WANA: Symbolic Execution of Wasm Bytecode for Extensible Smart Contract Vulnerability Detection

Bo Jiang†, Yifei Chen, Dong Wang
School of Computer Science and Engineering
Beihang University
Beijing, China
{jiangbo, chenyf, dongwang17}@buaa.edu.cn

Imran Ashraf and W.K. Chan
Department of Computer Science
City University of Hong Kong
Hong Kong
iashraf3-c@my.cityu.edu.hk and wkchan@cityu.edu.hk

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00102

***Abstract*—Many popular blockchain platforms support smart contracts for building decentralized applications. However, the vulnerabilities within smart contracts have demonstrated to lead to serious financial loss to their end users. In particular, the smart contracts on EOSIO smart contract platform have resulted in the loss of around 380K EOS tokens, which was around 1.9 million worth of USD at the time of attack. The EOSIO smart contract platform is based on the Wasm VM, which is also the underlying system supporting other smart contract platforms as well as Web application. In this work, we present WANA, an extensible smart contract vulnerability detection tool based on the symbolic execution for Wasm bytecode. WANA proposes a set of algorithms to detect the vulnerabilities in EOSIO smart contracts based on Wasm bytecode analysis. Our experimental analysis shows that WANA can effectively and efficiently detect vulnerabilities in EOSIO smart contracts. Furthermore, our case study also demonstrates that WANA can be extended to effectively detect vulnerabilities in Ethereum smart contracts.**

Keywords-WASM bytecode; Symbolic Execution; Smart Contract; Vulnerability Detection

I. INTRODUCTION

The blockchain technology has enabled decentralized value transfer networks among parties with limited trust [16]. With the support of smart contracts, the developers can build Decentralized Applications (DApps) on top of the blockchain platforms such that untrusted parties can cooperate with each other. Popular DApps include blockchain games, Decentralized Finance (DeFi), online gambling, decentralized exchanges, wallets, supply chain management, logistics tracking, etc.

EOSIO [31] is a popular public blockchain platforms supporting smart contracts. However, the vulnerabilities within EOSIO smart contracts have led to financial loss to their end users. For EOSIO smart contracts, the Block Information Dependency vulnerability, the Fake EOS Transfer vulnerability and the Forged Transfer Notification vulnerability have led to the loss of around 380K EOS tokens [34][38] in total. The accumulated amount of loss by these vulnerabilities was around 1.9 million worth of USD at the time of attack. Therefore, effective vulnerability detection tools are needed to safeguard the ecosystem of EOSIO blockchain platforms.

WebAssembly (**Wasm** for short) [50] is a binary instruction format for a stack-based virtual machine. The Wasm Virtual Machines (VMs) are adopted by not only EOSIO public blockchain platforms as the smart contract execution engine [35] but also the agenda items of other blockchain platforms. For instance, the Ethereum blockchain platform have planned to adopt the Ethereum Wasm (EWasm) VM for Ethereum 2.0 [14][21]. The Polkadot platform [40] further adopts Wasm as the compilation target for both of its parachain runtime and smart contract execution. Wasm is also popular in building Web applications because Wasm modules can call into the JavaScript context and access the browser functionality through the same Web APIs accessible from JavaScript. A ***comprehensive*** symbolic execution engine for Wasm is promising to provide an extensible security analysis framework for any applications targeting at Wasm.

In this work, we proposed **WANA**[1], a general symbolic execution engine for Wasm bytecode that can support vulnerability ANAlysis for EOSIO smart contracts. Unlike the existing work, which can only handle a small subset of bytecode instructions, WANA handles the complete instruction set of Wasm specification 1.0. In our experiment on about 65285 EOSIO smart contracts, WANA effectively detected more than 12,000 vulnerabilities. Furthermore, our exploratory case study showed that WANA can also be extended to detect five important classes of security vulnerabilities of Ethereum smart contracts, demonstrating its possibility to handle smart contracts targeted for different blockchain platforms.

The contribution of this work is fourfold. First, it presents a general symbolic execution framework for Wasm bytecode. Second, it presents a new extensible architecture of vulnerability detection tool for smart contracts based on the symbolic execution engine. Third, this work presents a comprehensive experimental study to evaluate the vulnerability detection effectiveness of WANA for EOSIO smart contracts. Finally, it presents a case study to evaluate the extensibility of WANA in terms of vulnerability detection on the Ethereum platform.

The organization of the remaining sections is as follows. In Section II, we introduce the preliminaries of WANA, which

†Correspondence Author.

[1] WANA is open sourced at https://github.com/gongbell/WANA

2693-9177/21/$31.00 ©2021 IEEE
DOI 10.1109/QRS54544.2021.00102

is followed by a review on typical vulnerabilities in EOSIO and Ethereum smart contracts. In Section IV, we present the the WANA framework. After that, we report an experimental study to evaluate the effectiveness of WANA in terms of vulnerability detection on EOSIO smart contracts in Section V. It is followed by a case study to evaluate the extensibility of WANA in terms of vulnerability detection on Ethereum platform in Section VI. Finally, we present the related works in Section VII and conclude this work in Section VIII.

II. Background on Wasm and Smart Contracts

In this section, we will revisit background information on Wasm and smart contracts.

A. Wasm

Wasm [50][51] is a binary instruction format for a stack-based virtual machine (VM). It is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications. As a result, it is widely supported by all major web browsers [50]. There are two convertible representations for Wasm: the binary format (".wasm" as suffix) for execution and the text format (".wast" as suffix) for reading by human.

B. Background on EOSIO Smart Contracts

EOSIO [26] is a public blockchain platform that focuses on the scalability of transactions. It uses the Delegated Proof of Stake (DPoS) consensus protocol which is more scalable than the Proof of Work (PoW) consensus protocol.

EOSIO supports two types of smart contracts. The first type is the system contracts, which are smart contracts deployed on an EOSIO platform by default to realize the core blockchain features such as consensus, fee schedules, account creation, and token economics [47]. For instance, the eosio.token smart contract [33] is one such system smart contract, which defines the data structures and actions for users to issue and transfer tokens on an EOSIO blockchain. The second type of smart contracts are user-defined smart contracts [42], which realizes specific business requirements.

Each EOSIO smart contract must realize the *apply* function as the entry function to handle actions. All incoming actions are routed to the *apply* function [29], which will in turn dispatch the actions to the corresponding handler functions for processing. To develop an EOSIO smart contract, programmers may use C++ [31]. The source code is then compiled into the Wasm bytecode for execution on an EOSIO Wasm VM. Invoking the *apply* function requires to provide a few parameters: *receiver*, *code*, and *action*. The receiver is the EOSIO account currently processing the action (denoted as the *current receiver*, or the receiver for short). The code is the account that the action is "originally" sent to (denoted as the *original receiver*). The action is the identity of the action. When a smart contract receives an action, it may forward the action to other contracts via the *require_recipient*() function.

C. Background on Ethereum Smart Contracts

On an Ethereum blockchain platform, there are two types of accounts: external accounts (i.e., owned by human) and smart contract accounts [38]. Smart contract is responsible for managing the balance and the persistent private storage of the contract account. A transaction is a message sent from one account to another account. Conceptually, Ethereum [35] can be viewed as a transaction-based state machine, where its state is updated on each transaction. When the target account of a transaction is a smart contract account, the code of the smart contract is executed and the payload is provided as input.

