Verification of Ethereum Smart Contracts Written in Vyper

Master's Thesis Project Description

Robin Sierra
Supervisors: Marco Eilers, Prof. Dr. Peter Müller
ETH Zürich, Switzerland

28.3.2019

1 Motivation

Blockchain-based cryptocurrencies like Bitcoin [1] offer an alternative to traditional fiat currencies. Some of them, e.g. Ethereum [2], allow the deployment of *smart contracts* which enable users to perform (monetary) transactions exactly as defined in the contract without using a trusted third party. These contracts are usually written in high-level progamming languages like Solidity [3] and subsequently compiled into bytecode that can be interpreted by the Ethereum Virtual Machine (EVM). Due to the monetary nature of the contracts it is vital that they are correctly implemented, especially since they cannot be removed from the blockchain once deployed. The DAO vulnerability [4] is an example of a bug in a contract that led to a loss of 60 million dollars.

To increase safety of smart contracts and to avoid common pitfalls of Solidity, the Ethereum team is currently developing the Vyper language [5] as an alternative. Vyper offers a clean Python-like syntax that is supposed make it as hard as possible to write confusing code. However, while this language removes some pitfalls present in Solidity, writing smart contracts remains challenging. Listing 1 shows an example of a vulnerable Vyper contract that can be exploited to steal money from the contract. It pays a reward to the first client calling `claim_reward`. In line 3 we define a Boolean `did_pay` indicating whether the reward has already been paid, and in line 4 we define its amount. The values get initialized in `__init__` just like in Python. The `raw_call` method is used to call another contract and transfer money to it. The intended effect of calling `claim_reward` is that the money gets sent to the caller once by using `raw_call`, after that, `did_pay` is set to `True` and the method will do nothing on subsequent calls. The problem, however, is that calling another contract not only transfers money to it but allows the contract to execute (arbitrary) code when it receives the money.

A malicious caller could therefore invoke `claim_reward` a second time on receiving the money before `did_pay` has been set to `True`. Therefore the test in line 13 will return `True` again and the money is sent twice. This process can be repeated until the contract runs out of money. A similar reentrancy bug caused the aforementioned loss of money for the DAO. To fix the contract, line 17 should come before lines 15/16.

```
1  # Reward Payment
2
3  did_pay: bool
4  reward: wei_value
5
6  @public
7  def __init__(amount: wei_value):
8      self.did_pay = False
9      self.reward = amount
10
11 @public
12 def claim_reward():
13     if (not self.did_pay):
14         # send reward to the caller
15         raw_call(msg.sender, b"", outsize=0,
16             value=self.reward, gas=msg.gas)
17         self.did_pay = True
```

Listing 1: Example of a vulnerable Vyper program implementing a contract to pay a reward to the first one claiming it.

A more subtle problem with the contract is that the outcome depends on the order transactions are executed in. On the Ethereum blockchain miners control which transaction is executed next based on the transaction fee they receive. Therefore, a miner that tries to claim the reward can influence the transaction ordering such that their own transaction has a higher chance of being chosen.

Various other potential vulnerabilities exist. Atzei et al. [6] provide a classification of such vulnerabilities and list attacks that can be carried out. Some (like type casts and non-uniform handling of exceptions) are not present in Vyper, but bugs can also be caused by faulty assumptions about how EVM bytecode and the blockchain work. As a result, there has been an effort to develop static analysis and verification tools that prove correctness of smart contracts. Grishchenko et al. [7] formulate properties that can be checked by such a tool to prove absence of certain errors. For example, in order to avoid the second vulnerability of the contract in List-

ing 1 the contract would have to be *independent of mutable account state*, i.e., the amount of money flowing out of the contract cannot be affected by previous executions, which seems overly restrictive. To avoid the reentrancy vulnerability the contract would need satisfy *call integrity*, i.e., the behavior of the contract must not depend on untrusted attacker code. Call integrity, just like many other desired properties, is a hyperproperty (a property of sets of executions as opposed to properties of single executions) which are hard to analyze automatically.

