EOSIOAnalyzer: An Effective Static Analysis Vulnerability Detection Framework for EOSIO Smart Contracts

Wenyuan Li† [1,2], Jiahao He†[3], Gansen Zhao* [1,2], Jinji Yang* [1,2], Shuangyin Li[1,2], Ruilin Lai[1,2], Ping Li[1,2], Hua Tang[1,2], Haoyu Luo[1,2] and Ziheng Zhou[4]

[1]*School of Computer Science, South China Normal University*
[2]*Key Lab on Cloud Security and Assessment technology of Guangzhou*
[3]*WeBank Co., Ltd, Block A Building 7 Shenzhenwan Keji Eco Park, Shenzhen, China*
[4]*VeChain Foundation*

***Abstract*—EOSIO smart contracts are programs that can be collectively executed by a network of mutually untrusted nodes. As EOSIO smart contracts manage valuable assets, they become high-value targets and are subjected to more and more attacks. Tools for protecting EOSIO smart contracts are imperative.**

This paper proposes EOSIOAnalyzer, an effective static security analysis framework for EOSIO smart contracts to counter the three most common attacks. The framework consists of three components, the control flow graph builder, the static analyzer and the vulnerability detector. This paper implements an approach to transforming low-level Wasm bytecode into a high-level intermediate representation (Register Transfer Language). Besides, this paper also implements vulnerability detection specifications for three popular EOSIO smart contracts vulnerabilities, including Fake EOS Transfer, Forged Transfer Notification and Block Information Dependency. As a proof of concept, this paper conducts experiments to evaluate the effectiveness and efficiency of the EOSIOAnalyzer. The experiment results show that the detection accuracy of the three vulnerabilities is 100%, 98.8% and 100%, respectively.

***Index Terms*—EOSIO smart contract, static analysis, intermediate representation, vulnerability detector**

I. Introduction

With the growing prosperity of cryptocurrencies (e.g., Bitcoin [1]), blockchain technology has attracted extensive attention. As a representative Delegated Proof-of-Stake (DPoS) [2] platform, EOSIO [3] has become one of the most dynamic global communities. Participants of the EOSIO platform jointly maintain a shared ledger and abide by consensus protocols. All transactions are recorded in this ledger and are immutable. Since its launch in June 2018, the total value of on-chain transactions of the EOSIO platform has reached over 6 billion USD in 2019.

EOSIO supports programmable transactions in the form of smart contracts. A smart contract is a computer program executed by mutually distrusting nodes. Smart contracts can be executed automatically on blockchain platforms without users' interference. The uniqueness of the execution results is guaranteed by consensus protocols. EOSIO smart contracts are generally written in C++ and compiled by the EOSIO Contract Development Toolkit (EOSIO.CDT) [4] into WebAssembly (Wasm for short) [5] bytecode. To improve the execution efficiency of smart contracts, EOSIO uses Wasm Virtual Machine (Wasm VM) instead of Ethereum Virtual Machine (EVM) [6].

As smart contracts manage valuable digital assets, they may be subjected to many malicious attacks. Similar to some traditional malicious attacks, the impact caused by the attack on EOSIO smart contracts is likely irreversible. However, developers can modify these contracts and redeploy new contracts when vulnerabilities are found, which is different from the EVM smart contracts. In 2018, due to the design flaws of EOSIO smart contracts, malicious users stole more than 370000 EOS (official cryptocurrency of the EOSIO platform) from EOSBet [7] and EOSCast [8] by utilizing the vulnerabilities of Fake EOS Transfer and Forged Transfer Notification. Therefore, it is necessary to identify such security vulnerabilities in EOSIO smart contracts. However, Wasm and EVM bytecode are two completely different instruction sets; therefore, the vulnerability detection and protection schemes based on EVM bytecode are not applied to Wasm bytecode.

However, there are several challenges in analyzing EOSIO smart contracts, not to mention the vulnerability detection mechanism. Firstly, since the EOSIO platform appears much later than EVM, there is only a limited number of tools and solutions for analyzing EOSIO smart contract vulnerabilities. In addition, although Wasm bytecode is more compact than EVM bytecode, EOS VM is more complicated than EVM. For example, the Wasm VM supports more data types, type conversions, and complicated jumps, which make analysis more difficult and time-consuming. Secondly, with the continuous update of the EOSIO platform and its compiler CDT, the characteristics of the vulnerabilities are constantly changing, which results in the diversity of vulnerabilities. Thirdly, the EOSIO platform also adopts a stack-based VM. The characteristics

† *These authors contributed equally to this work and should be considered co-first authors.*

* *Co-corresponding authors*
✉gzhao@m.scnu.edu.cn(G.Zhao); yangjj@scnu.edu.cn(J.Yang)

of the stack complicate the control and data flow analysis of EOSIO smart contracts, which brings great challenges to static analysis. For example, there are various call (`call()`, `call_indirect()`) and jump (`br`, `br_if`, `br_table`) instructions in EOSIO smart contracts, which causes control flow and data flow relationship between variables to become more ambiguous.

This Paper. This paper proposes and implements the EOSIOAnalyzer, an effective static analysis framework to detect vulnerabilities of EOSIO smart contracts. Specifically, the EOSIOAnalyzer first transforms low-level Wasm bytecode into a high-level intermediate representation (RTL, Register Transfer Language). After the transformation, the EOSIOAnalyzer applies a context-sensitive data flow analysis algorithm to obtain the complete data propagation relationships between variables. Finally, the EOSIOAnalyzer identifies suspicious functions (e.g., `transfer`) and then further analyzes them to obtain the complete execution paths of these vulnerabilities. By integrating the domain knowledge of experts, hidden vulnerabilities could also be identified.

The EOSIOAnalyzer is capable of detecting three prevailing vulnerabilities: Fake EOS Transfer, Forged Transfer Notification and Block Information Dependency. These are the most common vulnerabilities that have enough data and cases for research. Experiments test the EOSIOAnalyzer against 3963 EOSIO smart contracts collected from EOSFuzzer [9], which is the current state-of-art test dataset and baseline. In line with the experimental approach of other current studies, this experiment uses 82 smart contracts, which come with source code, to verify the effectiveness of the EOSIOAnalyzer in a white-box way. Another 3881 smart contracts are used to evaluate the efficiency of the EOSIOAnalyzer on a large scale in a black-box way. These 3881 smart contracts do not come with source code, but only come with Wasm bytecode and ABI (Application Binary Interface). This paper also set up attacks on all identified smart contracts by manually compiling, deploying, debugging and simulating to verify the existence of vulnerabilities in these smart contracts. Experiment results indicate that the proposed EOSIOAnalyzer outperforms other tools in these experiments.

The contribution of this paper is four-fold.

