EOSAFE: Security Analysis of EOSIO Smart Contracts

Ningyu He[1]*, Ruiyi Zhang[2]*, Haoyu Wang[3]‡, Lei Wu[4]‡, Xiapu Luo[5]
Yao Guo[1]‡, Ting Yu[6], Xuxian Jiang[2]

[1]*Key Lab on HCST (MOE), Peking University* [2]*PeckShield, Inc.*
[3]*Beijing University of Posts and Telecommunications* [4]*Zhejiang University*
[5]*The Hong Kong Polytechnic University* [6]*Qatar Computing Research Institute*
* Co-first authors ‡Co-corresponding authors

Abstract

The EOSIO blockchain, one of the representative Delegated Proof-of-Stake (DPoS) blockchain platforms, has grown rapidly recently. Meanwhile, a number of vulnerabilities and high-profile attacks against top EOSIO DApps and their smart contracts have also been discovered and observed in the wild, resulting in serious financial damages. Most of the EOSIO smart contracts are not open-sourced and typically compiled to WebAssembly (Wasm) bytecode, thus making it challenging to analyze and detect the presence of possible vulnerabilities. In this paper, we propose EOSAFE, the first static analysis framework that can be used to automatically detect vulnerabilities in EOSIO smart contracts at the bytecode level. Our framework includes a practical symbolic execution engine for Wasm, a customized library emulator for EOSIO smart contracts, and four heuristic-driven detectors to identify the presence of the four most popular vulnerabilities in EOSIO smart contracts. Experiments have shown that EOSAFE achieves promising results in detecting vulnerabilities, with an F1-measure of 98%. We have applied EOSAFE to all active 53,666 smart contracts in the ecosystem (as of November 15, 2019). Our results show that over 25% of the smart contracts are labeled vulnerable. We further analyze possible exploitation attempts on these vulnerable smart contracts and identify 48 in-the-wild attacks (27 of them have been confirmed by DApp developers), which have resulted in financial loss of at least 1.7 million USD.

1 Introduction

With the growing prosperity of cryptocurrencies (e.g., Bitcoin), blockchain techniques have become more attractive and been adopted in a number of areas. Due to the limited throughput (e.g., Transaction Per Second, aka TPS) derived from the inherent principle of the Proof-of-Work consensus, traditional blockchain platforms (e.g., Bitcoin and Ethereum) cannot be used to support high performance applications. Researchers have proposed different consensus protocols, e.g., Proof-of-Stack (PoS) [1] and Delegated Proof-of-Stake (DPoS) [2], to resolve the performance issues.

As one of the most representative DPoS platforms and the first decentralized operating system, EOSIO has become one of the most active global communities. EOSIO adopts a multi-threaded mechanism based on its DPoS consensus protocol, which is capable of achieving millions of TPS. The performance advantage of EOSIO makes it popular for Decentralized Application (DApp) developers. EOSIO has successfully surpassed Ethereum in DApp transactions just three months after its launch in June 2018 [3]. Currently, the transaction volume of EOSIO on average is more than a hundred times greater than Ethereum [4]. As of 2019, the total value of on-chain transactions of EOSIO has reached over $ 6 billion.

A *smart contract* is a computer protocol that allows users to digitally negotiate an agreement in a convenient and secure way. In contrast to the traditional contract law, the transaction costs of a smart contract are dramatically reduced, and the correctness of its execution is ensured by the consensus protocol. EOSIO smart contracts can be written in C++, which will be compiled to WebAssembly (Wasm) and executed in the EOS Virtual Machine (EOS VM). Wasm is a web standard specifying the binary instruction format for a stack-based VM. It can run in modern web browsers and other environments [5].

However, it is not easy to guarantee the security of the implementation of smart contracts, EOSIO in particular. A number of vulnerabilities have been discovered in EOSIO smart contracts, while severe attacks have been observed in the wild, which caused great financial damages. For instance, in fall 2018, a gambling DApp, EOSBet, was attacked twice within just a month [6, 7] due to *fake EOS* and *fake receipt* vulnerabilities, causing 40,000 and 65,000 EOS losses, respectively. Therefore, it is necessary to identify the security issues of smart contracts in order to prevent such attacks.

Unfortunately, most smart contracts on EOSIO are not open-sourced, and there are few analysis tools towards analyzing Wasm bytecode, which makes it more difficult to detect vulnerabilities for EOSIO smart contracts automatically. As Wasm bytecode can be converted to C code using the official tool *wasm2c* [8], which naturally provides a potentially promising approach that analyzes the converted C code

rather than raw Wasm bytecode, so we can apply widely used tools such as KLEE [9]. However, our investigation shows that such a solution is not practical, i.e., KLEE failed to perform the detection in most cases, due to reasons including timeout and out-of-memory (OOM) issues caused by path explosion. This can be possibly explained by the adopted memory model [10,11] which may lead to heavy memory and time consumption, as reported by [12]. Furthermore, the conversion from Wasm bytecode to C code requires extra human efforts to prepare all exported functions (including function signatures and the logic) for EOSIO smart contracts. Otherwise, the compilation and the symbolic execution cannot be successfully completed. Moreover, the quality of the converted C code cannot be guaranteed, because wasm2c itself is still under development and may not be stable considering the bugs that have been discovered so far [13]. In short, the C language based solutions rely on sophisticated conversion tool(s), so they are typically too heavy to perform the analysis for EOSIO smart contracts (see §7.1).

As such, this paper attempts to analyze Wasm bytecode directly to detect vulnerabilities in EOSIO smart contracts. Although many efforts have been made to analyze Ethereum smart contracts [14–19], none of them, however, can be applied to EOSIO smart contracts, as these two ecosystems are totally different, ranging from their virtual machines, the structure of bytecode, to the types of vulnerabilities. Specifically, there exist several challenges. Firstly, EOS VM is more complicated than Ethereum VM in regard to their instructions, including both quantity and variety. For example, EOS VM supports floating point operations, type conversion and advanced jump instructions [20], none of these features are supported in Ethereum VM at the opcode level [21,22]. Secondly, although with a well-structured format, the Wasm bytecode is complicated to analyze due to the multi-level nested structures, which makes it difficult to perform the semantic-level recovery for further analysis/detection. Thirdly, most EOSIO vulnerabilities discovered so far are more complicated than traditional simple vulnerabilities, e.g., integer overflow. Thus it usually requires more semantic information, e.g., fields of the platform-specific data structure as the indexes, to model and analyze them. For example, to detect the fake EOS vulnerability (see § 3.1), we need to check the specific value of the argument `code` in the function `apply`.

This Paper. We implement EOSAFE, the first systematic static analysis framework for detecting vulnerabilities in EOSIO smart contracts. Specifically, we first implement a native symbolic execution engine for Wasm bytecode, and mitigate the inherent path explosion problem by applying a heuristic-guided pruning approach. Second, to analyze an EOSIO smart contract and simulate its external interactive environment, we implement an emulator to mimic the behaviors of key EOSIO library functions that are crucial in vulnerability detection. Third, we propose a generic vulnerability detection framework, which allows security analysts to easily implement their own vulnerability detectors as plugins. In this work, we have implemented four detectors aiming to detect four high-profile vulnerabilities, including *fake EOS*, *fake receipt*, *rollback* and *missing permission check* (see §3).

To evaluate the effectiveness of EOSAFE, we first manually crafted a benchmark suite including 52 smart contracts, which is composed of vulnerable smart contracts collected from publicly verified attacks and their corresponding patched ones. Experimental results and further manual verification suggest that EOSAFE achieves excellent performance in identifying existing vulnerabilities. To measure the overall landscape of vulnerabilities in the EOSIO ecosystem, we further applied EOSAFE to all the smart contracts in the ecosystem (53,666 in total). Experiment results reveal that security vulnerabilities are prevalent: over 25% of the smart contracts (including historical versions) are flagged as vulnerable, and a large portion of them have not been patched timely. To further measure the impact of these vulnerabilities, we collect the transaction records (over 2.5 billion transactions in total), and design a set of conservative heuristic strategies to identify attacks targeting these vulnerable smart contracts. We have identified 48 attacks in total, as well as 183 missing permission check actions. By the time of this writing, 27 attacks have been confirmed by DApp developers, which have already caused the financial loss of over 1.7 million USD.

This paper makes the following main contributions:

- We propose EOSAFE, the first systematic static analysis framework for EOSIO smart contracts, which is capable of detecting four kinds of popular vulnerabilities. Experiment results demonstrate that EOSAFE achieves excellent performance.
- We propose a *valuable-function-centric* detection framework, which is based on our observed vulnerability-specific pruning strategies, to effectively mitigate the path explosion issue in symbolic execution.
- We apply EOSAFE to over 53K EOSIO smart contracts, and perform the first measurement study of the whole EOSIO ecosystem. Our results reveal the severity of the security issues, i.e., over 25% of the EOSIO smart contracts may have been exposed to the threats introduced by these vulnerabilities.
- We have identified 48 attacks (35 of them were first discovered) and 183 missing permission check actions related to the identified vulnerabilities, which have caused huge financial loss. Most of the severe attacks have been confirmed by DApp Teams.

To boost further research on EOSIO smart contracts, we have released the benchmark and experiment results to the research community at [23].

2 Background

As the first industrial-scale decentralized operating system [24], the EOSIO platform can achieve high performance, i.e., millions of TPS, to efficiently execute complicated

DApps. Such efficiency is in large part due to the consensus algorithm it uses, i.e., DPoS, which does not spend a vast amount of computing resources on the unnecessary mining process compared to traditional PoW. We next introduce some key concepts to facilitate the understanding of this work.

2.1 Account Management

An *account* in EOSIO is the basic unit to identify an entity. It can trigger transactions to other accounts. Additionally, to ensure account security and prevent identity fraud, EOSIO implements an advanced *permission-based access control system*. Specifically, an account can assign public/private keys to specific actions, and a particular key pair will only be able to execute the corresponding action. By default, an EOSIO account is attached to two public keys: the *owner* key (which specifies the ownership of the account) and the *active* key (which grants access to activities with the account). These two keys authorize two native named permissions: the *owner* and *active* permission, to manage accounts. Apart from the native permissions, EOSIO also allows customized named permissions for advanced account management.

```
1  void apply(uint64_t receiver, uint64_t code,
       uint64_t action) {
2    if(action == N(onerror)) {
3      check(code == N(eosio), "exception captured");
4    }
5    auto self = receiver;
6    if((code == self || code == N(eosio.token))) {
7      switch(action) {
8        case N(transfer): // action == N(transfer)
9        // deal with:
10       // 1. direct invocation to transfer function
11       // 2. notification emitted from transfer
12         ...
13     }
14   }
15 }
```

Listing 1: An example of the `apply` function with slight modification for better readability. The function `N` is used to retrieve the string literal.