In existing releases of Ethereum, the smart contract code is executed on an Ethereum Virtual Machine (EVM). Developers can write smart contracts in the Solidity language [44], which are then compiled into the EVM bytecode. In the upcoming releases of Ethereum 2.0, EWasm VM is planned to replace EVM to serve as the smart contract execution engine [14][21]. In particular, EWasm is a restricted subset of Wasm to be used for contracts in Ethereum. For example, it eliminates non-deterministic behaviors during smart contract execution generally available to Wasm bytecode.

III. Smart Contract Vulnerabilities

In this section, we will review smart contract vulnerabilities in both the EOSIO and Ethereum blockchain platforms.

A. EOSIO Smart Contract Vulnerabilities

We will present three typical EOSIO smart contract vulnerabilities detected by WANA in this section.

1) ***Fake EOS Transfer****:* In an EOSIO platform, if a smart contract (i.e., a sender contract) wants to send EOS tokens to another smart contract (i.e., a receiver contract), it must invoke the functions provided by the *eosio.token* system smart contract. This *eosio.token* contract manages the EOS accounts of all smart contracts within its internal storage. During a token transfer, the sender contract must call the transfer function of *eosio.token*. Within the transfer function of *eosio.token*, the balances of the sender and receiver contract accounts will be updated accordingly. Meanwhile, *eosio.token* will aksi call the function *require_recipient*() to notify these two contracts when performing the transfer.

```
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  }}}}
```

Figure 1. The EOSBet Contract with Fake EOS Transfer Vulnerability

A smart contract should be safe. One requirement of a safe smart contract is that the contract must ensure the original receiver (i.e., the code parameter of apply function) of the *transfer* action is *eosio.token*. However, if the contract under attack is vulnerable in that the execution the *apply* function does not check whether the code parameter of the *apply* function is *eosio.token* when the action is the identity of the

transfer function, an attacker may perform an inline call to its transfer function directly to fake an EOS transfer. As a result, the vulnerable contract may wrongly consider that the attacker has transferred EOS tokens to it. In the history, the EOSBet [27] and EOSCast [28] are two such contracts with Fake EOS Transfer vulnerability, and both have suffered significant loss.

As shown in Figure 1, the function body of the apply function in the EOSBet contract only checks whether the code is either the contract itself or *eosio.token* (line 5), but it does not check whether the code is *eosio.token* when the action is the identity of the transfer function. As a result, an attacker contract may send a transfer action directly to a victim contract to trigger its transfer function to process the action without checking whether the action comes from *eosio.token*.

2) ***Forged Transfer Notification****:* During a Forged Transfer Notification attack [26], the attacker controls two accounts: sender and notifier. The sender initializes the attack by transferring EOS tokens to the notifier via *eosio.token*. When the transfer is successful, both the sender and notifier will receive the transfer notification. However, the smart contract with the notifier as its account can deliberately forward the transfer action to a victim contract *C* with the *require_recipient*() function, which is a carbon copy of the transfer action.

As shown in Figure 2, if the contract *C* fails to check whether the destination (i.e., *data.to*) of an EOS transfer is itself within the function body of the transfer function, it may wrongly consider that it has received EOS from the sender rather than just receiving a notification. As a result, it may wrongly credit the sender, which in fact has sent nothing to *C*. The smart contract *C* is said to have the Forged Transfer Notification vulnerability [37].

```
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  }
9 }
```

Figure 2. The Smart Contract with Forged Transfer Notification Vulnerability

3) ***Block Information Dependency****:* A reliable source of randomness is hard to obtain in a blockchain platform. Developers may tempt to use the block information such as *tapos_block_prefix* and *tapos_block_num* to generate random numbers. In some cases, the generated random numbers may be used to determine the transfer of EOS token or the winner of a lottery. Unfortunately, both *tapos_block_prefix* and *tapos_block_num* are not reliable sources of randomness because they can be directly calculated from *ref_block_num*, which is the identity of the last irreversible block by default. Consider a gambling contract that uses a deferred action to determine the winner of a lottery. Suppose that the reference block to produce the randomness is the block right before the block making a bet. In such a case, when a smart contract uses *tapos_block_prefix* and *tapos_block_num* directly to generate a random number, the generated number can be predicted.

EOSRoyale [34] is an EOSIO smart contract with Block Information Dependency vulnerability. As shown in Figure 3, it uses the product of the block number and block prefix as the seed for random number generation (lines 7 to 9). Therefore, the variables used for random number generation can be computed before making the bet. As a result, the attackers successfully calculated the random number, won the game, and received the prizes in EOS tokens.

```
1   class EOSRoyale: public eosio::contract {
2   …
3   void rand() {
5   checksum256 result;
7   auto mixedBlock = tapos_block_prefix()*
    tapos_block_num();
8   const char * mixedChar = (const char *)(&mixedBlock);
9   sha256((char*)mixedChar, sizeof(mixedChar), &result);
10    }
11  }
```

Figure 3. The EOSRoyale Contract with Block Information Dependency Vulnerability

B. Ethereum Smart Contract Vulnerabilities

Vulnerabilities in Ethereum smart contracts have also led to severe financial losses. For instance, the vulnerability in the DAO contract [25] led to the loss of $60 million. The Freezing Ether and the Dangerous DelegateCall vulnerabilities resulted in a loss of $60 million and the frozen of $150 million in ether [49][50], respectively. Smart contract vulnerabilities in Ethereum [2][6][8][9][11][13][17][18][24] are extensively in existing works. In this section, we briefly revisit five common types of Ethereum smart contract vulnerabilities, which will used to evaluate the extensibility of WANA.

1) ***Greedy****:* A greedy smart contract can receive ethers, but it contains no functions to send ethers out. Therefore, this kind of smart contract will freeze all ethers sent to them. The second round of attack on the Parity wallet vulnerability [49] was due to the fact that many smart contracts only relied on the parity wallet library to manage their ethers through DelegateCall(). When the parity library was changed to a contract through initialization and then killed by the attacker, all the ethers within these wallet smart contracts relying on the parity library were frozen.

2) ***Dangerous DelegateCall****:* The DelegateCall is similar to a message call except that the code is executed with the data of the calling contract [38]. This is the way to implement the "library" feature in Solidity for code reuse. However, when the argument of the DelegateCall is *msg.data*, an attacker can manipulate the content of *msg.data* so that the attacker can make a victim contract to call a specific function. This vulnerability resulted in the outbreaks of the first round of the parity wallet attack [48], where each affected wallet

contract used a DelegateCall with *msg.data* as its parameter. As a result, an attacker could call any public function of the *_walletLibrary* library. Thus, the attacker first called the *initWallet* function of *_walletLibrary* and became the owner the wallet contract. Then, it sent the ethers in the wallet to the account of the attacker to finish the attack, which led to $30 million loss to the parity wallet users.

3) ***Block Information Dependency****:* The Block Information Dependency vulnerability exists when an Ethereum smart contract uses the block timestamp or block number to determine a critical operation (e.g., sending ethers or determining the winner of a lottery). Indeed, both the block timestamp and block number are variables that can be manipulated by miners. So, they cannot be used as reliable sources for critical operations. For example, a miner has the freedom to set the timestamp of a block within a small interval [40] in Ethereum. Therefore, if an Ethereum smart contract transfers ethers based on timestamp, an attacker can manipulate the block timestamp to exploit the vulnerability.