- To detect the three prevailing EOSIO smart contract vulnerabilities, this paper proposes a bytecode-based analysis method for identifying vulnerabilities. The proposed method constructs the control flow graph based on bytecodes, then builds the potential life paths of variables from definition to usages to identify the dependency among variables, followed by the writing of rules for detecting vulnerabilities based on Datalog.
- To tackle the problems caused by the program stack while analyzing smart contracts, this paper proposes an approach to transform low-level Wasm bytecode into a high-level intermediate representation RTL and translate the RTL into logic semantic relations, thereby implementing a logic-driven approach for expressing security analysis.
- To facilitate data flow analysis, this paper proposes a context-sensitive data flow analysis algorithm to obtain the complete data propagation relationships between variables. Thereby improving the accuracy of the experiment results.
- For the purpose of proof of concept, this paper designs and develops the EOSIOAnalyzer that implements the methods and algorithms mentioned above. Experiment results of the EOSIOAnalyzer suggest that it is capable of directly analyzing EOSIO smart contract bytecodes and detecting all three kinds of popular vulnerabilities. Moreover, users can design detection rules according to their requirements to achieve further analysis and detection purposes, which significantly improves the scalability of the EOSIOAnalyzer.

The rest of this paper is organized as follows. Sect.II introduces the related background knowledge on EOSIO smart contracts and theories. Sect.III proposes the EOSIOAnalyzer framework in detail. Sect.IV reports the experiment results. Following the discussion of related work (Sect.V), Sect.VI concludes this paper.

II. Background

This section introduces the relevant background knowledge on WebAssembly, the EOSIO smart contracts program model, the Datalog program analysis and the vulnerabilities of EOSIO smart contracts.

A. WebAssembly

WebAssembly (Wasm for short) is a new stack-based specification that has the characteristics of portability, small size and fast loading speed. The Wasm VM is widely used in the execution engine of browsers, blockchain and other platforms.

Developers can write smart contracts in various high-level languages (e.g., C++, Rust), compile Wasm bytecodes through the compiler and deploy them on various platforms. As EVM bytecodes have been optimized for deployment and running overhead purposes, their bytecode structures are far less friendly for debugging and analyzing than Wasm bytecode structures. Wasm has two equivalent file formats, a bytecode format with the suffix wasm (.wasm) and a text format with the suffix wast (.wast), which can be converted into each other. Generally speaking, the bytecode format is usually used for deployment and execution, while the text format is more intuitive and can be read and analyzed directly by developers.

B. EOSIO Smart Contracts Program Model

The EOSIO smart contract program model is shown in Figure 1, which consists of one or more accounts, each of which can be mounted with zero or one smart contract. Users rely on the client to interact with the EOS network, jointly maintained by all miners (21 block producers selected by all users) and packaged into the blockchain. Each EOSIO smart contract has a uniform calling interface `apply(uint64_t receiver, uint64_t code, uint64_t action)` as the entry point. Whenever

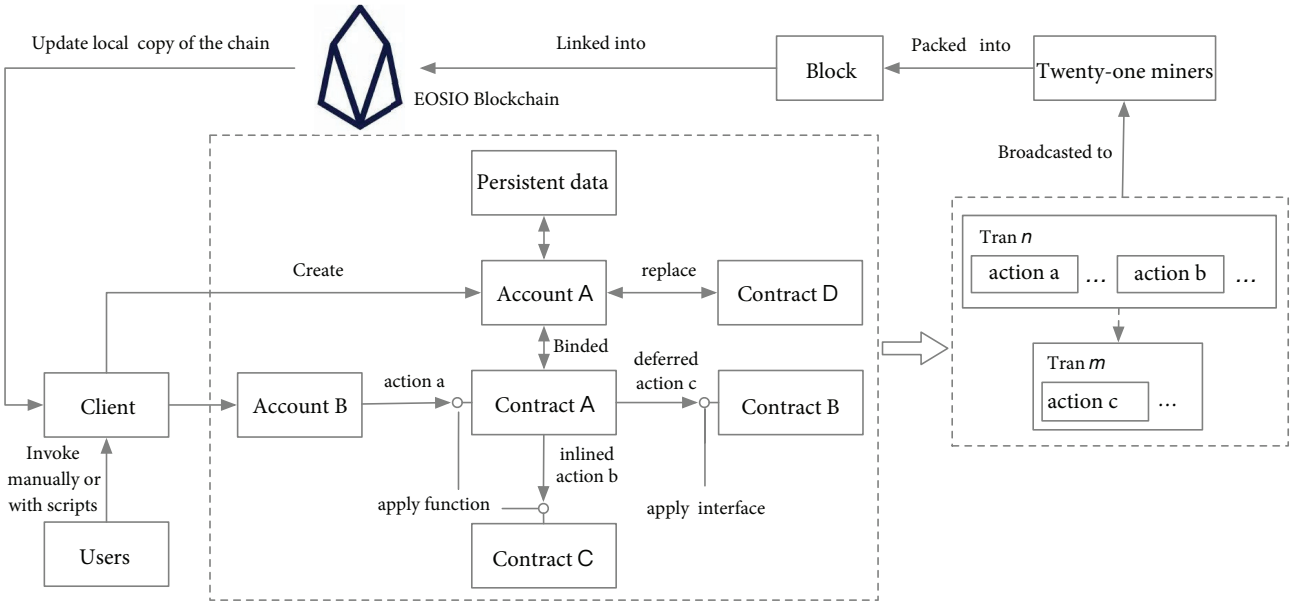

Fig. 1: EOSIO smart contract program model

users or other smart contracts initiate a call, the EOSIO smart contract first enters the `apply` function and maps the corresponding function according to the parameter `action` (includes `inlined action` and `deferred action`) to process related business logic.

C. Datalog Program Analysis

Datalog [10] is a declarative programming language. It has become a standard analysis approach used for static analysis in many high-level and low-level programming languages. Datalog is a subset of Prolog language, consisting of facts and rules. It can deduce new facts from known facts. At the same time, Datalog supports recursive rule structure, and static analysis tasks are typically recursive.

The execution of Datalog mainly includes an extensional database (EDB) and an intensional database (IDB). Figure 2 is a logic-driven program analysis approach [11] [12], in which EDB (some "`.facts`" files extracted by the extractor) is input into the execution engine of Datalog as the initial facts. Combined with some written rules, new facts are generated by reasoning and stored in IDB (some newly exported "`.csv`" files). The IDB can also generate new facts by reasoning, including the intermediate and final analysis results. When the IBD does not change anymore, the analysis reaches a fixed point, no more facts are generated, and the program terminates.

Common Datalog engines include LogicBlox [13], Soufflé [14], etc. Considering the tradeoff of performance, Soufflé is chosen as the underlying execution engine for vulnerability analysis of the EOSIO smart contracts.