Unlike Ethereum, an *EOSIO smart contract* is not treated as a separate entity. A smart contract is just a snippet of code stored in an account, which makes it easy to explain why a smart contract in EOSIO is *updatable*, rather than an Ethereum smart contract that cannot be changed freely by the owner. Therefore, when an account is invoked by another one, its smart contract will be responsible for handling the received invocation. In this way, it requires a *dispatcher* to dispatch the requests to the corresponding functions. Specifically, in EOSIO, this dispatcher is officially defined as a function named *`apply`* with a fixed function signature[1], as the example shown in Listing 1. As the entry point of each EOSIO smart contract, the `apply` function is responsible for handling all the requests, including invoked actions and received notifications (see §2.2), which will be forwarded to the corresponding processing functions. Besides, the `apply` function can be used to validate the input parameters if necessary. The details of the parameters and the mechanism of the `apply` function will be discussed in §2.2.

2.2 EOSIO Transactions

A *transaction* is the basic unit to be verified and packaged in blocks. Moreover, a transaction is composed of one or multiple *action*s, and an action is the basic unit to trigger functions. For example, the `action` in Listing 1 (line 1) specifies the target function name. The action is responsible for carrying permissions designated by the invoker. Moreover, another nested action can be triggered by `send_inline` as an *inlined actions*[2], which is still an ordinary action and inherits the context (including permissions) of its parent. Note that a failure in an action could lead to the revert of the whole transaction.

Besides transaction and action, there exists another exclusive mechanism named *notification*. Specifically, it is used to notify a target account of the current action being executed, indicating the name of the function that initiates the notification (let `fn` be the function name). After that, the notified account has to process the notification by triggering the function with the same name `fn` through the dispatcher.

Figure 1 provides a concrete example to illustrate the mechanism of the `apply` function (Listing 1). It is known that *EOS* is the official token issued by the account *eosio.token*, who maintains a table to record the holders and their balances. Thus, to transfer EOS to a DApp, a user has to request the `transfer` function in eosio.token. In step 1 shown in Figure 1, the `code` is assigned the value “eosio.token”, which indicates the account whose smart contract will be invoked; similarly, the `receiver` is also set to “eosio.token”, which represents the receiver of the action (or the notification). After updating the balance table, eosio.token will notify both payer (step 2) and payee (step 3). Note that the `code` in both steps remain unchanged, as the notification will not change the values. However, the `receivers` are set to the corresponding participants, i.e., account user and account dappeg1, respectively. Finally, the processing of incoming notifications depends on the type of the recipient account. To be specific, if it is a smart contract, the notification will be handled by the `transfer` function through the dispatcher (step 4); otherwise, if it is a normal account, the notification will be simply dropped.

Note that functions we studied in this paper can be divided into two categories. The first category includes functions that are declared by the official accounts, e.g., the `transfer` function in eosio.token. The second category consists of those declared and implemented by developers. For instance, to achieve the revealing logic in gambling DApps, developers

[1] The two terms, i.e., the dispatcher and the `apply` function, will be used interchangeably in the following.

[2] EOSIO also provides *deferred action*, which will be executed in a different transaction.

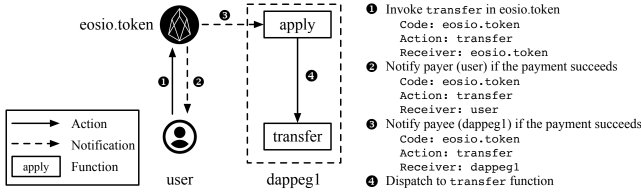

Figure 1: Transferring EOS from account user to account dappeg1 within a single transaction.

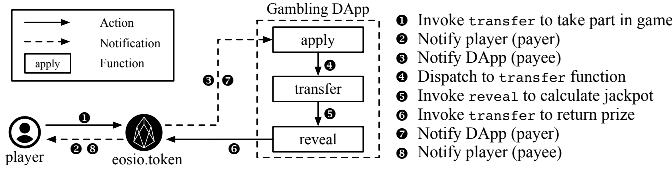

Figure 2: The general life-cycle of smart contract execution. Note that "reveal" here is used to represent the processing logic, while the name can vary in different smart contracts.

could arbitrarily name and implement their `reveal` functions (see Figure 2, which will be detailed in §3).

2.3 Wasm Bytecode and EOS VM

The EOSIO smart contracts are written in C++ and then compiled into Wasm bytecode, which will be executed in the EOS VM. Wasm is a binary instruction format for a stack-based virtual machine. Although it is designed to be an open standard to enable high-performance web applications, it can also be used to support other environments. Due to its efficiency and portability, besides EOSIO, other popular blockchains (e.g., Ethereum 2.0 [25]) are going to support Wasm.

An EOSIO Wasm binary is called a *module*. Inside a module, numerous *sections* exist. Specifically, in the **Function** section, the order of functions is determined, which corresponds to the order of the implementation of functions (in low-level instructions) in the **Code** section. All the indexes of functions that appear in the **Element** section can be treated as entries. Additionally, string literals are often used to initialize the **Memory** section and stored in the **Data** section.

In the EOS VM, all the operands and operators are pushed and popped from a virtual `Stack` as done in the Ethereum VM. However, two more particular structures are used to store data in the EOS VM, i.e., `Local` and `Global`. Specifically, data stored in the `Local` section can only be used inside the scope of the current function, while data stored in the `Global` section can be shared globally across functions. Also, EOS VM has an area called `Memory`, a random-accessible linear array of bytes, which can only be accessed by using specific instructions, e.g., `load` and `store`.

2.4 Threat Model

In this section, we introduce the adversarial threat model of this paper. Specifically, the adversary (attacker) in our study does not require any privileges to launch attacks against EOSIO smart contracts. Namely, any non-privileged account that is capable of interacting with the (up-to-date) deployed EOSIO smart contracts, can be used to launch the attacks. Note that, by default, the adversary can invoke any smart contract deployed by herself to automatically launch the attacks.

3 Vulnerabilities in EOSIO Smart Contracts

A number of attacks targeting the EOSIO ecosystem (including smart contracts) have been observed in the wild, and some of them have been reported [26, 27]. In this paper, we focus on four representative loopholes relevant to EOSIO smart contracts, including *fake EOS* (§3.1), *fake receipt* (§3.2), *rollback* (§3.3) and *missing permission check* (§3.4).

Before delving into the details, we introduce the general life cycle of a smart contract execution to facilitate further discussion. Here we take a gambling DApp as an example, as depicted in Figure 2. Firstly, the player invokes the `transfer` function in eosio.token to take part in the game. Then, on receiving the notification, the DApp will dispatch the request to `transfer` through the dispatcher. After that, `transfer` will call the `reveal` function (Note that "reveal" here is just used to represent the processing logic, and the real function names may be varied in different smart contracts.) to calculate a random number to determine if the player hits the jackpot this round. If it does, the DApp will trigger `transfer` in eosio.token to return the prize to the player. Unfortunately, the attackers can exploit the vulnerabilities in each step to gain profit. For example, in steps 3 and 4, failing to rigorously verify the values of the input parameters could be exploited by attackers. On top of that, this whole betting and revealing process has the potential to be maliciously rolled back.

3.1 Fake EOS

Anyone can create and issue a token called `EOS`, as the token names and symbols are not required to be unique in EOSIO. Therefore, the incorrect verification for `code` at step 3 in Figure 2 may lead to vulnerabilities.

Vulnerability Description. As the source code of *eosio.token* is entirely public, anyone can make a copy of its source code and issue a token with the identical name and symbol. However, due to the difference between issuers, if an attacker transfers the fake EOS to a gambling DApp via the `transfer` function of the copied contract, the `code` of the notification received by the DApp side will not be eosio.token. Thus, if the DApp happens not to check the value of the `code`, then the verification in the dispatcher will be bypassed.

To mitigate the above issue, some developers narrow down the scope of accepted `code`. As shown in line 6 of Listing 1, either "self" or "eosio.token" can be taken as the valid input value of `code`. However, such a mitigation can also be bypassed if the attacker directly calls the `transfer` function. As the condition "code == self" will always be satisfied (see

§2.2), due to the short-circuit evaluation [28] on line 6, the `transfer` function will be invoked even there is no notification from eosio.token, which indicates a transferring request.

As these two cases are only related to fake EOS tokens, in this work, we name both of them as *fake EOS* vulnerabilities.

3.2 Fake Receipt

If the DApp developer performs a comprehensive check against the `code`, the notification will then be forwarded by the dispatcher to `transfer`, as shown in step 4 in Figure 2. However, if the developer does not perform a verification in this step, the DApp can also be attacked.

Vulnerability Description. It is necessary to emphasize that the notification can be forwarded, and the `code` will not change. Therefore, DApp might be deceived by the attacker that plays the dual roles (accounts) of an *initiator* and an *accomplice* at the same time. To be specific, the initiator invokes a regular transfer to an accomplice (indicated by `to`, the argument of the `transfer` function) through eosio.token. When the accomplice is notified by eosio.token, it will immediately forward the notification to DApp without modification. In this way, the `code` is not changed, which is still the official issuer: eosio.token. Therefore, the dispatcher will be unaware of any anomalies. However, if the parameter `to` is not checked in `transfer`, the DApp will be fooled as the token transfer is completed between two accounts controlled by the attacker. It may result in direct financial loss for DApp developers.

As the notification is triggered by `require_recipient`, we name this vulnerability as *fake receipt* .

3.3 Rollback

In Figure 2, `transfer` and `reveal`[3] are the key functions. In the `transfer` function, DApp handles the bet that is received along with the player's transfer; in the `reveal` function, the developer often uses various *on-chain state* as seeds (e.g., `current_time`, indicating the timestamp when the action is executed) to generate a pseudo-random number[4] and finally obtains the result by comparing the generated number with the player's input via the *modulo operation*, which is achieved by the `rem` operator in Wasm bytecode [29]. Note that, in general, the rollback cases can only be found in gambling DApps. We assume it is always there and reachable from the dispatcher.

Vulnerability Description. Even if the developer does a thorough check on every input parameter and checks the caller's permissions before any sensitive actions, a game that matches the model in Figure 2 may still be attacked. To be specific, all the actions are invoked inline (see §2.2), i.e., locating in a single transaction. Therefore, when the player receives the notification after step 8, he could immediately invoke another inlined action to eosio.token to check his balance. If his balance is reduced, which means he did not win this round, he can use an assertion statement to force the current action to fail. We have mentioned in §2.2 that the failure of an action could lead to reverse of the whole transaction. To this end, the player can keep trying until he hits the jackpot. We refer to this malicious rollback as the *rollback* vulnerability.

[3]The "reveal" refers to the semantic meaning as we explained in §2.2

[4]The "pseudo" is due to all these seeds value are deterministic for lack of a true randomness source on blockchain temporarily.

3.4 Missing Permission Check

Before performing any sensitive operation, the developer should check whether the corresponding permission is carried by the action. For example, before step 5 in Figure 2, the DApp should check whether the caller could represent the actual payer to participate in the game.

Vulnerability Description. Permission checking is enforced by `require_auth(acct)` in EOSIO, which is used to check whether the caller has been authorized by `acct` to trigger the corresponding function. Note that the inlined actions inherit the context of their parents, including the permissions (see §2.2). Therefore, if an attacker carrying insufficient permission invokes a function, in which it performs sensitive operations via inlined actions and without permission checking, unexpected behaviors would happen. We regard all the functions lacking of permission checking as the smart contracts with the *missing permission check* vulnerability.

