] "Requirement violation," https://swcregistry.io/docs/SWC-123.

[9] "Smart contract weakness classification registry," https://swcregistry.io/.

[10] "Solidity documentation," https://docs.soliditylang.org.

[11] "Solidity expressions and control structures," https://docs.soliditylang.org/en/v0.4.25/control-structures.html.

[12] "Write to arbitrary storage location," https://swcregistry.io/docs/SWC-124.

[13] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[14] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2020, pp. 678–689.

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

[16] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," in *Proceedings of the International Conference on Formal Engineering Methods*, 2019, pp. 286–304.

[17] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018, pp. 2095–2108.

[18] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018, pp. 855–869.

[19] J. Choi, J. Jang, C. Han, and S. K. Cha, "Eclipser," https://github.com/SoftSec-KAIST/Eclipser, 2019.

[20] ——, "Grey-box concolic testing on binary code," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 736–747.

[21] J. Choi, K. Kim, D. Lee, and S. K. Cha, "NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2021, pp. 1973–1989.

[22] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[23] P. Daian, "Analysis of the dao exploit," https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 2016.

[24] M. di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *Proceedings of the IEEE International Conference on Decentralized Applications and Infrastructures*, 2019, pp. 69–78.

[25] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the International Conference on Software Engineering*, 2020, pp. 530–541.

[26] Ethereum, "Ethereum whitepaper," https://ethereum.org/en/whitepaper/.

[27] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proceedings of the International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019, pp. 8–15.

[28] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *Proceedings of the USENIX Security Symposium*, 2020, pp. 2577–2594.

[29] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[30] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in ethereum smart contracts," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2018, pp. 116:1–116:27.

[31] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.

[32] A. Groce, J. Feist, G. Grieco, and M. Colburn, "What are the actual flaws in important smart contracts (and how can we find them)?" in *International Conference on Financial Cryptography and Data Security*, 2020.

[33] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[34] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[35] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2019, pp. 531–548.

[36] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the International Conference on Automated Software Engineering*, 2018, pp. 259–269.

[37] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2R2: Building an efficient front-end for binary analysis," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[38] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[39] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 1317–1333.

[40] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[41] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2017, pp. 627–637.

[42] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2019, pp. 533–544.

[43] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and A. W. Roscoe, "ReGuard: Finding reentrancy bugs in smart contracts," in *Proceedings of the International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 65–68.

[44] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2016, pp. 254–269.

[45] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[46] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the International Conference on Software Engineering*, 2020, pp. 1024–1036.

[47] B. Mathis, R. Gopinath, and A. Zeller, "Learning input tokens for effective fuzzing," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2020, pp. 27–37.

[48] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[49] A. Møller and M. I. Schwartzbach, "Static program analysis," https://cs.au.dk/ amoeller/spa/, 2019.

[50] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proceedings of the International Conference on Automated Software Engineering*, 2019, pp. 1186–1189.

[51] L. D. Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[52] B. Mueller, "Smashing ethereum smart contracts for fun and actual profit," in *Proceedings of the HITB Security Conference*, 2018.

[53] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the International Conference on Software Engineering*, 2020, pp. 778–788.

[54] I. Nikoliundefined, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[55] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 729–743.

[56] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[57] Remix, "Ethereum ide and tools for the web," https://github.com/ethereum/remix, 2017.

[58] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020.

[59] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.

[60] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VeriSmart: A highly precise safety verifier for ethereum smart contracts," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2020, pp. 1678–1694.

[61] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su, "A survey on data-flow testing," *ACM Computing Surveys*, vol. 50, no. 1, pp. 1–35, 2017.

[62] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proceedings of the IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[63] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[64] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify2," https://github.com/eth-sri/securify2.

[65] ——, "Securify: Practical security analysis of smart contracts," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018, pp. 67–82.

[66] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "Vultron: Catching vulnerable smart contracts once and for all," in *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results*, 2019, pp. 1–4.

[67] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[68] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010, pp. 497–512.

[69] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," https://ethereum.github.io/yellowpaper/paper.pdf.

[70] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the International Symposium on Foundations of Software Engineering: Industry Papers*, 2020, pp. 1398–1409.

[71] ——, "Targeted greybox fuzzing with static lookahead analysis," in *Proceedings of the International Conference on Software Engineering*, 2020, pp. 789–800.

[72] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[73] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *Proceedings of the IFIP International Conference on New Technologies, Mobility and Security*, 2018, pp. 1–5.