Automatically Fixing Vulnerabilities in WebAssembly

Yubin Hu[1]

***Abstract*— With the widespread acceptance of blockchain and the Internet of Things, the world of technology seems to be converging once again. However, there are a large number of data leaks in IoT that jeopardize the development of IoT. And blockchain can help IoT solve these problems very well due to its own characteristics. However smart contracts on the blockchain can also have some vulnerabilities and developers are continuing to fix the code. We hope that we can use this part of the information to fix the vulnerabilities. In this paper, we propose a method to find security issues of EOSIO smart contracts by comparing the history version. After analyzing all the security issues in EOSIO smart contracts, we integrate the causes, patterns, and solutions to the security issues. Eventually, we will explore how to automatically fix vulnerabilities in smart contracts.**

I. Introduction

With the widespread acceptance of blockchain and the Internet of Things, the world of technology seems to be converging once again. The main purpose of IoT is to connect the physical world to the digital world. As the IoT permeates our devices, the issue of data security can be a major challenge when billions of devices are connected through a central communication channel. The chances of data being compromised are getting higher and higher and blockchain is the solution to this problem and more.

The core benefits of blockchain technology are the cost of authentication and the cost of networking. The cost of authentication is related to the capacity of the blockchain network, which is able to authenticate transactions in a cost-effective manner thanks to a decentralized authentication consensus approach. Blockchains are decentralized ledger setups in which the entire network is connected in a decentralized manner under cryptographic relationships. Once an entry is entered into the ledger it cannot be changed, thus leaving the entire network free of single points of failure and data leakage. Blockchain technology is granting anonymity and providing resistance to hacking to the networks that use it. It can help bridge the gap between IoT and data security and make IoT networks more secure and unchanging. Blockchain technology will make IoT device data more private than ever before.

The EOSIO is the most popular public blockchain platform supporting smart contracts and has grown rapidly in recent years. As of 2019, the total value of on-chain transactions of EOSIO has exceeded 6 billion USD. Furthermore, EOSIO is still in its early stages and has some experimental characteristics. As a result, as new bugs and security are discovered and new features are developed, the security threats we face constantly change. The vulnerabilities within smart contracts have led to severe financial loss. EOS Bet —a gambling dApp which uses EOS tokens, lost at least $338,000 from its operational wallets to hackers on 2018-10-15 [1].

Therefore, we need practical vulnerability detection tools to safeguard the ecosystem of blockchain. However, the vulnerability detection tools for EOSIO smart contracts are limited. EOSAFE [2] and WANA [3] support detecting specific bugs, such as Fake EOS, Fake Receipt, Rollback, and Missing Permission Check. [4] Although the current detection effect has reached the expected level, it will not have a good effect on new vulnerabilities or some artificially maliciously implanted vulnerabilities.

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. On top of the EOSIO core layer, a WebAssembly virtual machine, *EOS VM*, executes smart contract code, and it is designed from the ground up for the high demands of blockchain applications which require far more from a WebAssembly engine than those designed for web browsers or standards development. The EOSIO smart contracts are developed using cpp or rust. And they can be compiled into WebAssembly code for execution in the

corresponding WebAssembly VM implementation with *eosio.cdt* (EOSIO Contract Development Toolkit)

For Ethereum, the only way to destroy a smart contract on the blockchain system is using the *Selfdestruct* function when attackers find the bugs. Unlike Ethereum, applications deployed on EOSIO-based blockchains are upgradeable. This means you can deploy code fix, add features, and change the application's logic as long as sufficient authority is provided. As a developer, you can iterate your application without the risk of being locked into a software bug permanently. We can easily collect several versions of the same smart contract. Generally speaking, the code updated by the developer may be vulnerabilities, new functions, or even artificially implanted code with fraud.

Automatically fixing vulnerabilities in smart contracts poses unique challenges for existing repair techniques. First, existing repair techniques mainly rely on a test suite to detect incorrect behaviors of programs and validate the generated patches. However, a test suite that guards against all exploits is difficult, if not impossible, to obtain, causing the generated patches to fail to fix the vulnerabilities. Second, existing repair techniques explore the search space of repairs based on syntactic mutators, by leveraging search algorithms such as genetic programming or random search. However, the fix strategies of these techniques are mostly adding conditional checks or replacing a statement with another existing statement, being insufficient for fixing smart contract vulnerabilities that require temporary variable creations and statement reordering (e.g., fixing reentrancy vulnerabilities).、Also, simply adapting these techniques to include more complex fixing strategies can easily result in an exponential expansion of the search space. Third, applying patches beyond adding conditional checks may change the semantics of smart contracts, and even introduce new code faults or vulnerabilities.