3.5 The Generality of These Vulnerabilities

Note that these four vulnerabilities we studied are general vulnerabilities in EOSIO, rather than application-specific. Firstly, the fake EOS and fake receipt vulnerabilities impact the smart contracts with the verification of transferring requests. All the transferring requests in EOSIO, however, have to be processed by the `transfer` function that is limited by the notification mechanism in EOSIO. According to our statistics (see Table 3 in §7.2), there are 88.32% of deployed contracts using the `transfer` function. In other words, almost 90% of smart contracts can be influenced by these two vulnerabilities. Secondly, though the rollback vulnerability only affects the gambling DApps due to the mechanism they adopt (see §3.3), they constitute the most popular DApp category of EOSIO according to [30, 31]. Therefore, the detection of rollback vulnerability applies to a large portion of existing active DApps. Lastly, the missing permission check vulnerability may impact all the deployed smart contracts without carefully verifying permissions, which may lead to unexpected database modification or leakage of sensitive information. We will further measure the proportion of smart contract may be affected by the vulnerabilities in §7.2.

4 Technical Challenges and Our Solutions

We aim to design and implement a static analysis system to detect vulnerabilities for EOSIO smart contracts. To recover more semantic information, we use heuristic-based symbolic execution to perform in-depth analysis. Namely, semantic

information will be recovered in the constraints generated by symbolically executing the paths being analyzed. Thus, we can identify vulnerabilities with these constraints as patterns.

Comparing with Ethereum Smart Contract Analysis. Although there exists a number of static analysis tools proposed for Ethereum smart contracts, it is worth noting that they cannot be applied directly (or even after minor changes) to EOSIO smart contracts due to the differences between the two platforms, including VM models (e.g., allowing global variables), instructions (e.g., supporting floating-point operations) and system-level data structures (e.g., using *multi-index table* to store persistent data). In brief, these functionalities provided by EOSIO inevitably affect the design/implementation of the proposed system. For instance, we have to consider the side effect imported by external/system libraries (see §4.3 for details). Additionally, the vulnerabilities of EOSIO smart contracts are totally different from those of Ethereum's, which acquire different kinds of context information to support the detection. For example, the rollback vulnerability requires multiple actions being included in one transaction. As such, the detection relies on the propagation of some specific chain state variables (will be discussed in §5.3.4).

Comparing with C language based solution. As mentioned in §1, the Wasm bytecode can be converted to C code by the official tool named wasm2c, which enables the analysis to the corresponding C code by traditional tools like KLEE. However, there are many limitations to this approach. First, it has been reported [12] that the memory model adopted by KLEE [10, 11] may lead to heavy memory consumption and time consumption, which will inevitably affect the performance or even break the analysis. Second, the conversion from the Wasm bytecode to C code requires extra efforts. On the one hand, lots of symbols are missing after the conversion from Wasm to C by wasm2c. We have to manually re-declare all the imported functions, which are necessary for the subsequent analysis of KLEE. On the other hand, in order to get accurate results in vulnerability detection, we need to manually modify the converted C file to hook some functions, as a flag, to perform the vulnerability detection. Both of the above steps must be manually accomplished before analyzing each contract. Last, lots of the memory-checking code is appended by wasm2c[5], which may lead to extra performance overhead (around 85% [33]). To sum up, this solution is not applicable, and we will demonstrate it based on evaluation results in §7.1.

As a result, *no available native symbolic analysis framework could be used to analyze the EOSIO Wasm bytecode directly.* Specifically, we have to overcome several technical challenges to realize the proposed system. On the one hand, it is known that symbolic execution based solutions may suffer from inherent shortcomings, *path explosion* in particular. On the other hand, when applied to vulnerability detection for EOSIO smart contracts, there do exist platform-specific issues, including *memory overlap* and external/system *library dependency*, which will inevitably affect the effectiveness of symbolic execution further.

[5]Although the checking code could be optimized/disabled, however, it is not officially recommended and this issue is still under discussion [32].

4.1 Path Explosion

In EOSIO, the issue of path explosion is mainly due to two circumstances: *executing conditional jump instructions* (such as `br_if`) and *invoking function calls*. Specifically, unlike a normal conditional jump instruction that only generates two new branches, the `br_table` in EOSIO takes an array whose elements are pointers of destination as the argument. As a result, a single `br_table` can lead to n new branches, where n is the length of the array. Moreover, a function call also imposes many new branches to represent all possible callees. Obviously, the number of branches will increase exponentially if there exists a deep call stack. Unfortunately, a concatenation of several deep call stacks is common in EOSIO contracts. Thus, there is a practical need to mitigate this issue.

To this end, we propose a **heuristic-guided pruning approach** to solve the challenge. We rely on several *general* pruning strategies based on our hands-on experience to mitigate the issue derived from branches and deep function calls. For example, our operational observation suggests that discarding paths under a specific depth threshold, which is determined by the scenario, will not influence the precision of results for (almost) all cases. Specifically, we expose two options: `call depth`, which limits the depth of call stack; and `timeout` for users to limit the process of symbolic execution.

However, the effectiveness of the general mitigation strategies are limited in practice. Fortunately, this issue in EOSIO can be further (partially) resolved when performing vulnerability detection, as we only have to pay attention to some specific features/structures of the vulnerable code snippet. For example, when detecting fake EOS and fake receipt vulnerabilities, only `apply` and `transfer` functions are taken into consideration. All these technical details and *vulnerability-specific* pruning strategies will be discussed in §5.3.

4.2 Memory Overlap

The memory area of Wasm can be regarded as a vector of uninterpreted bytes [34], which means users can interpret these raw bits through `load` and `store` with different value types. The EOS VM adopts a *linear array* as its memory model, however, this is memory-consuming for the emulation due to mimicking the sparse memory layout of the EOSIO contract [35]. Therefore, we decide to use *key-value* mappings to emulate the memory, where the key is a tuple to specify the address range, and the value is the data being stored. To better articulate the problem itself and the model, we first define a notion of a memory area $\mathcal{M}$, which is a set of triplets that describe the values in (different) address ranges of the memory. The triplet t has the following form: $t := (l, h, \mathcal{D})$. Here l refers to the inclusive lower-bound, h means the exclusive

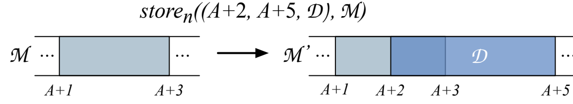

Figure 3: The interval $(A+2, A+3)$ is overlapped after applying $store_n((A+2, A+5, \mathcal{D}), \mathcal{M})$.

upper-bound, and $\mathcal{D}$ represents the data corresponding to the address range restricted by l and h. Moreover, $\mathcal{T}$ is the set of all legal triplets, hence $\forall t \in \mathcal{T}.\ell(\mathcal{D}) = h - l$ always holds.

Based on that, we can define a pair of naive operations $load_n(l, h, \mathcal{M})$ and $store_n(t, \mathcal{M})$ that describe memory accesses. To be specific, $load_n$ will load data from the address range between l and h and return a set that contains the triplets describing the memory contents in that range. $store_n$ will insert the given t into $\mathcal{M}$ and return the updated memory.

However, by this representation it is not guaranteed that the memory contents within a certain address range are defined at most once. Specifically, the overlapped memory interval may lead to ambiguity. For example, as shown in Figure 3, if there has already existed an interval addressed by $(A+1, A+3)$, the operation $store_n((A+2, A+5, \mathcal{D}), \mathcal{M})$ does not consider the relationship between these two intervals. As a result, the data addressed by key $(A+2, A+3)$ in the resulting memory area $\mathcal{M}'$ is ambiguous, meaning that $load_n(A+2, A+3, \mathcal{M}')$ would return the set $\{(A+2, A+3, \mathcal{D}_o), (A+2, A+3, \mathcal{D}_n)\}$ where $\mathcal{D}_o$ is the original data written at memory address $A+2$, and $\mathcal{D}_n$ is the data fraction of $\mathcal{D}$ that was inserted into the memory area by the $store_n$ operation.

The problem is due to the overlapping memory and the improper mapping strategy. Through further analysis, we observe that the memory overlap problem occurs mainly due to the implementation of the *store* instruction. As aforementioned, Wasm provides over 20 memory access related instructions, e.g., `i32.store`, `i64.store`, and `i32.load`. For *store*-related instructions, we can vary the length of $\mathcal{D}$ to make it suitable for any instructions that have different length of data to be stored; for those *load*-related instructions, setting different parameters to guarantee the length of retrieved data is enough. Consequently, we propose an implementation of storing and loading data with the memory area, namely the **memory-merging** method (see §5.1.2), to solve the problem by merging allocated memory. By doing so, we can successfully overcome the challenges raised in Figure 3.

4.3 Library Dependency

To facilitate the development of smart contracts, EOSIO allows the import of external functions as libraries, which means the bodies of these imported functions will not be compiled into Wasm bytecode. EOSIO officially provides plenty of such functions as the system library for DApp developers. They have been widely used in many (if not most) smart contracts. As a result, our analysis will be improperly terminated due to the lack of bodies of those imported function calls.

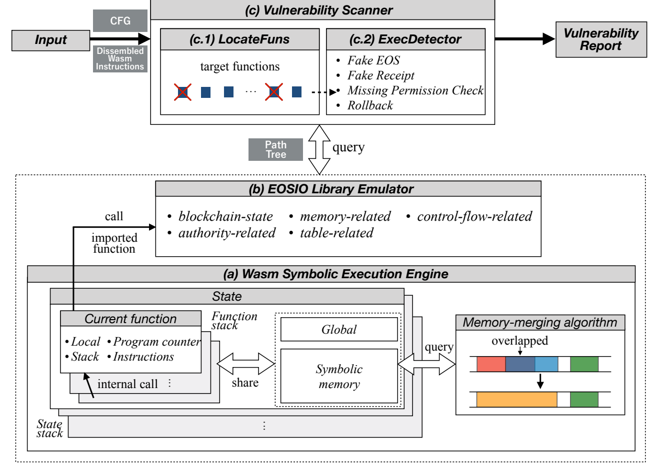

Figure 4: The architecture of EOSAFE.

To resolve the dependency, we propose an **on-demand and semantic-aware** approach (see §5.2) to emulate the imported functions. We only focus on functions whose functionalities and side effects are related to our analysis. We have to emulate such functions properly to guarantee the correctness of the final result. The strength and coverage of the emulation depend on our need to perform the analysis. For some functions, we have to cover the arguments, return value and side effect. For instance, for the memory-related function `memmov`, we need to consider all its side effect on the symbolic memory. For some others, we may only need to consider the possible side effects. For example, for those table-related functions which has no return value and no effect on vulnerability detection, e.g., `db_store_i64`, we can just balance the stack without mimicking its behaviors.

5 System Design

Figure 4 depicts the overall architecture of EOSAFE, which takes the Wasm bytecode of an EOSIO smart contract as the input and eventually determines whether the bytecode is vulnerable. Specifically, EOSAFE is based on *Octopus* [36], a security analysis framework for Wasm modules without supporting symbolic execution. Therefore, to avoid reinventing wheels, each smart contract will be sent to Octopus for building its corresponding Control Flow Graph (CFG) with the disassembled Wasm instructions in preprocessing.

