WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts

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

Zihan Sun
Beijing University of Posts and Telecommunications
Beijing, China
sunzihan@bupt.edu.cn

Haoyu Wang*
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

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

Haipeng Cai
Washington State University
Pullman, USA
haipeng.cai@wsu.edu

Lei Wu
Zhejiang University
Hangzhou, China
lei_wu@zju.edu.cn

ABSTRACT

WebAssembly (Wasm) smart contracts have shown growing popularity across blockchains (e.g., EOSIO) recently. Similar to Ethereum smart contracts, Wasm smart contracts suffer from various attacks exploiting their vulnerabilities. Even worse, few developers released the source code of their Wasm smart contracts for security review, raising the bar for uncovering vulnerable contracts. Although a few approaches have been proposed to detect vulnerable Wasm smart contracts, they have several major limitations, e.g., low code coverage, low accuracy and lack of scalability, unable to produce exploit payloads, etc. To fill the gap, in this paper, we design and develop WASAI, a new concolic fuzzer for uncovering vulnerabilities in Wasm smart contract after tackling several challenging issues. We conduct extensive experiments to evaluate WASAI, and the results show that it outperforms the state-of-the-art methods. For example, it achieves 2x code coverage than the baselines and surpasses them in detection accuracy, with an F1-measure of 99.2%. Moreover, WASAI can handle complicated contracts (e.g., contracts with obfuscation and sophisticated verification). Applying WASAI to 991 deployed smart contracts in the wild, we find that over 70% of smart contracts are vulnerable. By the time of this study, over 300 vulnerable contracts have not been patched and are still operating on the EOSIO Mainnet. One fake EOS vulnerability reported to the EOSIO ecosystem was recently assigned a CVE identifier (CVE-2022-27134).

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Software testing and debugging**.

*Haoyu Wang and Xiapu Luo are the corresponding authors.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534218

KEYWORDS

Concolic fuzzing; smart contracts; dynamic software analysis

ACM Reference Format:
Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. 2022. WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534218

1 INTRODUCTION

WebAssemly (Wasm) [59] was designed for Web applications to support the execution of Wasm programs in memory-safe, sandboxed environments. It has been recently adopted by many popular blockchain platforms, e.g., EOSIO [4] and NEAR [43], as their smart contract runtime, and thousands of Wasm smart contracts have been deployed on blockchains. Even the Ethereum has placed Ethereum flavored WebAssembly on the Ethereum 2.0 roadmap [21] as a replacement to the Ethereum Virtual Machine (EVM). Like Ethereum smart contracts [7, 20, 60], Wasm smart contracts have been suffering from various attacks exploiting their vulnerabilities. Being the first blockchain supporting Wasm smart contracts, EOSIO has been attacked from time to time due to the vulnerabilities in smart contracts [11, 31–33, 42]. Even worse, few developers released the source code of their Wasm smart contracts for security review, raising the bar for uncovering vulnerable contracts.

The State of the Art. Although many tools have been developed for detecting vulnerabilities in Ethereum smart contracts [6, 9, 12, 23, 34, 36, 40, 44, 45, 52, 67], they cannot be used to detect vulnerable Wasm smart contracts due to the significant differences between Ethereum and EOSIO, including calling convention, seed format, transaction dependency and notification mechanism. To the best of our knowledge, only a handful of studies focus on Wasm smart contracts, which can be classified into two categories: a) static analysis based approaches that leverage symbolic execution (SE for short) for detecting vulnerable Wasm contracts, such as EOSafe [29] and WANA [61]; and b) fuzzers that conduct dynamic analysis with generated inputs (or seeds) to construct transactions sequences for triggering vulnerabilities, e.g., EOSFuzzer [30]. Unfortunately, our experimental results showed that existing tools cannot handle complicated smart contracts. On the one hand, SE-based static analysis techniques are impeded by the difficulty of solving

complicated constraints and the path explosion issue. For example, EOSafe [29] and WANA [61], the state-of-the-art approaches, cannot identify reachable paths for complicated smart contracts (e.g., obfuscated contracts). On the other hand, fuzzers are limited by ineffective seed generation strategies. To the best of our knowledge, EOSFuzzer [30] is the only public available fuzzer for Wasm contracts. Unfortunately, it only generates *random seeds* without leveraging feedback, thus resulting in low code coverage, especially for Wasm contracts with a large number of conditional branches.

This Work. We propose WASAI, the first concolic fuzzer for analyzing Wasm smart contracts. Although there are concolic fuzzers for x86 binaries, e.g., [54], they cannot be applied to smart contracts because of the unique features of smart contracts and the underlying blockchain platforms. For example, smart contracts are stateful with states recorded in the blockchain. Moreover, the memory model and the calling convention of Wasm runtime are different from those of x86 native. Since EOSIO is the largest platform hosting the most popular Wasm smart contracts, we focus on EOSIO smart contracts in this paper, and our approach can also be generalized to analyze Wasm smart contracts on other blockchains.

It is non-trivial to design WASAI because of the following challenges. **C1:** capturing traces of Wasm smart contracts is not straightforward because 1) EOSVM can execute multiple smart contracts in parallel, and 2) one transaction may invoke several smart contracts. To address these issues, we insert Wasm hooks into individual smart contracts to capture traces during the smart contract execution. Besides, we redirect the traces to offline files once one EOSVM thread finishes (§ 3.3.1). **C2:** it is challenging to recover meaningful data (e.g., input data, structs, strings, call parameters) from the memory [37], because they are usually deserialized from bytes streams by a bunch of memory instructions, where some memory data will be overwritten several times. It prevents the symbolic execution engine from locating the desired memory content if the corresponding memory address is a symbolic expression. To approach this issue, we create a memory model based on the concrete addresses from the traces in § 3.4.1. **C3:** Before executing the desired functions, Wasm contracts have to deserialize the input data into meaningful data, such as arrays, structs and objects. Existing tools usually fail to perform symbolic execution on the deserializing methods due to path explosion [55]. Hence, we simulate the execution of desired functions directly and skip the paths from the execution entry to the desired functions (§ 3.4.2). However, by doing so, we may lose the mapping from the input data to the arguments of the desired functions, which will affect the seed mutation. To addres this issue, we build function models for the deserializing methods and create the symbolic expressions for the input data directly as the arguments of the desired functions.

WASAI is state-sensitive for generating adaptive input according to the on-chain states of smart contracts. Specifically, WASAI performs a contract-level instrumentation to capture the execution traces of transactions, enabling an EOSVM simulator to simulate the smart contract execution and find adaptive seeds based on symbolic execution. By replaying the captured traces, the EOSVM simulator constructs symbolic expressions for the executed Wasm instructions. Consequently, WASAI can find adaptive seeds to explore more branches by solving the symbolic constraints, which can feedback the fuzzing. Moreover, we design vulnerability oracles to uncover vulnerable smart contracts in § 2.3 and conduct verification dynamically to achieve a low false-positive rate in § 4.2. To evaluate the performance of WASAI, we construct a comprehensive dataset, including known vulnerable smart contracts in the wild and crafted vulnerable smart contracts by injecting vulnerabilities to the bytecode of Wasm smart contracts [25]. The results of extensive experiments show that WASAI outperforms all state-of-the-art techniques in terms of the effectiveness, efficiency, and robustness of vulnerability detection (see § 4). WASAI and its dataset are released at 10.5281/zenodo.6517515

This work makes the following main contributions:

- We propose WASAI, the first concolic fuzzer for Wasm smart contracts, after tackling several challenges. It 1) instruments the contract bytecode to capture the runtime traces; 2) builds a memory model to recover symbolic expressions from the memory faster than EOSafe; 3) and eases the path explosion of the symbolic execution by building function models and skipping redundant paths. With the feedback of the symbolic execution, WASAI can generate adaptive seeds to resolve complex branches.
- We develop a prototype of WASAI and construct a large-scale benchmark with 3,340 samples, which is by far the largest benchmark of Wasm smart contracts, to evaluate it. Extensive experimental results demonstrate that WASAI outperforms all state-of-the-art approaches, with an F1-measure of 99.2%. Specifically, the results of experiments using complicated smart contracts with code obfuscation or complicated verification show that WASAI retains high detection accuracy and is more robust than existing techniques. We also select 100 real-world contracts and then measure WASAI's code coverage comparing to EOSFuzzer. As a result, WASAI can explore $2x$ more distinct branches than EOSFuzzer.
- We apply WASAI to 991 deployed Wasm contracts in EOSIO Mainnet. The experimental results reveal that over 70% of smart contracts are vulnerable. By the time of this study, 58% of these contracts are still operating on the Mainnet, while over 300 of them have not been patched yet. We reported a Fake EOS vulnerability to the EOSIO ecosystem, which was recently assigned a CVE identifier (CVE-2022-27134).

2 BACKGROUND

2.1 Transactions in EOSIO

EOS is the official token in EOSIO blockchain [24]. In this paper, we focus on profitable smart contracts, which can accept EOS tokens to provide services. We use *eosponser* to refer to the Wasm function that can respond to EOS transactions. Usually, smart contracts implement *eosponser* as `void transfer(name, name, asset, string)` to receive the message from *eosio.token*. Any EOSIO smart contract can issue cryptocurrency. As the issuer of the official EOS, the smart contract, *eosio.token* creates the EOS tokens and allows users to transfer EOS via invoking a function called "transfer". Action functions are the public methods of a smart contract. A function header explicates its function name and the parameters types (a.k.a. function signature). In this paper, we use '@' to indicate the scope of an action function. For example, *transfer@eosio.token* denotes the action function named `transfer` in the smart contract *eosio.token*.