This paper first crawls all the verified (open-sourced) smart contracts with several versions from EOSIO. Then we compare the differences between different versions of the same smart contract from the symbolic execution path. Finally, we analyze the differences to summarize the security issues and other reasons for smart contract updates.

II. Backgroud

A. WebAssembly

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications [5]. The WebAssembly virtual machines can be embedded into Web browsers or blockchain platforms. The EOSIO blockchain has supported Wasm. Furthermore, in Ethereum 2.0, Wasm VM is the replacement of Ethereum VM (EVM).

The design choice of using Wasm enables EOSIO to reuse optimized and battle-tested compilers and toolchains which are being maintained and improved by a broader community. In addition, adopting the Wasm standard also makes it easier for compiler developers to port other programming languages onto the EOSIO platform.

There are two convertible and equivalent representations for WebAssembly. We have a binary format - ".wasm" as the suffix. To enable WebAssembly to be read and edited by humans, there is a textual representation of the Wasm binary format - ".wast" as the suffix.

B. EOSIO

EOSIO is a free, open-source blockchain software protocol that provides developers and entrepreneurs with a platform on which to build, deploy and run high-performing blockchain applications.

1) Accounts: An *account* is a human-readable name stored on the blockchain [4]. In order to ensure account security and prevent identity fraud, EOSIO has implemented an advanced access control system based on permissions. An *account* is required to transfer or push any valid transaction to the blockchain.

2) Delegated Proof of Stake (DPOS): The EOSIO platform implements a proven decentralized consensus algorithm capable of meeting the performance requirements of applications on the blockchain called the *Delegated Proof of Stake* (DPOS). Under this algorithm, if you hold tokens on a EOSIO-based blockchain, you can select block producers through a continuous approval voting system. Anyone can choose to participate in the block production and will be allowed to produce blocks,

provided they can persuade token holders to vote for them [4].

3) Smart Contracts: **Smart contract** is a piece of code that can execute on a blockchain and keep the state of contract execution as a part of the immutable history of that blockchain instance. Therefore, developers can rely on that blockchain as a trusted computation environment in which inputs, execution, and the results of a *smart contract* are independent and free of external influence. Every EOSIO smart contract must use *apply* functions as input functions to process operation.

Transactions are composed of one or more actions, which are the basic units for triggering functions in smart contracts.

Dispatcher When one account needs to adjust the smart contract of another account, and the smart contract needs to be processed and dispatched, then we need a dispatcher.

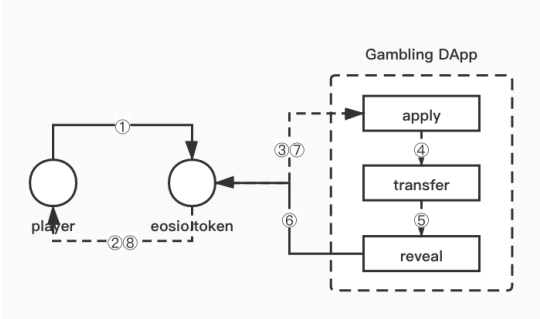

Fig. 1. life-cycle of smart contract execution [2] [6]

The life-cycle of smart contract execution [2]:

1) Invoke transfer to take part in game
2) Notify player (payer)
3) Notify DApp (payee)
4) Dispatch to transfer function
5) Invoke reveal to calculate jackpot
6) Invoke transfer to return prize
7) Notify DApp (payer)
8) Notify player (payee)

In this process, due to various reasons, such as failing to check the authentication parameters, security issues will occur.

III. Vulnerabilities

A. Fake EOS

Cause of vulnerability In EOSIO smart contract, we need *EOS* token at the 3rd step of the generate life-cycle of smart contract execution. However, anyone, including a hacker, can create a token through the public code of EOSIO.token. And the name of the token can be repeated. This means that the name of the token created by the hacker can also be called *EOS*.