4) ***Mishandled Exception****:* The mishandled exception vulnerability is due to the fact that Solidity allows flexibility in exception handling, which is dependent on the way contracts calling one another. When a contract calls a function of another contract, the function may fail due to diverse types of exceptions. For a chain of nested calls where at least one call is made through low-level call methods on address (e.g., address.call()), the rollback of the transaction will only stop at the calling function of the low-level call and return a boolean value of false. From that point, no other side effect can be reverted and no further throw of exception will be propagated. Such inconsistencies in exception handling will make the calling contracts unaware of the errors happened during the nested function calls.

5) ***Reentrancy Vulnerability****:* The reentrancy bug is due to the fact that some of the functions are not designed to be reentrant by the developers. For example, the ether transfer and the update of corresponding variable in storage is not performed atomically within the function of the DAO contract [25]. However, a malicious contract can deliberately invoke such functions in a reentrant manner (e.g., through fallback functions). As a result, a vulnerable contract may lose ethers.

IV. The Design of WANA Framework

In this section, we present the design of WANA symbolic execution framework for smart contract vulnerability analysis.

A. The Workflow of WANA

We first introduce the workflow of WANA and highlight its extensible design. The input of WANA is the Wasm bytecode, which can be compiled from the source code of a smart contract or downloaded from deployed applications in Wasm. For example, the Wasm bytecode of EOSIO smart contract can be either collected from the EOSIO public blockchain directly or compiled from the EOSIO smart contract source code.

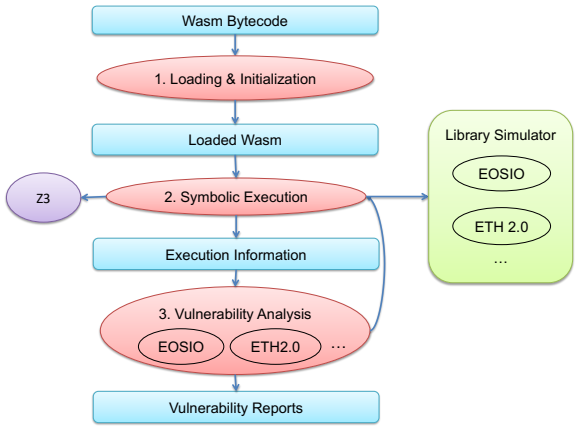

Figure 4 The Workflow of WANA

The workflow of WANA is shown in Figure 4. First, the Wasm bytecode is parsed, loaded, and initialized within the Wasm symbolic execution engine. As a result, the stack and memory of the symbolic execution engine are ready for execution. It then performs an iteratively process as outlined below: The symbolic execution engine starts to traverse the paths of the Wasm code with symbolic inputs. During symbolic execution, the symbolic execution engine will invoke the Z3 [15] constraint solver to check and prune unsatisfiable paths along the way. Meanwhile, the symbolic execution engine will also consult the library simulator when the Wasm bytecode is interacting with external functions. Then, the execution information useful for vulnerability analysis are collected during symbolic execution. Third, WANA will perform a vulnerability analysis based on the execution information collected during the symbolic execution process. When a vulnerability is detected, the corresponding report will be generated. Note that in the above iterative process, the vulnerability analysis and symbolic execution will interleave with each other. The whole workflow ends when all the code has been traversed.

The WANA tool is designed to be an extensible smart contract security analysis framework. Components for loading and initialization of Wasm code and that for the symbolic execution engine are general. The library simulator and the vulnerability simulator are platform specific. Different blockchain platforms will provide different external functions for the contracts to interact with. Similarly, the smart contracts on different platforms have different types of vulnerabilities, leading to different vulnerability analysis algorithms. When extending WANA to handle a new smart contract platform, software engineers need to simulate its library functions and define corresponding vulnerability analysis algorithms. Furthermore, when analyzing a Wasm module for different application domains (e.g., Web applications), software engineers can further extend WANA by simulating the corresponding library functions and designing customized vulnerability analysis algorithms.

B. The Symbolic Execution Engine

The symbolic execution engine of WANA is implemented as a new virtual machine that can evaluate both concrete and symbolic inputs according to the WASM specification. It has realized all instructions in Wasm specification version 1.0 [50]. Thus, WANA is more powerful than the previous symbolic execution engines in handling Wasm bytecodes.

Wasm datatypes: Wasm has four data types, including two integer data types and two floating-point data types under the IEEE 754 standard. The WANA symbolic execution engine supports ***all*** of them for analysis.

Wasm program architecture: To symbolic execute Wasm programs, the engine should follow how the functions and instructions are organized in these programs. More specifically, each EOSIO or Ethereum smart contract is compiled into a Wasm program. A Wasm program is organized as a module [39], which is the unit of deployment, loading, and compilation. In a module, the *exports* component defines a set of exported functions and data structures that other modules can access once the module has been instantiated. (In other words, these exported functions serve as the public interfaces for this Wasm module.) One of these exported functions is the entry function of the module. The entry functions for an EOSIO smart contract and an Ethereum smart contract are the *apply* function and the *main* function, respectively.

Wana Initialization: To start the symbolic execution on a module, WANA should first prepare an execution environment for it. When a module is loaded into the WANA Wasm VM (during the loading and initialization phase), the WANA symbolic execution engine prepares an *execution environment*, which includes the memories, various tables (including the function tables and symbol tables), global variables, and an execution stack to keep the execution contexts of the module.

Wana Execution: WANA is designed to iteratively performs symbolic executions on all exported functions. For each Wasm function invocation called by the instructions in the function body of each exported function, WANA first prepares a frame as its **execution context**, which includes arguments, local variables, return values, and references to its module. Then, it starts symbolically executing the instructions in the code section of the invoked function sequentially.

Among all Wasm instructions, there are four main kinds of instructions, including the numeric, memory, control, and function call instructions. We present how WANA handles them below.

Numeric instructions: WANA realizes the stack-based execution logic. The WANA symbolic execution engine first pops the operands, perform symbolic calculations, and push the result back to stack. *All* the numeric instructions on integer and floating-point value types are supported.

Control instructions: There are two kinds of branch instructions: unconditional branch (e.g., br) and conditional branch (e.g., br_if). (a) **Unconditonal branch**: In Wasm, executing an unconditional branch will directly jump to the label specified by the instruction argument to continue the execution, which is useful to realize loops. However, when the depth of the loop grows, the symbolic execution engine will become slow or trapped due to state explosion. Therefore, WANA sets an upper bound on the maximum depth of nested loop for each label during its path exploration. (b) **Conditonal branch:** For the instruction br_if, the execution will depend on the evaluation result of the expression of the instruction. WANA will record the execution context for each branch and traverse individual branches to realize the path coverage. For each branch, the corresponding path constraint will be input to z3 [15] for constraint solving before traversing, which can improve the execution efficiency by pruning those infeasible paths. Like the above, WANA also sets an upper bound on the maximum depth of nested loop for each label during its path exploration.

Memory and Memory instructions: In Wasm, there are three types of memory manipulation instructions: *load*, *store*, and *increasing the size of the memory*. Wasm adopts a linear memory model with a contiguous, mutable array of raw bytes, which also conforms to the Wasm specification [50]. A module can load and store values from/to a linear memory at any byte address. Since there are many bit manipulation instructions on the integer and floating-point data types in Wasm, WANA is designed to represent the integer and float data types with the bit-vector data type in z3. WANA uses 32-bit (64) bit-vector data types to represent 32-bit (64-bit) integer and float-point symbolic data types. Since a byte array cannot store a bit-vector or its expression, WANA uses a linear list to store the references of both concrete and symbolic values of bit vectors to emulate the memory of the module.

