Verification of Advanced Properties for Real World Vyper Contracts

Master's Thesis Project Description

Christian Bräm
Supervisors: Marco Eilers, Prof. Dr. Peter Müller
ETH Zürich, Switzerland

April 16, 2020

1 Motivation

Smart contracts are programs running on a distributed, decentralised blockchain network, like Ethereum [11]. The contracts facilitate agreements between multiple parties without the need of a trusted third party. The code of a contract specifies rules how an agreement can be established. If others fulfill the conditions of the contract, they conclude the agreement and the defined consequences are automatically executed. Hereby the principle *Code is law* applies [8]. This means not the intention behind a smart contract is legally binding but just the code.

If the smart contract has an unintended behavior from the perspective of the programmer and the smart contract is already deployed to the blockchain, the programmer cannot do anything to change the contract anymore. The attack on the *TheDAO* contract and the resulting theft of $50M worth of Ether [2] was only possible since this contract had a bug. Only a fork of the blockchain could undo these unwanted transactions.

Therefore, it is very important that writers of smart contracts can specify what their intentions are and that the code of their contracts can then be automatically verified. There are now various verifiers (for example: [1] [6] [7]) that try to help a programmer to verify some specification, one of the verifiers is *2VYPER* [9]. It allows users to formally specify the contract using annotations in the form of code comments.

With *2VYPER*, the behavior of the functions of a contract can be specified. But *2VYPER* is currently not able to verify some advanced properties of smart contracts.

For example, two such advanced properties would be

- invariants between contracts or
- properties of contracts with highly nonlinear arithmetic.

Inter-Contract Invariants Many popular smart contracts like UniSwap [12] and StableSwap [4] maintain invariants between multiple contracts.

Listings 1 and 2 show an example of such a contract. Here, the *interContractIncrease* in Listing 2 tries to maintain a constant difference between two *simpleIncrease* contracts of Listing 1. An invariant of the *interContractIncrease* contract could be: *self.token_A.get() - self.token_B.get() == _diff*. We cannot verify this invariant without further information about the return value of the *get* function in the *simpleIncrease* contract. This value can only be modified by the *increase* function of the same contract.

The execution of smart contracts takes place in a distributed and decentralised environment. Therefore, *increase* of *simpleIncrease* could be called by many others. Nevertheless, the invariant actually holds, since the amount of the *token_A* and *token_B* can only be changed by the *interContractIncrease* contract. Other callers of *increase* would have different addresses in *msg.sender* on line 7 of Listing 1.

Only with these or similar arguments, a verifier is able to prove such an invariant.

```
1  # simpleIncrease
2
3  amounts: map(address, uint256)
4
5  @public
6  def increase() -> bool:
7      self.amounts[msg.sender] += 1
8      return True
9
10 @public
11 @constant
12 def get() -> uint256:
13     return self.amounts[msg.sender]
```

Listing 1: A simple contract that stores an amount for each unique address.

```
1  # interContractIncrease
2
3  import simpleIncrease as Token
4
5  token_A: Token
6  token_B: Token
7  _diff: uint256
8
9  @public
```

```
10 def __init__(token_A: address, token_B: address):
11     self.token_A = Token(token_A)
12     self.token_B = Token(token_B)
13     value_A: uint256 = self.token_A.get()
14     value_B: uint256 = self.token_B.get()
15     assert(value_A >= value_B)
16     self._diff = value_A - value_B
17
18 @public
19 def increase() -> bool:
20     result: bool = False
21     if self.token_A.get() != MAX_UINT256 \
22       and self.token_B.get() != MAX_UINT256:
23         self.token_A.increase()
24         self.token_B.increase()
25         result = True
26     return result
```

Listing 2: A contract that maintains a constant difference between two *simpleIncrease* contracts.