D. Vulnerabilities in EOSIO Smart Contracts

This section analyzes the causes of three kinds of vulnerabilities: Fake EOS Transfer, Forged Transfer Notification and

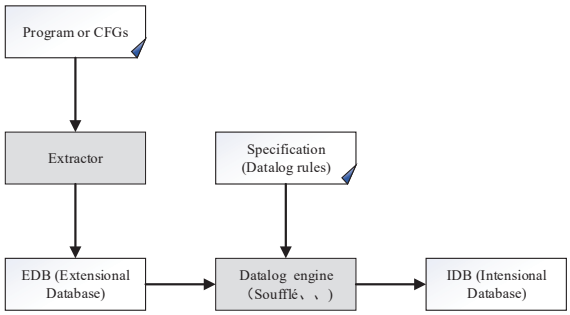

Fig. 2: Program analysis based on logic driver

Block Information Dependency.

1) Fake EOS Transfer: In the EOSIO platform, transfer records between contracts are managed through the `eosio.token` system smart contract. In an expected scenario, an EOSIO smart contract only accepts transfers via the `eosio.token` system smart contract. Since the official `eosio.token` system smart contract is entirely open source, and anyone can copy its source code and publish a cryptocurrency called "EOS". There is no difference between the official EOS cryptocurrency and the fake EOS except for the `issuer`.

As shown in Figure 3, when account A initiates a transfer to account B (step 2) and B enters the `transfer` function (step 4) to execute relevant business logic code after receiving the notification that the transfer is successful. If smart contract B is not validated effectively, there is a chance that malicious users will use the attack method of Fake EOS Transfer to execute the business logic code.

As shown in Table I is part of the EOSBet real code. The program only verifies `code` is contract itself or `eosio.token` in the `apply` function (line 5), but does not

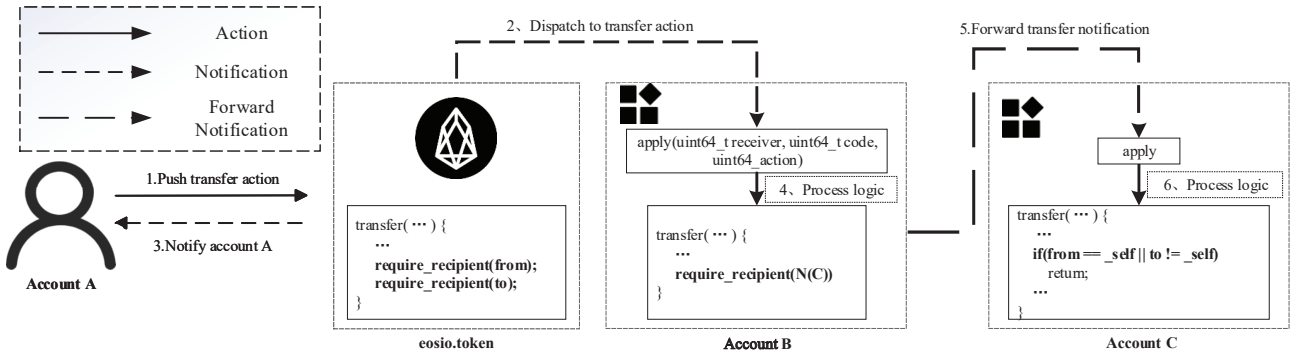

Fig. 3: The transfer life-cycle of smart contract execution

verify when `action==N(transfer)` whether meets `code == N(eosio.token)`. Therefore, the attacker executes the code of the `transfer` function in the smart contract without costing any EOS. At this time the `code == _self`.

TABLE I: The smart contract with Fake EOS Transfer vulnerability

```
1  #define EOSIO_ABI_EX(TYPE,MEMBERS)
2  extern "C"{
3     void apply(uint64_t receiver, uint64_t code, uint64_t action){
4        auto self=receiver;
5        if(code==self||code==N(eosio.token)||action==N(onerror)){
6           TYPE thiscontract(self);
7           switch(action){
8              EOSIO_API(TYPE,MEMBERS)
9  }}}}
```

2) Forged Transfer Notification: Unlike malicious attacks that exploit the Fake EOS Transfer vulnerability, the Forged Transfer Notification vulnerability does not require a malicious user to create a fake EOS token. The malicious user controls two accounts (A and B) in the EOS network and sends real EOS to account B through account A. As shown in Figure 3, the `eosio.token` system smart contract sends a notification to account B after a successful transfer (step 2). When account B receives the notification, it immediately calls the `require_recipient()` (step 4) function to forward the received notification to the victim smart contract C. If smart contract C does not strictly check whether it is the receiver of the notification, it will mistakenly believe that it has received the transfer from account A and enters the `transfer` function to execute the business logic (step 6).

Table II shows an EOSIO smart contract with a Forged Transfer Notification vulnerability. The `transfer` (line 5) function does not check whether the receiver of the transfer notification is itself in the vulnerable smart contract, i.e., it directly executes the business logic code without verifying `data.to==self`.

3) Block Information Dependency: Since an EOSIO smart contracts may use block information (such as `tapos_block_prefix()` or `tapos_block_num()`) to generate random numbers to determine the transfer of EOS or the the winner of a lottery. However, the random numbers obtained in this way are not safe because they can be calculated directly. Therefore, the attacker usually obtains block information by calling `tapos_block_prefix()` or `tapos_block_num()` functions. Then, these returned values will affect some critical behaviors (such as `send_inline` or `send_deferred`) to determine that the attacker is the final winner.

TABLE II: The smart contract with Forged Transfer Notification vulnerability

```
1  Class C:public eosio::contract{
2  public:
3     void transfer(uint64_t sender, uint64_t receiver){
4        auto data=unpack_action_data<st_transfer>();
5        if(data.from==self) //no check for data.to
6           return;
7        doSomething();
8  }};
```

Table III is part of the code of the game contract `eosfun`. The attacker uses the block information obtained by `tapos_block_prefix()` (line 5) function as the judgment condition of the control flow and finally calls `send_inline` or `send_deferred` functions to transfer EOS.

TABLE III: The smart contract with Block Information Dependency vulnerability

```
1  Class eosfun:public eosio::contract{
2  ...
3  void transfer(...){
4  ...
5     if((tapos_block_prefix()/75321+6984)%10){
6        action(
7           permission_level{_self,N(active)},
8           N(eosio.token),
9           N(transfer),
10          std::make_tuple(_self,from,quantity*11/10,memo)
11       ).send();
12 }}}
```

III. The architecture of EOSIOAnalyzer

This section will introduce the overall framework of the EOSIOAnalyzer and then depict each component in detail.

A. Overview

Figure 4 shows the overall architecture of the EOSIOAnalyzer, which accepts the EOSIO Wasm bytecode as the input. The framework consists of three main components: the control flow graph builder, the static analyzer and the vulnerability detector. The second part and the third part are the focus of this section and introduce them in detail.

B. Control Flow Graph Builder

Existing analyses for smart contracts are mostly performed on the control flow graphs (CFGs), which describe the behavior of these contracts. A CFG consists of base blocks and edges representing the control jump relationship of basic blocks. Each basic block is a single-entry, single-exit and sequential Wasm instruction with no jump or termination instructions. This tool inputs Wasm bytecode in binary form and outputs the corresponding CFG based on the cross-language constructor Octopus [15].

