STORYLINE

A. Introduction

- Overview
  - The Ethereum Virtual Machine is responsible for executing the bytecode.
  - The implementations of EVM in different programming languages may have **defects**.
  - Many efforts are put into the validation and vulnerabilities detection on smart contract. Few techniques exist to help systematically validate EVMs.
  - Existing work, EVMfuzzer, mutates the smart contracts and then use the Solidity compiler to compile the mutants into bytecode fed into differentially testing EVMs. However, bytecode generated by Solidity compiler is syntactically valid. Different EVM platforms may check the validity of bytecode in a different way. For example, go-ethereum and openethereum will refuse the bytecode if the length of bytecode is odd, but aleth will run normally and use up all the gas limit. We observe that bytecode-level mutating can directly generate some syntactically invalid bytecodes such as the length of bytecode is odd. Normally, the length of bytecode generated by Solidity compiler is even.
  - To perform efficient EVM testing, the difficulty lies in synthesizing inputs that exercise a **variety** of code paths in the EVM. That is to say, how can we select the seeds from a large number of mutants to perform the next iterations of mutans generation while ensuring that the resulting mutants can explore the EVM's execution paths as much as possible.
  - Inspired by one notable effort, classming, alters the **control flow** of Java bytecode for JVMs testing. EVMs may execute uncommon bytecode instruction sequences in different manners.
  - This paper tackles this problem by introducing a ***coverage-directed fuzzing*** approach that focuses on representative bytecodes for differential testing of EVMs. Our core idea is to continuously (1)manipulate the **control flow** of seed bytecodes of smart contracts to generate semantically different mutants and (2)execute the mutants on a *reference* EVM and use ***coverage uniqueness*** as a guideline to selectively accept the generated mutants to steer the mutation process toward diverse mutants. The generated mutants are then employed to differentially test EVMs.
- Contributions
  - We designed 5 mutators to generate a large number of runable, semantically different test cases..
  - We adopt the *unique branch coverage*in the EVM as the fitness function to guide the fuzzing. Given a seed, our approach can generate a large number of test bytecodes with **diverse semantics** to help expose EVM implementation differences..
  - We have evaluated our framework on mature and popular Ethereum Virtual Machines such as **go-ethereum**, **parity-ethereum** and **aleth**. We have analyzed the **crashes** and **differences**. Accompanied by manual root cause analysis, we found some previously unknown issues.

- Thesis Arrangement
  - Chapter2 introduces background knowledge including Ethereum Virtual Machine, smart contract, opcodes, bytecode and an motivating example.
  - Chapter3 introduces our approach.
  - Chapter4 introduces the implementation and evaluation.
  - Chapter5 introduces the discussions.
  - Chapter6 introduces related works about coveraged-guided fuzzing, differential testing, Java Virtual Machines testing and Ethereum Virtual Machines Testing.
  - Chapter7 shows the conclusion.

B. Preliminaries and Motivating Examples

- Ethereum Virtual Machine
  - There are tens of thousands of transactions happened via the clients based on different EVM implementations everyday. Therefore, the vulnerability hidden in any EVM version might result in serious consequences such as the financial loss.
- Smart Contract, Opcodes and Bytecode
  - Smart contracts are compiled by Solidity compiler to low-level machine instructions(called **opcodes**). The EVM uses a set of opcodes to execute specific tasks. Especially, this paper focus on **Program counter related opcodes** (JUMP, JUMPI, JUMPDEST). And we use these instructions to design mutators. It has several reasons. Firstly, these opcodes can alter a program's control–flow. Secondly, these mutators can generate a large number of mutants in a short time. Thirdly, the more public functions in a smart contract, and the more complex the structure of the function and the more mutants that can be generated. Therefore, our approach do not need too many seed contracts. We only need to select enough contracts with more inner public function and rich structure in these functions as the seed contracts.
  - In order to efficiently store opcodes, **opcodes** are encoded to **bytecode**
  - Bytecode is part of the input fed into the EVM.
- An Illustrative Example
  - **infinite loop** can cause two set of different behaviours
    - mutator3 + mutator4
    - mutator2

C. Design of Approach

- Overview
  - Given a set of seed bytecode files *f*, our approach aims at creating many mutants of files in *f* by manipulating *f's* bytecode,

  - and then these mutants are executed in test program. After execution, we collect the *branch coverage* of each mutants, and use the *unique branch coverage* as a guidline to choose the resulting mutants for the next iteration.
  - The generated bytecode mutants are then employed to differentially test EVMs
  - The whole process is splited into following parts, including opcode and function block generation, abnormal opcodes generation, translate opcodes into bytecode, CGF guide the seed selection and unified EVM execution.

- Opcode and Function Block Generation
  - We use the *disassembler* in **Vandal** to convert seed bytecode to opcodes and *decompiler* to generate the the byteocde's function blocks for further mutation

- Abnormal Opcodes Generation
  - extract body blocks of each public function
  - extract all addresses of JUMP or JUMPI insturctions
  - mutate the bytecode by a set of *pre-defined mutators*

- Translate Opcodes into Bytecode
  - After we mutate the opcodes, we need to translate the opcodes into bytecode, which is one of the input arguments for Ethereum Virtual Machine.