Generally speaking, a hacker creates an *EOS* token and name is *EOS*, then transfer the amount of these fake EOS tokens to a EOSIO smart contract. If the DApp's code is robust enough, then the hacker's attack should have failed. Although the name is the same as the real EOS token, the issuer is different. The hacker transmits the EOS token to the DApp through the copy, and then the DApp will not receive the notified *code* EOS token. If it happens that the DApp does not check the value of the *code* at this time, the verification of the dispatcher will be bypassed.

In order to avoid the above problems as much as possible, developers simply narrowed the scope of accepting code, but it can also be bypassed. Obviously, this approach is not rigorous enough. It is this place that gives us the possibility of detecting vulnerabilities.

Harm of vulnerability Early morning on 2018-10-31, online media reported that gambling platform EOSCast was attacked by hackers and lost more than 70K EOS tokens. [7]

B. Fake Receipt

Cause of vulnerability In the generate life-cycle of smart contract execution, the notification will be forwarded when the developer checks the *code*. Also, notification can be forwarded by the dispatcher even the code will not be changed. This is the key to this vulnerability being attacked.

Hackers play two roles in this game, and one is *initiator*, the other is *accomplice*. The initiator invokes a *transfer* to the accomplice through the EOS token. When the EOS token notifies the accomplice, the code is not changed and is directly forwarded to the DApp. However, if the parameters are not checked during *transfer*, the

EOS token will be passed between the two roles of the hacker, which will lead to EOS loss.

C. Rollback

Cause of vulnerability The transaction can be rolled back in some malicious ways, which is not a good thing for some gambling DApps. Suppose that in the gambling DApp, in the generate life-cycle of smart contract execution, when the eighth step, "Notify player" is executed, the hacker can immediately check whether the EOS balance of his account has decreased. If it reduces, the hacker will immediately call the interrupt to roll back the gambling. Such gambling is meaningless, and hackers will always win.

Harm of vulnerability On the day 2018−12−19, Most of the gambling dapp was suffered rollback attack, including Betdice, EOSMax, Tobet, etc. Caluclating at the price of 18 RMB each EOS, it had loss over 500 million rmb. [8]

Through the transaction records of the account, we can find that the account has only revealing records and no betting records.

D. Missing Permission Check

Cause of vulnerability In the generate life-cycle of smart contract execution, the fifth step, "Invoke reveal to calculate jackpot", the DApp should check whether the caller is actually the payer. We have such a function *require_auth()* in EOSIO, it is used to check whether the caller is authorized to call some methods. There is a pre-knowledge here, inlined actions inherit the permissions of the father. Therefore, hackers can execute inlined actions by calling functions to perform operations that the hacker does not have the authority to perform, which leads to security problems.

E. Reentrancy

In July 2016, a fault in TheDAO contract allowed an attacker to steal $50M. The atomicity and sequentiality of transactions may make developers believe that it is impossible to re-enter a non-recursive function before its termination. However, this belief is not always true for smart contracts

First, the *attack*() function in the attacker contract is called, which deposits some ethers in the victim contract and then invokes the victim's vulnerable $refund()$ function. Then, the $refund()$ function sends the deposited ethers to the attacker contract, also triggering the unnamed fallback function in the attack contract. Next, the fallback function again calls the $refund()$ function in the victim contract. Since the victim contract updates the userBalances variable after the ether transfer call, userBalances remains unchanged when the attacker re-enters the $refund()$ function, and thus the balance check can still be passed. As a consequence, the attacker is able to repeatedly siphon off ethers from the victim contract and exhaust its balance

F. Missing Input Validation

The arguments of a function should be validated before their uses. If developers forget to assign correct values to the arguments, EVM will execute the function using the default values based on the argument types. This mechanism makes smart contracts vulnerable to the attacks on function arguments.

G. Unhandled Exception

In Solidity, there are multiple situations where an exception may be raised. Unhandled exceptions can affect the security of smart contracts. In February 2016, a vulnerable contract forced the owner to ask the users not to send ether to the owner because of an unhandled exception in the call instruction.

H. Arithmetic Vulnerability

Arithmetic vulnerability is a common bug, and it is no exception on smart contracts. Integer overflow vulnerabilities and divide-by-zero vulnerabilities are all vulnerabilities that need to be focused on.