EOSAFE is mainly composed of three modules, i.e., *Wasm Symbolic Execution Engine* (*Engine* for short), *EOSIO Library Emulator* (*Emulator* for short), and *Vulnerability Scanner* (*Scanner* for short). As shown in Figure 4, the input after preprocessing (CFGs) is fed to the Scanner to perform vulnerability detection in a two-step process (*locating suspicious functions* and *detecting vulnerabilities*) with the Engine and Emulator. Specifically, the Engine performs symbolic execution accordingly along with path constraints, which will be used by the Scanner to perform vulnerability detection. Additionally, the Engine requests Emulator to implement the

modeled behaviors when the Engine encounters the call for imported functions. Note that the challenges discussed in §4.1 and §4.2 are addressed in §5.1 and §5.3, while the challenge discussed in §4.3 is addressed in §5.2.

5.1 Wasm Symbolic Execution Engine

We design the symbolic execution engine as a generic framework to simulate the execution of a smart contract on the stack-based EOS VM. It accepts the CFGs and the disassembled Wasm instructions as the input, and symbolically executes instructions within basic blocks in order for all feasible paths. During the process, the path constraints are generated accordingly. This module needs to maintain two crucial components, i.e., *path tree* and *state*, for further analysis. Specifically, the *path tree* is composed of *feasible path*s, which are ***possible*** control flows of the current smart contract. A path would diverge into two paths when encountering some conditional instructions (like `br_table`). To obtain feasible paths, the Engine first relies on the SMT solver to check the path conditions, and then prunes all the infeasible paths that are *unsolvable*. Along each feasible path, we not only record the corresponding constraints , but also all the signatures of invoked imported functions. The maintained path tree significantly contributes to the analysis of vulnerability detection (see §5.3). As to the *state*, we maintain some necessary state-related information, including local/global variables, linear memory, stack, and the subsequent instructions with its corresponding program counter. Specifically, we address the technical challenges mentioned in §4.1 and §4.2 as follows.

5.1.1 General strategies for alleviating path explosion

We provide two options, including `call depth` and `timeout`, for users to mitigate the path explosion issue. On the one hand, the option `call depth` is used to confine the depth of the call stack to prevent the analysis from getting into trouble to deal with complicated branches or deep function calls. As we know, a single function could have several sets of constraints corresponding to feasible paths within the function, which may lead to an exponential growth of the number of paths. Thus we limit the depth of call stack to improve the coverage. On the other hand, we may still be in trouble when encountering some cases that are extremely time-consuming. To guarantee the progress for the whole system, the Engine offers another option named `timeout` to control the maximum execution time for the path-level analysis. The timeout results will be recorded for further investigation. Note that, the path explosion issue will be further addressed in the vulnerability scanner (see § 5.3), as we only have to pay attention to some specific features of the vulnerable code snippets.

5.1.2 Eliminating memory overlap

Recall the definition in §4.2, the key-value pair in the memory $\mathcal{M}$ is defined as a triplet t, i.e., $(l, h, \mathcal{D})$, where $\mathcal{D}$ is a sequence. To be specific, a sequence is a concatenation of elements of

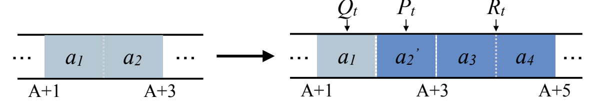

Figure 5: The structure of $\mathcal{M}$ before and after the instruction $store_r((A+2,\ A+5,\ a_2' \| a_3 \| a_4), \mathcal{M})$.

the same size, where the element is represented as a. We should notice that, in $\mathcal{M}$, a is the smallest and indivisible element, whose length is 1. Therefore, we have:

$$\mathcal{D} := a_0 \,\|\, a_1 \,\|\, a_2 \,\|\, ... \,\|\, a_n$$

in which the element can be obtained by its index (i.e., a non-negative integer):

$$\mathcal{D}[i] := a_i, \text{ where } i \in [0,\ n]$$

Moreover, the length of the sequence can be obtained by the ℓ operator (e.g., $\ell(a_0 \,\|\, a_1 \,\|\, a_2) = 3$). We ensure the length of a sequence equals to its corresponding address range, i.e., $\ell(\mathcal{D}) \equiv h - l$.

Based on the above definitions, we can formally define the $load_n$ and $store_n$ operations described in §4.2 as:

$$load_n : \mathbb{N} \times \mathbb{N} \times 2^{\mathcal{T}} \rightarrow 2^{\mathcal{T}} \text{ with}$$

$$load_n(l',h',\mathcal{M}) := \{(l',h',\mathcal{D}') | \exists f.\ (\forall i \in [l',h').\ f(i) \in M \wedge i \in [f(i).l, f(i).h)) \wedge \mathcal{D}' = \big\|_{j=l'}^{h'-1} f(j).\mathcal{D}[j - f(j).l]\}$$

$$store_n : \mathcal{T} \times 2^{\mathcal{T}} \rightarrow 2^{\mathcal{T}} \text{ with } store_n(t',\mathcal{M}) := \mathcal{M} \cup \{t'\}$$

Specifically, $load_n$ will traverse the index space from l' to h', concatenate elements from the memory fragments in $\mathcal{M}$ that overlap with the index space, and finally return the set containing all triplets that describe the different data values that can be read from $\mathcal{M}$ within the index space. Meanwhile, $store_n$ will return the updated $\mathcal{M}$. Here f represents a function that maps an arbitrary index to the corresponding interval in $\mathcal{M}$ from which the data for this index is taken. As discussed in §4.2, there exists a memory overlap problem, i.e., there might be different functions f that map the same index to different intervals. To address this issue, we have to take care of the intervals that are overlapped with the newly incoming ones. Formally, when a new sequence is going to be stored into $\mathcal{M}$, say $(l', h', \mathcal{D}')$, we will first filter out a set I that consists of all the overlapped triplets with $(l', h', \mathcal{D}')$, as follows:

$$I := \{t \mid t \in \mathcal{M} \wedge \exists i \in [t.l, t.h).\ i \in [l', h')\}$$

After that, we will remove all the intervals in I from $\mathcal{M}$. According to the overlapped relationship between them and $(l', h', \mathcal{D}')$, all these intervals will be divided into the following three types: 1) overlapped sub-intervals that need data updating; 2) sub-intervals existing in I but not overlapped by $(l', h', \mathcal{D}')$; and 3) sub-intervals of $(l', h', \mathcal{D}')$ that not conflict with any existing sub-intervals in I. These three types of intervals are constructed by predicates P_t, Q_t, and R_t, respectively.

Finally, the reconstructed set of intervals will be merged into $\mathcal{M}$. Therefore, we define a refined version of the store

operation, called $store_r$:

$$store_r : \mathcal{T} \times 2^{\mathcal{T}} \to 2^{\mathcal{T}} \text{ with}$$
$$store_r(t', \mathcal{M}) := (\mathcal{M} - I) \cup \{t \mid t \in \mathcal{T} \wedge (P_t \vee R_t \vee Q_t)\}$$

Moreover, the predicates are related to the picked out t and detailed in Table 1. Specifically, each one of them is the conjunction of its corresponding two sub-predicates: *Interval Correctness* (denoted as $pred_{IC}$), and *Interval Maximality* (denoted as $pred_{IM}$). $pred_{IC}$ describes an interval with an updated sequence. However, as there may exist multiple intervals that satisfy $pred_{IC}$, we have to define $pred_{IM}$ to enforce the maximality of that interval. To better illustrate the meanings of the predicates and the operation $store_r$, we give a concrete example in Figure 5. Originally,

$$\mathcal{M} = \{(A+1, A+3, a_1 \| a_2)\}$$

Before executing $store_r(A+2, A+5, a_2' \| a_3 \| a_4)$, the set I will be calculated immediately as:

$$I = \{(A+1, A+3, a_1 \| a_2)\}$$

According to the defined formalism of $store_r$, it will construct three triplets as:

$$t_1 = (A+2, A+3, a_2') \text{ satisfies } P_{t_1}$$
$$t_2 = (A+1, A+2, a_1) \text{ satisfies } Q_{t_2}$$
$$t_3 = (A+3, A+5, a_3 \| a_4) \text{ satisfies } R_{t_3}$$

Note that, $(A+3, A+4, a_3)$, $(A+4, A+5, a_4)$ and $(A+3, A+5, a_3 \| a_4)$ all satisfy the $pred_{IC}$ of their corresponding R_t. Meanwhile, the $pred_{IM}$ guarantees that the range $(A+3, A+5)$ will be returned for predicate R_{t_3} instead of the ranges $(A+3, A+4)$, and $(A+4, A+5)$ which would also satisfy $pred_{IC}$. Consequently, $\mathcal{M}$ becomes:

$$(\mathcal{M} - I) \cup \{t_1, t_2, t_3\} \equiv \{t_1, t_2, t_3\}$$

After that, $load_n(A+1, A+4, \mathcal{M})$ will concatenate the necessary parts of elements in $\mathcal{M}$ and finally return a set which contains the single element $(A+1, A+4, a_1 \| a_2' \| a_3)$.

In brief, the $store_r$ (with the original $load_n$) guarantees data consistency by forcing all valid addresses appearing only once in the key space. By doing so, we can solve all the issues raised in Figure 3 effectively.

5.2 EOSIO Library Emulator

We introduce an *on-demand and semantic-aware* approach to resolve EOSIO library dependency. We have manually analyzed the smart contracts of the top 100 popular DApps and existing known vulnerable smart contracts (see §7.1) to extract all the imported functions from their **Function** section (see §2.3). Then, we classify all the imported functions into five categories according to their main functionalities to conduct the emulation. Lastly, we can retrieve the side effects from the emulated imported functions.

The corresponding side effects of these five imported function categories are summarized in the following.

Blockchain-state functions. These functions return constants related to the blockchain system, e.g., `current_time`, which are mostly used by the smart contracts as the seeds, to generate the pseudo-random numbers. As they do not introduce any side effect, we just emulate them by directly returning a symbolic value to represent the blockchain state.

Memory-related functions. As the name suggests, functions in this category are related to the symbolic memory we have implemented. Therefore, we imitate the behaviors as their original intention, and apply the memory-merging algorithm when inserting the new data. Note that, we throw an exception for undefined behaviors, e.g., the negative length of the `memcpy` function due to the constraint solving.

Control flow related functions. These functions may alter or terminate the control flow of a smart contract according to their return results. Therefore, we will fork two paths if necessary. For example, two paths will be generated if the predicate of the `eosio_assert` function is a symbolic value rather than a specific boolean value.

Authority-related functions. As the authority system is merely related to the detection of *missing permission check* vulnerability, we only have to examine the existence of these functions, e.g., `require_auth`, without concerning about the specific permission. Hence, we just return a symbolic value to balance the stack.

Table-related functions. There is a special data structure in EOSIO that allows for persistent storage of data. Similar to the concept of *storage* in Ethereum, this kind of data is saved on the blockchain that is called *table*. *Table* can be regarded as a database that supports *CRUD* operations (i.e., Create, Retrieve, Update and Delete) by several platform-specific instructions. For these functions, we only have to focus on the side effects to the memory rather than the internal operations. Specifically, we have implemented them with return values used to update the memory, as follows:

$$A = \texttt{db_get_i64}(itr, data, length)$$
$$\texttt{i64.store}(base, A)$$

For functions (e.g., `db_update_i64`) that do not have any return value but modify the contents of the *table*, we record their function names and arguments in the constraints.