C. Static Analyzer

Generally speaking, program vulnerability detection problems are converted into data flow analysis problems. Firstly, we need to analyze the execution path of a smart contract in a fine-grained way to obtain the corresponding intra-procedural control flow graph within the function boundary of the smart contract. However, intra-procedural analysis can only make conservative assumptions about function calls, leading to more false positives. Therefore, we also need to analyze the data propagation relationships between functions when they call each other and analyze the function call graph (CG) according to the function call relationships. The composition of the inter-procedural control flow graph (ICFG) is shown in (1), which is constructed by combining fine-grained CFG and coarse-grained function CG.

$$ICFG = CFGs + call\ edges + return\ edges \quad (1)$$

In the Algorithm 1, the `WorkList` consists of five tuples $(pc, s_{op}, m, g, s_{call})$. pc is a program counter; s_{op} is a operand stack; m is a memory; g is the global variable; s_{call} is a function call stack, represented by the triple tuples $(ret, curr, l)$, where ret represents the return address, $curr$ is the current function, and l is the local variable. The process of the algorithm is to carry out state transformation in the generated ICFG.

The semantics of opcodes are shown in Table IV in the form of execution rules, each of which is associated with several special opcodes, which describes the iteration analysis of the `next` function (line 10 of Algorithm 1) based on operational semantics. Equation (2) describes the state transformation function `transfer` (line 14 of Algorithm 1), where $IN[s]$ represents the definition before the start of the variable, $KILL[s]$ represents the definition that is invalid due to the redefinition of the variable, $GEN[s]$ represents the redefined definition of the variable and $OUT[s]$ represents the definition after the end of the variable. Whenever the `stmt` of nodes (basic blocks) in ICFG is traversed, the `transfer` function will execute according to the corresponding semantics of opcodes to get a new state and record the corresponding data propagation relationships `DF`. For example, s_{op} needs to be updated whenever it encounters an arithmetic opcode, or s_{call} needs to be updated when `call` or `return` are encountered. The algorithm will stop analyzing when `WorkList` is empty, meaning that a fixed point has been reached.

Algorithm 1 Data propagation relationship analysis algorithm

Require: EOSIO smart contract inter-procedural control flow graph $icfg$
Ensure: Data propagation relationships DF
1: $WorkList = [(0,\ empty,\ m_i,\ g_i,\ empty)]$
2: **while** $WorkList$ is not empty **do**
3: $(pc,\ s_{op},\ m,\ g,\ s_{call}) = WorkList.pop()$
4: $ret,\ curr,\ l = s_{call}.top()$
5: **if** $Stmt[pc]$ in $[unreachable, nop, else, end]$ **then**
6: $return$
7: **else if** $Stmt[pc]$ is $drop$ **then**
8: $s_{op}.pop()$
9: **else**
10: $PC_n = next(pc,\ icfg)$
11: **end if**
12: **while** pc in PC_n **do**
13: $DF' = transfer(pc,\ s_{op},\ m,\ g,\ s_{call})$
14: $DF = DF \cup DF'$
15: $\sigma' = (pc',\ s'_{op},\ m',\ g',\ s'_{call})$
16: **if** σ' is not visited **then**
17: $WorkList = WorkList \cup \sigma'$
18: **end if**
19: **end while**
20: **end while**
$Stmt[pc]$: a set of all instructions, pc is program counter
σ: represents the current state

$$\begin{aligned} f_{s_i}(IN(s_i)) &= GEN[s_i] \cup (IN[s_i] - KILL[s_i]) \\ OUT[s_i] &= f_{s_i}(IN(s_i)) \\ IN[s_{i+1}] &= OUT[s_i] \end{aligned} \quad (2)$$

EOSIOAnalyzer performs static analysis based on ICFG, translating stack-based Wasm instructions into intermediate representation and abstracting the stack data that are difficult to analyze into variables. Use the intermediate representation to obtain the data and control flow relationships within EOSIO smart contracts. The processing process is as follows: Step 1.

1) Traverse all Wasm instructions and transform them into the intermediate representation with variables according to semantics. For example, a constant instruction "$i32.const\ 0x1$" transforms into "$V_1 = 0x1$"; a binary arithmetic instruction "$i32.sub$" transforms into "$V_4 = i32.sub\ V_2\ V_3$", where V_2 and V_3 is the operand, V_4 is operation result; a variable access instruction "$get\_local\ 0x0$" transforms into "$V5 = LM0$"; a memory access instruction "$i32.load\ 0x2\ 0x0$" transforms into "$V6 = i32.load(2, 0)\ V5$"; a control flow instruction "$br\_if\ 0x0$" transforms into "$br\_if\ V6$". If a Wasm instruction does not semantically push or pop any data, it will not be transformed into an intermediate rep-