I. Feasibility & Versatility

The above four vulnerabilities are common problems that may exist in EOS smart contracts, rather than special cases. According to research in EOSafe, 88.32% of EOSIO smart contracts are deployed using the transfer function. And Fake EOS and Fake receipt need to use the transfer function. Although Rollback is only found in gambling games, from the perspective of DappRadar [9] and DAppTotal [10], gambling games are the most popular Dapp. And Missing Permission Check is a common and high-risk vulnerability.

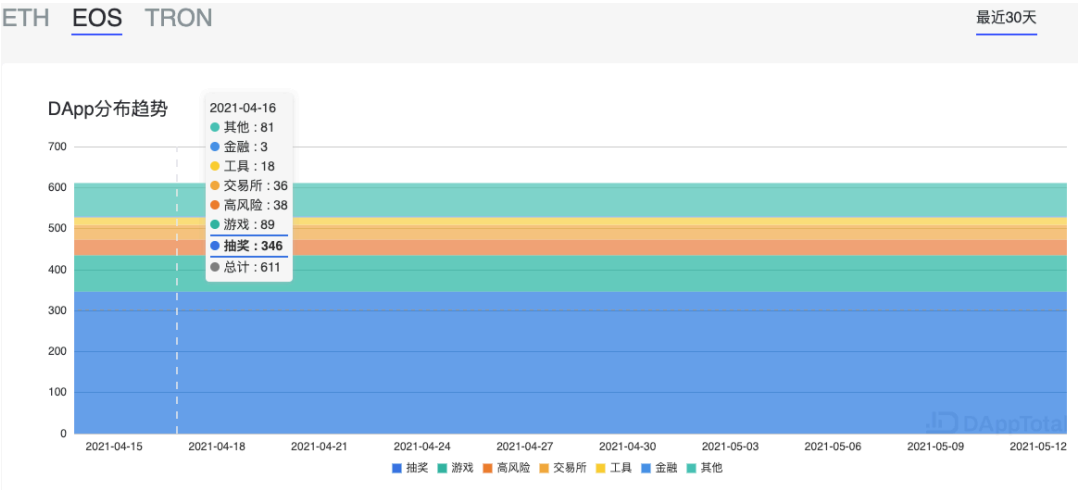

Fig. 2. DApp distribution trend.
From the figure, we can see that in 2021-04-16, out of a total of 611 DApps, 346 DApps are of the gambling type, and 89 DApps are of the game type.

IV. Research Questions

A. Differences Between History Version

We aim to compare the differences between different versions of the same smart contract. So we are thinking about how to visually and effectively analyze the difference between the two codes.

1) Compare Codes Line By Line: Compare codes line by line is not a good choice. First, our code is in wasm bytecode format, which is not conducive to reading. Even if converted to wat format, it isn't easy to get specific meaning from the surface for codes similar to assembly language. So obviously, we will not take this approach.

2) Control Flow Graph: The control flow graph is an abstract representation of a process or program. It is an abstract data structure used in the compiler. It is maintained internally by the compiler and represents all the paths traversed during the execution of a program. It expresses the possible flow of execution of all basic blocks in a process in the form of a graph and can also reflect the process's real-time execution process.

3) Sybolic execution path: Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

```
1   int twice(int v) {
2       return 2*v;
3   }
4
5   void testme(int x, int y) {
6       z = twice(y);
7       if(z == x) {
8           if(x > y + 10)
9               ERROR;
10      }
11  }
12
13  /* simple driver exercising testme() with sym */
14  int main() {
15      x = sym_input();
16      y = sym_input();
17      testme(x, y);
18      return 0;
19  }
```

Fig. 3. Sybolic Code Example

The path constraint generated by symbolic execution makes it easier for us to detect the difference between the two codes.

