Automatically Fixing Vulnerabilities in WebAssembly

Yubin Hu[1]

***Abstract*— This reading report mainly records papers on how to detect vulnerabilities in smart contracts which based on Webassembly and how to fix them automatically and generate patches automatically to accumulate methods and experience for subsequent experiments.**

I. Introduction

A. Blockchain

Blockchain is a public list of records which are linked together.

B. Smart Contracts

Smart Contracts, once deployed on the blockchain network, become an unchangeable commitment between the involving parties. Because of that, they have the potential to revolutionize many industries such as financial institutes and supply chains. However, like traditional programs, smart contracts are subject to code-based vulnerabilities, which may cause huge financial loss and hinder its applications.

C. 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.
- The WebAssembly virtual machines can be embedded into Web browsers or blockchain platforms.
- Furthermore, in Ethereum 2.0, Wasm VM is the replacement of Ethereum VM (EVM).

However, smart contracts are subject to code-based vulnerabilities, which casts a shadow on its applications. As smart contracts are unpatchable (due to the immutability of blockchain), it is essential that smart contracts are guaranteed to be free of vulnerabilities. Unfortunately, smart contract languages such as Solidity are Turing-complete, which implies that verifying them statically is infeasible.

The following papers will explain how to detect vulnerabilities in smart contracts which based on Webassembly from various perspectives, and how to automatically fix them and automatically generate patches. It will present multiple papers detailing the more innovative approaches to each process, and will conclude with a summary of the ideas we have absorbed from these papers.

II. Finding Ethereum Smart Contracts Security Issues by Comparing History Versions [1]

A. Abstract

Ethernet's smart contracts are complete programs that run on the blockchain. They cannot be modified even if an error is detected. The $Selfdestruct$ function is the only way to destroy a contract on the blockchain system and transfer all items on the contract balance. Therefore, many developers use this function to destroy contracts and redeploy a new one when an error is detected. In this paper, we propose an in-depth learning approach to find security issues in Ethernet smart contracts by finding updated versions of the destroyed contracts. After finding the updated version, we use Open Card Sorting to find the security issue.

B. Research Questions

1) How to find old and new versions of the same smart contract
2) How to check the version difference between old and new versions of the same smart contract by

C. Detailed Approach

1) Collecting data from the following two stages. Our data contains three parts, verified contracts, self-destruct contracts, and contract transactions.

2) Finding upgrade version of a destructed contract.
3) Finding the security issues by comparing the difference between a self-destruct contract and its upgrade version.

III. SGUARD: Towards Fixing Vulnerable Smart Contracts Automatically

[2]

A. Abstract

In this work, we propose an approach and a tool called SGUARD, which will automatically repair potentially vulnerable smart contracts. SGUARD is inspired by program fixing techniques for traditional programs such as C or Java, and yet are designed specifically for smart contracts. First, SGUARD is designed to guarantee the correctness of the fixes. Furthermore, fixes for smart contracts may suffer from not only time overhead but also gas overhead and SGUARD is designed to minimize the run-time overhead in terms of time and gas introduced by the fixes.

B. Research Questions

In this work, a method was developed to automatically convert smart contracts so that they are proven to be free of the 4 common vulnerabilities. The key idea is to apply runtime verification in an efficient and transparent way.

C. Definition

- Control dependency
- Data dependency
- Intra-function reentrancy vulnerability
- Cross-function reentrancy vulnerability
- Dangerous tx.origin vulnerability
- Arithmetic vulnerability

D. Detailed Approach

1) Enumerating Symbolic Traces: Our definition of vulnerability is based on symbolic traces. We simply apply the symbolic semantic rule iteratively until it terminates. However, in the presence of a loop, the process will not terminate as the condition of leaving the loop is usually symbolic. This is a well-known problem with symbolic execution, and the remedy is usually to enlighten the number of iterations bound. However, this approach does not work in our environment because we have to identify all data/control dependencies to identify all potential vulnerabilities. In the following, we establish bounds on the loop that are sufficient to identify the vulnerabilities we focus on.

2) Dependency Analysis: Given the set of symbolic traces, we then identify the dependencies between all symbolic signs among all opcodes, which are designed to check whether the trace suffers from any vulnerability.

3) Fixing the Smart Contract: After identifying the dependencies, we check each symbol for any previously defined vulnerabilities and then fix the smart contract accordingly.