Note that the focus on the side effect of the library functions is critical for both the symbolic execution engine and the vulnerability scanners in terms of correctness. For instance, a piece of data in the memory area, say $\mathcal{D}$, which will be used later as the branch condition, is overwritten as $\mathcal{D}'$ by invoking `memcpy`. If we do not consider the side effect, namely, taking $\mathcal{D}$ as the branch condition directly instead of $\mathcal{D}'$, it will inevitably affect the accuracy of the further analysis.

5.3 Vulnerability Scanner

To detect multiple vulnerabilities, the Scanner is designed as a generic framework to perform the detection. It mainly consists of two steps, i.e., *locating suspicious functions* and

Table 1: The formal definition of predicates P_t, Q_t, and R_t. Specifically, each one of them is the conjunction of its corresponding two sub-predicates: *Interval Maximality* (i.e., $pred_{IM}$), and *Interval Correctness* (i.e., $pred_{IC}$), e.g., $P_t = P_t^{pred_{IM}} \wedge P_t^{pred_{IC}}$.

	Interval Maximality*	***Interval Correctness***
P_t	$\forall i \in (\mathcal{U} - [t.l, t.h)).\ \forall t' \in I.\ i \notin [t'.l, t'.h) \vee i \notin [l', h')$	$\forall i \in [t.l, t.h).\ (\exists t' \in I.\ i \in [t'.l, t'.h) \wedge i \in [l', h')) \wedge t.\mathcal{D}[i - t.l] = \mathcal{D}'[i - l']$
Q_t	$\forall i \in (\mathcal{U} - [t.l, t.h)).\ \forall t' \in I.\ i \notin [t'.l, t'.h) \vee i \in [l', h')$	$\forall i \in [t.l, t.h).\ \exists t' \in I.\ i \in [t'.l, t'.h) \wedge i \notin [l', h') \wedge t.\mathcal{D}[i - t.l] = t'.\mathcal{D}[i - t'.l]$
R_t	$\forall i \in (\mathcal{U} - [t.l, t.h)).\ \exists t' \in I.\ i \in [t'.l, t'.h) \vee i \notin [l', h')$	$\forall i \in [t.l, t.h).\ (\forall t' \in I.\ i \notin [t'.l, t'.h) \wedge i \in [l', h')) \wedge t.\mathcal{D}[i - t.l] = \mathcal{D}'[i - l']$

* $\mathcal{U}$ refers to the whole legal address space.

detecting vulnerabilities. Accordingly, our goal is to realize detectors for the four vulnerabilities introduced in §3.

The general strategies proposed in §5.1.1 can alleviate the path explosion problem to some extent, however, it is still not enough to meet our needs. Fortunately, one key insight can help further mitigate this issue, i.e., we only have to focus on *valuable* functions that call imported functions with the ability to invoke actions or change the on-chain state, e.g., `send_inline` (see §2.2), `db_update_i64` and `db_store_i64` (see §5.2). These valuable functions are the key targets of our detection. For example, attacking a smart contract that is vulnerable to the rollback vulnerability requires the capability to invoke the `transfer` function. In total, there are 18 functions that can lead to the modification of permanent data [37], and our investigation shows that `send_inline`, `db_update_i64` and `db_store_i64` are the most used ones.

As a result, the two steps of the detection framework can be further transferred and simplified as a *valuable-function-centric* process: 1) *locating valuable functions*; and 2) *verifying their reachability to launch attacks*. Note that the second step of the process is optional since the reachability can always be guaranteed. Based on this framework, we will introduce the details for the four detectors.

5.3.1 Notations

To better explain the logic of detecting vulnerabilities, we first define several symbols here:

- $\mathcal{A}$, the set of names of all the valid accounts in EOSIO;
- $\mathcal{B}$, the set of signatures of all the blockchain-state functions as detailed in §5.2;
- $\mathcal{F}$, the set of signatures of functions that are reachable from the dispatcher;
- $\mathcal{P}$, the set of signatures of functions (18 in total, see §5.3) that can lead to the modification of permanent data;
- $\mathcal{S}$, the set of signatures of invoked imported functions during symbolic execution.

Moreover, as introduced in §5.1, we mainly focus on the constraints and invoked imported functions, which are both recorded in the path tree. Specifically, when the Engine symbolically executes the i-th feasible path of function `func`, we need to verify the existence of certain constraints and invoked imported functions. To this end, we define the following three predicates:

- $Eeq^i_{\texttt{func}}(a, b)$, which indicates the existence of the constraint $a = b$;
- $Eneq^i_{\texttt{func}}(a, b)$, which indicates the existence of the constraint $a \neq b$;
- $Em^i_{\texttt{func}}(sig_{target})$, which indicates there exists a signature in $\mathcal{S}^i_{\texttt{func}}$[6] that string matches the sig_{target}, i.e., $\exists s \in \mathcal{S}^i_{\texttt{func}}.s \sim sig_{target}$[7].

For example, if predicate:

$$Eeq^i_{\texttt{apply}}(action, \texttt{"transfer"})$$

is true, it means that there exists a path constraint of the form $action = \texttt{"transfer"}$ on the i-th path of the apply function, so given that this path is feasible, this means that there is a potential path of apply leading to the `transfer` function. In addition, if predicate:

$$Em^i_{\texttt{apply}}(\texttt{"send_inline(*)"})$$

holds, it indicates that there exists an invocation of the `send_inline` function with arbitrary arguments along the previous path.

5.3.2 Fake EOS detection

As discussed in §3.1 and depicted in Figure 2, the fake EOS vulnerability can only be triggered by invoking the `transfer` function. Moreover, the `transfer` function must be reachable from the dispatcher by attackers, which means there does not exist proper verification of `code` in the dispatcher. Accordingly, the detector traverses all the feasible paths generated by symbolically executing `apply` to examine:

$$Eeq^i_{\texttt{apply}}(action, \texttt{"transfer"}) \wedge \\ \forall a \in (\mathcal{A} - \{self\}).\neg Eeq^i_{\texttt{apply}}(code, a)$$

Specifically, it restricts that only the paths associated with the `transfer` function can be analyzed. To accelerate the analysis, the Engine will terminate irrelevant paths (if the destination is not `transfer`) in advance to avoid further execution. Then, the detector will examine the value in *code*, as discussed in §3.1. Thus, the satisfaction of any of the conditions associated with *code* implies the existence of improper verification. In summary, a smart contract that meets the above conditions is considered to be vulnerable.

5.3.3 Fake receipt detection

The root cause of this vulnerability comes from inadequate verification inside the `transfer` function. Therefore, it is unnecessary to perform symbolic execution from the dispatcher

[6] $\mathcal{S}^i_{\texttt{func}}$ indicates the set of signatures of functions recorded when symbolically executing the i-th path of function `func`.

[7] $\sim$ represents string matching that allows wildcard character $*$.

to the `transfer` function. Instead, it only needs to symbolically execute the `transfer` if we can identify it directly.

To this end, we adopt a heuristic-based method to accelerate the process. Specifically, the detector first identifies the `apply` function, then enumerates all the relevant basic blocks to verify their jump targets whose indices may point to the suspicious `transfer` functions. After locating the suspicious `transfer` functions, the detector will filter out valuable ones according to their corresponding call graphs.

Note that for a given candidate, there exists at most one `transfer` function (like Figure 2), which implies that the `transfer` function is either one of the suspicious functions, or inlined in the `apply` function. For either of the above cases, we would symbolically execute the function that is suspected of being the `transfer` function (indicated by `sus`). Formally, the detection logic will be:

$$Eeq^{i}_{\texttt{sus}}(to, self) \land \exists p \in \mathcal{P}.Em^{i}_{\texttt{sus}}(p) \land \\ Eneq^{j}_{\texttt{sus}}(to, self) \land Em^{j}_{\texttt{sus}}(\text{"}\texttt{eosio_assert()}\text{"})$$

Specifically, we would examine if there exists two paths (i and j) that forked from a point in which it verifies the value of *to*. For the path i that verifies the equality of *to* and *self*, we would further examine if it calls functions that can change blockchain state. For the other path j, which identifies the inequality between *to* and *self*, it will call `eosio_assert` to terminate the current path immediately. *We should pay attention that the above logic means that there does exist a protection against fake receipt vulnerability.* Therefore, if no any two paths satisfy above conditions, then we consider the EOSIO smart contract is vulnerable to the fake receipt vulnerability.

We further apply early termination to accelerate the process. For the valuable `transfer` function, the protection should be verified before updating changes for related on-chain states. Thus, it is reasonable to terminate the current path to investigate the collected constraints when encountering functions like `send_inline`. If there are two paths meeting the protection criteria, the smart contract is immune from the fake receipt vulnerability according to our heuristic strategy and the analysis will be terminated.

5.3.4 Rollback detection

As shown in Figure 2, the `reveal` function[8] often generates random numbers to determine the jackpot winner, and invokes the `transfer` function in eosio.token by an inlined action, i.e., `send_inline`, to return the prize.

In some circumstances, however, the computational burden has to be considered when handling the `reveal` function, as the call depth of the `send_inline` function is too deep for the Engine to reach, which may lead to call depth overflow (similar to the fake receipt detection in §5.3.3).

[8]Note that "reveal" here is just used to represent the processing logic, and the actual names may be varied in different smart contracts.

Fortunately, as it is not necessary to consider the reachability of the `send_inline` function in a path for any target gambling DApp (see §3.3), we are able to apply two strategies to accelerate the process to locate the `reveal` function. Specifically, the first strategy is to traverse feasible paths on demand. Instead of enumerating all paths, we only examine paths that can be used to resolve the data/variable dependency of the target `send_inline` function. The second strategy reduces the size of the *path set* being examined by the Engine after extracting valuable functions, namely, removing redundant paths whose basic blocks are thoroughly the subset of other paths. Consequently, we can achieve the smallest path set to cover as many basic blocks as possible.

Finally, the detection logic is associated with two properties. Firstly, our investigation suggests that the `reveal` function will generate random numbers with the `rem` instruction (see §3.3) along the path inside the constructed path set. Secondly, if the operands of the modulo calculation are (partially) generated by blockchain-state functions (see §5.2), the smart contract will be affected by the rollback vulnerability. In summary, the detection logic must satisfy:

$$Em^{i}_{\texttt{reveal}}(\text{"}\texttt{rem(op_1, *)}\text{"}) \land \exists b \in \mathcal{B}.\ \texttt{op_1} \sim b$$

According to our investigation, here op_2 is always a constant or a variable that has nothing to do with the blockchain state. If the above conditions are met, the smart contract will be labeled vulnerable to rollback vulnerability. Note that we will remove all the `rem` instructions generated by EOSIO official libraries, e.g., eoslib, to reduce the false positives.