B. Path Explosion

There are two instructions for branch jump in EOSIO, *br_if* and *br_table*. *br_if* will generate two branches. And *br_table* takes a vector of an arbitrary number n of label indices as its first immediate, and

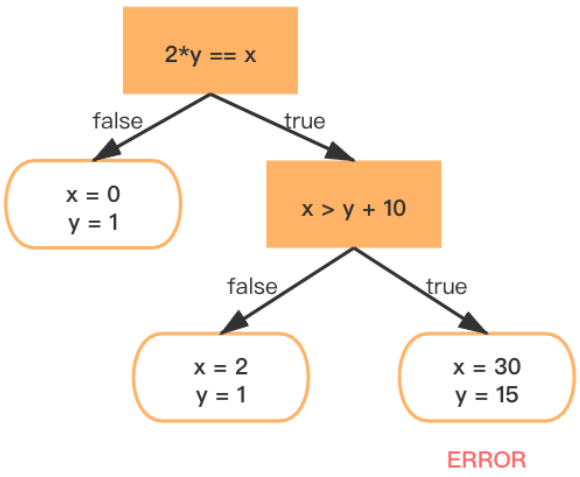

Fig. 4. Sybolic Execution Tree

another label index as its second immediate. This means that *br_table* can generate *n* branches. Once the code has a deep call stack, the complexity of program analysis will increase exponentially. [2]

Under the framework of EOSafe, it has used heuristic pruning to try to solve the above problems, and added *call_depth* to limit the call depth. On this basis, we only focus on some critical functions, and other functions that have little impact on our research content can be ignored.

Finally, set the timeout period to prevent the analysis from being unable to continue due to excessive code complexity.

C. Automatic Fixes

When a vulnerability is detected, the focus of attention is on how to automatically fix the vulnerability.

Automated repair of vulnerabilities in smart contracts poses unique challenges to existing repair techniques. First, existing repair techniques rely heavily on test suites to detect incorrect program behavior and to validate the generated patches. However, test suites that are difficult to obtain, if not impossible, to address all vulnerabilities are difficult, if not impossible, to obtain, resulting in generated patches that do not fix vulnerabilities. Second, by utilizing search algorithms such as genetic programming or random search using search algorithms, existing repair techniques explore the repair space based on syntax mutators. However, the repair strategies of these techniques, which mainly add conditional checks or replace a statement with another existing statement, are not sufficient to repair intelligent contracts that require temporary variable creation and statement reordering (e.g., repairing re-entrant vulnerabilities). Moreover, simply adapting these techniques to include more complex fixing strategies can easily lead to an exponential expansion of the search space. Third, applying patches beyond adding conditional checks may change the semantics of smart contracts and even introduce new code faults or vulnerabilities.

V. System Design

For this stage, we collect data for the following stages. We crawl the verified smart contracts, which are the open-sourced contracts. We have synchronized the EOS mainnet nodes. The complete history node provides an analysis layer on top of historical data, making DApp easier to process and use. In order to process this type of data, nodes need to run full history plugins and solutions, which require a lot of RAM and high maintenance costs. DApp developers can use the complete history node to track and query the data of a specific user account, and we can extract all operations or transactions associated with a specific account to obtain every update of a specific smart contract.

Although there are some public EOS data sets on the Internet, we need the special data required for the experiment which are different versions of the same smart contract. And only the code API of the requesting node cannot get the historical information, it only returns the latest code. So we need to filter out the EOSIO ‘setCode‘ action, grab all the ‘setCode‘ information of the related account, then parse the data packet, and get the Wasm code through decoding.

But there are too many node history records, and there are many repetitive data, it isn’t very meaningful for the experiment. So we chose to use the records before 2019-03-27, the data volume is top 50 million blocks, and the size is about 91GB.

A. Difference comparison

First, we select the cleaned data and choose EOSIO smart contracts with multiple versions.

Then, among multiple versions, we select two adjacent versions in turn for different analysis. The reason why two adjacent versions are chosen instead of any two versions is that one is that the number of codes to be compared increases, and the other is that the code modification of the two adjacent versions can best reflect the modified content. If the old version contains vulnerabilities and the new version fixes the vulnerabilities, we can clearly determine the type of vulnerabilities and how to fix them by analyzing the differences between the two codes.

After the two versions of the code are selected, emulate the code is executed, and the control flow diagram and symbolic execution path of the two versions are produced respectively. The control flow graph is convenient for experimenters to observe the difference of the code visually, and the symbolic execution path and some recorded states are used to automatically analyze whether there are vulnerabilities and the types of vulnerabilities.

We have implemented three types of vulnerability detectors in the experiment. They are Fake EOS Detector, Fake Receipt Detector, and Missing Permission Check Detector. The reason why the Rollback detector is not implemented is that the detection complexity of the rollback is extremely high. Generally, one can only try to substitute a particular method to perform the detection. Due to the time factor, the Rollback detector is temporarily not implemented.

