# Fuzzing Ethereum Smart Contracts
(research statement)

Roberto Ponte
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa - Portugal
roberto.ponte@tecnico.ulisboa.pt

Ibéria Medeiros
LaSIGE, Faculdade de Ciências,
Universidade de Lisboa - Portugal
imedeiros@di.fc.ul.pt

Miguel Correia
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa - Portugal
miguel.p.correia@tecnico.ulisboa.pt

## I. Introduction

Blockchain has been gathering a lot of public attention due to the success of Bitcoin [1] and to high hopes in its benefits. A blockchain is a data structure – an append-only sequence of blocks – that stores transactions. Moreover, the term also denominates a distributed system that maintains copies of that data structure in a set of nodes. To add new information to a blockchain there is a consensus protocol that ensures that the blocks are appended in the same order in all the nodes. Bitcoin and most blockchains tolerate arbitrary (Byzantine) faults.

Ethereum is a blockchain that both provides a cryptocurrency and supports the execution of *smart contracts* written in a Turing-complete language [2]. Smart contracts are similar to legal contracts but instead of having the terms recorded in a legal language they are coded as a computer program, written in a programming language [3], [2]. The advantages of the usage of smart contracts include low contracting enforcement, compliance costs, and no need for a trusted third party to validate the contract.

Building software that works as expected is easier than assuring that nobody can use it in a way that it is not supposed to. This is troublesome with smart contracts as they often handle valuable assets. Furthermore, every execution happens in a public network and the source code is often available, so having security in perspective is important while developing smart contracts. There have been huge attacks exploiting vulnerabilities in smart contracts, such as the DAO attack that led to losses of 60M USD [4] and the Parity Multi-Sig Wallet attack with losses of 30M USD [5].

This work introduces a tool to detect vulnerabilities encoded in smart contracts developed for Ethereum. The goal is to provide a tool that uses *fuzzing* [6] or attack injection [7] to search for vulnerabilities in smart contracts by doing input injection.

There are already a few experimental tools to discover vulnerabilities in smart contracts, but no fuzzers. Oyente creates a control flow graph and does symbolic execution to discover vulnerabilities [8]. Solidity-coverage is a tool that helps achieving full coverage during smart contract testing [9]. Manticore is a prototyping tool for dynamic binary analysis, with support for symbolic execution, taint analysis, and binary instrumentation [10]. Mythril uses concolic analysis to detect various types of issues in Ethereum smart contracts [11]. Formal verification has also been proposed for detecting program logic vulnerabilities at the bytecode level [12].

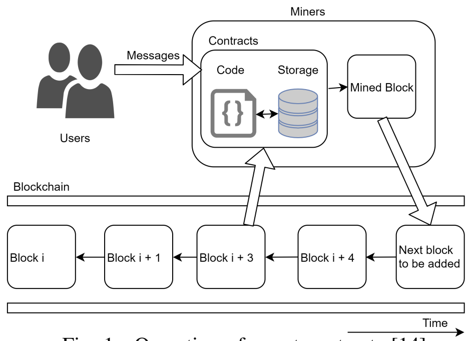

Fig. 1. Operation of smart contracts [14]

Our fuzzing tool aims to explore a different point in the design space with the typical benefits of fuzzing tools – real execution in a real environment – that have led to the discovery of large numbers of vulnerabilities in many applications.

## II. Smart Contracts and Vulnerabilities

Smart contracts have evolved way beyond the definition of Szabo in 1994 [3]. Nowadays smart contracts are computer programs, or scripts, that can be developed in high level languages, like Solidity [13], a contract-oriented language for implementing smart contracts which is designed to target the Ethereum Virtual Machine (EVM).

In Ethereum, smart contracts have code, a storage space and an account balance. These programs are user-created and can be posted on the blockchain. Once a smart contract enters the blockchain it cannot be removed, the process is irreversible due to the append-only nature of the blockchain. After posting the smart contract on the blockchain it is activated upon receiving a message from a user or from another contract, as shown in Figure 1.

The major bugs that were in the origin of the DAO attack allow altering user data and take over the control flow of the smart contract making changes that were not expected. This class of bug can be seen in multiple forms, one of them are race conditions. Example variants are those caused by reentrant functions [15], [16] and cross-function races [15].

Transaction-Ordering Dependence (TOD) vulnerabilities come from the fact that the outcome of the execution of a

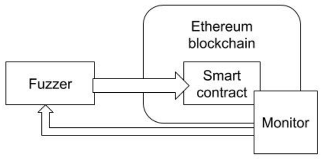

Fig. 2. Smart contract fuzzing tool architecture

transaction can depend on the order in which it is executed (e.g., if it is executed before or after another that costs a certain amount) [8]. A miner can exploit that fact to order transactions in a way that is convenient for its purposes.