During symbolic execution, WANA will insert a key-value pair ⟨address, value⟩ into a dictionary when performing a *store* operation. During a *load* operation, if the symbolic memory address exists in the dictionary, WANA will fetch the corresponding value from the dictionary directly. Otherwise, WANA will randomly choose a memory address from the linear list and return its value. The symbolic memory address will be bound to a randomly chosen memory address thereafter.

Function call: In Wasm, there are two types of function calls: *direct* and *indirect*. For indirect function calls, the WANA symbolic execution engine first gets an index from the top frame of the execution stack. Then the symbolic execution engine uses the index to get the concrete function address from the function table. For the functions in the current Wasm module, WANA directly steps into the functions to continue the symbolic execution. For the library functions of the blockchain platform, WANA emulates their behaviors in terms of their effects on symbolic execution, which we discuss in the subsection below.

C. Handling the Library Functions

A smart contract may invoke various library functions provided by the underlying blockchain platform. WANA handle these library function calls based on their effects and simulate their behaviors accordingly. Since the library functions are platform-dependent, WANA simulates different library function behaviors for different platforms. When extending WANA to a new platform, its corresponding library functions should be simulated.

1) Handling Library Functions for EOSIO: WANA classifies the libraries of the EOSIO platform into four kinds: assertion functions, output related functions, input related functions, and memory manipulation functions.

To handle an assertion function (e.g., eosio_assert, eosio_assert_code), WANA will continue the symbolic execution only if the predicate of the assertion is evaluated to true. Otherwise, the execution will terminate. Therefore, WANA adds a path constraint where the assertion predicate is true to continue the symbolic execution.

For output related functions (e.g., printf, printi, printsf), they have no effect on the control flow or data flow of the symbolic execution. Therefore, WANA will just pop their arguments from the stack and continue the symbolic execution.

For input related functions, they may change the internal memory state of the smart contracts through different forms of input values. These functions include the reading information from the blocks of the underlying blockchain (e.g., current_time, tapos_block_num), these reading information from an account for authentication (e.g., require_auth, check_transaction_authorization), and the these updating the memory based on the information of external tables (e.g., db_get_i64). Since the values from these input-related functions may impact the execution path, WANA will return new symbolic values for these variables to continue the symbolic execution.

For memory manipulation functions (e.g., multi3, addtf3, memcpy, memmove), WANA will perform the symbolic execution on them to generate function summaries and reuse the summaries for compositional symbolic execution.

2) Handling Library Functions for Etherenum: The library functions for Ethereum are also classified into several types and handled separately by WANA.

For output related functions (e.g., eth.log), WANA will just pop their arguments from the execution stack and continue the symbolic execution because these functions have no impact on the control flow or data flow of the symbolic execution. One except is the *eth.storageStore* function, which will store data to persistent storage. WANA will keep all the data that have been persisted by *eth.storageStore* in a list during the symbolic execution. The list will be used for detecting the reentrancy bugs.

For input related functions, they may return values directly (e.g., eth.getGasLeft, eth.callDelegate, eth.getReturnDataSize, eth.call, eth.getBlockTimestamp, eth.getBlockNumber) or they may return values by writing the shared memory during their execution (e.g., eth.callDataCopy, eth.getCallValue, eth.getCaller, eth.getExternalBalance). WANA will return new symbolic value for them.

For some complex and frequently invoked functions (e.g., $lt_320x320_64, $lt_512x512_64, $lt_256x256_64), WANA will perform the symbolic execution on them to generate function summaries. Then WANA will reuse the function summaries to speed up the symbolic execution of the smart contracts.

Finally, for cryptographic functions (e.g., keccak256), performing the symbolic execution with constraint solver will lead to a significant performance degrade. Therefore, WANA chooses to return a new symbolic value when handling such functions. Meanwhile, WANA will store the ⟨input, output⟩ symbolic value pair in a dictionary to record their relationships, which can be used to track data dependency for vulnerability detection.

D. The Vulnerability Detection Algorithm

In this section, we will present the vulnerability detection algorithms for EOSIO and Ethereum smart contracts. In general, the vulnerability detection algorithms are defined within an independent module in a separate file. The interface between the symbolic execution engine and the vulnerability detection module is well defined. The vulnerability detection algorithms will be regularly triggered through the interface during symbolic execution. The global variables tracking the executions states related to vulnerability detection are also managed uniformly. Therefore, the vulnerability detection module can be easily extended to support the detection of new vulnerabilities. Furthermore, when extending WANA to a new platform, it is also important to extend the vulnerability detection module to support the detection of vulnerabilities specific to the platform.

```
1  bool isFakeEOSTransfer() {
2    if apply function is reachable
3      if the apply function have 3 i64 parameters
5        find all feasible paths P in apply function where code is
   neither equal to eosio.token nor receiver
6        label the code invoking transfer function as sink
7        if there is any path in P reaching sink
8          return true
9    return false
```

Figure 5. Algorithms Detecting Fake EOS Transfer

1) Vulnerability Detection for EOSIO Smart Contracts:

a) Fake EOS Transfer: In general, to detect the Fake EOS Transfer vulnerability, WANA checks whether there is a feasible path with code different from *eosio.token* that finally invokes the transfer function within the *apply* function.

The algorithm to detect Fake EOS Transfer is shown in Figure 5. At line 2, WANA first checks whether the *apply* function is reachable. Then, WANA further checks whether the *apply* function contains three 64-bit integer parameters at line 3. The three 64-bit integer parameters correspond to the *receiver*, *code*, and *action* parameters. In lines 5, WANA performs the symbolic execution within the *apply* function to find all feasible paths *P* where *code* is neither equal to *eosio.token* nor *receiver*. In line 6, WANA labels the Wasm code that invokes the *transfer* function as a sink. Finally, it considers the contracts vulnerable if there are a path in *P* that can reach the sink (line 7). Otherwise, the contract is not reported as vulnerable (line 8 to 9).

```
1   ...
2   block $B2
3     local.get $p1 //$p1 is code
4     //6138663591592764928 is i64 encoding of
    eosio.token
5     i64.const 6138663591592764928 //eosio.token
6     i64.ne
7     br_if $B2  //jump if not equal to eosio.token
8     // branch where code equals to eosio.token
9     i32.const 0
10    local.set $l4
11    br $B1
12  end
```

Figure 6. WAST Code Comparing Code and eosio.token

WANA performs the analysis at the bytecode level. For ease of presentation, in the sequel of this section, we use the Wasm text format (Wast) to describe our analysis algorithm.

```
1   ...
2   local.get $p2
3   //-3617168760277827584 is the i64 encoding of transfer
4   i64.const -3617168760277827584
5   i64.ne
6   br_if $B10  //jump if not equal to transfer
7   //equal to transfer, so preparing the invocation of transfer
8   local.get $l3
9   i32.const 0
10  ...
11  call $f162 //invoking transfer
```

Figure 7. WAST Code Comparing Action and Invoking Transfer

One possible Wast code pattern to compare the code parameter with *eosio.token* is shown in Figure 6. At line 3, the wast code pushes $p1 (which represents the *code* parameter) to the stack. It further pushes the 64-bit integer corresponding to *eosio.token* to the stack (line 5). Next, the code and *eosio.token* are compared at line 6. Line 7 represents the control flow branch when they are unequal and lines 9 to 11 represent the control flow branch when they are equal. The branch for *code* not equal to *eosio.token* can be automatically identified during the symbolic execution by adding the path constraint to the symbolic execution at the start of the branch: $p1 != encoding("eosio.token").