B. Frame

We use Octopus and EOSafe as framework. Octopus is a security analysis framework for WebAssembly module and Blockchain Smart Contract. The purpose of Octopus is to provide an easy way to analyze closed-source WebAssembly modules and smart contracts bytecode to understand deeper their internal behaviors.

VI. Vulnerabilities Dectection

A. Fake EOS Detector

According to the life-cycle of smart contract execution in II-B.3 and the cause of the Fake EOS vulnerability we know in III-A, we found that the smart contract containing the vulnerability of Fake EOS must be called by the hacker to transfer the function, and successfully bypassed the code. An examination. Then we only need to perform symbolic execution on the apply function, generate a symbolic execution path, and determine whether the difference constraints between the new and the old version is:

$$\begin{aligned} & action == transfer, (\text{ in apply function}) \\ \bigwedge\ & code == account\ name,\ not\ self, (\text{ in apply function}) \end{aligned} \tag{1}$$

If the new and old versions of the EOSIO smart contract meet the above conditions, we will consider the old version of the EOSIO smart contract to be fragile and may contain the vulnerability of Fake EOS, and it has been fixed in the new version.

We only need to analyze the differences in the paths related to transfer, and the differences in other paths will hardly cause Fake EOS. This is a direction that can be optimized for pruning.

B. Fake Receipt Detector

According to the life-cycle of smart contract execution in II-B.3 and the cause of the vulnerability of Fake Receipt we know in III-B, we found that the critical factor of the smart contract that contains the vulnerability of Fake Receipt is insufficient verification in the transfer function. Then our approach is straightforward: find all the functions that may call/jump to the transfer function, perform symbolic execution, generate a symbolic execution path, and determine whether there are operations to modify the state of the blockchain between the new and old versions of the suspicious functions, and whether there is a function such as *eosio_assert()* to interrupt the path that does not jump.

$$\begin{aligned} & self == to,\ modification, (\text{ in suspicious function}) \\ \bigwedge\ & self! = to,\ eosio\_assert(), (\text{ in suspicious function}) \end{aligned} \tag{2}$$

C. Missing Permission Check Detector

According to the life-cycle of smart contract execution in II-B.3 and the cause of the Missing Permission Check vulnerability we know in III-D, we only need to pay attention to whether functions check permissions

before performing sensitive operations. And all functions are connected with apply function in the call graph. So we can perform symbolic execution on the apply function, find all functions, and check whether a permission check has been added before the instructions for some sensitive operations that we have counted. If permission check is added, we can think that the old version of EOSIO smart contract has the Missing Permission Check vulnerability, and it has been fixed in the new version.

$$\begin{aligned} &sensitive\ operations == db\_update_i64(), etc. \\ &\exists sensitive\ operations \bigwedge no\ require\_auth() \end{aligned} \tag{3}$$

VII. Automatic Fixes

In this section, we present the fix patterns used for patch generation. Our program analysis supports fix patterns in three granularity levels: statement level, method level, and contract level. Next will be a description of the vulnerabilities that can be fixed and how to generate patches.

A. Reentrancy

The preferred fix pattern for Reentrancy is to move all writes to storage ahead so that there is no write to storage after an external method call or a coin transfer call.

B. Missing Input Validation

The fix pattern for MissingInputValidation is to add conditional checks to validate the method parameters at the beginning of method body.

C. Unhandled Exception

The fix pattern for this vulnerability is to check the return value of each call statement. Program adds a *require*() function call to validate the return values of *send*() and *value*() and ensures that their executions are successful before completing the whole transaction.

D. Algorithm vulnerability

The fix for the arithmetic vulnerability is to replace all unsafe arithmetic functions with safe ones.

VIII. Evaluation

A. Dataset

We use the EOSIO records before 2019-03-27, the data volume is top 50 million blocks. In the data set after data cleaning, there are *1373* smart contracts, and *610* smart contracts have multiple versions. There are *2849* pairs of old and new version codes in total.

B. The Accuracy of Vulnerability Detection

In order to evaluate the detection level of this system, we inquired about the reports issued by blockchain security companies such as PeckShield [11] and SlowMist [12] to confirm that the vulnerabilities we found were accurate and real. And at the same time, use EOSafe to assist judgment. EOSafe is an excellent detection framework. In its previous experiments, its precision and recall were 100% and 96.30%, respectively. We believe that EOSafe can serve as an effective referee.