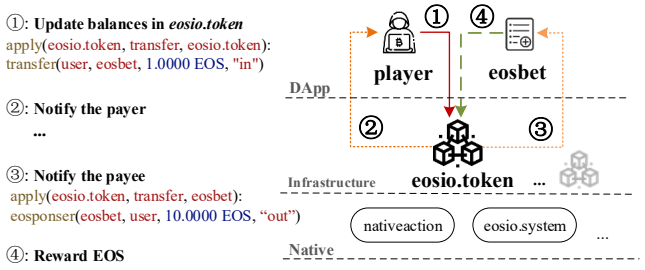

Figure 1: The process of transferring EOS tokens.

Figure 1 shows the procedure of an EOS transaction. The player pays EOS to the lottery smart contract named *eosbet* for the jackpot. ①: the player invokes *transfer@eosio.token* to pay EOS. Next, *eosio.token* forwards the invocation message to notify the payer (②) and the payee (③) accordingly. *eosbet* should response the EOS notification with an action function, i.e, *eosponser*. ④: At last, *eosbet* interacts with *eosio.token* to reward a lucky player with 10 EOS. To this end, *eosponser@eosbet* must have the same function signature with the *transfer@eosio.token* to parse the notification message.

2.2 EOSIO Smart Contract

EOSIO smart contracts are developed in C++ SDK (Software Development Toolkit) [16]. Its compiler turns the source code into a Wasm contract and an ABI describing its function signatures of the action functions (see § 2.1). Each Wasm contract uses an identical dispatcher as the execution entry, i.e., `void apply(name receiver, name code, name action)`. The dispatcher loads the input data and then performs an indirect call [49] to invoke the action function indicated by its parameter `action`. EOSVM can invoke a Wasm function using the unique function ID [63].

EOSIO Virtual Machine (EOSVM). EOSIO smart contracts are running in the EOSVM [18]. EOSVM is a stack-based virtual machine, including a call stack, a `Local` section, a `Global` section, a linear memory and a key-value database. The call stack consists of multiple sub-stacks for isolating the function namespace. When a function is invoked, EOSVM will allocate an array in `Local` section for saving local data, e.g., the parameters of the function invocation. `Global` section is an array as well, saving the global data across the execution. The type of each element in these three components can be i32 (32-bit integer), i64 (64-bit integer), f32 (32-bit floating-point), or f64 (64-bit floating-point). The linear memory [58] is a byte-addressable pool. We denote the storing instructions and loading instructions as `store` and `load`, respectively. In total, there are 23 kinds of memory instructions in Wasm.

Library API. A Wasm contracts can access blockchain states using the library APIs provided by the EOSVM. `has_auth`, `require_auth`, and `require_auth2` verify the permissions of the caller (permission APIs for short); `tapos_block_pre` and `tapos_block_num` fetch the blockchain states such as timestamp; `eosio_assert` asserts the constraints. To maintain persistent data, each smart contract can access its database with database APIs, such as `db_find` and `db_erase`.

2.3 Vulnerabilities

Following previous studies [29, 30, 61], we aim to detect five kinds of commonly-seen vulnerabilities in EOSIO, which is a superset of those considered in the existing studies. Note that the Fake EOS and the Fake Notification bugs only exist in EOSIO contracts. The other three bugs are examined for a fair comparison with the two baselines that claim to be able to detect them (see § 4.2).

```
1  void apply(name receiver, name code, name action) {
2    // the initiation is omitted
3    if (action == N(transfer)) {
4      // patch:assert(code==N(eosio.token), "");
5      run(eosponser);// execute eosponser
6    }
7    else if(code==receiver||code==N(eosio.token)){
8      EOSIO_API(action);// execute other actions
9    }
10 }
```

Listing 1: An example of Fake EOS vulnerability.

2.3.1 Fake EOS. EOSIO allows anyone to issue tokens with any name, enabling attackers to release fake EOS tokens with identical name of the official one. If the victim accepts the fake tokens from attackers, it falls in the Fake EOS vulnerability. Listing 1 shows a vulnerable smart contract. Attackers can exploit the vulnerability in two ways. On the one hand, they can invoke the *eosponser* of the victim directly. On the other hand, they can deploy a smart contract, e.g., *fake.token*, to issue the fake token named "EOS", and then invoke *eosponser@fake.token* to transfer fake EOS to the victim.

The exploit is feasible because the victim does not check whether the token issuer is *eosio.token*. Hence, no matter who sends the transaction, the *eosponser* will be invoked to provide services (Line 5). To patch it, developers must implement the guard code in Line 4 to check the token issuer, where the current caller is recorded in the parameter `code` of `void apply()`. In Wasm bytecode, there should be a comparison instruction (e.g., `i64.ne` and `i64.eq`), in which the name of the contract *eosio.token* and value of the parameter `code` are the two operands.

```
1  [[eosio::action]] void eosponser( name from, name to,
       asset quantity, string memo) {
2    // patch: if (to != _self) return;
3    action(...).send();//sensitive operation
4  }
```

Listing 2: An example of a Fake Notification vulnerability.

2.3.2 Fake Notification. The Fake Notification (`Fake Notif` for short, also known as Fake Receipt in [29]) vulnerability evolves from the Fake EOS. Attackers can play the dual accounts and forge an EOS notification to the example shown in Listing 2: 1) they deploy an agent smart contract called *fake.notif*; 2) they invoke *transfer@eosio.token* to transfer official EOS to the *fake.notif*; 3) the pre-defined code of *fake.notif* forwards the notification from *eosio.token* to the victim. At this time, the parameter `code` remains the original value: the official issuer *eosio.token*; 4) as a result, the victim executes *eosponser* accidentally without receiving any EOS.

In this exploit procedure, the attacker is able to forward a fake notification from *eosio.token* to the victim. Thus the guard code of the Fake EOS can be bypassed. Any smart contract does not validate the notification in its *eosponser* is vulnerable, because the attacker can pretend a friendly user who has paid EOS to the contract. Since

the header of *eosponser* must follow *transfer@eosio.token* (see § 2.1), the second parameter of the *eosponser* indicates the payee name. Developers can use the guard code in Line 2 to fix the bug by ensuring the EOS payee (`to`) is the contract itself (`_self`), where `_self` is a global variable maintaining the contract identification.

```
1 void eosponser(name from, name to, asset quantity,
      string memo) {
2   // patch: require_auth(from);
3   action(permission_level{from, N(active)},
4     N(eosio.token), N(transfer),
5     make_tuple(from,to,quantity,memo)
6   ).send();
7 }
```

Listing 3: An example of using permission check at line 2.

2.3.3 Missing Authorization Verification. Smart contracts can execute sensitive operations, hence it is necessary to verify the authorization status of the caller. Otherwise, it exposes authorization vulnerability (`MissAuth` for short). Taking Listing 3 as an example, the smart contract should check whether the caller in the parameter `from` is the actual payer before it invokes *eosio.token* at Line 3-6. To patch it, smart contracts should utilize permission APIs to perform authorization checking (see § 2.2).

2.3.4 Blockinfo Dependency. Due to the lack of native pseudorandom number generators (PRNG), EOSIO smart contracts may utilize blockchain states (e.g., timestamp) as the source of randomness by calling specific APIs such as `tapos_block_pre` and `tapos_block_num`. However, such randomness would be predicted by the attacker. Following previous work [30], we flag a vulnerable contract (`BlockinfoDep` for short) if it uses blockchain states to generate pseudorandom numbers. For the security of pseudorandom, developers are suggested to use the verified PRNG services, e.g., `Niguez Randomity Engine` [15].

```
1 void apply(name receiver, name code, name action) {
2   if (action == N(transfer))
3     run(reveal);// execute the function reveal(...)
4   ...
5 }
6 void reveal(name from, name to, asset quantity, string
      memo) {
7   eosio_assert(quantity>=asset("10.0000 EOS"),"exit");
8   int a = tapos_block_prefix()*tapos_block_num(); //
      use verified PRNG for the BlockinfoDep safety
9   int b = tapos_block_prefix()+tapos_block_num();
10  if (a % b) {
11    action(permission_level{_self, N(active)}, // use
      defer scheme for the Rollback safety
12      N(eosio.token), N(transfer),
13      std::make_tuple(_self, N(eosio.token),
14      asset(1000), std::string("test"))) .send();
15  }
16 }
```

Listing 4: An example of a Rollback vulnerability.

2.3.5 Rollback. EOSIO allows smart contracts to invoke each other by using inline actions and defer actions. All inline actions will be packed into the same transaction under the control of the caller, while the defer actions could not be reverted by the caller directly. Attackers can rollback the inline actions to enforce the blockchain to deny the transactions. Listing 4 shows a vulnerable lottery game, in which the attacker can make a profit via an evil contract. The evil contract first participates in the game with an inline action and then immediately checks its balance after asking for the revealing with another inline action. If its balance increased, it means the attacker won from the reveal behavior. Otherwise, the evil contract can revert itself to avoid loss.

This exploit scenario is reliable since the participate behavior and the reveal behavior are two inline actions in one identical transaction, which can be reverted by the attacker. To patch it, the developer should implement a defer mechanism (see Line 11), ensuring the participate behavior and the reveal behavior would be packed into two different transactions.

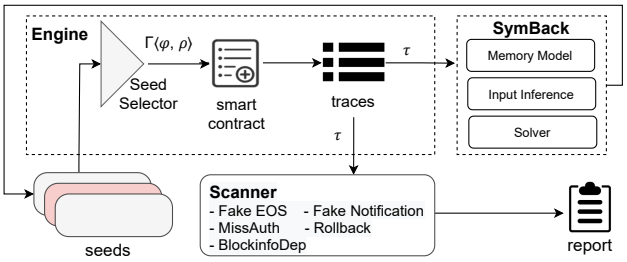

Figure 2: The architecture of WASAI. The red blocks in *seeds* are generated from the adversary oracles (see § 2.3).