The algorithm to identify the invocation of the *transfer* function is shown in Figure 7. At lines 2 to 5, the algorithm compares the *action* parameter with *transfer*. Note that "$p2" represents the *action* parameter and the 64-bit integer at line 4 represents *transfer*. If *action* equals to *transfer* (line 7), the algorithm will prepare for function call and call the *transfer* function (line 8 to 11). When detecting the Fake EOS Transfer vulnerability, WANA will use the code pattern in Figure 7 to locate the code invoking the *transfer* function (i.e., a sink).

b) Forged Transfer Notification: When a smart contract receives a *transfer* notification, if it fails to check the destination (i.e., data.to) of the EOS transfer is itself within its *transfer* function, a Forged Transfer Notification occurs.

The algorithm to detect Forged Transfer Notification is shown in Figure 8. At line 2, WANA first performs a concrete execution on the *apply* function with both *action* and *code* parameters are fixed to *transfer* and *eosio.token*, respectively. The goal of the concrete execution is to locate the position of the transfer function. During the concrete execution, if the Wasm instruction *call_indirect* is executed, WANA will consider the *transfer* function reachable (line 3). This is because the *call_indirect* instruction is used in the Wasm bytecode of any EOSIO smart contract to invoke the ABI functions where the parameter of *action* is fixed to *transfer*. The *call_indirect* instruction will pop a 32-bit integer as an *index* of a table storing the function addresses to get the address of the *transfer* function in the Wasm module. Then, WANA records the index of the *transfer* function in the Wasm module for follow-up analysis (line 4).

```
1   bool isForgedTransferNotification() {
2       concrete execution of apply with action fixed to transfer and
        code equals to eosio.token.
3       if transfer function is reachable:
4         label the position pt of transfer function in Wasm
5         label the variables or stack storing  self in transfer
6         label the Wasm comparing  self and to in transfer as C
7         symbolically executing the transfer function at pt
8         if there is any path not reaching C
9             return true
10        else
11            return false
```

Figure 8. Algorithms Detecting Forged Transfer Notification

From lines 5 to 11, the WANA analyzes the Wasm bytecode of the *transfer* function. At line 5, WANA first labels the set of variables storing _self, which is usually read from $p0 (the first parameter of *transfer*) and stored in local variables or the stack. Then, it continues to label the Wasm bytecode comparing _self (in local variables or stack) with "to" as the label C (line 6). Note that the "to" can be read from $p2 (the third parameter of transfer) or from the action data returned by unpack_action_data(). Finally, WANA performs the symbolic execution within the *transfer* function to traverse all the paths starting from pt (line 7). If there is any path not reaching C during the symbolic execution, the smart contract is considered containing the Forged Transfer Notification vulnerability (line 8 and 9). Otherwise, the smart contract is considered not vulnerable (line 11).

c) Block Information Dependency: The algorithm to detect the Block Information Dependency vulnerability is relatively straightforward, and we will use the Wast representation for illustration. WANA first checks whether there is any invocation of the $env.tapos_block_prefix or $env.tapos_block_num functions to get the block information through static analysis. Then, WANA further checks whether there are invocations to the $env.send_inline or $env.send_deferred functions to send EOS tokens within the smart contract under analysis. Both the positions for collecting the block information and for sending EOS tokens are labelled during analysis. Finally, WANA performs the

symbolic execution to check whether there is at least one feasible path from the block information collection to the *transfer* function of EOS. If there is one or more such feasible paths, the smart contract is reported to have the Block Information Dependency vulnerability.

2) Vulnerability Detection for Ethereum Smart Contracts: In this section, we will present the algorithms to detect the Greedy, Dangerous DelegateCall, Block Information Dependency, Mishandled Exception, and Reentrancy vulnerabilities in Ethereum smart contracts. For ease of understanding, we will also use the function names in the Wast representation to describe the vulnerability detection algorithm in this section. In Wasm, each function name corresponds to a 32-bit integer. The mapping between the function name and the 32-bit integer can be collected when the smart contract is loading into the symbolic execution engine.

a) Greedy: If an Ethereum smart contract can receive ethers but cannot send any ethers, it is considered greedy. A greedy smart contract will freeze any ether it received.

To check whether an Ethereum smart contract can receive ether, WANA checks whether there is at least one payable function reachable from the main function. If a function is not payable, the Wasm code of the smart contract generated by solc compiler will invoke an internal function called $callvalue and check whether its return value is 0. If it is not 0 (i.e., it has received ether), it will revert. Otherwise, it will continue execution. Therefore, if there is such check in the Wasm code, WANA will consider the function as not payable. Otherwise, the function is payable.

To check whether a smart contract cannot send any ether, WANA first checks whether there are any $ethereum.call functions within the bytecode of the smart contract. If there is no any $ethereum.call function within the bytecode of the smart contract, the contract cannot send any ether. However, if there are some $ethereum.call functions, WANA will further check whether there is any feasible path from the main function to any of the $ethereum.call functions through symbolic execution. If there is no feasible path to any of them, the smart contract also cannot send any ether.

b) Dangerous DelegateCall: To detect the Dangerous DelegateCall vulnerability, WANA checks whether the invocation of DelegateCall() function (i.e., $ethereum.callDelegate in the Wast) is reachable from the entry function and whether the input parameter specifying the function called by DelegateCall() is manipulatable by an attacking contract. To determine whether the function invoked is manipulatable, WANA checks whether the argument of $ethereum.callDelegate is a constant value. If it is not a constant value, a vulnerability is detected.

c) Block Information Dependency: The detection of the Block Information Dependency vulnerability for Ethereum smart contracts is similar to that of EOSIO smart contracts. WANA also checks the existence of at least one feasible path from the block information collection to the callsite of ether transfer based on symbolic execution. The only difference lies in the implementation details. In Ethereum, the block information collection functions are $ethereum.getBlockNumber, $ethereum.getBlockHash, and $ethereum.getBlockTimestamp andthe ether transfer function is $ethereum.call.

d) Mishandled Exception: To detect the Mishandled Exception vulnerability, WANA analyzes whether the smart contract under analysis has checked the return value against zero after invoking other contracts. If there are no such checks, WANA will report the mishandled exception vulnerability.

During symbolic execution, whenever the CALL instruction is encountered, WANA will first check whether its arguments contain $ethereum.call or $ethereum.callCode. If it is a call in Ethereum smart contracts, WANA will return a symbolic value corresponding to the CALL. Meanwhile, WANA will record the symbolic value and the instruction count of the CALL into a dictionary. During symbolic execution, when an if instruction is encountered, WANA will check whether there is a condition comparing the symbolic value and zero within the path constraints. If there is such a condition in the path constraint, the return value of the CALL is handled properly and WANA will remove the symbolic value from the dictionary. When the symbolic execution ends, if there are still any symbolic values in the dictionary, then there must be some unhandled CALL return values. Therefore, WANA will report the mishandled exception vulnerability.

e) Reentrancy Vulnerability: To detect the Reentrancy vulnerability, WANA uses symbolic values to represent the data in storage retrieved by $ethereum.storageLoad. During symbolic execution, before the $ethereum.call is executed, WANA will retrieve the symbolic values for the data in storage under the current path constraints. If these symbolic values have not been persisted by $ethereum.storageStore, a reentrancy vulnerability is detected to reflect the fact that an attacker can exploit it to invoke $ethereum.call more than once.