Through comparison, we found that EOSIO smart contracts with vulnerabilities detected by this system can all be confirmed on EOSafe. However, there is a phenomenon of underreporting in this system, and the general reasons are as follows:

- The timeout setting time is relatively short, resulting in *587* EOSIO smart contracts that cannot be fully analyzed (the three detectors cannot be successfully executed within the specified time).
- The detection conditions of the detector need to be strengthened.
- Because the data is the first 50 million blocks, and the last block is dated 2019-03-27, EOSIO has just developed, many vulnerabilities have not been discovered, and no developers have fixed these blockchain security issues, so there are a lot of historical version codes. Both have the same problem, and it is impossible to check out the vulnerabilities by comparing the differences.

C. Distribution of vulnerabilities

Since there are multiple versions in an EOSIO smart contract, as long as there is a pair of old and new versions of the code and vulnerabilities are detected, we can consider the EOSIO smart contract to be fragile and insecure. We found that among the EOSIO smart

Type	# Candidates	# Vulnerable (%)
Fake EOS	23	1(4.35%)
Fake Receipt	23	2(8.70%)
Missing Permission Check	23	4(17.39%)
Total	23	7(30.43%)

TABLE I
VULNERABILITIES DETECTION RESULT.
CANDIDATES ARE SMART CONTRACTS THAT CAN RUN UNDER SPECIFIED TIME AND CONDITIONS. VULNERABLE (%) IS THE PROPORTION OF SMART CONTRACTS THAT HAVE VULNERABILITIES IN THE SMART CONTRACTS THAT ARE DETECTED.

contracts that can fully execute the 3 vulnerability detectors, 30.43% of the smart contracts have security issues. The most frequent problem is Missing Permission Check, followed by Fake Receipt, and the least one is Fake EOS.

Among the smart contracts in the data set, the smart contract with the most historical versions is guydgnjygige, a popular gaming platform. Of course it also has the most vulnerable versions.

By manually inspecting the EOSIO smart contract for which vulnerabilities were detected, we found the following problems:

1) The vulnerabilities repaired by DApp developers are generally very fast. For example, the smart contract danakilblock introduced the vulnerability at 2018-09-05T 19:41:37.500, and updated the version at 2018-09-05T 20:00:42.000 to solve the Missing Permission Check.
2) When DApp developers update the code, they may fix vulnerabilities in the code and update the business code more often. Of course, as long as the code is updated, new vulnerabilities are likely to be introduced. According to our manual inspection, there are two EOSIO smart contracts that have not been detected vulnerabilities in the old version, but were detected after updating the business code. This reminds DApp developers to avoid creating new vulnerabilities as much as possible when developing new features.
3) After relevant blockchain security companies released security reports, DApp developers updated smart contracts more intensively.

D. Automatic Fixes result

Automatic fixes for Arithmetic Vulnerability, Unhandled Exception, Missing Input Validation vulnerability can solve the original problem. However, more work needs to be done on other vulnerabilities such as reentrant vulnerabilities.

IX. CONCLUSIONS

In this paper, we proposed the idea of searching for security issues in EOSIO smart contracts by comparing historical versions, and implemented three types of vulnerability detectors. Finally automatic fixes were made for specific vulnerabilities. According to the experimental results, in the current EOSIO ecological environment, there are still many smart contracts facing the situation of being exploited by hackers to obtain illegal benefits. Our research results have further verified that EOSIO's blockchain security needs to be improved.

References

[1] Q. Le, "How hackers attack eos contracts and ways to prevent it," nov 2018.

[2] 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.

[3] D. Wang, B. Jiang, and W. K. Chan, "Wana: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," *ArXiv*, vol. abs/2007.15510, 2020.

[4] "Eosio developer portal."

[5] "Webassembly 1.0 has shipped in 4 major browser engines.."

[6] ""transactions protocol,"."

[7] ""fake eos attack" upgraded, 60k eos tokens lost by eoscast."

[8] "Roll back attack about blacklist in eos."

[9] ""dappradar, a dapp browser,"," dct 2020.

[10] "Dapptotal„" nov 2019.

[11] ""blogs about blockchain security events,"," 2020.

[12] ""blockchain security events,"."