3 WASAI

3.1 Overview

Figure 2 depicts the overall architecture of WASAI, which consists of three major modules, i.e., Engine, Symback, and Scanner. Given the binary of a Wasm contract and the corresponding ABI as input, WASAI strategically generates seeds as transaction sequences to trigger the vulnerabilities described in §2.3 in the Wasm contract (if any). To boost the fuzzing in exploring as many paths as possible, WASAI performs symbolic execution to feedback the seed mutation, which should generate adaptive seeds that can satisfy the path constraints. Since smart contracts are stateful, the execution of one transaction may depend on the persistent data which can be accessed by executing another specific transaction. We call it transaction dependency [28], which can be resolved by Engine. In particular, Engine will instrument the smart contract binary to capture execution traces (§3.3.1) and select new seeds that fulfill the transaction dependency for testing (§3.3.2). Symback will construct symbolic constraints for Wasm branches from the traces and generate new seeds to explore more code (§3.4). To recover the memory accesses by symbolic execution, Symback proposes a new memory model to handle the complex memory addresses used in EOSVM, e.g., different symbolic expressions as the address may point to the same memory content at EOSVM. To ease the path explosion during the symbolic execution, Symback simulates the action function and skips its prior paths. Since the seed mutation requires the mapping from the input data to the arguments of the

action functions, Symback infers the mapping based on the calling convention in Wasm smart contracts. Scanner will analyze the exploit events from the traces and output a vulnerability report (§3.5).

Before describing the algorithm of WASAI and its modules, we first define the following concepts.

Seed. Each seed is a pair $\Gamma\ \langle \varphi, \vec{\rho} \rangle$, where φ is the name of the action function in the fuzzing target, and $\vec{\rho}$ is the parameters.

Trace. Traces refer to the executed Wasm instructions. Each trace is a tuple $\tau(i, \vec{p})$, where i is an executed instruction, and $\vec{p}$ records its operands, respectively.

Machine State. A machine state is an internal state of the EOSVM, denoted as μ. Each machine state contains the information in the following components: a code section (i.e., the instructions), a linear memory μ_m, a stack μ_s, a `Local` section μ_l, a `Global` section μ_g and a list μ_r containing the returns from the currently invoked function. Following the Wasm design, we create an EOSVM simulator for the symbolic execution in this paper.

Conditional State. A conditional state can change the control flow depending on its condition:

(1) execute the branch instruction, i.e., `br_if` and `if`; or
(2) call the assertion API, i.e., `eosio_assert`,

Algorithm 1 shows the three-stage workflow of WASAI. **1) Instrumentation (L1)**: we instrument the Wasm smart contract to generate bin' (§ 3.3.1); **2) Initiation (L2)**: we initiate a local blockchain with necessary smart contracts, e.g., bin', *eosio.token* and some agent contracts used in the adversary oracles. *seeds* is the seed pool, which is initiated with random data at first. **3) Fuzzing loop (L3-12)**: during each iteration, WASAI selects a seed from *seeds* to execute. To tackle the transaction dependency (§ 3.3.2), we first choose an action function, φ, and then enquiry the parameters (denoted as $\vec{\rho}$) from the seed pool to construct a concrete seed for fuzzing, denoted as $\Gamma\ \langle \varphi, \vec{\rho} \rangle$. We build an EOSVM simulator (§ 3.4) for the feedback fuzzing. We first follow the calling convention of Wasm contracts to infer the data layout of the Wasm bytecode and then initiate the EOSVM simulator with the symbolic expressions of the user input generated on-the-fly (§ 3.4.2). The machine state (i.e., μ) of the EOSVM simulator is updated along the traces. We flip the symbolic constraints for resolving the unexplored branches, and analyze the execution traces to detect the vulnerabilities (§ 3.5). The fuzzing will be terminated after timeout period.

Algorithm 1 The workflow of WASAI

```
1:  bin' ← instrumentation(bin)
2:  fill seeds with random data
3:  repeat
4:      select an action function for transaction dependency
5:      select a seed, Γ
6:      τ⃗ ← run(bin', Γ)                          ▷ capture traces
7:      report vul(τ⃗)                    ▷ vulnerabilities detection
8:      create symbolic expressions for ρ⃗ and store in μ
9:      update μ along the symbolic traces
10:     constraints ← flip(μ, τ)
11:     solve constraints and find new seeds
12: until timeout
```

3.2 Challenges and Solutions

The existing tools [54, 65] cannot analyze the EOSIO smart contract directly because of the architecture differences between EOSIO and x86, and those between EOSIO and Ethereum. Comparing with x86, EOSIO Wasm uses a different memory structure. (1) x86 uses heap for memory allocation, which shares the same address space with the stack. In contrast, Wasm's memory has its unique space; (2) The heap data in x86 is not stored in a continuous piece, but Wasm memory data is stored in a linear vector. The differences between EOSIO and Ethereum lie in the persistent storage and the calling convention. First, the structured storage in Wasm makes persistent data more complicated because Wasm supports several kinds of APIs to operate the persistent data in the smart contract database, and thus WASAI has to identify their patterns to recover the database dependency for generating adaptive seeds. Second, the structured function isolation requires the symbolic execution engine to handle its calling convention. Since the calling convention of EOSIO is quite different from that of EVM, WASAI has to identify the stack frame for each invoked function to update its symbolic states. In specific, there are three challenges in building WASAI.

C1: Trace Acquisition. When interacting with smart contracts, WASAI needs to fetch the traces from EOSVM to know the states of the smart contract. Existing fuzzers usually instrument the modified runtime (e.g., Qemu for X86 fuzzers, EVM for Ethereum fuzzers) to export traces. It is not straightforward to apply their technology to EOSIO because the traces from different smart contracts can be generated in parallel in EOSVM. Different from EVM that only executes one smart contract at a time, EOSVM can allocate several threads to execute multiply smart contracts in parallel for a larger throughput. EOSFuzzer [30] instruments EOSVM to collect the traces, and thus it has to scarify the efficiency to execute smart contracts one by one to avoid mixing up the traces from different smart contracts. Without differentiating traces of each executed smart contract, EOSFuzzer redirects all of them into one offline file.

Our Solution: We perform a contract-level instrumentation, which injects low-level hooks [39] before each Wasm instruction in the individual Wasm bytecode. Each low-level hook is a piece of several Wasm instructions, executing along with the instructions of smart contracts. Specifically, the hooks duplicate the corresponding operands of the hooked instruction and invoke library APIs to print the traces. Once an EOSVM thread finishes, we export the traces to Symback for analysis. Since only the instrumented smart contracts can output traces, WASAI is not affected by the unnecessary traces from the auxiliary smart contracts, e.g., *eosio.token*. The instrumentation framework makes it easy to migrate WASAI to other Wasm platforms because it is much easier to invoke new library APIs than to instrument a new Wasm virtual machine (§ 3.3.1).

C2: Memory Model. Memory is essential in the execution of Wasm smart contracts. Each unit space of the EOSVM memory is one byte [58]. The 23 kinds of memory instructions can obtain memory contents with addresses. EOSVM memory can be used to load the arguments of a function invocation and the returns from a function. Crucially, EOSVM stores and retrieves any complex data

structure larger than 64 bits with an i32 pointer, which points to the data content in the memory. During the symbolic execution, we construct symbolic expressions as the addresses used by the memory instructions, which implicitly maintain the data flow in the Wasm contracts. The high-precision analyses require to recover high-level concepts (e.g., structs, strings, call parameters) from the input data through an accurate memory model [37]. However, apart from handling all 23 kinds of memory instructions with a unified model, resolving the symbolic accesses is also challenging in modelling memory because the memory instructions may use a symbolic expression as the address to load memory content. Different symbolic expressions as the addresses may point to the same memory content, but they stay symbolic until the constraints are solved. Here is an example to explain the symbolic accesses. Considering a and b are two symbolic expressions, we may write two bytes from a to $a+2$ as `0x0000` and then write `0xffff` from b to $b+2$. If a is equal with b, then this piece of memory should be written in `0xffff`. However, the symbolic execution engine cannot find it until it solves some constraints to confirm the relation between a and b. Therefore, some memory contents can be overwritten multiply times with a series of memory addresses during the symbolic execution [29]. We have to resolve the symbolic accesses, otherwise we may fail to identify that the overlapped the memory content should be `0xffff`. To ease the problem, the state-of-the-art SE tool in EOSIO, EOSafe [29] adopts a mapping structure to map the address and the memory content. Unfortunately, it is time-consuming for EOSafe to analyze memory accesses when analyzing deeper code in the smart contract because, in each memory access, it needs to search all items in its memory model to merge the overlapped contents.

Our Solution: We create a memory model based on the concrete address from the runtime traces, enabling WASAI to recover symbolic expressions from the memory faster than EOSafe, which is essential to improve the fuzzing throughput. The memory model returns a symbolic content once the address used in `load` hits the key, which is created when we simulate a `store`. To be specific, we build a Z3 [51] array to model each byte in the symbolic data. For each memory instruction, our memory model finds out the length of the memory content. When simulating the `store`, we split the memory content into bytes and store them into the Z3 array with the concrete address from the runtime traces. For *load*, we load the specific symbolic bytes from the Z3 array with the concrete address and concatenate them into an expression whose size is assigned by the inline arguments in the `load`. Moreover, since not all memory contents are recorded in the simplified traces, we construct a *symbolic load object* for simulating the load instruction that reads the implicit memory data. We will discuss more details in § 3.4.1.

C3: Calling Convention. The meaningful input data will be serialized into a byte stream before being fed to the smart contract, according to the function signatures declared at the ABI. The existing methods (e.g., Driller [54], EOSafe [29]) for building symbolic data for input data will usually fail due to timeout because their symbolic execution engines have to emulate a massive of code to deserialize the arguments, starting the analysis from the execution entry, i.e., `void apply()`. To boost the symbolic execution, WASAI simulates the execution of action functions and skips the execution of the deserializing methods. However, we may lose the mapping from the input data to the arguments of the action functions, and consequently seed mutation will be affected. Without the arguments, WASAI cannot build symbolic constraints for the conditional branches that depend on the input data.

