Concolic Fuzzing for Smart Contracts
Mohammadreza Ashouri
University of Potsdam
ashouri@uni-potsdam.de

Abstract
In recent years we have seen a great deal of attention to the topic of security analysis in smart contracts, especially those developed for the Ethereum blockchain. As a result, there seems to be an ever-growing demand for secure smart contracts to protect what could potentially be worth billions. In this poster, we introduce a practical approach for fuzzing Ethereum smart contracts. Our method works based on the combination of dynamic taint analysis and concolic testing on the bytecode. We implemented our technique in the Rust programming language with the help of an instrumented Ethereum virtual machine, which is called Parity. So far, we have evaluated our approach on a benchmark suite, including 9 smart contracts and the result of our preliminary evaluation reveals 39 security vulnerabilities in our benchmark suite.

What is an Ethereum smart contract?
The Ethereum platform allows users to run “smart contracts” on its distributed infrastructure. Smart contracts are programs that define a set of rules for the governing of associated funds, typically written in a Turing-complete programming language called Solidity [2]. The language is object-oriented and similar to JavaScript for writing the contracts on various blockchain platforms, most prominently, Ethereum. Programs written in Solidity are compiled into low-level untypied bytecode to be executed on the Ethereum platform by the Ethereum Virtual Machine (EVM). It is also possible to write the EVM contracts by other tools such as Vyper [8] (similar to Python). Similar to other software ecosystems, there are various vulnerabilities in smart contracts as well. For instance, in Solidity “Integer” data types have not any built-in security against over- or underflow (BOF or underFOF attacks). Thus, if a loop counter were to overflow, generating an infinite loop, the funds of a contract would become fully frozen. In other words, attackers can exploit this bug by increasing the number of iterations of a loop, for example, by introducing new users to the contract [9]. Also, some low-level operations (e.g. send method) in Solidity have not thrown an exception on failure; instead, inform the state by delivering a Boolean. Hence, the contract does not check the returned Boolean, it will maintain its execution even if the payment dropped so that this situation can lead to serious security issues like Re-Entrancy. This vulnerability is one of these issues that has taken Ethereum security communities by storm so that it can be exploited when a contract attempts to send Ether before having updated its internal state. If the target address is a different contract, the contract code will be executed and can invoke the function to ask Ether again and again, which results in generating extra funds.

1 Why analyzing smart contracts is challenging?
1. Smart contract is immutable after deployment; thus, they must be secured before getting released.
2. Existing security analyzers mostly have been written based on static code review so that they need the source code. However, the source code is not always available, particularly for commercial off-the-shelf (COTS) contracts.
3. Static analyzers (e.g., [9, 5, 6]) cannot adequately cover the runtime behavior of smart contracts with untrusted input coming from the network, thereby missing new attacks and giving false negatives/positives is a common weakness of prior work.
4. Dealing with low-level instructions in the EVM is difficult.
5. The owners of smart contract owners are anonymous, i.e., responsible disclosure usually is infeasible.

How our approach works?
Through our introduced approach, we can detect vulnerabilities and zero-day attacks based on the observing the runtime behavior of the smart contract instructions during the execution time in the EVM. Hence, in the first step, we extended an instrumented Ethereum client, which is called Parity [1], for performing the monitoring process.

Early Results
Although there exist many available open-source synthetic smart contracts on the Internet that might be helpful for detecting classic vulnerabilities, finding a ready benchmark suite including various real-world contracts that contains modern vulnerabilities was a challenge for our evaluations. Therefore, for measuring our fuzzing method near to real-life circumstances, we carefully collected a benchmark suite including a real-world and educational contracts. We also deliberately included a broad range of security issues in our benchmark such as Integer Overflow (IOF) - Underflow (IUF), Bad Randomness (BR), Re-entrancy (RE), Locked Ether (LE), Unhandled Exceptions (UE), Denial of Service (DoS), Short Address Attack (SA), Race Condition (RC), Shadow Memory (SM). The description of each bug can be found in many references (e.g. [9]). Moreover, all tests executed in our experiment were conducted on an 8 Core Xeon W 3.2 GHz machine with 32GB RAM running Ubuntu 18.04 LTS. Table 2 represents the summary of our experiment.

Our contributions
We achieved the following contributions in this work:
1. Performing security analysis on smart contract at the EVM binary level (with no need to source code).
2. Generating accurate tests for discovered vulnerabilities in the target contracts.
3. Reducing false positives by checking the potential protections in the code.
4. Evaluating our approach on various benchmarks with different security levels.
5. Mitigating path/state explosion problem in concolic testing by providing context information of each tainted data flow for our test generator engine.

References