5.3.5 Missing permission check detection

As discussed in §3.4, we focus on the functions that are valuable and lacking authority validation before the sensitive operations. Again, such functions should be reachable through the `apply` function. After filtering all the valuable functions by call graph, we would symbolically execute `apply` to filter out all the reachable ones from the dispatcher:

$$Eeq^{i}_{\texttt{apply}}(code, self) \land \exists f \in \mathcal{F}.Eeq^{i}_{\texttt{apply}}(action, f)$$

Then, we would symbolically execute `func` to obtain its path tree with the corresponding constraints. To be specific, for any feasible path j of `func`, if it invokes sensitive functions, e.g., `db_update_i64`, without checking the permission of caller by `require_auth`, i.e.,:

$$Em^{j}_{\texttt{func}}(\text{"}\texttt{db_update_i64(*)}\text{"}) \land \\ \neg Em^{j}_{\texttt{func}}(\text{"}\texttt{require_auth(*)}\text{"})$$

we regard the smart contract as vulnerable to the missing permission check vulnerability.

6 Implementation and Experimental Setup

Implementation We take advantage of Octopus [36] to construct the CFG of Wasm bytecode, and use the Z3 Theorem Prover (version 4.8.6) as our constraint solver to prune infeasible paths. All the other major components, including Engine,

Emulator and Scanner are all designed and implemented by ourselves. The implementation is based in Python, which includes over 5.5k lines of code.

Experimental Setup Our experiment is performed on a server running Debian with four Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz and 64G RAM. As mentioned in §5.1.1, the Engine has provided two configuration options (i.e., *call depth*, and *timeout*) to partially address the path explosion issue. During our experiments, we empirically set the call depth as *2 layers*, as we find it is enough to identify most vulnerabilities. As to the exploration time, we empirically set the upper bound as *5 minutes*, due to the following two main reasons. First, within 5 minutes, all the smart contracts in our benchmark can be fully analyzed and detected with promising results (see §7.1). Second, as we seek to apply EOSAFE to all the EOSIO smart contracts, we have to make a trade-off between accuracy and scalability. Therefore, the exploration time for each contract is set at a maximum of 5 minutes. Note that all these settings could be easily configured and customized in our tool, to fulfill the different requirements.

To compared with the C language based solution discussed in §4, we also setup the KLEE environment to perform the evaluation. Specifically, for *wasm2c*, we adopted the latest version in the main branch[9] as it is still under development; meanwhile, for KLEE, we choose the latest stable version (KLEE:2.1[10]) released within the official docker image. Initially, we set the exploration time as *5 minutes*, the same with EOSAFE. Unfortunately, as almost all the tasks could not be completed due to the 5-minute timeout, we decided to give it another trial and increase the exploration time to *30 minutes*, which shall lead to a better performance. Beyond that, we left the remaining configuration items unchanged and performed subsequent experiments on the same server used by EOSAFE.

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

RQ1 *How accurate is* EOSAFE *in detecting vulnerabilities in EOSIO smart contracts?*

RQ2 *Are these vulnerabilities prevalent in the ecosystem?*

RQ3 *How many smart contracts have been exploited by attackers and what are the impacts of these attacks?*

To answer RQ1, in the absence of established benchmarks in the research community, we propose to collect real-world attacks and manually examine the victim smart contracts to craft a reliable benchmark. To answer RQ2, we collect all the available smart contracts on EOSIO and their historical versions. Then we apply EOSAFE to detect the presence of security vulnerabilities, and characterize the evolution of vulnerabilities. To answer RQ3, we further collect all the on-chain transactions related to the flagged vulnerable contracts, and then propose heuristics to pinpoint possible attacks.

[9]The hash is `be5e8bf8ec698f9ad3a1b6fbb412680995fe39bf`.

[10]The sha256 digest is `33a568ccee52efc1fbcce4fb33bab476ce666bef2fa3e628627881bdd70c9d0f8`.

7 Experimental Results

7.1 RQ1: Accuracy of Vulnerability Detection

Creating the Benchmark. To evaluate EOSAFE, we first make efforts to craft a benchmark. EOSIO attacks were observed and reported ad hoc from time to time. Thus, we resort to the security reports released by well-known blockchain security companies to collect all the related publicly verified attacks [26, 27] as the ground-truth. We have collected 38 attacks, targeting 34 unique vulnerable smart contracts in total. Although these attacks were confirmed by the official team of the corresponding DApps, we found that some attacks are irrelevant to the smart contract itself but concern other external factors, e.g., the server's issues [38]. Thus, we further manually examined all the involved smart contracts. Specifically, we found that 3 out of the 10 fake EOS attacks are related to server issues (e.g., [38]). For rollback, 11 out of 21 attacks are due to the wrong reveal strategy of the server (e.g., [39]). Besides, 2 of them were variants of rollback, which are related to the configuration of some nodes on EOS MainNet (see [40]). At last, we excluded all the above contracts to make sure all the attacks are resulted from the vulnerability in smart contracts. The benchmark can be accessed at [23].

Overview of the Benchmark. The distribution of the benchmark is shown in Table 2. Note that we also collected the corresponding patched smart contracts (without vulnerabilities) as comparison to evaluate the effectiveness of EOSAFE. Additionally, only two vulnerable smart contracts related to the missing permission check vulnerability were reported, and neither of them has been patched yet. Thus, we further manually created 4 pairs of smart contracts (with and without such vulnerabilities) to complement our benchmark. At last, we have included 52 smart contracts in our benchmark. As the benchmark is small-scale, which may not be sufficient to comprehensively evaluate the effectiveness of EOSAFE, we will further perform a manual investigation to verify the detection results in the wild (see §7.2.1).

Results. Among the 52 smart contracts, EOSAFE flagged 26 as vulnerable, with only one false negative case (belongs to rollback) and no false positives, leading to *precision and recall of 100% and 96.30%*, respectively. Table 2 shows the detailed results. For the only false negative case of rollback, i.e., `fairdogegame/betdogewallt`, as the number of suspicious `reveal` functions is too high to build paths, it is difficult to symbolically execute each of them for a given `timeout` (*5 minutes* here). After manually locating the vulnerable function, i.e., `func73`, we can get a correct result. Therefore, the false negative is introduced by the optimization strategies, which is a trade-off between accuracy and scalability. It is easy to tune our approach to cover it, e.g., by exploring more paths and increasing the analyzing time. Nevertheless, the exceptional case is rarely seen during experiments, as most smart contracts are not too complicated to handle.

Comparison with KLEE. To enforce fair comparison, we

Table 2: A Comparison of EOSAFE and KLEE on the benchmark. (TP – True Positive, FP – False Positive, TN – True Negative, FN – False Negative, SR – Success Rate, P – Precision, R – Recall, F1 – F1 Measure)

Type	# Samples(Vul/Non-Vul)	EOSAFE								KLEE							
		TP	FP	TN	FN	SR**	P	R	F1	TP	FP	TN	FN	SR**	P	R	F1
Fake EOS	14 (7/7)	7	0	7	0	100.00%	100.00%	100.00%	100.00%	5	0	7	2	50.00%	100.00%	71.43%	83.33%
Fake Receipt	10 (5/5)	5	0	5	0	100.00%	100.00%	100.00%	100.00%	0	0	5	5	0.00%	-	-	-
Rollback	18 (9/9)	8	0	9	1	94.44%	100.00%	88.89%	94.12%	0	0	9	9	0.00%	-	-	-
Permission	10 (6/4)*	6	0	4	0	100.00%	100.00%	100.00%	100.00%	5	0	4	1	90.00%	100.00%	83.33%	90.91%
Total	**52 (27/25)**	**26**	**0**	**25**	**1**	**98.08%**	**100.00%**	**96.30%**	**98.11%**	**10**	**0**	**25**	**17**	**30.77%**	**100.00%**	**37.04%**	**54.05%**

* 4 pairs of the missing permission check samples are manually crafted. ** Timeout or memory error caused path explosion will be regarded as failed cases.

apply KLEE to the same benchmark to evaluate its overall performance. The detailed results are shown in Table 2. Note that we conservatively treat those contracts that are failed in analysis but bug-free as true negatives (TN), which consequently results in 100.00%, 37.04%, and 54.05% of precision, recall and f1-measure, respectively. To be specific, for those 27 de facto vulnerable contracts and 25 non-vulnerable ones, KLEE can only successfully identify 10 and 6 of them, respectively. Interestingly, all these 16 cases are related to either fake EOS vulnerability or missing permission check vulnerability. For the other 36 cases (including all the contracts under the categories of fake receipt and rollback vulnerabilities), KLEE failed to analyze them. In other words, the results are all timeout (under 30 minutes) or OOM. After an in-depth investigation, we ascribed the failure to the massive number of jump and call instructions. Entering from the dispatcher without optimization makes it difficult to complete the analysis under the required time and limited memory.

7.2 RQ2: Prevalence of Vulnerabilities

Dataset. We consider all the 53,666 smart contracts (including history versions) from June 9, 2018 (the very beginning of EOS MainNet) to November 15, 2019. Note that different from Ethereum smart contracts that cannot be modified once deployed, EOSIO contracts could be updated and bind with the same account as explained in §2.1. Thus, we use the *EOSIO account* to label each *unique smart contract*, i.e., one account may correspond to multiple *contract versions*. As a result, we have 53,666 different versions of contracts, belonging to 5,574 EOSIO accounts. As the rollback vulnerability is only related to the gambling DApps, we can shrink our candidate list here. We refer to DAppTotal [31] – a credible multi-platform DApp browser, to label the gambling DApps and use such contracts (17,394) for rollback vulnerability detection. Moreover, for both the fake EOS and the fake receipt vulnerabilities that only link to the `transfer` functions, we identified the candidates, i.e., EOSIO smart contracts with `transfer` functions. Specifically, 47,396 versions of contracts and 4,678 unique ones are extracted. For the missing permission check vulnerability, we apply the detector to all the 53,666 contracts (see Table 3).

7.2.1 Overall results

Table 3 shows the overall results. Surprisingly, over 25% of the 53,666 smart contracts are labeled vulnerable (see Column 3). The missing permission check vulnerability is the most prevalent, affecting over 15% of the smart contracts. The fake receipt vulnerability is also quite common (13%). For the rollback vulnerability, although we only analyzed 17K smart contracts of gambling DApps, over 1,000 of them are labeled vulnerable. The fake EOS vulnerability affects roughly 2.7% of the smart contracts. *It suggests that security vulnerabilities are prevalent in EOSIO smart contracts, revealing the urgency to identify and prevent such vulnerabilities.*

Table 3: Vulnerability detection results in the wild.

Type	# Candidates	# Vulnerable (%*)	# Unique	# Vulnerable (%*)
Fake EOS	47,396	1,457 (2.71%)	4,678	272 (4.88%)
Fake Receipt	47,396	7,143 (13.31%)	4,678	2,192 (39.33%)
Rollback	17,394	1,149 (2.14%)	913	84 (1.51%)
Permission	53,666	8,373 (15.60%)	5,574	662 (11.88%)
Total	**53,666**	**13,752 (25.63%)**	**5,574**	**2,759 (49.50%)**

*The percent is calculated based on all the EOSIO smart contracts with their versions.