More generally, contracts calling each other are very similar to multiple threads accessing shared memory since arbitrary state changes can happen unexpectedly [8]. There are various tools for verifying concurrent programs, e.g. the Viper[1] verification infrastructure [9]. There exists a Python front-end for Viper, Nagini [10], that supports verification of hyperproperties for information flow security [11]. It does so by using *Modular Product Programs* [12], a technique that makes it possible to verify hyperproperties by reducing them to trace properties of a transformed program. Because of that, and due to the fact that Vyper is a Python-like programming language, Nagini offers an excellent basis for a Vyper verifier to build upon.

2 Core Goals

The goal of this project is to build a static verifier for Ethereum smart contracts written in the Vyper language. The verifier will be based on Nagini. The following steps are required:

- Search for important security problems of smart contracts in the literature, in particular consider [6] and [13].
- Assess whether the security problems have already been solved in the Vyper language (compared to Solidity) or to what extent, and if not, whether their absence can be proved using the Viper infrastructure with hyperproperties, e.g. by proving the properties outlined in [7]. If so, provide an encoding into Viper.
- Design a speficiation language to express the security properties that should be verified. The language should be expressive enough to verify simple token implementations following the ERC-20 standard.
- Build a verifier for smart contracts written in Vyper. Since Vyper is a Python-like language and Nagini already supports verification of hyperproperties the verifier will be able to reuse a large portion of Nagini code. The verifier will provide the specification language designed in

[1] Not to be confused with the Vyper language for Ethereum smart contracts

an earlier step to state what properties the contract should have. All Vyper features needed for ERC-20 with the exception of event handling should be supported.

- Evaluate the verifier by using it on example contracts, in particular ERC-20 implementations.

3 Extension Goals

- Expand the supported language features beyond what is needed to verify ERC-20. Verify more advanced examples and evaluate usability of the verifier for those.
- Support proving secure information flow, i.e., that the contract does not leak secret information. Vyper allows private fields, however, this does not guarantee that the data itself is private. Since the transaction execution is publicly available on the blockchain, anyone can inspect the operations that were executed and infer the private state of the contract. Therefore, cryptographic techniques like hashing and commitments have to be used.
- Integrate event logging into the verifier by providing a way to specify what events are logged under which circumstances.
- Provide a soundness argument or proof for (a subset of) the encoding.

References

[1] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2008. http://www.bitcoin.org/bitcoin.pdf.

[2] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, 2014. https://ethereum.github.io/yellowpaper/paper.pdf.

[3] *Ethereum Foundation: The Solidity Contract-Oriented Programming Language*. https://github.com/ethereum/solidity.

[4] *Understanding the DAO attack*. http://www.coindesk.com/understanding-dao-hack- journalists.

[5] *Ethereum Foundation: Pythonic Smart Contract Language for the EVM*. https://github.com/ethereum/vyper.

[6] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, (New York, NY, USA), pp. 164–186, Springer-Verlag New York, Inc., 2017.

[7] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust* (L. Bauer and R. Küsters, eds.), (Cham), pp. 243–269, Springer International Publishing, 2018.

[8] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," *CoRR*, vol. abs/1702.05511, 2017.

[9] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, (New York, NY, USA), pp. 41–62, Springer-Verlag New York, Inc., 2016.

[10] M. Eilers and P. Müller, "Nagini: A static verifier for python," in *Computer Aided Verification* (H. Chockler and G. Weissenbacher, eds.), (Cham), pp. 596–603, Springer International Publishing, 2018.

[11] S. Meier, "Verification of information flow security for python programs," Master's thesis, ETH Zürich, 2018.

[12] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," in *Programming Languages and Systems* (A. Ahmed, ed.), (Cham), pp. 502–529, Springer International Publishing, 2018.

[13] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 254–269, ACM, 2016.