Timestamp Dependence is a security problem that a contract may have when using the block timestamp as a triggering condition to execute some operations [15]. All indirect or direct uses of the timestamp should be considered since the timestamp of the block can be manipulated by the miner.

Integer overflow and underflow vulnerabilities are known for long but can also appear in smart contracts [15].

There are several other classes of vulnerabilities in smart contracts [8], [15], [16].

## III. Fuzzing Ethereum Smart Contracts

We propose a tool to fuzz smart contracts, i.e., for invoking transactions in loop until vulnerabilities are discovered. The basic architecture of the tool is represented in Figure 2. The main components are the following:

- *Ethereum:* This component is a private deployment of the Ethereum blockchain.
- *Fuzzer:* This component manages the injection process. It (optionally) starts by setting up the state of whole blockchain (e.g., restarting it) or of the contract in the blockchain (e.g., creating a new contract even if with the same code), then issues one or more transaction requests to the contract, and finally repeats the whole process. As many other fuzzers, the transaction inputs will be generated using a combination of predefined tokens (called fuzzing vectors) and random values.
- *Monitor:* This component is concerned with detecting if a transaction exploited a vulnerability. This is a challenging task that can be achieved in different ways. The most common is detecting conspicuous events such as a crash or high use of memory, CPU and other system resources [7]. Our approach involves extracting a trace of the execution using a debugger, then analyzing it.

Figure 3 shows an example of a smart contract vulnerable to integer underflow and overflow. When a transaction requests the execution of function `underflow` (resp. `overflow`), there may be an integer underflow (resp. overflow) of variable zero (resp. max). Detecting such an attack involves analyzing the debug trace searching for such events.

We are currently developing a prototype of our fuzzing tool. To run Ethereum we use Truffle, a development environment and testing framework for Ethereum. To obtain the debug trace we use the debugger provided by Tuffle. The injector itself and the monitor are being implemented by us.

```
contract OverflowUnderFlow {
   uint public zero = 0;
   uint public max = 2**256-1;

   // zero will end up at 2**256-1
   function underflow() public {
      zero -= 1;
   }

   // max will end up at 0
   function overflow() public {
      max += 1;
   }
}
```

Fig. 3. Example vulnerable smart contract written in Solidity

This overview of the projects suggests that there are still many open questions: (1) Which classes of smart contract vulnerabilities can be detected using fuzzing? (2) How can they be detected and what is the simplest way to detect each one? (3) Do we need the blockchain to have several replicas or is a single replica sufficient to detect all smart contract vulnerabilities? (4) How can we make fuzzing efficient, dealing with state explosion? (5) Is it possible to parallelize the fuzzing process?

*Acknowledgements.* Research supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2013 (INESC-ID) and UID/CEC/00408/2013 (LASIGE).

## References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[2] V. Buterin and Ethereum team, "Ethereum - a next-generation smart contract and decentralized application platform," 2014-17, white Paper.
[3] N. Szabo, "The idea of smart contracts," *Nick Szabo's Papers and Concise Tutorials*, 1994.
[4] K. Finley, "A $50 million hack just showed that the DAO was all too human," 2016.
[5] H. Qureshi, "A hacker stole $31M of Ether — how it happened, and what it means for Ethereum," 2017.
[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[7] J. Antunes, N. F. Neves, M. Correia, P. Veríssimo, and R. F. Neves, "Vulnerability discovery with attack injection," *IEEE Transactions on Software Engineering*, vol. 36, no. 3, pp. 357–370, 2010.
[8] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," *Proceedings of the ACM SIGSAC CCS'16*, pp. 254–269, 2016.
[9] A. Rea, "Solidity-coverage - Code coverage for Solidity testing." https://github.com/sc-forks/solidity-coverage, 2017.
[10] T. o. Bits, "Manticore - Dynamic binary analysis tool with EVM support," https://github.com/trailofbits/manticore, 2017.
[11] B. Mueller, "Mythril - Reversing and bug hunting framework for the Ethereum blockchain," https://github.com/b-mueller/mythril/.
[12] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018, pp. 66–77.
[13] "Solidity," https://solidity.readthedocs.io/en/develop/.
[14] A. Juels, A. Kosba, and E. Shi, "The Ring of Gyges: Investigating the Future of Criminal Smart Contracts," *Conference on Computer and Communications Security*, pp. 283–295, 2016.
[15] ConsenSys, "Smart Contract Security Best Practices," https://github.com/ConsenSys/smart-contract-best-practices/, 2017.
[16] I. Sergey and A. Hobor, "A Concurrent Perspective on Smart Contracts," pp. 1–15, 2017.