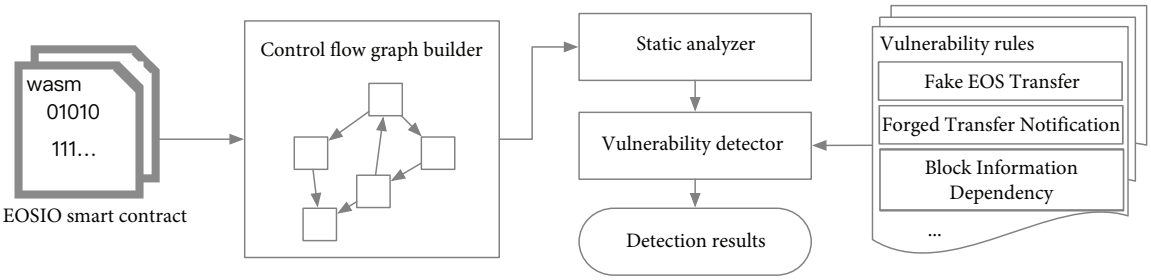

Fig. 4: The architecture of EOSIOAnalyzer

TABLE IV: Wasm instructions conversion function table

Wasm Instructions	Transfer Function	Description
Constant	$\frac{g \leftarrow <s,m,v_g>, \sigma = <f.A,g>}{\sigma \rightarrow \sigma[g[s \rightarrow s.push(val)]]}$	Push val into the operand stack s
Control	$\frac{g \leftarrow <s,m,v_g>, \sigma = <f.A,g>}{\sigma \rightarrow \sigma[g[s \rightarrow eval(s)]]}$	According to CFG, the control flow is transferred in the current function frame $f.A$, and the contents of the operand stack are popped according to different control statements
Arithmetic	$\frac{g \leftarrow <s,m,v_g>, \sigma = <f.A,g>}{\sigma \rightarrow \sigma[g[s \rightarrow s.push(eval(s))]]}$	Read the operand from the operand stack s, and store the result in s after the operation is completed
Load Local Variable	$\frac{f \leftarrow <id,v_l>, g \leftarrow <s,m,v_g>, \sigma = <f.A,g>, ind \leftarrow s[0]}{\sigma \rightarrow \sigma[g[s \rightarrow s.pop(1), s \rightarrow s.push(v_l[ind])]]}$	Load the local variable identified as the top of operand stack s in the current function frame $f.A$
Store Local Variable	$\frac{f \leftarrow <id,s,v_l>, g \leftarrow <s,m,v_g>, \sigma = <f.A,g>, ind \leftarrow s[0], val \leftarrow s[1]}{\sigma \rightarrow \sigma[f[v_m[ind] := val].A, g[s \rightarrow s.pop(2)]]}$	Read two elements ind, val from the operand stack s, and store val in the local variable of the current function frame $f.A$, identified as ind
Load Memory	$\frac{g \leftarrow <s,m,v_g>, \sigma = <f.A,g>, ind \leftarrow s[0]}{\sigma \rightarrow \sigma[g[s \rightarrow s.pop(1), s \rightarrow s.push(m[ind+offset])]]}$	Load the memory value at ind to the operand stack s
Store Memory	$\frac{g \leftarrow <s,m,v_g>, \sigma = <f.A,g>, ind \leftarrow s[0], val \leftarrow s[1]}{\sigma \rightarrow \sigma[g[s \rightarrow s.pop(2), m[ind+offset] := val]]}$	Read two elements ind and val from operand stack s and store val in memory $m[ind]$
Load Global Variable	$\frac{g \leftarrow <s,m,v_g>, \sigma = <f.A,g>, ind \leftarrow s[0]}{\sigma \rightarrow \sigma[g[s \rightarrow s.pop(1), s \rightarrow s.push(v_g[ind])]]}$	Load the global variable identified as the top element ind of the operand stack s, and store the result in s
Store Global Variable	$\frac{g \leftarrow <s,m,v_g>, \sigma = <f.A,g>, ind \leftarrow s[0], val \leftarrow s[1]}{\sigma \rightarrow \sigma[f[v_g[ind] := val], g[s \rightarrow s.pop(2)]]}$	Read the two elements ind and val from the operand stack s, and store val in the global variable identified as ind
Function Call	$\frac{f \leftarrow <id,v_l>, g \leftarrow <s,m,v_g>, \sigma = <f.A,g>, f' = (id',0,v_l'), params = \{s_0,...,s_n\}}{\sigma \rightarrow \sigma[g[s \rightarrow s.pop(n)], f'[v_l' \rightarrow params], \rho \rightarrow f'.f.A]}$	After loading n elements from the operand stack s as the calling parameter $param$, a new function frame f' is created, and the local variable v_l' is set to $param$ according to the order. Finally, push f' into the function frame stack
Return	$\frac{f \leftarrow <id,s,v_l>, \sigma = <f'.f.A,g>, f' = (id',s',v_l')}{\sigma \rightarrow \sigma[\rho \rightarrow f.A]}$	Pop the top function frame f' from the function frame stack, and return control to the function frame $f.A$

resentation. For example, "nop", "end", "$unreachable$" instructions, etc.

2) Initialize the stack $Stack = \{S_0, S_1, ..., S_n\}$, S_0 is the top stack element and S_n represents the deepest element accessible by Wasm instructions within the block. Iterate over the intermediate representation instruction sequence in the basic block. Algorithm 1 is used to analyze the data propagation relationships, update the operand stack according to the instruction semantics, and record the definition relationship $def(var,\ stmt)$, the usage relationships $use(var,\ stmt,\ i)$ and the data flow relationships $FlowsFrom(to,\ from)$. For example, a binary arithmetic instruction "$V_3 = i32.sub\ V_1\ V_2$", operand stack pop the top two elements S_0 and S_1, after to push the result into the stack, assigned to "V_4", and record $def(var1,\ stmt1)$, $def(var2,\ stmt2)$, $def(var3,\ stmt3)$, $use(var1,\ stmt3,\ 0)$, $use(var2,\ stmt3,\ 1)$, $FlowsFrom(stmt3,\ stmt1)$ and $FlowsFrom(stmt3,\ stmt2)$ relationships of variables. Other Wasm instructions perform similar operations. In order to avoid the problem of path explosion, a constraint condition is added in the algorithm implementation: when the value of variables exceeds n(10), the variable is set as $\top$.

3) To get the complete data transformation relationships as much as possible, according to the data flow transitivity, if $FlowsFrom(temp,\ from)$ and $FlowsFrom(to,\ temp)$ exist, there is a data flow $FlowsFrom(to,\ from)$.
4) Iterate over the above steps until a fixed point is reached and get the complete Def-Use and data flow relationships.

D. Vulnerability Detector

Finally, EOSIOAnalyzer uses the vulnerability detection rules and the complete data propagation relationships obtained by the static analyzer to achieve the purpose of vulnerability detection.

Based on II-D, this paper makes a detailed analysis of the popular EOSIO smart contracts vulnerabilities. However, as compiler versions (CDT) continue to evolve, even the same logic may have different representations of Wasm instructions. Therefore, we have tried our best to summarize and cover as many different cases as possible to ensure the robustness and completeness of the detector. Figure 5 contains all the variables for different vulnerability characteristics. Then it establishes a set of relatively complete vulnerability rules, which can effectively detect three common vulnerabilities.

1) Fake EOS Transfer Analysis Rules: According to II-D1, the Fake EOS Transfer vulnerability is mainly caused by the contract developer not checking the code is `eosio.token` when the action is `transfer` within its `apply` function, which leads to attackers illegally entering into `transfer` to execute relevant business codes. Therefore, if there is an execution path that calls `transfer` function in `apply` function, which means the `transfer` function is reachable from `apply` function by attackers.

As shown in Listing 1, which is a part of the Fake EOS Transfer analysis rules, `NotEosioToken` indicates no comparison between `code` and `eosio.token`. `Call_Transfer` indicates that the `transfer` function is called, and there is a dominant relationship[1] between the two steps.

```
.decl DetectFakedEosTransfer(call_stmt: Statement)
.output DetectFakedEosTransfer
DetectFakedEosTransfer(call_stmt) :-
NotEosioToken(true_stmt),
Call_Transfer(call_stmt),
Dominates(call_stmt, true_stmt).
```

Listing 1: Fake EOS Transfer analysis rules

As shown in (3), if all the following conditions are met, that is, in `apply` function, when "`code == self`" and "`action == transfer`", there is no assertion judgment between `code` and `eosio.token`, it indicates that the contract has a Fake EOS Transfer vulnerability.

