UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Denys Kovalenko

Static analysis of smart contracts as a way to finding security bugs

Literature survey

Supervisor: Luciano Banuelos

Tartu 2017

Table of Contents

Introduction

Blockchain entered our world with Bitcoin launched in 2009. Blockchain is a public, decentralized ledger of transactions, which means records can only be added, and not modified/deleted. Many computers connected to Internet store this ledger, and when new records is about to be added, those computers, called "miners" are checking whether transaction is valid (i.e. there is no double-spenditure problem - someone tries to spend money they have already spent) and are solving cryptographic puzzle to add record to block. These blocks are connected intro chain mathematically, that's why it's called blockchain.
One of the main implications of blockchain today is cryptocurrency, with current market cap leader Bitcoin, followed by Ethereum. But many reasons have caused development of next-generation platforms, that are capable of more sophisticated tasks, using so called smart contracts.
Smart contract is a piece of software running on blockchain.
Today smart contracts start getting more and more important role in IT infrastructure, and carry more monetary value, that's why it's important to have them behave in a predictable way, according to specifications and with high level of security, at least the same we require from banking applications.

In case of Ethereum (Ethereum.org) which will be used as a reference platform smart contract is represented as EVM (Ethereum Virtual Machine) code - turing complete language (which means it can solve any problem that can be represented by algorithm), and there are high-level languages such as Solidity that get compiled into EVM code.

Smart contract gets executed when message (with some monetary value - eg. some "amount" of cryptocurrency) is transferred. As it requires electricity and CPU time for miners to execute contract's code, this should be somehow rewarded. One of the ways was to take a transaction fee but Ethereum have chosen another part - concept of "Gas" as resource needed for running software. Each transaction that fires smart contract requires some amount of Gas, which depends on complexity of code. If not enough gas was handled over to contract to run all lines, exception is thrown, code stops executing and due to transaction-like nature of smart contracts all changes are reverted and no effect is produced, and also already consumed gas isn't refunded. Most popular platform now is Ethereum (Ethereum.org) , and recently IBM have announced that they are launching blockchain with smart contracts capability. [5]

Although, smart contracts become more popular, it's still in it's infancy - community is small, there are no guidelines and best practices polished, which leads to many smart contracts are deployed having bugs in them. That's why this paper is aimed to find tools to prevent buggy contracts from being deployed and to make world of smart contracts more secure and beneficial.
We have to imply security-first approach to development of smart contracts, rather than try-break-fix, reasons for that are inside nature of smart contracts and are discussed later in text.

Research question

Following **research questions** were identified and will be pursuit:

1) What are the security bugs to which smart contracts are vulnerable?
2) How they can be identified?
3) How they can be avoided or fixed?

I have identified following search **keywords**:

1) Static analysis smart contracts
2) Smart contracts bugs

Criteria for selection:

- Article shouldn't be older than **5 years**
- It has to address **smart contracts**
- It has to express some **vulnerabilities** of smart contracts or **tools**

To find answers to this questions I have searched for literature in following resources:

Results from http://ieeexplore.ieee.org/

Included: NONE. Articles at this resources weren't related to smart contracts at all

Excluded:

1) Improved network analysis by using data from smart meters
2) LeakMiner: Detect Information Leakage on Android with Static Taint Analysis
3) Experimental characterization of indoor multi-link channels
4) All other articles for "smart contracts bugs" search term

Results from http://scholar.google.com

Included:

1) Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab
2) Making smart contracts smarter
3) A survey of attacks on Ethereum smart contracts
4) Formal verification of smart contracts: Short paper

Excluded:

1) Town crier: An authenticated data feed for smart contracts
2) Capability-based financial instruments

Results from http://dl.acm.org/

Included:

1) Making smart contracts smarter

Review

Root causes of bugs

Main cause of bugs is that majority of developers come from traditional software industry, that encourages try-fail-fix approach, as many problems can be reverted after being identified by testers/users. Also, development of smart contracts require economical way of thinking which is unusual for developers. And finally, smart contracts' execution flow can be represented by state machine, which might have logical problems, but developers are not used to visualising their code as state machine and in general tend not to do formal verification, in best case some edge-value based testing.
So logical flaws in design can be used by malicious users to benefit from.

But more important in the context of security bugs is another reason. There is a difference in semantic of Ethereum and developer's perception of that semantic, which leads to bugs. It means that developers don't know some limitation and implementation details of Ethereum platform, which leads to inconsistency in contract executing and gives a field for manipulation to benefit malicious user or just pure losses for everyone (when value (ethers) is stuck forever in smart contract, like in a black hole).