- CGF Guide the Seed Selection
  - In each iteration, new inputs are generated by running the mutation program and the test program is then executed on each input.
  - During the execution, the **branch coverage** of each seed will be recorded. If the corresponding input covers a **new coverage point** that has not been exercised by any previous valid input, then the input is added to the seed pool for the next iteration

- Unified EVM Execution
  - The first step is to get the data that can directly feed into EVM platform, namely the contract runtime bytecode and input parameters.
  - The second step is to call the execution interface of each EVM to execute the contract data, standardize the output format of the execution result and save the output.
  - The third step is to analyze the output files of each platform and the crashes during the execution to uncover potential bugs.

D. Implementation and Evaluation

- overview

  - In this chapter, we present the implementation details of our algorithm. Our approach has conducted large-scale mutations on 52 real-world smart contracts and several EVM discrepancies and vulnerabilities have been found. We answer the following three questions: (i) Is there any inconsistency among EVMs? (ii) Could our approach achieve higher branch coverage? (iii) Is it possible to find EVM bugs through our approach?

- Data and Environment Setup

  - with 8 cores (Intel i7-4870HQ @2.50GHz), 16GB of memory, and Ubuntu 16.04.6 LTS as the host operating system. The 52 real-world contracts we used for mutation were crawled from the Etherscan and we obtained the corresponding runtime bytecode and abi function via the Solidity language compiler solc 0.6.2. EVMFuzz tested three widely used EVM platforms for each mutated contract, that is, go-evm,parity-evm, aleth-vm

- we mainly evaluate the effectiveness and efficiency of our method. To be specific, we are going to answer the following four research questions:

  - Can our approach generate sufficient test bytecode files for EVM testing?
  - Could our approach achieve higher branch coverage compared to EVM-Fuzzer?
  - Is the branch coverage-guided algorithm effective?
  - Is there any inconsistency and vulnerabilities among EVMs?

E. Discussion

- **Mutators design**.

  - Considering the characterics of control-flow, we designed 5 mutators, whose quantity and qualities still need improving. Currently, we just created some mutators based on JUMP, JUMPI, JUMPDEST opcodes and their targets address. In the future, we will make more semantic level mutations based on bytecode, taking the optimization of bytecode into consideration.

- **More efficient fitness function**

  - In this paper, we use **unique branch coverage** as the fitness function. For instance, if a test case covers a new branch in the program under test, then we add it to the pool as it meets the fitness criterion.Despite its wide use, it suffers from critical information loss . In the future, we will improve the fitness function, considering the relationships between program executions.

- **Testing more EVM implementation**.

  - The core idea of differential testing is testing on at least two objects with the same test inputs. Therefore, we analyze the differences in three EVM platforms: geth, parity-EVM, aleth. However, in the real world, many platforms based on other programming languages are also widely used, and they may also have some potential vulnerabilities in implementation. We will further incorporate them into our framework.

F. Related Work

- Coverage-Guided Fuzzing
  - CGF instruments the test program to provide dynamic feedback in the form of *code coverage* for each run. If the mutated inputs can lead to new **branch coverage** in the test program, they are saved for next fuzzing iteration.
- Differential Testing
  - Differenetial testing attempts to detect bugs by providing the same input to a series of similar test programs
- Java Virtual Machine testing
  - Chen et al. achieves JVM implementations testing via systematic manipulation of the control- and data-flow of live bytecode to effectively testing verifiers and execution engines at the backend in the JVM.
- Ethereum Virtual Machine testing
  - EVMFuzzer directly mutated smart contracts to differentially testing EVMs. However, seed contracts generated by EVMFuzzer has no syntax error and are not diverse.
- Our main difference
  - Different from the above work, our work mainly focuses on discovering the defects and vulnerabilities in EVM by mutating the **bytecode** of smart contracts. In the case that most researchers are concerned about smart contracts validation and no researchers focus on the bytecodes when testing the EVM, motivated by this, we proposed our approach which aims at manipulating the **control-flow** of bytecodes and integrated the **CFG** into **differential testing**, using the *branch coverage uniqueness* as the *fitness function* to select the diverse seeds.

G. Conclusion

- In this paper, we presented a technique that incorporates ideas from coverage-guided fuzzing into differential testing, the first bytecode level automated fuzzing framework, to effenciently find vulnerabilities of EVM implementations.
- We design 5 mutators related to changing the control-flow of bytecode and use the *branch coverage uniqueness* as the guideline during the fuzzing to explore the scode paths to generate more diverse input tests.
- We evaluated our approach on 3 different EVM implementations, we found that there exists gasUsed inconsistencies and error information inconsistencies among EVMs, our approach achieved higher branch coverage than baseline techniques and our generated corner cases could find some previously unknown problems.
- Our future work mainly includes developing more specific muators which can generate more corner cases to uncover the vulnerabilities, and conducting more extensive evaluations on more EVMs.