[1]*Dominates(call_stmt, true_stmt): true_stmt must be called before call_stmt is called*

$$\{\neg eosio\_assert\{code == N(\text{“}eosio.token\text{”})\}\} \wedge \{action == N(\text{“}transfer\text{”})\} \wedge \{\{code == self\} \vee \{\forall acct \in accounts,\ acct \neq code\}\} \subseteq func_{apply} \quad (3)$$

2) Forged Transfer Notification Analysis Rules: According to II-D2, the Forged Transfer Notification vulnerability is mainly caused by the contract developer not strictly checking whether the values of `_self` and `to` are equal when writing the `transfer` function. EOSIOAnalyzer does this by checking if an execution path in the `transfer` function does not result in an abnormal rollback.

As shown in Listing 2, part of the Forged Transfer Notification analysis rules, `IsTransfer` means locating all `transfer` functions in the program, `getSelfVar` represents getting the parameter `_self`, and `getToVar` represents getting the parameter `to` in the function.

```
.decl DetectForgedTranNotification(stmt: Statement
  , func: Function)
.output DetectForgedTranNotification
DetectForgedTranNotification(cmp_stmt, func) :-
IsTransfer(func),

getSelfVar(self_var, func),
getToVar(to_var, func),
(op(cmp_stmt, "i64.eq", _); op(cmp_stmt, "i64.ne",
  _)),
use(self_var, cmp_stmt, _),
use(to_var, cmp_stmt, _),
def(cond_var, cmp_stmt),

((op(cond_stmt, "br_if", _),
use(cond_var, cond_stmt, 0),
in_function(cond_stmt, func));

(FuncAssertStr(assert_stmt, func, _),
use(cond_var, assert_stmt, 1))).
```

Listing 2: Forged Transfer Notification analysis rules

As shown in (4), if all the following conditions are met, that is, when there is an assertion judgment between the parameters `to` and `_self` in the `transfer` function, or as long as there is a comparison between `to` and `_self`, there is no vulnerability in the contract.

$$\{eosio\_assert(to == \_self)\} \vee \{to == \_self\} \vee \{to \neq \_self\} \subseteq func_{transfer} \quad (4)$$

3) Block Information Dependency Analysis Rules: According to II-D3, the Block Information Dependency vulnerability is caused by using functions such as `tapos_block_prefix()` or `tapos_block_num()` in EOSIO smart contracts to obtain block information and use its return value as the judgment condition of control flow. Therefore, EOSIOAnalyzer detects whether there is an execution path in EOSIO smart contracts that uses block information as a judgment condition and finally triggers `send_inline()` or `send_deferred()` functions to transfer money.

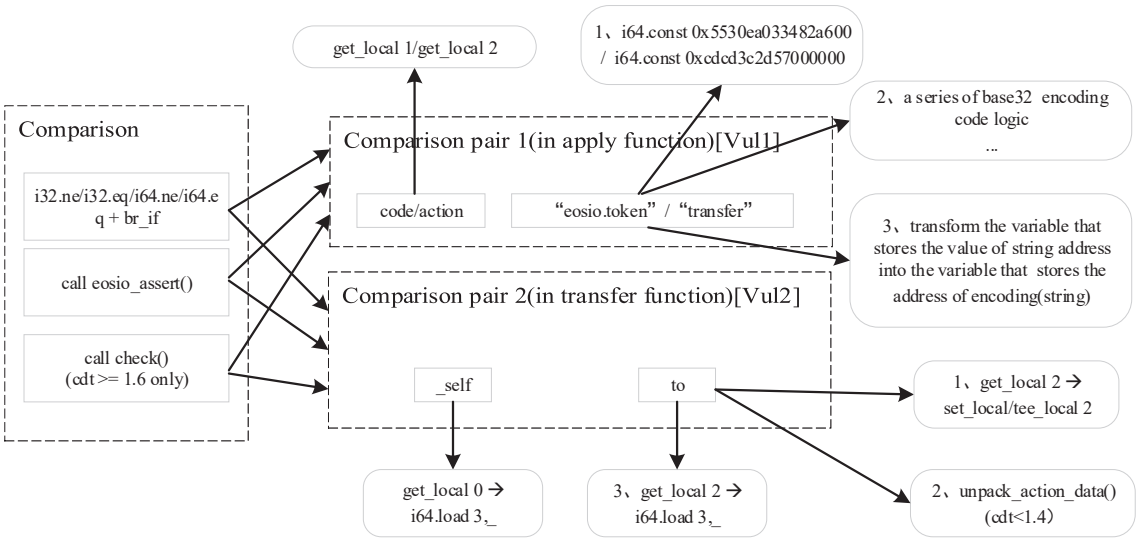

Fig. 5: Summary of covering variants

As shown in Listing 3, part of Block Information Dependency analysis rules, `is_function` indicates calling the specified function, and `FlowsFrom` indicates the data flow relationship between variables.

```
.decl DetectBlockInfoDependency(stmt: Statement)
.output DetectBlockInfoDependency
DetectBlockInfoDependency(stmt) :-
op(stmt, "call", _),
imm(func_id, stmt, 0),
(is_function(func_id, "tapos_block_prefix");
  is_function(func_id, "tapos_block_num")),
def(var, stmt),

FlowsFrom(inter_var, var),

op(cont_stmt, "br_if", cond_block),
use(inter_var, cont_stmt, _),

op(target_stmt, "call", call_block),
imm(target_func_id, target_stmt, _),
(is_function(target_func_id, "send_inline");
  is_function(target_func_id, "send_deferred")),
block_dom(call_block, cond_block).
```

Listing 3: Block Information Dependency analysis rules

As shown in (5), if all the following conditions are met, when `func1` is a function to obtain block information and `func2` is a function to transfer money, there is a dominant[2] relationship between them. If it exists, it is considered that there is a Block Information Dependency vulnerability in the contract.

$$\begin{aligned} &fun1 \in \{tapos\_block\_prefix, tapos\_block\_num\} \\ &\wedge\ func2 \in \{send\_inline, send\_deferred\} \\ &\wedge\ Dom(fun2, fun1) \end{aligned} \tag{5}$$

[2]*Dom(func2,func1): func1 must be called before func2 is called*

IV. Experiment and Results Analysis

This section evaluates the EOSIOAnalyzer and compares it with other EOSIO smart contract vulnerability detection tools.

A. Research Questions

RQ 1:

1) Can EOSIOAnalyzer detect vulnerabilities effectively within EOSIO smart contracts?
2) Can the average operating efficiency of EOSIOAnalyzer be used for vulnerability detection in real EOSIO smart contracts?

B. Experiment Subjects

To answer RQ 1, 82 EOSIO smart contracts with source code are collected to analyze the effectiveness of EOSIOAnalyzer. The experiment subjects include EOSBet, EOSCast and other smart contracts that cause huge economic losses in real events.

To answer RQ 2, the second dataset is collected on the EOS main network for a total of 3881 non-source EOSIO smart contracts to analyze the execution efficiency of EOSIOAnalyzer and to evaluate the security of existing EOSIO smart contracts.