Bugs overview

Many bugs in smart contracts are introduced by developers who don't consider malicious behaviour of users.

1) **Transaction-Ordering Dependence.** Block includes several transactions for example, Ti, Tj that manipulate with same contract. Depending on order between Ti and Tj contract might be in different state. So, it's uncertain in which state contract will be when transaction is added to the block, and only the miner who "mines" the block can choose order of these transactions, and in that way order of update of contract state. For example, contract specify valuable reward for solving puzzle and submitting solution, (find value hash of which is smaller than some number) and this reward can be modified by contract owner. Owner can set reward to zero and if this transaction get it the same block where someone has submitted solution, and miner decides to put owner's transaction first user won't get any reward, which was unexpected. We say that such transactions are order-dependent [2]
2) **Blockhash Bug**. Another Ethereum-specific quirk is that the block.prevhash instruction supports only the 256 most recent blocks, presumably for efficiency reasons. This limitation also affected Etherpot (lottery based on smart contract) and potentially other contracts that went into production. Miller proposed one potential fix to this problem by

implementing a global “blockhash service” contract that allows other contracts to retrieve block hashes beyond 256 blocks. [1].

3) **Timestamp dependence.** In another article [2] there problem is referenced as “timestamp dependence”. As I found out, both blockhash and timestamt bugs refer to contract that can be found in references [2] it was references not because of 256 last blocks limitation, but the fact that timestamp of the block is used as a salt to random number generator. In general, when block is added to blockchain it uses local timestamp of proposer’s machine, and most miners should agree on timestamp. But miner can propose timestamp in a range of +- 9000 seconds and it still will be accepted, so malicious miner can precompute outcomes of smart contract and change timestamp that would be beneficial for him.
4) **Mishandled Exceptions.** In Ethereum, contracts can be called by another contract via send instruction or call another contract’s function directly (for example, aotherContract.function1()). If there is an exception raised in the callee contract, the callee contract terminates, reverts state and yeilds false as result. This can be caused by not enough gas, exceeding call stack limit or other reasons..If the call is made via the send instruction, the caller should explicitly check the return value to see whether callee was executed successfully, but there is no such requirement when calling method directly. This can break transaction atomicy, and KingOfEtherThrone [3] smart contract which transfer “crown” to user that wants to pay more for status of kind and returns money back to previous kind. exception in call returning money to previous king will result in transfer of the king status even if payment back wasn’t made.
5) **Reentrancy Vulnerability** With a recent TheDao (Decentralized Autonomous Organization) hack [4], where the attacker exploited this to steal over 3, 600, 000 Ether (60 million US Dollars at the time when it happened). Basically, this vulnerability refers to problem with multithreading, which is well known among software developers. Smart contract has 2 types of memory - one is used as temporary memory during execution of contract call, and another is storage representing persistent state of contract. And as some operation may take long time to execute, the same contract can be called again with old state. Imagine bank system, that store account balance of a client, and this balance is 100 dollars. Hacker sends many withdrawal requests almost at the same time, and until value of his account hasn’t been set to 0 we can make smart contract to send money to him multiple times.

According to “making smart contracts smarter” [2] smart contracts on Ethereum hold value totaling to 30 million US dollars, a lot of contracts have one of the mentioned above vulnerabilities.

Which means that the are a lot vulnerabilities and a lot of contracts with at least one of them that can be exploited.
And as mentioned above, we can’t just patch or redeploy smart contract. Once published, it can’t be modified.

And also, as opposite to any website Ethereum can’t just make update to protocol to fix issues highlighted above and make users use new version. Protocol change require all users of network to update their software.

Which means it’s critically important to do throughout check of contract to identify any problems beforehand.

Tools for identifying security bugs

As identified above, we should perform security checks on smart contracts, either when we are on a developer's side to ensure contract will work as supposed, or when we want to use any contract to be sure that we will get what expect and won't become victim of hackers.

And there are possibilities to do that.
Singapore scientists have developed tool called Oyente, that performs static analysis of smart contract and searches for security issues mentioned in "bugs overview" section

Oyente is small program in Python with few thousands lines of code. Currently, it uses Z3 [6] as solver to decide satisfiability. It simulates Ethereum Virtual Machine (EVM) code..