Our Solution: WASAI infers the specific arguments in the Wasm layout of each input data. As mentioned in § 2.2, the initial values in the `Local` section indicate the arguments of a Wasm function. Thus, we can find the deserialized input in the `Local` section of the action function and build symbolic expressions for the arguments directly without simulating the deserializing methods. To locate the executed action function, we parse the indirect calls in the apply function, i.e., `void apply()` and identify the executed action function's ID [49]. The indirect call pattern is widely used in all EOSIO smart contracts as it is introduced by the EOSIO SDK [49]. At the beginning of symbolic execution, we initiate the `Local` section of the action function to create symbolic expressions as the input data. Therefore, we can analyze the execution of the action function without emulating its prior traces, e.g., produced from the dispatcher function. This solution relaxes the constraints resolution and allows WASAI to mutate specific parameters based on executed seeds. Besides the basic data types, i.e., i32/64 and f32/64, WASAI can find seeds for array, struct, name, string, etc. (§ 3.4.2).

3.3 Engine

Engine is the skeleton of WASAI, which is responsible for capturing the runtime traces and feeding seeds to the fuzzer.

3.3.1 Trace Acquisition. To tackle **C1**, we insert low-level hooks after each Wasm instruction. Each low-level hook consists of several Wasm instructions, which not only capture the instruction name but also duplicate the runtime values, i.e., operands. During the smart contract execution, the hooks invoke some library APIs to output the traces through the EOSVM IO methods. For example, to capture the trace of `i32.const 1024`, we replace the Wasm code with `i32.const 1024; i32.const 1024; call logi`, where `logi` is a library API provided by the EOSVM. At runtime, the EOSVM can bind the low-level hooks and print the trace, i.e., τ("const", 1024). As a result, WASAI has the capability to instrument all kinds of Wasm instructions. Additionally, some low-level hooks can output specific traces for labeling the different stages of the function invocation. For example, each function invocation in Wasm will be instrumented with the five low-level hooks shown in the Table 1.

All the low-level hooks are developed based on the Wasabi [39]. Wasabi generates a Node.js script to bind the instrumented low-level hooks in the Wasm binary and to output traces. To capture traces for Wasm smart contract, we further extend the Wasabi hooks to output the traces through the library API provided by the EOSVM without depending on the Node.js scripts. Specifically, we only need to extend a few library APIs in the EOSVM to print different types of data, such as i32, i64, f32, and f64. The extended instrumentation tool can analyze other Wasm blockchain platforms as long as they implement the library APIs at their customized Wasm virtual machines. For EOSIO, we propose an optimization for filtering unnecessary traces. We redirect the traces to offline files when the action function finishes its execution, enabling WASAI to analyze specific action functions on demand. In the cross-contracts

Table 1: Five low-level hooks for function invocations.

Hook name	Instrumentation place	Behavior
call_pre	before the invocation	duplicate the invocation parameters from callers' stack
call	in the invocation	duplicate the function ID of the invoked function;
function_begin	begin of the body of the invoked function	print a function_begin label
function_end	end of the body of the invoked function	print a function_end label
call_post	after the invocation	duplicate the returns

analysis, this strategy enables us to understand the internal behavior of each smart contract.

3.3.2 Seed Selector. We use the data dependency of the persistent data to guide Engine to construct transactions sequences for tracking the transaction dependency. The persistent state variables are all stored in the database tables, which can be accessed by the library APIs provided by EOSVM (see § 2.2). For example, if an action function asserts that its database must contain data, we first invoke another action function, which can write the database. We use *DBG* (database dependency graph) to record the database accesses, representing the transaction dependency implicitly. At the beginning, Engine executes a random seed (denoted as Γ_1) to initiate the *DBG* and the seed pool. We analyze the used database APIs from the traces and then update *DBG* with the data dependency of the persistent data. Each database API is a pair $\langle \triangle.read | \triangle.write, tb \rangle$, referring to an operation on a database table tb. In the next round, Engine selects an adaptive seed based on the *DBG* content and the seed pool. Firstly, if Γ_1 reads tb, Engine will select a new action function φ_2 from *DBG* to write the same table. Secondly, taking φ_2 as the index, Engine selects Γ_2 from the seed pool. The seed pool is a mapping, where each key is an action name and each item is a circular queue saving the seed candidates. Engine pops the head of the seed candidates of φ_2 and then pushes it back to the queue tail (denoted as $seeds[\varphi_2]$).

3.4 Symback

We build an EOSVM simulator for constructing symbolic constraints from the Wasm traces and solving them to feedback the seed mutation. We show the memory model, calling convention, trace simulation and constraint solver in §3.4.1, §3.4.2, §3.4.3 and §3.4.4, respectively.

Since each Wasm function has its `Local` section frame and stack frame, we use $\hat{}$ as the namespace of the executing function. For example, $\mu_{\hat{s}}$ is the stack frame of the executing function, and $\mu_{\hat{s}}[0]$ is the current stack top.

3.4.1 Memory Model. We first describe the semantics of memory instructions and then introduce the strategies used to address the memory modelling challenge (see § 3.2).

Memory Instructions. There are 23 kinds of Wasm memory instructions. ext_1.load$[size]_[ext_2]$ shows the format of `load` instructions, which loads *size* bits of data from the memory and pushes a result to the stack. The `store` instructions store *size* bits of `ext` data into the memory without type conversion, as follows: ext.store$[size]$.

Table 2: An example of input inference

ρ	Type	`Local`	Linear Memory
from	name	$\mu_{\hat{l}}[1]$	-
to	name	$\mu_{\hat{l}}[2]$	-
quantity	asset	$\mu_{\hat{l}}[3]$	$\mu_m[\mu_{\hat{l}}[3] : \mu_{\hat{l}}[3]+8] \leftarrow$ amount item $\mu_m[\mu_{\hat{l}}[3]+8 : \mu_{\hat{l}}[3]+16] \leftarrow$ symbol item
memo	string	$\mu_{\hat{l}}[4]$	$\mu_m[\mu_{\hat{l}}[4]] \leftarrow$ string length $\mu_m[\mu_{\hat{l}}[4]+1 : \mu_{\hat{l}}[4]+\mu_m[\mu_{\hat{l}}[4]]+1] \leftarrow$ string content

To simulate the byte-addressable memory, we define the simplified models as follows:

- $\triangle.load(\mu_m, addr, size) \mapsto val$: Load *size* bytes memory data from the offset *addr*, and return the result *val* to the stack.
- $\triangle.store(\mu_m, addr, size, val) \mapsto \mu_m$: Store *size* bytes of *val* into memory from the offset *addr*, and return the updated memory.

Memory Access. To address **C2**, we build a memory model with a Z3 [51] byte array. The symbolic expression is split into byte vectors and stored in the memory model. Specifically, we use the Z3 API, i.e., *Store* to insert data and *Select* to load data from the memory model. The address is concrete data read from the runtime traces. Since WASAI performs symbolic execution starting from the action function rather than `void apply()`, we may fail to load some memory contents because not all of them are recorded from the simplified traces. To recover the semantics of memory instructions, we construct a *symbolic load object* for each `load` instruction that reads the unsaved memory data. A *symbolic load object* would be an operand of a Wasm instruction and become the part of the symbolic constraints. If a symbolic constraint contains a *symbolic load object*, the SMT solver will resolve its value by finding a certain address first. To be specific, each *symbolic load object* is a symbolic pair $\langle a, s \rangle$, showing the memory maintains s bytes data at the offset a.

For instance, `f32.store32` is modelled as follows:

$$\mu_m \leftarrow \triangle.store(\mu_m, \tau_p, 4, \mu_{\hat{s}}[1]),$$

where τ_p is the concrete address read from the trace, and the four bytes of data $\mu_{\hat{s}}[1]$ is inserted into the memory μ_m. If there are symbolic expressions for `i32.load16_u` to load, its model is as follows:

$$val \leftarrow \triangle.load(\mu_m, \tau_p, 2)$$

In this example, we return the 16-bits result (*val*) to the stack by converting the loaded data into unsigned 32bits-integer data. When there is no symbolic content in the $\mu_m[\tau_p]$, we assign *val* as $\langle \mu_{\hat{s}}[0], 2 \rangle$.

3.4.2 Calling Convention. To accelerate the feedback process, we skip the redundant paths and perform symbolic execution starting from the action function directly. We first identify the user input from the context of the action function and create symbolic expressions for them (**C3**). According to the specifications of EOSIO SDK, the input parameters (denoted as $\vec{\rho}$) must be in the `Local` section of the action function, i.e., $\mu_{\hat{l}}$. Specifically, there is a consistent one-to-one mapping between the runtime values in $\mu_{\hat{l}}$ and the seed

Table 3: The operational semantic of Wasm instructions. We build symbolic machine states along the traces.