We publish all the subjects, test cases, experiment results and comparative experiment results on the website (https://github.com/lwy0518/datasets_results)

C. Experiment Setup

The experiment environment of this paper is a server with a CentOs7.9 system, Intel Xeon CPU E5-2620 and 78GB of RAM. Datalog's execution engine Soufflé[3] is currently the latest version 2.0.2 (GCC and G++ version are not less than 7.0.0) and python3 version is not less than 3.6. The required installation packages are in the `requirements.txt` file and the TIMEOUT of the experiment is set to 5min.

[3]*https://souffle-lang.github.io/*

D. RQ1: The Effectiveness of EOSIOAnalyzer

This section shows the vulnerability detection results of EOSIOAnalyzer and compare it with the tool EOSFuzzer [9], WANA [16] and EVulHunter [17].

1) Experiment Results and Analysis: Table V shows the vulnerability detection results of EOIOAnalyzer. We calculate seven indicators of the results, namely false positive (FP), false negative (FN), precision (P), accuracy (A), recall (R), F1-Measure and average analysis time. These metrics indicate that the framework's accuracy, effectiveness, and completeness are ideal.

TABLE V: Results of EOSIOAnalyzer on smart contracts with source code

Vulnerability types		**Fake EOS Transfer**	**Forged Transfer Notification**	**Block Information Dependency**
Total contracts (source code)		82		
EOSIO-Analyzer	**Results**	1	5	3
	FP	0	1	0
	FN	0	0	0
	Precision	100%	80%	100%
	Accuracy	100%	98.8%	100%
	Recall	100%	100%	100%
	F-Measure	100%	88.9%	100%
	Aver-time	7.6s		

And we manually checked these 82 smart contracts with source code to identify the false positive or false negative cases. There are no timeout and error contracts. EOSIOAnalyzer detected one smart contract with Fake EOS Transfer vulnerability and three smart contracts with Block Information Dependency vulnerability. And we confirmed that EOSIOAnalyzer reports neither false positives nor false negatives in detecting these two types of vulnerabilities.

For the Forged Transfer Notification vulnerability, five contracts have been detected with vulnerabilities, and there is a false positive contract `charity` and no false negatives. As shown in Table VI, which is part of the code of the contract `charity`. The vulnerability rules directly compare whether the parameter `to` in the `transfer` function is equal to `_self` to determine a vulnerability. The contract `charity` compares `to==N(charity)` in the `transfer` function to determine whether itself is the recipient of the transfer. However, this is inconsistent with the vulnerability detection rules we have written. Unless we directly check the corresponding CFG, we can determine the value of `N(charity)`. Moreover, this contract is also a special case (that is, if the contract name is changed, the value will be different).

2) Comparison of Results: Table VII shows the experiment comparison results of EOSIOAnalyzer with EOSFuzzer, WANA and EVulHunter. Because the code of EOSAFE [18] is not open source and the experimental dataset is not the same as our experiment, we did not use EOSAFE as the experimental comparison subject. In addition, since EVulHunter cannot detect Block Information Dependency vulnerability, it can only compare the results of the other two vulnerabilities.

TABLE VI: A false postive case of Forged Transfer Notification by EOSIOAnalyzer

```
1  class charity : public eosio::contract {
2  ......
3      void transfer(account_name from, account_name to, asset quan-
       tity, std::string memo){
4          if(from != N(charity) && to == N(charity) && quan-
           tity.symbol == S(4, EOS) )
5          ......
6  }}
```

TABLE VII: Comparison of experiment results EOSIOAnayzer, EOSFuzzer, WANA and EVulHunter

Vulnerability types		**Fake EOS Transfer**	**Forged Transfer Notification**	**Block Information Dependency**
Total contracts (source code)		82		
EOSIO-Analyzer	**Results**	1	5	3
	FP	0	1	0
	FN	0	0	0
EOSFuzzer	**Results**	2	4	2
	FP	1	0	0
	FN	0	0	1
WANA	**Results**	1	0	0
	FP	0	0	0
	FN	0	4	3
EVulHunter	**Results**	9	11	N/A
	FP	8	10	N/A
	FN	0	1	N/A

The results indicate that the tool EOSFuzzer has a false positive and a false negative. The tool WANA has seven false negatives. The tool EVulHunter has eighteen false positives and two false negatives. Since EOSFuzzer's analysis time per contract is linearly related to the number of test cases, this can lead to longer times. Since EOSFuzzer's analysis time per contract is linearly related to the number of test cases, this can lead to longer times. In addition, when the tool EOSFuzzer locates the suspicious `transfer` function, it finds the function with the same function signature according to the parameters and return value of the `transfer` function in the existing source code and then treats it as a suspicious function. However, since the amount of data with source code is not much in that paper, the function obtained in this way is not convincing.

Compared with WANA and EVulHunter, their false positive and false negative rates are higher, resulting in the accuracy of experimental results is not ideal. Therefore, in summary, the experiment results of EOSIOAnalyzer are relatively more ideal.

E. RQ2: Execution Efficiency of EOSIOAnalyzer

Table VIII shows the results of EOSIOAnalyzer detecting 3881 EOSIO smart contracts without source code, of which 441 smart contracts are at risk of being successfully attacked by malicious users. And the EOSIOAnalyzer has identified 128 Fake EOS Transfer vulnerabilities, 237 Forged Transfer Notification vulnerabilities, and 38 Block Information Dependency vulnerabilities, which account for 3.30%, 6.11%, and 0.98%,

respectively. However, since these smart contracts only have Wasm bytecode and ABI, it is difficult to verify the accuracy of the detection results manually, but the final detection results show that there are many vulnerable smart contracts in real life. Therefore, this is a reminder that security checks must be carried out before deploying the contract to reduce user losses.

TABLE VIII: Results of EOSIOAnalyzer on smart contract without source code

Vulnerability types	Total contracts (no source code)	EOSIOAnalyzer	
		Results	Percentage
Fake EOS Transfer	3881	128	3.30%
Forged Transfer Notification		237	6.11%
Block Information Dependency		38	0.98%

Through analyzing the execution time of these smart contracts without source code, excluding 75 timeout contracts (account for 1.93%) and 51 error contracts (account for 1.31%), the average analysis time for each contract is about 15 seconds. After verification, it is found that the Wasm file size in 51 error contracts is 0, so it can not be successfully detected. Excluding these error contracts, the success rate of contract decompilation is 98.0%, which is acceptable. To this end, the efficiency of EOSIOAnalyzer is sufficient to support realistic EOSIO smart contracts vulnerability detection.

V. Related work

A. Ethereum Smart Contract Analysis and Validation

As one of the largest blockchain technologies in the world, which has received extensive attention in various fields. At present, many effective vulnerability detection schemes for Ethereum smart contracts have been proposed in the relevant literature on vulnerability analysis and detection of smart contracts [11], [19]–[28]. For example, Luu et al. proposed Oyente [19], the first symbolic execution(SE) [29] tool for detecting vulnerabilities in Ethereum smart contracts. X Wang et al. [25] embedded the intrusion detection system into the bytecode of the smart contracts using instrumenting so that the smart contracts can monitor illegal transactions and intercept them in time after deployment to prevent losses to users.