This tool is based upon symbolic execution, that represents program values as symbolic expressions of the input symbolic values. It reasons about a program path-by-path, compared to input-by-input.
It has 4 main components:

1) **CFG Builder.** It creates a skeletal control flow graph with all the basic blocks as nodes, and some edges representing jumps to the corresponding source nodes.
2) **Explorer**. It has a loop that gets a state and instruction to run, and does transition
3) **Core Analysis.** detect contracts that are transaction-order-dependent, timestamp-dependent or have mishandled exceptions. It detects differences in ether-flow with different order of transactions in the block and so can find whether contract is transaction-order-dependent
4) **Validation**. The last component attempt to remove false positives - contracts marked as buggy which are not in reality

Oyente is built on EVM bytecode, nevertheless most contracts are written in high-level Solidity because only a small fraction of contracts have source code publicly available.

It has identified many contracts having security issues, as shown below.

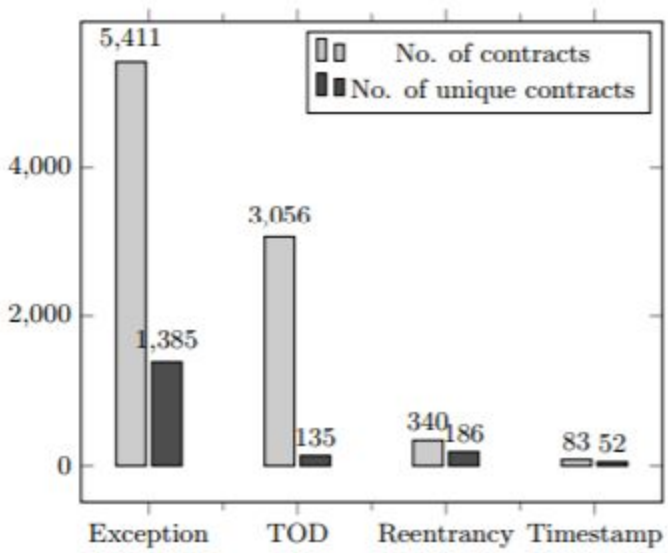

Figure 1. Number of contracts with bug per each security problem reported by Oyente tool, among analysed.

Another approach is to use formal verification language like F*[7]. Researches have implemented EVM* and Solidity* in OCaml to show problem with reentrancy vulnerability described above.

Conclusion

Smart contracts is emerging field that still has few tools to perform security check, compared to other fields of programming. And there is need to introduce security-first approach among developers as well as make existing tool Oyentee highly available and widespread to ensure that new contracts won't get deployed with obvious problems, and that user's won't have their ethers stolen.
In future, many organisation will rely on smart contracts in their day-to-day business activities, so it's worth exploring this emerging field more.

This literature review have given such answers to research questions:

- What are the security bugs to which smart contracts are vulnerable?

Alhtough there are many possibilities to get buggy smart contract, most problems arises from gap in understanding of semantic of programming languages, such as Solidity and perception of their semantics, meaning that developer don't know how code will behave. There are 5 main types of such problems: transaction-ordering dependence, timestamp dependence/blockhash bug, reentrance vulnerability and most popular - mishandled exceptions.

- How they can be identified?

There are few tools for formal analysis - using Oyentee tool, which is performs static analysis of smart contract and shows potential problems with 5 listed bugs; and there is F* formal language that can help to prove that contracts will behave as expected

- How they can be avoided or fixed?

Now it's impossible to replace contract, or cancel or alter it, that's why it's critical to be sure that deployed contracts is valid. So using such tool as Oyentee can be helpful. Probably, Ethereum.org should include it into their web IDE for Solidity programming.

Further research should be conducted to find more tools for smart contract analysis, which may be already out there but haven't got that much attention; and considering that smart contracts are in their infancy many more tools are about to come soon

References

[1] Delmolino, Kevin and Arnett, Mitchell and Kosba, Ahmed and Miller, Andrew and Shi, Elaine. (2016). Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab

[2] Luu, L., Chu, D.H., Olickel, H., Saxena, P. and Hobor, A., 2016, October. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (pp. 254-269). ACM.

[3] KingOfTheEtherThrone smart contract. https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol

[4] TheDAO smart contract. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code

[5] Atzei, N., Bartoletti, M. and Cimoli, T., 2016. *A survey of attacks on Ethereum smart contracts*. Cryptology ePrint Archive: Report 2016/1007, https://eprint. iacr. org/2016/1007.

[6] Microsoft Corporation. The Z3 theorem prover. https://github.com/Z3Prover/z3.

[7] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N. and Zanella-Béguelin, S., 2016, October. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (pp. 91-96). ACM.