E. Discussions on Threats to Validity

WANA may lead to false positives or false negatives due to the follow reasons. Firstly, WANA currently realizes part of the library functions for input manipulation. Some of the cryptographic functions are not precisely simulated because they are intrinsically hard to process by the SMT solvers. Second, the vulnerability detection algorithms of WANA are based on the manual summarized vulnerability patterns, which may not be comprehensive due to corner cases. Third, WANA performs symbolic execution on the Wasm bytecode. The Solidity to Wasm compilers or EVM to Wasm transcompilers are still under development by the Ethereum community. Therefore, a complete support of Ethereum platform still needs some time. Finally, WANA returns new symbolic value when dealing with some library functions, which may lead to false positives.

V. Experiment on EOSIO Smart Contract Vulnerability Detection

In this section, we will present our experiment on EOSIO smart contract detection with WANA.

A. Research Questions

RQ1: Is WANA effective to detect the vulnerabilities within smart contracts?

RQ2: Is WANA efficient to detect the vulnerabilities within smart contracts?

B. Subject Programs

For EOSIO smart contracts, we collected 83 smart contracts with source code and 65202 smart contracts without source code for evaluation. The 83 smart contracts were downloaded from GitHub. And the 65202 smart contracts were downloaded from EOSPark [29]. The 65202 smart contracts correspond to 6453 distinct EOSIO accounts. As discussed in previous sections, the input of WANA tool is WASM bytecode. For EOSIO smart contracts with source code, they can be compiled into WASM for analysis with WANA. For EOSIO smart contracts in WASM bytecode, they can be analyzed by WANA directly.

C. Experiment Setup

We used a desktop PC as our experiment environment. The PC was running Ubuntu 18.04 and was equipped with Intel i7-6700 8-core CPU and 16GB of memory. The WANA tool was implemented by Python3 and we used the official Python release version 3.7. Finally, WANA used the z3 version 4.8.0 as the constraint solver. The default loop depth of WANA was set as 10 in the experiment.

D. Effectiveness of WANA

In this section, we want to evaluate the vulnerability detection effectiveness of WANA. We first compared the WANA tool and the EVulHunter tool on 83 EOSIO smart contracts with source code. Then we further analyzed 65202 EOSIO smart contracts without source code using WANA to estimate the percentage of vulnerabilities in the wild.

Table 1. Comparison of WANA and EVulHunter

Vulnerability	Total	WANA			EVulHunter		
		Reported	FP	FN	Reported	FP	FN
Forged Transfer Notification	83	5	0	0	12	10	3
Fake EOS Transfer	83	2	0	0	9	8	1
Block Information Dependency	83	3	0	0	\	\	\

Results on smart contracts with source code. The vulnerability detection results on EOSIO smart contracts with source code are shown in Table 1. We can see that among the 83 smart contracts, WANA reported 5 smart contracts with Forged Transfer Notification, 2 smart contracts with Fake EOS Transfer, and 3 smart contracts with Block Information Dependency. After manually checking the source code of the smart contracts, we confirmed that there were no false positives or false negatives for the 3 types of vulnerabilities by WANA in these 83 contracts. Therefore, WANA is effective to detect vulnerabilities in EOSIO smart contracts with high accuracy in the experiment.

We further compare WANA with the EVulHunter [20] tool for vulnerability detection. Since the EVulHunter tool does not the support the detection of Block Information Dependency vulnerability, we compare WANA and EVulHunter in terms of the Forged Transfer Notification and Fake EOS Transfer vulnerability only.

Table 2. Vulnerability Detection Effectiveness on EOSIO Smart Contracts without Source Code

Vulnerability	Total	Vulnerabilities Detected	Percentage
Block Information Dependency	65202	37	0.056%
Forged Transfer Notification	65202	7164	10.92%
Fake EOS Transfer	65202	5394	8.22%

Among the 83 smart contracts, the EVulHunter successfully analyzed 74 smart contracts. It fails to generate output for the other 9 contracts. Within the 74 smart contracts, The EVulHunter has reported 12 smart contracts with Forged Transfer Notification vulnerability. However, after manual check, we found 10 of them are false positives. Furthermore, EVulHunter missed 3 vulnerable smart contracts. For the Fake EOS Transfer Vulnerability, WANA identified 2 vulnerabilities while the EVulHunter tool reported 9 vulnerabilities. However, after manual check, we found EVulHunter generates 8 false positives and 1 false negative. Therefore, WANA in general incurs much lower false positives and false negatives than EVulHunter.

Results on smart contracts without source code. We further evaluated WANA on smart contracts without source code, the results are shown in Table 2. We can see that among the 65202 smart contracts, WANA has detected 37 Block Information Dependency vulnerability, 7164 Forged Transfer Notification vulnerabilities, and 5394 Fake EOS Transfer vulnerabilities. Among the 65202 smart contracts, the percentage of Block Information Dependency, Forged Transfer Notification, and Fake EOS Transfer are 0.056%, 10.92%, and 8.22%, respectively.

It is hard to manually verify those reported vulnerable smart contracts to calculate the precise number of false positives and false negatives. But the results give us an estimation of the number of vulnerable smart contracts deployed in EOSIO platform in the wild. Given the relatively large percentages of Forged Transfer Notification and Fake EOS Transfer vulnerabilities, the developers are strongly recommended to perform security checks and harden their contract before releasing their EOSIO smart contracts.

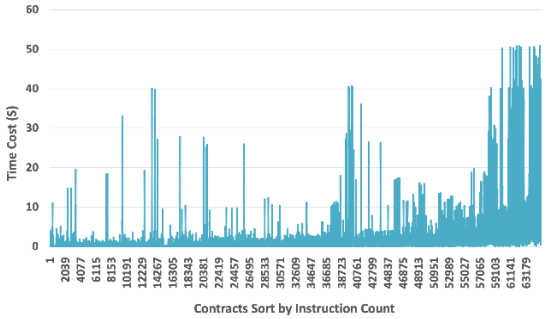

Figure 9. Smart Contract Analysis Time of WANA

E. *Efficiency of WANA*

In this section, we further analyze the vulnerability detection efficiency of WANA to answer RQ2. We set the loop depth of WANA is as 10 (default value). The smart contract analysis times with respect to the instruction count is shown in Figure 9.

We sort the smart contracts in ascending order of their instructions counts (i.e., bytecode size). The x-axis represents the 65202 smart contracts ordered in ascending order of their instruction counts and the y-axis represents the analysis time for the corresponding smart contract in seconds. Across the smart contracts, we can see that the smart contract analysis times range from 0.1 second to 51 seconds, which are still practical. In general, when the sizes of the smart contracts grow, the analysis time tend to grow accordingly. But the trend is not strictly linearly increasing because the analysis time is also affected by program structure. The mean smart contract analysis time for EOSIO smart contract is 1.59 second. We further increased the loop count to 20 and 50, and the smart contract analysis times were all within 90 seconds. Therefore, WANA is efficient to perform vulnerability analysis on EOSIO smart contract for practical use.