B. EOSIO Smart Contract Analysis and Validation

Compared with the literature on vulnerability analysis and detection of Ethereum smart contracts, there is very little literature on vulnerability analysis and detection of EOSIO smart contracts, mainly because the new blockchain technology proposed by EOSIO in 2018 is far behind the most popular and largest blockchain platform Ethereum. Over time, however, EOSIO has attracted more and more attention from the industry, and EOSIO smart contracts security incidents have caused more detrimental effects than the Ethereum attacks. Moreover, due to the upgrade of Ethereum 2.0 and the use of Ethereum WebAssembly (EWasm VM) instead of EVM as the execution engine of the smart contracts, developing vulnerability analysis tools for Wasm bytecode has become the focus of supporting the development of EOSIO and Ethereum.

In the relevant literature of EOSIO smart contracts vulnerability analysis and detection, Y Huang et al. [9] built a general black-box analysis tool EOSFuzzer through fuzzing testing. N He et al. [18] used symbolic execution, EOSIO smart contracts simulator and other technologies to find out whether there are vulnerabilities in the bytecode of the EOSIO smart contract. L Quan et al. [17] also used similar ideas to design EVulHunter. Compared with other vulnerability detection tools, D Wang et al. [16] proposed a cross-platform vulnerability detection tool for smart contracts, which can simultaneously identify the vulnerabilities of smart contracts on both Ethereum and EOSIO platforms. Huang et al. [30] proposed a method to identify the bot-like accounts in EOSIO based on transaction analysis. Lee et al. [31] introduced and studied four attacks stemming from the unique design of EOSIO.

VI. Conclusions

EOSIO smart contracts are getting widespread in various scenarios, managing high-value assets. They have suffered a large number of attacks and resulted in great losses in many cases. It is imperative to have an effective tool to detect vulnerabilities in EOSIO smart contracts, to avoid further attacks and losses. This paper proposes a method to detect vulnerabilities based on EOSIO bytecodes using static analysis. The proposed method converts bytecodes into control flow graphs and builds the dependency relations of variables. By using Datalog for reasoning, the proposed method is capable of detecting the three most common vulnerabilities. This paper implements the proposed method and tests it against 3963 different smart contracts, 82 smart contracts of which are with source codes while the others are not. The experiment result suggests that the proposed method effectively detects the targeted vulnerabilities.

The contribution of this paper is as follows.

- This paper proposes a bytecode-based analysis method for identifying vulnerabilities. The proposed method constructs the control flow graph based on bytecodes, then identifies the dependency among variables, followed by the detection of vulnerabilities with the help of Datalog reasoning.
- This paper proposes an approach to transform low-level Wasm bytecode into a high-level intermediate representation RTL and translate the RTL into logic semantic relations. This transformation enables the logic-driven approach for expressing security analysis.
- This paper proposes a context-sensitive data flow analysis algorithm to obtain the complete data propagation relationships between variables.
- This paper implements the methods and algorithms mentioned above and tests against a large number of EOSIO smart contracts. The experiment results suggest the effectiveness of the proposed method.

This paper is not without its limitations. There needs a better way to generate the detection rules of vulnerability. This

requires an automatic algorithm to engineer the features that can distinguish vulnerabilities.

In the future, we plan to expand EOSIOAnalyzer to detect more new types of EOSIO smart contracts vulnerabilities and improve the efficiency and effectiveness of its detection. In addition, since Ethereum 2.0 uses EWasm VM instead of EVM, the approach of transforming low-level Wasm bytecode into a high-level intermediate representation implemented in this paper can be directly applied to detect Ethereum smart contract vulnerabilities.

ACKNOWLEDGMENT

This work is supported by the Key-Area Research and Development Program of Guangdong Province(No.2020B0101090003), National Key-Area Research and Development Program of China (2018YFB1404402), Key-Area Research and Development Program of Guangdong Province(No.2019B010137003), National Social Science Foundation Major Project (19ZDA041) and VeChain Foundation (No.SCNU2018-01).

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[2] D. Larimer, "Delegated proof-of-stake (dpos)," *Bitshare whitepaper*, vol. 81, p. 85, 2014.

[3] "EOSIO - blockchain software architecture," accessed August, 2021. [Online]. Available: https://eosio.io

[4] "EOSIO.CDT," 2021, accessed August, 2021. [Online]. Available: https://developers.eos.io/manuals/eosio.cdt/latest/index

[5] "WebAssembly," accessed August, 2021. [Online]. Available: https://webassembly.org

[6] G. Wood, "ETHEREUM: a secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[7] "EOSBet," accessed August, 2021. [Online]. Available: https://eosbet.io

[8] "EOSCast," accessed August, 2021. [Online]. Available: https://www.eoscast.cc

[9] Y. Huang, B. Jiang, and W. K. Chan, "EOSFuzzer: Fuzzing eosio smart contracts for vulnerability detection," *arXiv preprint arXiv:2007.14903*, 2020.

[10] "Datalog," accessed November, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Datalog

[11] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[12] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Asian Symposium on Programming Languages and Systems*. Springer, 2005, pp. 97–118.

[13] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1371–1382.

[14] "Soufflé," https://souffle-lang.github.io/, 2021, accessed November, 2020.

[15] "Octopus," accessed August, 2021. [Online]. Available: https://github.com/pventuzelo/octopus

[16] D. Wang, B. Jiang, and W. Chan, "WANA: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," *arXiv preprint arXiv:2007.15510*, 2020.

[17] L. Quan, L. Wu, and H. Wang, "EVulHunter: Detecting fake transfer vulnerabilities for eosio's smart contracts at webassembly-level," *arXiv preprint arXiv:1906.10362*, 2019.

[18] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "EOSAFE: Security analysis of EOSIO smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1271–1288. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/he-ningyu

[19] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[20] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[21] 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.

[22] J. Krupp and C. Rossow, "TEETHER: Gnawing at ethereum to automatically exploit smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1317–1333.

[23] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.

[24] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts." in *NDSS*, 2018.

[25] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung, "ContractGuard: defend ethereum smart contracts with embedded intrusion detection," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 314–328, 2019.

[26] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi, "EVM*: from offline detection to online reinforcement for ethereum virtual machine," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 554–558.

[27] G. Ayoade, E. Bauman, L. Khan, and K. Hamlen, "Smart contract defense through bytecode rewriting," in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019, pp. 384–389.

[28] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He *et al.*, "SODA: a generic online detection framework for smart contracts," in *Proceedings of the 27th Network and Distributed System Security Symposium*, 2020.

[29] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[30] Y. Huang, H. Wang, L. Wu, G. Tyson, X. Luo, R. Zhang, X. Liu, G. Huang, and X. Jiang, "Characterizing eosio blockchain," *arXiv preprint arXiv:2002.05369*, 2020.

[31] 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.