Instruction	Machine state changes	Instruction	Machine state changes	Instruction	Machine state changes
const im	$\mu_{\hat{s}}.push(im)$	global.get im	$x = \mu_g[im]; \mu_{\hat{s}}.push(x)$	br	-
drop	$\mu_{\hat{s}}.pop()$	global.set im	$x = \mu_{\hat{s}}.pop(); \mu_g[im] = x$	br_if	$\mu_{\hat{s}}.pop()$
select	$z, x, y = \mu_{\hat{s}}.pop(3); \mu_{\hat{s}}.push(z?y : x)$	load	$\triangle.load$	br_table	$\mu_{\hat{s}}.pop()$
unary	$x = \mu_{\hat{s}}.pop(); \mu_{\hat{s}}.push(op(x))$	store	$\triangle.store$	return	-
binary	$x, y = \mu_{\hat{s}}.pop(2); \mu_{\hat{s}}.push(op(x, y))$	call_pre	$args = \mu_{\hat{s}}.pop(\$); \mu_l.push(args)$	unreachable	-
local.get im	$x = \mu_{\hat{l}}[im]; \mu_{\hat{s}}.push(x)$	call_post	$\mu_l.pop(); rts = \mu_r.pop(); \mu_{\hat{s}}.extend(rts)$	nop	-
local.set im	$x = \mu_{\hat{s}}.pop(); \mu_{\hat{l}}[im] = x$	begin_function	$\mu_s.push(\emptyset)$	memory.grow	$\mu_{\hat{s}}.pop(); \mu_{\hat{s}}.push(4096)$
local.tee im	$\mu_{\hat{l}}[im] = \mu_{\hat{s}}[0]$	end_function	$rts = \mu_s.pop(); \mu_r.push(rts)$	memory.size	$\mu_{\hat{s}}.push(4096)$

parameters ($\vec{\rho}$) in the context of the action function:

$$\mu_{\hat{l}}[i+1] \Longleftrightarrow \vec{\rho}_i \text{ ;when } 0 \leq i \leq \left|\vec{\rho}\right|$$

Since the maximum unit of Wasm is 64-bit long, Wasm leaves the oversized data in the memory and allocates a local data as a pointer to fetch it. Taking the `asset` type as an example, which is a 128-bit struct composed of two 64-bit elements named `amount` and `symbol`, respectively. When a Wasm function uses the `asset` type parameter, a pointer is allocated in its `Local` section. The pointer points to a 128-bit memory piece, which saves a `amount` type data and a `symbol` type data, accordingly. Similarly, to utilize `string` data, EOSVM allocates a pointer for a continuous piece of memory. The first byte is the length of the string, and the following bytes save the string content.

Table 2 shows the data layout of an action function, i.e., `transfer (name from, name to, asset quantity, string memo)`. The table guides us to build symbolic expressions for the input data. In the entry of the action function, we initiate $\mu_{\hat{l}}$ with the following symbolic expressions, where $\mu_{\hat{l}}[i]$ denotes the i^{th} unit of `Local` section. `from` is the first argument declared in action header, denoted as $\vec{\rho}_0$, thus the second local data stores its value (`from`). Similarly, the third local data saves `to`. $\mu_{\hat{l}}[3]$ is an i32 address pointing to the certain content of the `asset` struct. $\mu_{\hat{l}}[4]$ points to a string indicating the parameter named `memo`.

3.4.3 Traces Simulation. During symbolic execution, we need to lift low-level information in runtime traces to high-level machine states, so that Symback can build and solve symbolic constraints for fuzzer feedback. According to the operational semantics of Wasm instructions shown in Table 3, we update the machine states (i.e., μ) along the runtime traces. The symbols are defined in § 3.1. For the memory instructions, we use $\triangle.load$ and $\triangle.store$ to lift the semantic of `load` and `store`, respectively (see § 3.4.1 for more details.). We use $push()$ and $pop()$ to control the stack-based Wasm components, i.e., μ_s, μ_l and μ_r. For example, $\mu_{\hat{s}}.push(constant)$ means we push a constant into the current stack frame of the executing function. $pop(x)$ means we pop x element(s) from the stack top, where the default value of x is one. $a.extend(b)$ is used to extend the stack a with the values in the b. Based on the low-level hooks shown in Table 1, we label function invocations and capture corresponding traces such as function parameters and the return results. For example, we update μ along the traces to recover the calling convention in Wasm:

- call_pre: the callee's `Local` section ($\mu_l[0]$) is initiated with the invocation parameters from callers' stack ($\mu_{\hat{s}}[0 : \$]$), '$' is the parameter amount of the invoked function, fetched from the traces;
- function_begin: an empty stack frame (denoted as $\emptyset$) is created for callee;
- function_end: callee prepares the returns (denoted as rts) from the callee's stack, maintaining in the μ_r;
- call_post: the context switches to the caller who loads the returns from μ_r to its stack, i.e., $\mu_{\hat{s}}$

Hence, we can obtain the returns from library APIs without simulating their function bodies in the EOSVM simulator. Since we perform the symbolic execution along the executed traces, we omit the jump destinations used in the control flow traces such as `call`, `br`, `br_if` and `br_table`. For `memory.grow` and `memory.size` Wasm instructions, we only balance the stack with a constant value, i.e., 4096, because they cannot affect the control flow or the data flow. Note that 4096B is the default size of the Wasm memory in EOSIO.

3.4.4 Constraints Solver. For each fuzzing seed, we mutate one parameter in $\vec{\rho}$ to touch new branches of the executed paths. To this end, we construct new constraints for the parameter we choose. Firstly, we flip the constraint used in the conditional state as long as it contains the symbolic input built in § 3.4.2. We flip the jumping condition for the opposite branch. Besides, we require the constraint used in the assertion must be satisfied. The formal representation is shown as follows:

$$flip(\mu, \tau(i, \vec{p})) := \begin{cases} \mu_{\hat{s}}[0] == (\vec{p}_1 \oplus 1), \text{ if } i \in \{br\_if, if\} \\ \mu_{\hat{s}}[0] == 1, \text{if eosio\_assert is invoked} \end{cases}$$

Secondly, the path to the conditional state must be feasible. We concatenate the flipped constraint and the path constraints with an `AND` operator. Symback can identify what additional conditions should be satisfied in ρ if it wants to explore deeper. At last, we generate adaptive seeds by solving the constraints with the Z3 Solver [51]. To improve the throughput, we collect the target constraints together and solve them in parallel. Thus we can solve more constraints at the same time and generate more adaptive seeds before reaching the timeout.

3.5 Vulnerability Scanner

We abstract the function calls chain as a list called $\vec{id}$, recording the IDs of the executed functions. For example, id_0 represents `void apply()`. id_e records the function ID of the *eosponser*, which is located from a valid EOS transaction traces. To trigger the vulnerabilities, Engine executes payload transactions according to the oracles discussed in § 2.3. After that, Scanner detects the exploit events via $vul(\tau)$ and outputs *True* for a vulnerable sample.

Fake EOS Detector. After transferring fake EOS to the target contract (see § 2.3.1), if the *eosponser* is actually invoked, Scanner will report a Fake EOS vulnerability, as follows:

$$vul := id_e \in \vec{id}$$

Fake Notif Detector. During the fuzzing, if the guard code is executed for resisting the adversary oracle (see § 2.3.2), the contract is safe. Therefore, it is necessary to search as many paths as we can, otherwise, a false positive may be produced. The guard code should be an instruction (i.e., `i64.eq`, `i64.ne`) in the *eosponser*, where the two stack operands $\tau_p[0], \tau_p[1]$ are the values of `fake.notif` and `_self`, respectively. If no guard code is detected (till timeout), the detector would label it as vulnerable; that is:

$$vul := id_e \in \vec{id} \wedge \vec{\tau} \not\ni (i64.\text{eq}|i64.\text{ne}, (\texttt{fake.token}, \texttt{\_self}))$$

MissAuth Detector. The side-effect behavior is achieved via some library APIs, such as `send_inline` and `db_*`, denoted as $Effects$. If a side-effect is produced without checking the granted authority before, we report a MissAuth vulnerability. We use *Auths* denotes the permission APIs (see § 2.2).

$$vul := any(\{\underset{0\rightarrow i}{\vec{id}} \cap Auths = \emptyset \wedge id_i \in Effects \mid i > 0\})$$

BlockinfoDep Detector. Following previous work [30], we report a BlockinfoDep vulnerability when WASAI identifies an invocation of an API to load blockchain states such as `tapos_block_pre` and `tapos_block_num`. For a formal description, we use `#t` to map the function ID of the function named t.

$$vul := \vec{id} \cap \{\#tapos\_block\_prefix, \#tapos\_block\_num\} \neq \emptyset$$

Rollback Detector. If an inline action is invoked by `send_inline`, we report a vulnerability warning:

$$vul := \#send\_inline \in \vec{id}$$

4 IMPLEMENTATION AND EVALUATION

Implementation. We implemented WASAI with over 4,300 lines of Python code. The contract-level instrumentation is based on Wasabi [39] (commit-0x01f0e26). We use 24 kinds of low-level hooks to capture the traces, which guide WASAI to update the machine states during the symbolic execution. Each low-level hook will invoke the virtual machine to print out traces with library APIs. Specifically, we build a local blockchain using *Nodeos* [46] (version 1.8.6) and further extend *Nodeos* to support three library APIs (see § 3.3.1), including `logi()`, `logsf()` and `logdf()` for i32/i64, f32 and f64 data, respectively. The transaction traces are exported to offline files once one EOSVM thread finishes, i,e, executing the `apply_context::finalize_trace()`. The EOSVM simulator uses Z3 (version 4.8.6) as the SMT backend, and hence all data used in symbolic execution are represented as Z3 bit vectors, e.g., `BitVec` and `FPVal`.

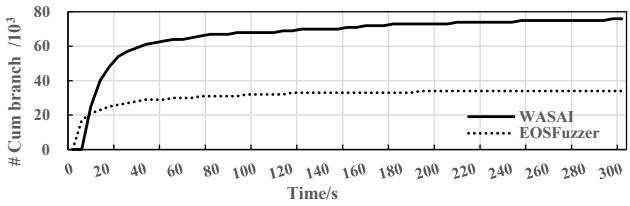

Figure 3: Branch coverage of WASAI and EOSFuzzer (RQ1).

Experimental Setup. Our experiments were performed on a server running Ubuntu 18.04 with an i9-9900 CPU and 64 GB RAM. We empirically set the *timeout* as 5 mins and spend at most 3,000 ms in solving an SMT problem, because we find it is enough to cover most cases (see § 4.1). We also apply a state-of-the-art dynamic Wasm fuzzer (i.e., EOSFuzzer) and a static analyzer (i.e., EOSAFE) with the default settings to the same benchmark for a fair comparison.