Vulnerable Unique Smart Contracts. As one smart contract may correspond to multiple versions, we further characterize the distribution of vulnerabilities from the perspective of unique contracts (accounts). As shown in Column 5 of Table 3, for the 5,574 unique contracts, roughly half of them have at least one vulnerable version. 10% of unique smart contracts account for 61.24% of vulnerable versions, which indicates *most of vulnerable versions are imported by a small portion of smart contracts.* Besides, there are 1,793 unique smart contracts, whose versions are all vulnerable (41% of them have at least two versions). The contract `eossanguoone`, which is a popular game DApp, has the most number of vulnerable versions (356 versions). By manual inspection, we found that all its versions released before Sep. 4th, 2019 have suffered from the fake receipt vulnerability, and then it was patched by the developer. The missing permission check vulnerability has been found since Aug. 2019, which may be due to the import of the new functions without authority check.

Manual Verification To further verify the veracity of the results, we manually sampled some contracts labeled by EOSAFE. Specifically, we randomly sampled 10 labeled vulnerable contracts and 10 labeled bug-free contracts for each type of vulnerabilities. For these collected 80 samples, we manually reverse-engineered all of them to verify the labeling results[11]. The results show that, there exists only *one* false negative case which cannot be successfully detected as the

[11] The verification is a time-consuming process, and it took the first two authors three whole days to analyze them. These samples are also attached into the benchmark at [23].

Table 4: The time to fix the vulnerabilities.

Type	# Unique (Vul)	# Latest with Vul (%)	# Patched (%)	Patch Time
Fake EOS	272	207 (76.10%)	65 (23.90%)	14.85d
Fake Receipt	2,192	1,735 (79.15%)	457 (20.85%)	24.01d
Rollback	84	28 (33.33%)	56 (66.67%)	4.24d
Permission	662	313 (47.28%)	349 (53.72%)	4.38d*
Total	**2,759**	**2,080 (75.39%)**	**679 (24.61%)**	**16.84d**

*The average patch time for missing permission check is calculated on the action level.

rollback vulnerability[12], while all the other *79* ones are correct. We then conducted an in-depth analysis to understand that failed case. The investigation showed that the constructed path (see §5.3.4) had indeed reached the target `reveal` function. The failure, similar to the false negative case mentioned in §7.1, was due to the extreme complicated control flow inside the function. However, after adopting the same method in §7.1, i.e., manually feeding the `reveal` function into the scanner, it was still timeout even after 30 minutes. As such, due to the conservative strategy, EOSAFE mislabeled this contract as safe to produce the false negative. In nutshell, *the result is inline with our evaluation on the benchmark*.

7.2.2 Time to fix the vulnerability

We next investigate the *time to fix the vulnerabilities* for each smart contract, which could be used to measure the *window period* for the attackers to exploit these vulnerabilities.

Result. As shown in Table 4, for the 2,759 unique smart contracts with vulnerable versions, over 75% of them still have at least one security vulnerability in their latest version by the time of our study. 679 unique smart contracts have patched all their vulnerabilities during their evolution, and the average window period is 16.84 days.

Patch Rate. We further analyze the patch rate across vulnerabilities. The rollback vulnerability has the highest patch rate (over 66%), and the average window period is roughly 4 days. The reason for its timely response might be that the rollback vulnerability only exists in game/gambling DApps, which usually have high balance in their accounts. The financial loss could be devastating if developers leave the vulnerability alone. For the missing permission check, 349 smart contracts have patched all their missing check actions. Note that we measured the average patch time on the action level here, as one vulnerable contract may have more than one missing permission check actions. There are 647 patched actions in total – roughly 500 of them are patched within only one day, while the overall patch time is 4.38 days. It suggests that most of the missing permission checking actions are patched timely, while a few contracts take relative long time to fix. In contrast, the fake EOS and the fake receipt vulnerabilities have the lowest patch rates (i.e., roughly 20%), and the patching time is relative long (i.e., 2 to 3 weeks on average). Our manual check found that, half of the smart contracts related to fake receipt are patched within 24 hours, which further indicates that some *inactive* smart contracts drag the average patch time. Most of the inactive smart contracts (accounts) have no balance and very few transactions, which are usually not the targets of attackers.

[12]Named as `eospindealer`, deployed at 2018-12-28 03:14:10

7.3 RQ3: The Presence of Attacks

7.3.1 Approach

It is non-trivial to explore how many of the vulnerable smart contracts have been successfully exploited. Until recently, a lot of ad hoc (often manual) efforts of security researchers [26, 27] are necessary to verify them. Thus, we first collected all the *on-chain* transactions including the ones of labeled non-vulnerable contracts, and then designed a set of heuristics to locate the suspicious attacks, which will be used to facilitate further manual verification to determine the real attacks. In total, we have collected over *2.5 billion* transaction records.

Fake EOS Attack. The most important behavior of this attack is to defraud the *official EOS tokens* from the vulnerable smart contract by using the *fake EOS tokens*, which can be identified through the transaction records storing the information of token issuers. According to the observation, we will first filter out all the transactions of token transfer whose token symbols are “EOS”. Then, these transactions will be grouped according to the following definitions:

- *fake-sending* transactions that send fake EOS tokens.
- *true-sending* transactions that send true EOS tokens.
- *true-receiving* transactions that receive true EOS tokens.

As a result, we can define a *potential* attack as a sequence of a fake-sending transaction followed by a true-receiving transaction. Note that a fake-sending transaction A can be joined with a true-receiving transaction B, if and only if they appear on the same period while A occurs before B. For all these potential transactions, we focus mainly on those who have gained more true EOS tokens than they spent. To this end, we further examine the input-output ratio between the attacker and the vulnerable contracts to determine the *suspicious* attacks. Finally, based on the suspicious attacks, we will verify whether the vulnerable smart contracts will resume the normal execution (e.g., running a lottery for a real player) after receiving the fake EOS tokens. If so, we will mark the suspicious transaction as a fake EOS attack.

Fake Receipt Attack. The key feature of this attack is that the vulnerable smart contract is misled by the fake notification to receive tokens, while the actual token transfer occurs between the two accounts belonging to the same attacker (see §3.2). For simplicity, we will use *from_account* and *to_account* to represent the two accounts in the following, where to_account will send the fake receipt to vulnerable contract, and from_account is the ultimate beneficiary.

Accordingly, we will first query all the transactions of token transfer whose tokens are issued by `eosio.token` and token symbols are “EOS”, to get all the true EOS token transfers. Then, we will filter out the transactions whose receivers are neither `eosio.token`, nor the from_account or to_account. These transactions will be regarded as the fake receipts with crafted notifications. Next, if a from_account sends a fake

Table 5: Overall results of attack detection.

Type	# Attacks	# Attackers / Victims	Financial Loss ($)	# Verified
Fake EOS	9	10 / 9	652,428.48	8
Fake Receipt	27	28 / 17	1,020,831.94	7
Rollback	12	12 / 9	52,984.00	12
Permission	183	- / 144	-	-
Total	**48***	**50 / 34***	**1,726,244.42**	**27**

* Exclude the results of missing permission check.

receipt before making profits from the vulnerable contract, we will mark the corresponding transaction as potential. After that, by eliminating the unrelated EOS spending transactions (e.g., for testing purpose initiated by the attacker), we focus mainly on those who have gained more true EOS tokens than they spent. If the input-output ratio are still high, the corresponding transactions are labeled as *suspicious*.

Finally, we will manually check the suspicious transactions whether the vulnerable smart contract will resume the normal execution after receiving the fake receipts. If so, we will mark such a transaction as a fake receipt attack.

Rollback Attack. As mentioned in §3.3, the transaction of this attack is composed of sequential invocations of actions, which can be used as the pattern to identify the attack.

Specifically, we will first filter out all the transactions which contain at least four actions as the potential transactions. Among them, we will select suspicious ones that meet the following four conditions: (1) the first and the last actions must be invoked in the same contract, where the first means to start the attack, and the last will determine whether the rollback is necessary after receiving the reward from the vulnerable smart contract. (2) the two actions in the middle must be token transfers through `eosio.token`, and the sender and the receiver (either one must be the vulnerable smart contract) of the two actions are arranged opposite to each other. (3) at least one of the counterparties, i.e., either the sender or the receiver, is labeled as the gambling or game DApp. (4) the amount of tokens transferred from the vulnerable smart contract is more than it received. Besides, it is worth noting that, the rollbacked transactions will not be recorded on the chain. As a result, we have to manually check the player's successful rate per unit time, namely, if it is oddly high than the others, we will mark the suspicious transaction as a rollback attack.

Missing Permission Check Attack. Because authority information is along with the invoked transaction, we can examine whether it belongs to the callee contract to identify this attack. More precisely, we will first screen out all the transactions whose target actions are the vulnerable actions, to get suspicious transactions. Then, if the transaction's authority does not belong to that smart contract the action belongs to, we will mark it as a missing permission check attack.

7.3.2 Results

The overall result is shown in Table 5. We have identified 48 attacks in total, including 9 fake EOS attacks, 27 fake receipt attacks, and 12 rollback attacks. Note that 35 attacks among them were first discovered by our approach. Additionally, we also identified 183 invoked actions (belonging to 144 contracts) which missed the permission checking. Note that for these missing permission check actions, some of them are designed intentionally instead of unexpected implementation. It is hard to differentiate whether they are attacks or not, and it is impossible to estimate the financial loss. Therefore, we regard them as misuse actions instead of attacks.

Table 6: Top 5 identified attack events.

Type	Attacker Account(s)	Victim Account	Financial Loss (EOS/$)
Fake Receipt	il***23 wh***r1	eosbetdice11	138K/757K
Fake EOS	re***et	eoscastdmgb1	63K/328K
Fake Receipt	re***om re***et	nkpaymentcap	54K/201K
Fake EOS	aa***fg	eosbetdice11	44K/234K
Fake Receipt	be***s1 be***s2	epsdcclassic	17K/42K
Total	-	-	**341K/1,639K**

Impact of Attacks. *The 48 identified attacks lead to over 341K EOS loss, which is roughly 1.7M USD according the close price of the date of attacks.* Note that we have collaborated with a leading blockchain security company to report these attacks to the DApp developers, and 27 of them have been confirmed, accounting for more than 99% of the total loss. All the unconfirmed suspicious attack events only relate to a few EOS, and most of them are no longer active. The Top-5 confirmed attack events are listed in the Table 6.

Unexploited Vulnerable Contracts. It is interesting to observe that, although thousands of contracts are vulnerable (see Table 3), only a few of them have been successfully exploited by attackers in the wild. Thus, we have manually sampled 40 labeled vulnerable smart contracts (10 for each vulnerability), for reverse engineering and inspecting their transactions and balances. We observe two major reasons leading to this. First, the popular smart contracts (with high balances) were the main targets of attackers, but these vulnerable contracts were patched in time according to the results from §7.2.2, which left a very short window for attackers. Based on the transaction data, we observed that attackers were always trying to exploit the popular contracts. Although some attacks were successful (see Table 6), most of them failed. Second, most of the unpatched smart contracts were inactive with low balances. As a result, it was hard, if not impossible, for them to attract attackers, who must have considered the trade-off between the low profits and the costs of attacks.