IV. ContractFix: A Framework for Automatically Fixing Vulnerabilities in Smart Contracts

A. Abstract

In this paper, we present a composite document of a new framework for automatically generating security patches for vulnerable smart contracts. ContractFix is intended to be a generic framework that can incorporate different types of vulnerabilities into different remediation models and be used as a security ”fixer” tool to automatically apply patches and validate patched contracts before they are deployed. To address the unique challenges of remediating smart contract vulnerabilities, given the input smart contracts, contract files are first validated using a collaborative approach that combines multiple static validation tools to identify automatically patched vulnerabilities using a majority voting mechanism to improve detection accuracy. Contractfix then generates patches using template-based fix patterns and leverages static program analysis (program dependency calculations and pointer analysis) to accurately infer and populate the values of the fix patterns. Finally, ContractFix performs two-step validation, statically verifying the elimination of vulnerabilities in the patch contract and dynamically testing the patch contract to ensure that the patch does not introduce additional faults.

B. Research Questions

1) Synergy of Static Verification Tools
2) Effectiveness in Patch Generation
3) Runtime Performance

C. Detailed Approach

1) Vulnerability Detection: In this phase, Contractfix first uses static validation to detect vulnerability candidates. ContractFix applies static validation to check security properties for identifying four types of vulnerabilities: Reentrancy, MissingInputValidation, LockedEther, and UnhandledException.

To identify these types of vulnerabilities that can be used for automated remediation, Contractfix post-processes reported vulnerabilities based on syntactic analysis from AST and in-program control and data flow analysis. For example, for reported re-entry vulnerabilities, detecting whether the external function call used to control the execution of a state variable update will require control and data flow analysis.

2) Patch Generation: Contractfix performs static program analysis to extract contextual information about detected vulnerabilities, and this method is used by the remediation mode to generate the corresponding source code patches. Our program analysis supports three levels of granularity in the remediation model: statement level, method level, and contract level.

3) Validation and Testing: In this phase, ContractFix conducts a two-step validation to ensure that the detected vulnerabilities are eliminated while the expected behaviors are preserved in the patched contract.

Static Verification: Once a patched contract candidate is generated, ContractFix applies static verification again on the patched contract and checks whether the vulnerabilities reported in Phase I are eliminated. If the static verification tool no longer reports the same vulnerabilities, the patched contract is considered to pass the static verification.

Semantics Validation: Semantics validation is conducted using a smart contract testing platform named Truffle.

V. Conclusions

Through these papers, we understand how to DETECT vulnerabilities from blockchains such as Ether, EOS; Clarify the types of vulnerabilities on blockchains and how to fix them; And how to automatically fix vulnerabilities at the source level or binary level; And how to improve the accuracy and efficiency of automatic fixes.

A. Vulnerability

1) 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

2) 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.

3) Locked Ether: In 2017, a vulnerable contract led to the frozen of million dollars. The reason is that this contract relies on another library contract to withdraw its funds (using delegatecall). Unfortunately, a user accidentally removed the library contract from the blockchain (using the kill instruction), and thus the funds in the wallet contract could not be extracted anymore.

4) 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.

B. Vulnerability Detection

- Comparison with historical versions
- Find control flow, data flow characteristics
- Fuzzing [3]
- Using multiple existing tools, and seting thresholds to determine if it is a vulnerability

C. Fixing vulnerabilities

- generates patches using template-based fix patterns and leverages static program analysis
- Binary rewriting. [4] Binary rewriting has also been applied to retrofit security hardening techniques such as control-flow integrity, to compiled binaries, but also to dynamically apply security patches to running programs. For binary rewriting on traditional architectures two flavors of approaches have been developed: static and dynamic rewriting.

D. Evaluation

- False Positives
- Runtime Performance
- Extra Gas

Based on the above, we can use a more appropriate approach at each stage to conduct our own experiments to detect vulnerabilities in smart contracts on EOS, as well as to fix specific vulnerabilities.

References

[1] J. Chen, "Finding ethereum smart contracts security issues by comparing history versions," *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1382–1384, 2020.

[2] T. D. Nguyen, L. H. Pham, and J. Sun, "sguard: Towards fixing vulnerable smart contracts automatically," 2021.

[3] Y. Huang, B. Jiang, and W. K. Chan, "Eosfuzzer: Fuzzing eosio smart contracts for vulnerability detection," *ArXiv*, vol. abs/2007.14903, 2020.

[4] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Evmpatch: Timely and automated patching of ethereum smart contracts," 2020.