Research Questions. Our evaluation is driven by the following four research questions (RQs):

- RQ1 **How much code coverage can WASAI achieve?**
- RQ2 **How accurate is WASAI in detecting vulnerabilities?**
- RQ3 **Is WASAI robust to complicated smart contracts with code obfuscation or sophisticated logic?**
- RQ4 **Can WASAI identify vulnerabilities in real-world smart contracts? Are the vulnerabilities prevalent?**

4.1 RQ1: Code Coverage of WASAI

Approach. We randomly select 100 real-world smart contracts from the EOSIO blockchain and then measure WASAI's code coverage, which is defined as the number of distinct branches explored by a fuzzer. As EOSFuzzer is the only available fuzzer for EOSIO smart contracts, we compare WASAI with it in this setting. For a fair comparison, we only analyze the traces generated by the fuzzing target and ignore those from auxiliary smart contracts, e.g., *eosio.token*.

Results. Figure 3 shows the results of WASAI and EOSFuzzer, where the x-axis is the fuzzing time, and the y-axis is the cumulative number of explored distinct branches for all smart contracts. In the beginning, WASAI has to spend extra time on SMT solving, resulting in a slightly lower code coverage than EOSFuzzer. However, it surpasses the EOSFuzzer after ten seconds, and the advantage of WASAI becomes very significant in the following minutes. Overall, WASAI obtains over 75,000 distinct branches for all these 100 smart contracts, which is about $2x$ of that EOSFuzzer gets. WASAI obviously outperforms EOSFuzzer.

> **Answer to RQ1:** *WASAI achieves about twice as much code coverage than the baseline.*

4.2 RQ2: Accuracy of Vulnerability Detection

Approach. Due to the lack of ground truth, we curate a large-scale dataset via injecting verified vulnerabilities into real-world smart contracts for comparing WASAI with the baselines [29, 30]. The

Table 4: Evaluation results on the ground truth (RQ2).

Types	# Cnt (Vul/Non-Vul)	WASAI			EOSFuzzer			EOSafe		
		P	R	F1	P	R	F1	P	R	F1
Fake EOS	254(127/127)	100%	100%	100%	90.7%	84.3%	87.3%	98.3%	44.9%	61.6%
Fake Notif	1,378(689/689)	100%	100%	100%	94.9%	78.7%	86.0%	67.4%	98.3%	79.9%
MissAuth	890(445/445)	100%	96.0%	97.9%	-	-	-	100%	38.9%	56.0%
BlockinfoDep	400(200/200)	100%	100%	100%	0%	0%	0%	-	-	-
Rollback	418(209/209)	100%	95.7%	97.8%	-	-	-	50.5%	97.6%	66.6%
Total	**3,340(1,670/1,670)**	**100%**	**98.4%**	**99.2%**	**94.2%**	**63.9%**	**76.1%**	**67.7%**	**75.6%**	**71.4%**

benchmark includes 3,340 samples in total, which incorporates 34 samples labeled from the disclosed exploits [29] and 3,306 samples generated by us via inserting exploitable vulnerability into the real-world smart contracts [13]. In this balance benchmark, half of them (#1,670) are vulnerable samples. *To the best of our knowledge, it is by far the largest vulnerable Wasm contract dataset (*100*x of existing benchmarks), and we make it available to the community.*

- For Fake EOS and Fake Notif vulnerability, we first identify non-vulnerable contracts (i.e., $vul(\tau) = False$) if the corresponding guard code is executed. After that, we remove the guard code to generate new vulnerable samples.
- For MissAuth vulnerability, we remove/add the invocation of the permission APIs (see § 2.2) to generate both vulnerable and non-vulnerable contracts.
- For BlockinfoDep and Rollback vulnerabilities, we directly generate the smart contract with vulnerabilities in C++ code and then compile it into Wasm bytecode. We generate several nested `if-else` branches and insert a fixed template code with both two kinds of vulnerabilities (see in Line 9-16 in Listing 4) at the branches ends. Each branch verifies several function parameters with random constants. Notably, by generating inaccessible branches, we can generate non-vulnerable samples with ground truth.

Results. Table 4 shows the overall results. WASAI outperforms all existing techniques. Among the 3,340 smart contracts, WASAI successfully flags 1,643 as vulnerable with 0 FP and 27 FNs, leading to the precision and recall of 100% and 98.4%, respectively. Our manual investigation identifies two main reasons for the FNs. First, it is difficult for WASAI to resolve the transaction dependency when the smart contract can operate several database tables at the same action function, thus leading to FNs. Besides, some smart contracts with the Rollback vulnerability can only be invoked by the caller with the specific address, i.e., its administrator. However, we did not implement an address pool to invoke smart contracts with different identifications. Therefore, WASAI accidentally reports 9 FNs for them because the generated seeds cannot pass the address checking. Nevertheless, WASAI designs oracles to cover all the vulnerabilities considered in the existing studies.

By contrast, EOSFuzzer only performs well in the Fake EOS detection and Fake Notif detection and achieves 87.3% F1-score and 86.0% F1-score, respectively. The FPs of EOSFuzzer come from the flaws in its oracle. For example, it reports positive no matter which action is invoked after receiving fake EOS. As a result, it may produce FP for the honeypot contracts. Besides, due to the lack of feedback strategy, EOSFuzzer produces FNs for Fake Notif because of the unexplored guard code and identifies 0 TP in Rollback because of the unexplored vulnerable code.

EOSafe also performs worse than WASAI. To find the paths to the action function, EOSafe depends on a heuristic strategy to match the dispatcher patterns (e.g., `code == N(eosio.token) && action == N(transfer)`). Since the EOSIO SDK does not require this pattern, i.e., developers can implement it in diverse ways, EOSafe may fail to locate the paths to action functions and report FNs due to the timeout. As a result, it reports 359 FNs, leading to a low recall (75.6%). When detecting Fake Notif, EOSafe regards timeout as a positive sample, and hence it acquires a high recall of 98.3% but a poor precision of 67.4%. To detect Rollback, EOSafe analyzes all branches in the conditional states, even if the constraints are impossible to be satisfied, and thus produces FPs. As a result, EOSafe achieves a precision of 50.5% for detecting Rollback.

> **Answer to RQ2:** *WASAI outperforms all state-of-the-art techniques on a large-scale benchmark, with 100% of precision and 98.4% of recall.*

4.3 RQ3: Robustness of WASAI

Approach. We create two types of complicated samples and use them to evaluate the robustness of WASAI and the baselines [29, 30].

• *code obfuscation.* Based on the benchmarks we harvested in § 4.2, we further generate their obfuscated versions. Since there is no available obfuscation tool for Wasm bytecode, we develop one with two obfuscation methods. First, it obfuscates the data flow by encoding function arguments with the popcount algorithm[27], which counts the number of '1' bits in the given value. Second, it obfuscates the control flow by inserting recursion invocations to the bytecode, where the entry condition is impossibly satisfied. As a result, we generated 3,340 obfuscated samples.

• *complicated verification.* Some real-world contracts may verify the input before allowing users to participate. It requires dynamic tools to generate adaptive seeds to satisfy the complicated path constraints, otherwise, they are prone to FNs due to the low code coverage. To generate samples with complicated verification, we inject several `if` code constructs, which verify the input data with random data. If the verification fails, the injected code will enforce the smart contract to terminate the execution by a Wasm instruction, i.e., `unreachable`. For example, we can inject the following Wasm code into the entry of the action function of `transfer@eosio.token` to ensure the value of `quantity` must be `"100.000 EOS"`. As a result, we generate 2,924 (87.5%) samples with complicated verification logic.

```
if(i64.ne local.get 3 (i64.load) i64.const 100000)(then unreachable)
if(i64.ne local.get 3 (i64.load offset=8) i64.const 1397703940)(then
 unreachable)
```

Results on the Obfuscation Benchmark. As shown in Table 5, WASAI can remain a high accuracy under code obfuscation, with 96.6% of precision and 97.9% of recall. The F1-score of EOSFuzzer reaches 76.5%, which is not affected much by the code obfuscation. However, the performance of EOSafe decreases greatly due to code obfuscation, with 881 TPs (decreased by 20.7%) and 943 FPs. Notably, EOSafe cannot find any feasible paths to detect Fake EOS vulnerability and MissAuth vulnerability, leading to 0 TP. To the end, EOSafe reaches a low F1-score of 61.2% (decreased by 10.2%).

Table 5: The impact of code obfuscation (RQ3).

Types	# Cnt (Vul/Non-Vul)	WASAI			EOSFuzzer			EOSafe		
		P	R	F1	P	R	F1	P	R	F1
Fake EOS	254(127/127)	100%	100%	100%	91.4%	92.1%	91.8%	0%	0%	0%
Fake Notif	1,378(689/689)	92.4%	100%	96.0%	94.6%	78.1%	85.5%	67.5%	98.4%	80.0%
MissAuth	890(445/445)	100%	94.2%	97.0%	-	-	-	0%	0%	0%
BlockinfoDep	400(200/200)	100%	100%	100%	0%	0%	0%	-	-	-
Rollback	418(209/209)	100%	95.7%	97.8%	-	-	-	50.4%	97.1%	66.3%
Total	**3,340(1,670/1,670)**	**96.6%**	**97.9%**	**97.3%**	**94.0%**	**64.5%**	**76.5%**	**62.6%**	**59.9%**	**61.2%**

Table 6: The impact of complicated verification (RQ3).