8 Threats to Validity

First, *our system inherits the limitation of symbolic execution*, i.e., path explosion. Although we have implemented several optimization strategies, EOSAFE still reports false negative cases, as discussed in §7.1. However, we believe this is not a big issue for our system. On the one hand, most of the smart contracts are not as complicate as other software. A large portion of smart contracts can be fully analyzed in a short time. On the other hand, we have proposed specific

optimization methods when searching for the vulnerabilities, which could eliminate most irrelevant paths. Nevertheless, we can further take advantage of advanced symbolic execution techniques [9, 41–45] to alleviate this issue.

Second, *we rely on heuristics and semi-automated methods to verify attacks* (see §7.3). This, of course, might not be scalable and could mean that we only offer a coarse estimate of the attacks. However, a large portion of the attacks we identified are confirmed by DApp teams, which suggests that our approach is quite reliable. Nevertheless, some other techniques (e.g., dynamic testing) can be applied to help us automatically identify attacks. In this paper, our main contribution is automatically detecting the security vulnerabilities, while attack verification is not a main focus in this work.

Third, *there might exist some new vulnerabilities we did not cover in this current prototype, as well as the general vulnerabilities in other software systems*, such as buffer overflow. In this paper, we focus only on the EOSIO-specific vulnerabilities, the main reason is that we are lacking ground-truth for other security bugs. Nevertheless, we have tried our best to minimize the burden for further development efforts. Specifically, we have adopted a modular design scheme, hence the Engine and the Emulator can be treated as black boxes and used directly. Moreover, the pruning strategy in Engine is generic rather than vulnerability-specific. However, building vulnerability scanner always requires prerequisite domain knowledge for any security analyst. Finally, EOSAFE can also work on the Wasm bytecode from other platforms (e.g., web), where the only extra effort is to resolve the library dependency for the corresponding platform.

9 Related Work

WebAssembly Bytecode Analysis WebAssembly is the new low-level language for the web. There are only a handful work on analyzing the Wasm bytecode [46–50]. For example, Lehmann et al. [48] has proposed a general-purpose dynamic analysis system for Wasm, which allows developers or researchers to implement heavyweight dynamic analysis, e.g., instruction counting and memory access tracing. However, all of them were focused on web applications, which were mainly dynamic analysis. In this paper, we implemented a general symbolic execution framework for Wasm, and made effort to support the security analysis of EOSIO smart contracts.

EOSIO Analysis There are several work focused on the EOSIO [51–53]. For example, Huang et al. [52] proposed to identify the bot-like accounts in EOSIO based on transaction analysis. Lee et al. [53] introduced and studied four attacks stemming from the unique design of EOSIO. Several technical blogs [6, 7, 38, 40, 54] from the industry have reported the security attacks of EOSIO. However, there are no available work on detecting the security vulnerabilities in EOSIO.

Vulnerability Detection of Ethereum Smart Contracts Ethereum has received lots of attention from academia, and a number of studies were focused on vulnerability detection [14–19, 55–57]. For example, [16] was mainly focused on the overflow vulnerabilities. Luu et al. [18] proposed Oyente, the first symbolic execution tool for detecting vulnerabilities in Ethereum smart contracts. Machine learning and fuzz testing techniques [55] were also adopted to identify the vulnerabilities in Ethereum smart contracts. As we mentioned earlier, *the two ecosystems (Ethereum and EOSIO) are totally different, and no previous work on Ethereum can be applied to analyze EOSIO smart contracts directly.* Nevertheless, we admit that the general idea of Ethereum vulnerability detection can be incorporated to improve our work.

10 Conclusion

To the best of our knowledge, this paper presents the first work on detecting security vulnerabilities in EOSIO smart contracts. We propose EOSAFE, an accurate and scalable framework based on a well designed native Wasm symbolic execution engine. Experiment results suggest the promising performance of EOSAFE. Our large-scale measurement study further reveals serious security issues in the ecosystem, i.e., over 25% of the smart contracts are vulnerable and a number of high-profile attacks have been successfully carried out.

Acknowledgement

We would like to thank our shepherd Clara Schneidewind and all anonymous reviewers for their helpful suggestions and comments to improve the paper. This work was supported by the National Key Research and Development Program (2016YFB1000105), National NSF of China (62072046, 61772042), the Fundamental Research Funds for the Central Universities (No. 2020QNA5019), Hong Kong RGC Projects (No. 152193/19E, 152223/20E), QNRF grant QNRF-AICC01-1228-170004 from Qatar National Research Fund (a member of Qatar Foundation). The findings herein reflect the work, and are solely the responsibility of the authors.

References

[1] QuantumMechanic, "The proposal of PoS," Jul. 2011. [Online]. Available: https://bitcointalk.org/index.php?topic=27787.0

[2] "The DPoS consensus," Jan. 2020. [Online]. Available: https://en.bitcoinwiki.org/wiki/DPoS

[3] CRAIG RUSSO, "EOSIO surpasses Ethereum in transaction volume," Sep. 2018. [Online]. Available: https://sludgefeed.com/eos-surpasses-ethereum-in-daily-dapp-users-and-transaction-volume/

[4] Alfredo de Candia, "Increase of EOSIO transaction volumes," Sep. 2019. [Online]. Available: https://en.cryptonomist.ch/2019/09/03/eos-porn-transaction-volumes/

[5] Mozilla, "Basic concepts for Wasm." [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly

[6] EOS, "EOSBet was attacked by Fake EOS vulnerability," Sep. 2018. [Online]. Available: https://www.reddit.com/r/eos/comments/9fpcik/how_eosbet_attacked_by_aabbccddeefg/

[7] PeckShield Inc., "EOSBet was attacked by Fake Recipt." Oct. 2018. [Online]. Available: https://blog.peckshield.com/2018/10/26/eos/

[8] WebAssembly, "Project home of wasm2c," Oct. 2020. [Online]. Available: https://github.com/WebAssembly/wabt/tree/master/wasm2c

[9] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[10] "Memory model of KLEE," Apr. 2012. [Online]. Available: http://formalverification.cs.utah.edu/gklee_doxy/overview.html

[11] J. Novák, "Improvements of memory management in klee."

[12] "Performance issue for klee." [Online]. Available: https://stackoverflow.com/questions/5742618/limits-of-klee-the-llvm-program-analysis-tool

[13] WebAssembly, "Open issues for wasm2c," Oct. 2020. [Online]. Available: https://github.com/WebAssembly/wabt/issues

[14] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[15] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *USENIX Security 18*, 2018, pp. 1317–1333.

[16] 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.

[17] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *CSF*. IEEE, 2018, pp. 204–217.

[18] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *CCS*. ACM, 2016, pp. 254–269.

[19] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 654–675.

[20] A. Rossberg, "WebAssembly Specification," Feb. 2020. [Online]. Available: https://webassembly.github.io/spec/core/index.html

[21] "Supported opcodes in EVM," Oct. 2020. [Online]. Available: https://github.com/crytic/evm-opcodes

[22] "Type conversion of Solidity," 2019. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.3/types.html#conversions-between-elementary-types

[23] N. He, "EOSafe benchmark," Feb. 2021. [Online]. Available: https://github.com/HNYuuu/EOSafe-benchmark

[24] EOSIO, "EOSIO official site," 2019. [Online]. Available: https://eos.io/

[25] "Ethereum WebAssembly." [Online]. Available: https://ewasm.readthedocs.io/en/mkdocs/

[26] PeckShield Inc., "Blogs about blockchain security events," 2020. [Online]. Available: https://blog.peckshield.com/blog.html

[27] SlowMist Zone, "Blockchain security events," 2020. [Online]. Available: https://hacked.slowmist.io/en/

[28] "Short-circuit mechanism," May. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Short-circuit_evaluation

[29] "Modulo operation in wasm," Oct. 2020. [Online]. Available: https://github.com/sunfishcode/wasm-reference-manual/blob/master/WebAssembly.md#integer-remainder-signed

[30] "DappRadar, a DApp browser," Oct. 2020. [Online]. Available: https://dappradar.com/

[31] "DAppTotal," Nov. 2019. [Online]. Available: https://dapptotal.com/

[32] "Discussion about bounds checks in wasm2c," Oct. 2020. [Online]. Available: https://github.com/WebAssembly/wabt/pull/1432

[33] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: Webassembly as a practical path to library sandboxing," *arXiv preprint arXiv:1912.02285*, 2019.

[34] WebAssembly Community Group, "Wasm memory module," 2017.

[35] "Memory layout of eos vm," Oct. 2019. [Online]. Available: https://github.com/EOSIO/eos-vm/blob/master/README.md

[36] QuoScient, "Octopus," GitHub repository, Nov. 2019. [Online]. Available: https://github.com/quoscient/octopus

[37] "Functions can modify data in table," Sep. 2019. [Online]. Available: https://github.com/EOSIO/eosio.cdt/blob/master/libraries/eosiolib/contracts/eosio/multi_index.hpp

[38] David Canellis, "Newdex was attacked by Fake EOS," Sep. 2018. [Online]. Available: https://thenextweb.com/hardfork/2018/09/18/eos-hackers-exchange-fake/

[39] SlowMist, "Rollback attack for betdiceadmin," Jun. 2019. [Online]. Available: https://github.com/slowmist/eos-smart-contract-security-best-practices/blob/master/README_EN.md#random-number-practice

[40] ——, "Roll Back Attack about blacklist in EOSIO," Jan. 2019. [Online]. Available: https://medium.com/@slowmist/roll-back-attack-about-blacklist-in-eos-adf53edd8d69

[41] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[42] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 443–446.

[43] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of data-parallel floating-point code," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 710–737, 2014.

[44] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *FSE*, 2015, pp. 842–853.

[45] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *ICSE*, 2018, pp. 350–360.

[46] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," *arXiv preprint arXiv:1807.08349*, 2018.

[47] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," *arXiv preprint arXiv:1802.01050*, 2018.

[48] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *ASPLOS*. ACM, 2019, pp. 1045–1058.

[49] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position paper: Progressive memory safety for webassembly," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019, pp. 1–8.

[50] M. Vassena and M. Patrignani, "Memory safety preservation for webassembly," *arXiv preprint arXiv:1910.09586*, 2019.

[51] L. Bach, B. Mihaljevic, and M. Zagar, "Comparative analysis of blockchain consensus algorithms," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, pp. 1545–1550.

[52] Y. Huang, H. Wang, L. Wu, G. Tyson, X. Luo, R. Zhang, X. Liu, G. Huang, and X. Jiang, "Understanding (mis) behavior on the eosio blockchain," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 2, pp. 1–28, 2020.

[53] S. Lee, D. Kim, D. Kim, S. Son, and Y. Kim, "Who spent my EOS? on the (in) security of resource management of eos. io," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[54] PeckShield Inc., "EOSCast was attacked by Fake EOS." Nov. 2018. [Online]. Available: https://blog.peckshield.com/2018/11/02/eos/

[55] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, 2020.

[56] R. Ji, N. He, L. Wu, H. Wang, G. Bai, and Y. Guo, "Deposafe: Demystifying the fake deposit vulnerability in ethereum smart contracts," *arXiv preprint arXiv:2006.06419*, 2020.

[57] B. Gao, H. Wang, P. Xia, S. Wu, Y. Zhou, X. Luo, and G. Tyson, "Tracking counterfeit cryptocurrency end-to-end," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–28, 2020.