VI. Case Studies on Ethereum Smart Contracts Vulnerability Detection

In this section, we perform a case study to evaluate the extensibility of WANA by extending it to detect the vulnerabilities of Ethereum smart contracts.

At the time of writing, the official support of Wasm in Ethereum (i.e., the EWasm) is still under development due to the slow transition to Ethereum 2.0. As a result, there is no official toolchain to generate Wasm bytecode from Solidity or from EVM bytecode right now. But there are indeed ongoing projects working on Solidity-to-Wasm compilers such as SOLL [46], solang [43] and solc. And the solc [45] is the official compiler from the Ethereum community. After manual checking, we chose solc because it supports more language features. In our experiment, we use the solc version 0.7.5.

However, since the full support of EWasm is still under development, even the solc compiler still does not support all language features of smart contracts. Therefore, most of the Ethereum smart contracts cannot be successfully compiled into EWasm yet. In our experiment, we have to manually modify the Ethereum smart contracts by removing unsupported features such that they can be supported by solc (without affecting their semantics and vulnerable behavior). Then we used these modified smart contracts to evaluate WANA. Due to the large manual effort to modify those unsupported features, we cannot perform large scale experiment on Ethereum smart contract yet, which is only feasible when the transition to ETH 2.0 is done.

A. *Ethereum Smart Contracts Used in Case Study*

We chose Ethereum smart contracts with 5 typical vulnerabilities supported by WANA to perform the case study: Greedy, Dangerous DelegateCall, Block Information Dependency, Reentrancy, and Mishandled Exception. To make them usable with the version of solc at the time of writing (version 0.7.5), we manually modified the source code of each smart contract to remove unsupported syntax features. In total, we modified 89 Ethereum smart contracts from the vulnerable smart contract dataset [36]. Note that all these smart contracts are manually confirmed as vulnerable.

B. *Setup of the Case Study*

We used the same desktop as in the experiment for EOSIO smart contract analysis to perform the case study. The desktop was installed with Ubuntu 18.04 as operating system. We used the SOLL version 0.0.5 to compile Ethereum smart contract to EWasm. And SOLL depended on LLVM version 8.0.

C. *Results and Analysis*

The vulnerability detection results on Ethereum smart contracts are shown in Table 3. We can see that WANA has successfully identified all vulnerable smart contracts for each type of vulnerability. WANA produced no false negative cases in the case study. Since all smart contracts used in the experiment are vulnerable as confirmed by manual inspection, the false positive rate is not applicable.

We have also measured the efficiency of WANA on Ethereum smart contract. The smart contract analysis times ranged from 0.01 second to 0.6 second and the average smart contract analysis time was 0.08 second per smart contract. The results show that WANA is also efficient for analyzing Ethereum smart contracts.

Table 3. Vulnerability Detection Results on Ethereum Smart Contract

Vulnerability	WANA	
	Total	Reported
Greedy	22	22
Dangerous DelegateCall	20	20
Block Information Dependency	21	21
Reentrancy	9	9
Mishandled Exception	17	17
Total	89	89

VII. Related Work

In this section, we present closely related work on smart contract vulnerability detection.

Atzei et al. [2] performed a comprehensive survey on vulnerabilities and attacks on Ethereum smart contracts. Parizi et al. [19] performed an experimental evaluation of current Ethereum smart contracts security testing tools. Chen et al. [5] proposed the TokenScope tool, which could analyze the transaction traces from Ethereum to automatically check whether the behaviors of the token contracts are consistent with the ERC-20 standards.

There are several works on the formal verification of smart contracts. Abdellatif and Brousmiche [1] proposed a formal modeling approach to verify the behavior of smart contract in its execution environment. They further performed security analysis on the smart contracts with a statistical model checking approach. Hirai [8] used Isabelle/HOL tool to verify the Ethereum smart contract Deed.

Fuzzing is also an effective approach for vulnerability detection. Jiang et al. proposed ContractFuzzer [9], a black-box fuzzer for detecting vulnerabilities in Ethereum smart contracts. They also proposed the test oracles for detecting 7 typical Ethereum smart contract vulnerabilities. Nguyen et al. proposed a grey-box fuzzing tool called sFuzz [17] for smart contract vulnerability detection. The sFuzz tool adopts a lightweight multi-objective adaptive strategy to cover those hard-to-cover branches.

Symbolic execution [3][10] is a popular technique for vulnerability detection. There are several symbolic execution tools for security analysis of Ethereum smart contracts. Luu et al [13] proposed the Oyente symbolic verification tool for Ethereum smart contract. Oyente first builds the control-flow graph of smart contracts and then performs symbolic execution on the graph to check code patterns corresponding to smart contract vulnerabilities. Nikolic et al. [18] designed MAIAN, another symbolic execution tool to analyze the execution traces of Ethereum smart contracts. MAIAN can detect the greedy, the prodigal and the suicidal contracts in a scalable manner. Tsankov et al. proposed the Securify tool to detect Ethereum smart contract vulnerabilities. The Securify tool [22] first symbolically analyzes the contract's dependency graph to extract precise semantic information from the code. Then, it checks compliance and violation patterns that capture sufficient conditions for proving if a property holds or not.

Jhannes Krupp [11] proposed TEETHER, a tool that automatically generates an exploit for an Ethereum smart contract given only its binary bytecode through symbolic execution. ConsenSys proposed the Mythril [6] security analysis tool for EVM bytecode. Mythril used symbolic execution, SMT solving and taint analysis detect a variety of security vulnerabilities. Trailofbits proposed the Manticore [23] symbolic execution tool for dynamic binary analysis of smart contracts and Linux ELF binaries. Manticore combined symbolic execution, taint analysis, and `instrumentation to analyze binaries. After checking the source code of Manticore, we found Manticore can support the symbolic execution of both Ethereum and EOSIO smart contracts. However, Manticore essentially integrates two independent symbolic execution engines, one for EVM bytecode and one for Wasm. In comparison, WANA uses a uniform symbolic execution engine for WASM to support the analysis of contracts on both platforms. Moreover, Manticore still have not supported the vulnerability detection of EOSIO smart contracts at the time of writing, which makes it infeasible to compare Manticore and WANA for smart contracts vulnerability detection.

Wang et al. [24] proposed the VULTRON tool based on the observation that almost all the existing transaction-related vulnerabilities were due to the mismatch between the actual transferred amount and the amount reflected on the contract's internal bookkeeping. The VULTRON tool provided a general test oracle that can be used to drive a range of smart contract analysis techniques.

EOSafe [7] is a static analysis framework to detect vulnerabilities within EOSIO smart contracts based on symbolic execution on WASM bytecode. WANA is different from EOSafe in two aspects. First, WANA is designed to be easily extensible to other smart contract platforms. Second, the symbolic execution engine of WANA is implemented as a new virtual machine that can execute both concrete and symbolic values while the symbolic execution engine of EOSafe is based on static symbolic execution. Wasabi [12] is a general-purpose dynamic analysis framework for Wasm. It adopts binary instrumentation to insert calls to analysis functions written in JavaScript into a Wasm binary. With Wasabi, dynamic analyses such as instruction counting, call graph extraction, memory access tracing, and taint analysis can be realized easily.