Types	# Cnt (Vul/Non-Vul)	WASAI			EOSFuzzer			EOSafe		
		P	R	F1	P	R	F1	P	R	F1
Fake EOS	190(95/95)	100%	100%	100%	50.0%	100%	66.7%	100%	43.2%	60.3%
Fake Notif	1,178(589/589)	99.6%	83.0%	90.6%	0%	0%	0%	68.1%	99.3%	80.8%
MissAuth	756(378/378)	100%	97.4%	98.7%	-	-	-	100%	40.5%	57.6%
BlockinfoDep	400(200/200)	100%	100%	100%	0%	0%	0%	-	-	-
Rollback	400(200/200)	100%	100%	100%	-	-	-	50%	100%	66.7%
Total	**2,924(1,462/1,462)**	**99.9%**	**92.5%**	**96.0%**	**50%**	**10.7%**	**17.7%**	**67.4%**	**77.6%**	**72.1%**

Results on the Complicated Verification Benchmark. With the effective feedback strategy, WASAI retains 99.9% precision and 92.5% recall. Since the injected code only creates several short paths in the CFG, EOSafe can cover them by exhaustive searching. As a result, it can still maintains its performance with 72.1% of F1-score. However, EOSFuzzer is affected greatly, achieving only 50% precision and 10.7% recall. Since EOSFuzzer cannot choose a valid payload for the next fuzzing round after the transaction is reverted by the injected code, it cannot report any positive cases for the Fake Notif and BlockinfoDep. Specially, there are some problems in the EOSFuzzer's oracles, e.g., it outputs a positive report in detecting Fake EOS if none of the transactions is executed successfully. Since only the elaborate input can bypass the complicated verification, the random seeds generated by EOSFuzzer usually fail to pass the verification and terminate the execution of smart contract. According to its flaw in the oracle, EOSFuzzer fails to execute the fuzzing target every time and flags all samples as vulnerable in detecting the Fake EOS. Hence, EOSFuzzer achieves $\frac{1462}{1462+1462} \times 100\% = 50.0\%$ of precision for the Fake EOS benchmarks.

Note that WASAI produces FPs in the detection of the Fake Notif samples in the both benchmarks. It is caused by the assumption used in the oracle (§ 3.5). We will explain more in RQ4 through manually analyzing several FPs.

> **Answer to RQ3:** *WASAI maintains high detection accuracy when handling samples with code obfuscation and complicated verification logic, which is more robust than the baselines.*

4.4 RQ4: Vulnerabilities in the Wild

Approach. We want to analyze the profitable smart contracts (§ 2.1) because the vulnerabilities in them may cause losses of cryptocurrency. The authors of EOSFuzzer[30] have collected 3,881 real-world smart contracts from EOSIO Mainnet, where 991 (25.5%) of them are profitable. These 991 samples are identified with the following steps: 1. we allocate some EOS tokens to the fuzzing target; 2. and label it as profitable if any action is activated in our local blockchain. Next, we apply WASAI to these 991 smart contracts. *Note that EOSIO allows developers to update their smart contracts in Mainnet.* Thus, we further analyze whether these flagged vulnerable smart contracts have been patched in their later versions, which are downloaded from the Mainnet blockchain via an RPC service [17]. Moreover, we manually verify the results. For each kind of vulnerability, we randomly select ten vulnerable samples and ten bug-free samples labeled by WASAI(i.e., 100 smart contracts) for verification in total. We rely on dynamic debugging and reverse engineering to verify the results.

Overall results. 707 contracts (71.3%) are flagged as vulnerable, i.e., WASAI identifies 241 Fake EOS, 264 Fake Notif, 470 MissAuth, 22 BlockinfoDep and 122 Rollback. For the latest versions of these 707 flagged contracts, we observe that 58.4% (413) of them are still operating on the Mainnet, and the remaining flagged vulnerable ones are abandoned by their developers (i.e., the latest versions are replaced with empty files). For the 413 working ones, only 72 of them have been patched[1]. In other words, 341 vulnerable contracts are exposed to attackers. We next present the zero-day vulnerability in *batdappboomx*[2] identified by WASAI. This bug has been assigned with a CVE identifier (CVE-2022-27134). WASAI found that anyone can activate the *eosponser* (i.e., *transfer@batdappboomx*) of *batdappboomx* directly with an fake EOS. Thus attackers can receive the reward from *batdappboomx* as long as they set the parameter `memo` as "action:buy". Unfortunately, this smart contract has not been patched yet.

Manual Analysis. We observed only 2 false positives and 1 false negative among the 100 samples. WASAI accidentally reports two false positives in detecting Fake Notif (i.e., *paytobtckey1* (deployed at 2019-05-25, 09:08:30) and *zlkggamerobs* (deployed at 2019-01-22, 20:24:38)) because of the assumption used by its oracle. The oracle may flag a false positive due to the insufficient code coverage (§ 3.5). When testing the *paytobtckey1* (see § 2.3.2), WASAI cannot set the transaction parameter 'memo' as a 26 bytes string, thus it fails to touch guard code in the deeper program states. WASAI also fails to detect *eospindealer* (deployed at 2019-01-09, 15:05:54) as Rollback vulnerability, since its extreme complicated control flow prevents us from analysing its deep code in five minutes.

> **Answer to RQ4:** *WASAI reveals the urgency of fixing the vulnerabilities in EOSIO smart contracts.*

5 THREATS TO VALIDITY

Although WASAI achieves promising results, our work faces several limitations. First, we limit the resources of the SMT solver for a larger throughput, and hence WASAI produces some FNs because of unsolved branches. Actually, we can get better results by extending the fuzzing time, while it is a trade-off between scalability and efficiency. Second, we construct the dependency of the persistent data at the table level, which is a coarse-grained strategy that may not guide WASAI to select a proper seed. In the future, we can parse the database index and construct more comprehensive transaction sequences to explore more code. Third, we do not measure the branch coverage in RQ1. Since the vulnerabilities can only be triggered in the action functions, WASAI only focuses on exploring branches in the action functions rather than the entire

[1]Note that we further applied WASAI to analyze their latest version to investigate whether the vulnerability has been patched.
[2]SHA256: 1327c04cf4b56183eddb1a897bbebf5a873d3421272b708829a4ed0765bef820

smart contract. Fourth, the benchmark should be improved. On the real-world benchmark, WASAI outperforms the baselines. To emphasize the novelty of WASAI, we further extend the existing benchmark by modifying the smart contract binary, following the idea of LAVA [13]. Although the synthetic data only focus on several kinds of vulnerabilities, it indeed demonstrates the advantages of WASAI in terms of robustness and effectiveness. In the future, we will construct a larger benchmark from cryptocurrency forms and disclosed exploits.

It is worth noting that we provide interfaces for the ease of extending WASAI. In particular, the bug detectors can be extended in two steps: (1) adding oracles and constructing the payload templates. During the fuzzing, WASAI mutates specific arguments in such payload templates; (2) analyzing traces to confirm the exploit events. Moreover, the proposed memory model can be applied to other languages, as long as we set the size of the memory unit (e.g., 8 bits for EVM bytecode) and know the length of data accessed by the memory instructions. Last but not least, WASAI can be extended to analyze other Wasm applications in different platforms, such as browsers, IoT, and other blockchain platforms (e.g., Polkadot), through three steps: (1) obtaining function signatures of the fuzzing target; (2) customizing Wasm client's hooks to export traces; and (3) informing WASAI what APIs can access the persistent storage in order to find out the transaction dependency.

6 RELATED WORK

Ethereum Smart Contract Analysis. Many studies have been proposed for security analysis of Ethereum smart contracts [5, 8, 10, 12, 14, 22, 23, 26, 30, 34, 36, 40, 41, 45, 47, 50, 53, 56, 57, 60, 64, 66, 68]. ContractFuzzer [34] is the first fuzzer framework in detecting vulnerabilities in Ethereum [19] smart contracts. Tai et al. implement the first AFL-Favour fuzzer in EVM. Valentin and Maria [65] propose a greybox fuzzer, Harvey, which flips the input bytes according to the code-coverage information. Harvey measures the energy cost of comparison operators and storage assignments without identifying whether the operands are tainted by user input, hence it may fail to explore deep paths because of the complex path constraints. These work cannot be used directly to analyze Wasm contracts due to the following difference: (1) Calling convention: A Wasm contract usually contains several functions in its binary, and thus the symbolic execution engine needs to identify the function namespace to construct the symbolic constraints. By contrast, all functions in an Ethereum smart contract share the same stack frame. (2) Seed format: In EOSIO, the input data of a smart contract is a byte stream which will be deserialized and stay in the memory. Such complex memory operations may make traditional symbolic execution fail with timeout due to path explosion. Note that for EVM, smart contracts must load the user input from a fixed-size vector (i.e., calldata), and thus it is straightforward to perform symbolic execution and find the new seeds. (3) Transaction dependency: Smart contract is a stateful program that can store persistent data in its database with specific APIs. The data flow in the persistent data may indicate transaction dependency. Unlike Ethereum fuzzers that just need to handle the two Opcodes that can access the storage (i.e., SLOAD and SSTORE), WASAI has to analyze more storage APIs to construct the transaction sequences. (4) A unique attack surface in EOSIO due to its notification mechanism.

Wasm Binary Analysis. There are some recent studies on Wasm applications [1, 2, 35, 38, 62]. To address the security issues, Jonathan et al. [48] applied formal verification techniques to those sophisticated cryptographic components written in Wasm. Weikang et al. [3] model the application's behavior using the Wasm instruction execution trace. However, all of these studies focus on Web applications and cannot be applied to EOSIO directly.

EOSIO Smart Contract Analysis. There are only a few studies[29, 30, 61] on EOSIO contracts. For symbolic execution based techniques, the path explosion is the major challenge, as there are several kinds of indirect jumping instructions in Wasm. The state-of-the-art approaches such as WANA [61] and EOSafe [29] are unable to find exploit payloads. EOSFuzzer, the only fuzzer [30], lacks a feedback phase and thus it cannot generate adaptive seeds to explore deep code.

7 CONCLUSION

We design and develop WASAI, the first concolic fuzzer for uncovering the vulnerabilities in Wasm smart contracts. Extensive experiment results on well-labeled benchmarks demonstrate the great performance of WASAI, outperforming state-of-the-art techniques. Applying WASAI to the deployed smart contracts in the wild, we detect over 300 unpatched vulnerable smart contracts.

DATA AVAILABILITY STATEMENT

WASAI and its dataset are released at 10.5281/zenodo.6517515.

ACKNOWLEDGMENT

We sincerely thank our shepherd and the anonymous reviewers for their valuable feedback and suggestions. This work was supported by the National Key R&D Program of China (2021YFB2701000), National Natural Science Foundation of China (grants No.62172360, No.U21A20467 and No.62072046) and Hong Kong RGC Projects (No.PolyU15219319 and No.PolyU15224121).

REFERENCES

[1] T. Lukasiewicz B. McFadden and J. Engler. 2020. Security Chasms of WASM. (Oct. 2020). [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf.

[2] J. Bergbom. 2021. Memory safety: old vulnerabilities become new with WebAssembly. [Online]. Available: https://www.forcepoint.com/sites/default/files/resources/files/report-web-assemblymemory-safety-en.pdf.

[3] W. Bian, W. Meng, and Y. Wang. 2019. Poster: Detecting WebAssembly-Based Cryptocurrency Mining. In *Proc. ACM SIGSAC Conference on Computer and Communications Security.*

[4] Block.one. 2021. A blockchain protocol with industry-leading transaction speed and flexible utility. [Online]. Available: https://eos.io/.

[5] L. Breidenbach, P. Daian, F. Tramèr, and A. Juels. 2018. Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts. In *Proc. USENIX Security Symposium.*

[6] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. 2022. DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Transactions on Software Engineering* (2022).

[7] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022).

[8] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. In *Proc. Network and Distributed System Security Symposium.*

[9] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proc. ACM Conference on Computer and Communications Security.*

[10] Z. Chen, C. Wang, J. Yan, Y. Sui, and J. Xue. 2021. Runtime Detection of Memory Errors with Smart Status. In *Proc.ACM SIGSOFT International Symposium on Software Testing and Analysis.*

[11] Cointelegraph. 2021. EOS DApps Lose Almost $1 Million to Hackers Over the Last Five Months. [Online]. Available: https://cointelegraph.com/news/eos-dapps-lose-almost-1-million-to-hackers-over-the-last-five-months.

[12] ConsenSys. 2021. Mythril, security analysis tool for EVM bytecode. [Online]. Available: https://github.com/ConsenSys/mythril.

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

[14] T. Durieux, JF. Ferreira., R. Abreu, and P. Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proc. ACM/IEEE 42nd International Conference on Software Engineering.*

[15] Niguez Randomity Engine. 2019. [Online]. Available: https://niguezrandomityengine.github.io/.

[16] EOSIO. 2021. C++ API. [Online]. Available: https://developers.eos.io/manuals/eosio.cdt/v1.5/group__cpp__api.

[17] EOSIO. 2021. Nodeos RPC API Reference. [Online]. Available: https://developers.eos.io/welcome/v2.2/reference/nodeos-rpc-api-reference.

[18] EOSIO. 2021. The repository of the EOS VM. [Online]. Available: https://github.com/EOSIO/eos-vm.

[19] Ethereum. 2021. Ethereum Official Site. [Online]. Available: https://ethereum.org/.

[20] Ethereum. 2021. A smart contract is simply a program that runs on the Ethereum blockchain. [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/.

[21] Ethereum. 2021. A virtual machine to run Ethereum smart contracts. [Online]. Available: https://ethereum.org/en/developers/docs/evm/.

[22] Y. Feng, E. Torlak, and R. Bodik. 2020. Summary-Based Symbolic Evaluation for Smart Contracts. In *Proc. IEEE/ACM International Conference on Automated Software Engineering.*

[23] J. Frank, C. Aschermann, and T. Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *Proc. USENIX Security Symposium.*

[24] Jake Frankenfield. 2021. EOS ICO. [Online.] Avaliable: https://www.investopedia.com/terms/i/initial-coin-offering-ico.asp.

[25] A. Ghaleb and K. Pattabiraman. 2020. How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis.*

[26] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proc. International Conference on Software Engineering.*

[27] HackersDelight. 2021. Hakmem Popcnt Algorithm. [Online]. Available: http://www.hackersdelight.org/.

[28] J. He, B. Mislav, A. Nodar, T. Petar, and V. Martin. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proc. ACM SIGSAC Conference on Computer and Communications Security.*

[29] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang. 2021. EOSAFE: Security Analysis of EOSIO Smart Contracts. In *Proc. USENIX Security Symposium.*

[30] Y. Huang, B. Jiang, and W.K. Chan. 2020. EOSFuzzer: Fuzzing EOSIO Smart Contracts for Vulnerability Detection. In *Proc. Asia-Pacific Symposium on Internetware.*

[31] Huobi Inc. 2021. Million EOS Disappears in a Hack Attack EOS Accounts Blocked by huobi. [Online]. Available: https://www.forexcrunch.com/eos-news-update-2-09-million-eos-disappears-in-a-hack-attack-eos-accounts-blocked-by-houbi.

[32] PeckShield Inc. 2021. Blogs about blockchain security events. [Online]. Available: https://blog.peckshield.com/blog.html.

[33] SlowMist Inc. 2021. Blockchain security events. [Online]. Available: https://hacked.slowmist.io/en/.

[34] B. Jiang, Y. Liu, and W.K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proc. International Conference on Automated Software Engineering.*

[35] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. 2021. SFI safety for native-compiled Wasm. In *Proc. Network and Distributed System Security Symposium.*

[36] J. Krupp and C. Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proc. USENIX Security Symposium.*

[37] S. Lagouvardos, N. Grech, I. Tsatiris, and Y. Smaragdakis. 2020. Precise Static Modeling of Ethereum Memory. *ACM Program. Lang. OOPSLA* (2020).

[38] D. Lehmann, J. Kinder, and M. Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *Proc. USENIX Security Symposium.*

[39] D. Lehmann and M. Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems.*

[40] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. 2016. Making Smart Contracts Smarter. In *Proc. ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.*

[41] B. Mariano, Y. Chen, Y. Feng, S. Lahiri, and I. Dillig. 2020. Demystifying Loops in Smart Contracts. In *Proc. IEEE/ACM International Conference on Automated Software Engineering.*

[42] R. Mitra. 2021. EOS/USD market drops by 4% following $7.7 million EOS hack attack. [Online]. Available: https://www.fxstreet.com/cryptocurrencies/news/eos-usd-market-drops-by-4-following-77-million-eos-hack-attack-201902262151.

[43] NEAR. 2021. A blockchain platform that accelerates the development of webAssembly smart contract. [Online]. Available: https://near.org/.

[44] TD. Nguyen, LH. Pham, and J. Sun. 2021. SGUARD: Towards Fixing Vulnerable Smart Contracts Automatically. In *Proc. Symposium on Security and Privacy.*

[45] TD. Nguyen, LH. Pham, J. Sun, Y. Lin, and QT. Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proc. ACM/IEEE 42nd International Conference on Software Engineering.*

[46] Nodoes. 2019. The core service daemon that runs on every EOSIO node. [Online]. Available: https://developers.eos.io/manuals/eos/v2.1/nodeos/index.

[47] D. Perez and B. Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *Proc. USENIX Security Symposium.*

[48] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. 2019. Formally Verified Cryptographic Web Applications in WebAssembly. In *Proc. IEEE Symposium on Security and Privacy.*

[49] L. Quan, L. Wu, and H. Wang. 2019. EVulHunter: Detecting Fake Transfer Vulnerabilities for EOSIO's Smart Contracts at Webassembly-level. (2019).

[50] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai. 2021. Empirical Evaluation of Smart Contract Testing: What is the Best Choice?. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis.*

[51] Microsoft Research. 2021. Z3, a theorem prover from Microsoft Research. [Online]. Available: https://github.com/Z3Prover/z3.

[52] M. Rodler, W. Li, G. O. Karame, and L. Davi. 2021. EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts. In *Proc. USENIX Security Symposium.*

[53] S. So, S. Hong, and H. Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *Proc. USENIX Security Symposium.*

[54] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proc. Annual Network and Distributed System Security Symposium.*

[55] S. Cha T. Avgerinos, A. Rebert and D. Brumley. 2014. Enhancing symbolic execution with veritesting. In *Proc. International Conference on Software Engineering.*

[56] C. Torres, M. Baden, R. Norvill, B. Fiz, H. Jonker, and S. Mauw. 2020. ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks. In *Proc. ACM Asia Conference on Computer and Communications Security.*

[57] C. Torres, M. Steichen, and R. State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *Proc. USENIX Security Symposium.*

[58] A. Turner. 2021. WebAssembly Linear Memory. [Online]. Available: https://wasmbyexample.dev/examples/webassembly-linear-memory/webassembly-linear-memory.rust.en-us.html.

[59] W3C. 2021. "The main page of webassembly.org". [Online]. Available: https://webassembly.org/.

[60] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo, and X. Yang. 2021. Smart Contract Security: a Practitioners' Perspective. In *Proc. IEEE/ACM International Conference on Software Engineering.*

[61] D. Wang, B. Jiang, and W.K. Chan. 2020. WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection. In *arXiv preprint arXiv:2007.15510.*

[62] C. Watt, A. Rossberg, and J. Pichon-Pharabod. 2019. Weakening WebAssembly. *ACM Program. Lang. OOPSLA* (2019).

[63] WebAssembly. 2021. Operation Semantic of WebAssembly. [Online]. Available: https://webassembly.github.io/spec/core/text/instructions.html.

[64] S. Wu, L. Wu, Y. Zhou, R. Li, Z. Wang, X. Luo, C. Wang, and K. Ren. 2022. Time-Travel Investigation: Towards Building A Scalable Attack Detection Framework on Ethereum. *ACM Transactions on Software Engineering and Methodology* (2022).

[65] V. Wüstholz and M. Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering.*

[66] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng. 2020. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *Proc. IEEE/ACM International Conference on Automated Software Engineering.*

[67] P. Zhang, F. Xiao, and X. Luo. 2019. SolidityCheck : Quickly Detecting Smart Contract Problems Through Regular Expressions. *arXiv preprint arXiv:1911.09425* (2019).

[68] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey. 2018. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. In *Proc. USENIX Security Symposium.*