VIII. CONCLUSIONS AND FUTURE WORK

The vulnerabilities within smart contract have caused financial loss to its end users. In this work, we proposed WANA, a general symbolic execution engine for Wasm bytecode and an extensible smart contract vulnerability detection tool. The experiment on EOSIO smart contracts shows that WANA is effective and efficient to detect smart contract vulnerabilities. Furthermore, the case study on Ethereum shows that WANA is also extensible to other smart contract platforms for vulnerabilities detection.

The hardware-, language-, and platform-independent nature of Wasm also makes it popular for use in other environments. For future work, we plan to extend WANA to support the vulnerability detection of other applications targeting at Wasm.

IX. ACKNOWLEDGEMENTS

This research is supported in part by the National Key R&D Program of China under Grant 2019YFB2102400, NSFC (project no. 61772056), Innovative Technology Fund of HKSAR (project no. 9440226) and CityU MF_EXT (project no. 9678180).

REFERENCES

[1] T. Abdellatif, K.L. Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. *In Proceedings of IFIP NTMS International Workshop on Blockchains and Smart Contracts (BSC)*, Paris, France, 2018.

[2] N. Atzei, M. Bartoletti and T. Cimoli. A survey of attacks on ethereum smart contracts. *International Conference on Principles of Security and Trust.* Springer, Berlin, pages 164-186, 2017.

[3] R. S. Boyer, B. Elspas, K N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6): 234-245, 1975.

[4] V. Buterin. A next-generation smart contract and decentralized application platform. white paper, 3:37, 2014.

[5] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. *In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, New York, NY, USA, 1503–1520, 2019.

[6] ConsenSys. Mythril: Security analysis tool for ethereum smart contracts. https://github.com/ ConsenSys/mythril. Last access, 2021.

[7] N. He, R. Zhang, L. Wu, H. Wang, X. Luo, Y. Guo, T. Yu, X. Jiang. "Security Analysis of EOSIO Smart Contracts." ArXiv abs/ 2003.06568, 2020.

[8] Y. Hirai. Formal verification of Deed contract in Ethereum name service. http://yoichihirai.com/deed.pdf, 2016.

[9] B. Jiang, Y. Liu, and W.K. Chan, ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection, in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering* (ASE 2018), Montpellier, France, September 2018.

[10] James C. King. A new approach to program testing. *In Proceedings of the international conference on Reliable software*. Association for Computing Machinery, New York, NY, USA, 228–233, 1975.

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

[12] D. Lehmann and M. Pradel. Wasabi: A Framework for Dynamically Analyzing Wasm. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19). Association for Computing Machinery, New York, NY, USA, 1045–1058, 2019.

[13] L. Luu, D.H. Chu, H. Olickel, P. Saxena, A. Hobor. Making Smart Contracts Smarter. *In Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, pp:254-269, Vienna, Austria, 2016.

[14] T. McCallum. Diving into Ethereum's Virtual Machine(EVM): the future of Ewasm. https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy. Last access, 2021.

[15] Microsoft Corporation. The Z3 theorem prover. https://github.com/Z3Prover/z3.

[16] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[17] D. T. Nguyen, L. H. Pham, J. Sun, Y. Lin and M. Q. Tran. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts, *In Proceedings of the 42nd International Conference on Software Engineering*. IEEE Computer Society, 2020.

[18] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, A. Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. https://arxiv.org/pdf/1802.06038, 2018.

[19] R. M. Parizi, A. Dehghantanha., K. K. R. Choo, and A. Singh. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. *In Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering* (pp. 103-113). IBM Corp, 2018.

[20] Lijin Quan, Lei Wu and Haoyu Wang. EVulHunter: Detecting Fake Transfer Vulnerabilities for EOSIO's Smart Contracts at Wasm-level. ArXiv abs/1906.10362, 2019.

[21] C. Schwarz. Ethereum 2.0: A Complete Guide. Ewasm. https://medium.com/chainsafe-systems/ethereum-2-0-a-complete-guide-ewasm-394cac756baf

[22] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical Security Analysis of Smart Contracts. *In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 67–82, 2018. DOI:https://doi.org/10.1145/3243734.3243780

[23] Trailofbits. Manticore. https://github.com/ trailofbits/manticore, 2017.

[24] H. Wang, Y. Li, S. Lin, L. Ma, and Y. Liu."VULTRON: Catching Vulnerable Smart Contracts Once and for All," *In Proceedings of IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Montreal, QC, Canada, 2019, pp. 1-4.EOS.IO Technical White Paper, 2018.

[25] Analysis of the DAO exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. Last access, 2021.

[26] Attack on EOS Bet. https://medium.com/leclevietnam/hacking-in-eos-contracts-and-how-to-prevent-it-b8663c8bffa6. Last access, 2021.

[27] EOSBet. https://eosbet.io. Last access, 2021.

[28] EOSCast. https://www.eoscast.cc. Last access, 2021.

[29] EOS Jungle. https://monitor.jungletestnet.io/. Last access, 2021.

[30] EOS Park – EOS data service provider. https://eospark.com/.

[31] EOSIO.CDT. https://eos.io/build-on-eosio/eosio-cdt/. Last access, 2021.

[32] EOSIO Platform. https://eos.io/. Last access, 2021.

[33] EOSIO.token. https://developers.eos.io/welcome/latest/getting-started/smart-contract-development/deploy-issue-and-transfer-tokens. Last access, 2021.

[34] EOSRoyale Smart Contract. https://www.eosroyale.com, Last access, 2021.

[35] EOS-VM. https://eos.io/build-on-eosio/eos-vm/. Last access, 2021.

[36] Ethereum Vulnerable Smart Contract Benchmark. https://github.com/gongbell/ContractFuzzer/tree/master/examples. Last access, 2021.

[37] Forged Transfer Notification in EOS smart contracts. https://blog.peckshield.com/2018/10/26/eos/. Last access, 2021.

[38] Introduction to Smart Contracts. http://solidity.readthedocs.io/en/v0.4.21/introduction-to-smart-contracts.html. Last access, 2021.

[39] Modules. https://Wasm.github.io/spec/core/syntax/modules.html#syntax-module. Last access, 2021.

[40] Polkadot Platform. https://wiki.polkadot.network/docs/en/getting-started. Last access, 2021.

[41] Predicting Random Members in Ethereum Smart Contracts. https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620. Last access, 2021.

[42] Smart Contract Development on EOS. https://developers.eos.io/welcome/latest/getting-started/index.

[43] Solang–a Solidity to wasm compiler written in rust. https://github.com/hyperledger-labs/solang. Last access, 2021.

[44] Solidity. https://solidity.readthedocs.io/en/v0.6.10/introduction-to-smart-contracts.html. Last access, 2021.

[45] Solc compiler. https://github.com/ethereum/solidity/releases/tag/v0.7.5.

[46] SOLL. https://github.com/second-state/soll. Last access, 2021.

[47] System Smart Contracts on EOSIO. https://developers.eos.io/manuals/eosio.contracts/latest/index. Last access, 2021.

[48] The Parity Wallet Hack Explained. https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7. Last access, 2021.

[49] The Wallet Smart Contract Frozen by the Parity Bug. https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol. Last access, 2021.

[50] Wasm. https://Wasm.org/. Last access, 2021.

[51] Wasm Core Concepts. https://Wasm.github.io/spec/core/intro/overview.html#concepts. Last access, 2021.