Nonlinear Arithmetic Often smart contracts execute some nontrivial and most importantly nonlinear math calculations. Since SMT solvers' theory (e.g. Z3 [3]) of nonlinear arithmetic tends to be slow and unstable and it is undecidable in the integer domain, they have to rely only on heuristics. Just some small code changes can lead to unpredictable verification failures [5].

Real world contracts like StableSwap contain many nonlinear expressions. For example StableSwap uses Newton's method to get the roots of a nonlinear function. An example implementation of Newton's method in Vyper can be seen in Listing 3. In this example the function would be x^2-4x+3 and the zeros of this function would be 1 and 3. Therefore, a possible postcondition of this function could be: $\neg loop\_break \vee (x > 0 \wedge x < 4)$.

The update step of Newton's method on line 13 of Listing 3 contains a nonlinear expression. Since we have nonlinear integer arithmetic, this is undecidable for SMT solvers and we would not be able to prove this postcondition.

```
1 # newtonsMethod
2
3 @public
4 @constant
5 def newton(start_value: int128) -> int128 :
6     if start_value == 2:
7         start_value = start_value - 1
8     x_prev: int128 = 0.0
9     x: int128 = start_value
10    loop_break: bool = False
11    for _i in range(255):
12        x_prev = x
```

```
        x = x - (x*x - 4*x + 3) / (2*x - 4)
        if x > x_prev:
            if x - x_prev <= 1:
                loop_break = True
                break
        else:
            if x_prev - x <= 1:
                loop_break = True
                break
    return x
```

Listing 3: A contract that calculates the zeros of $x^2 - 4x + 3$ using Newton's method.

2 Core Goals

The goal of this project is to support more complex contracts in *2VYPER*. The following steps are required:

- **Inter contract invariants:** Design and implement a way for users to express invariants between smart contracts and enable them to prove properties of these inter-contract dependencies.
- **Nonlinear arithmetic:** Design and implement one approach to deal with nonlinear arithmetic, so that the SMT solver does not use its heuristics. For example, one approach could be to enable pre-defined lemmas (e.g. $(x > 0 \land y > 0) \Rightarrow xy > 0$) as well as lemmas specified by the user. Using those lemmas, the SMT solver does no longer depend on its heuristic [5]. Another approach could be to change integers into reals, since nonlinear real arithmetic is decidable for SMT solvers [10].
- **Make verification more scalable:** Currently, in *2VYPER* function calls get inlined and loops get fully unrolled before the verification. This leads to a performance loss during verification.
  This goal, to enable fast verification of real world contracts with advanced properties, is twofold. First, we have to find and identify such performance losses. Second, we have to design and implement a way so that *2VYPER* must no longer rely on these parts.
- **Enable a way to referring to constant functions:** Vyper supports an annotation to declare a function to have no side effects on the contract state. We want to design and implement a way so that one can refer to this functions in the formal specification of *2VYPER*. For example in an invariant or a postcondition one can write ($f() > old(f())$), where f would be such a constant function.
- **Evaluation:** Evaluate each of the features mentioned in the previous goals on real world contracts.

3 Extension Goals

- **Comparison of nonlinear approaches:** As described in the Core Goals, there are multiple approaches in dealing with nonlinear arithmetic. Another approach should be designed and implemented, so that an evaluation and comparison of these two can be established.
- **StableSwap evaluation:** Prove correctness properties of the StableSwap contract.

References

[1] Karthikeyan Bhargavan et al. "Formal Verification of Smart Contracts: Short Paper". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. 2016, pp. 91–96.

[2] Phil Daian. *Analysis of the DAO exploit*. 2016. URL: `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/` (visited on 03/31/2020).

[3] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008, pp. 337–340.

[4] Michael Egorov. *stableswap*. 2020. URL: `https://github.com/curvefi/curve-contract/blob/master/vyper/stableswap.vy` (visited on 03/31/2020).

[5] Chris Hawblitzel et al. "Ironclad Apps: End-to-End Security via Automated Full-System Verification". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 165–181.

[6] Everett Hildenbrandt et al. "KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 2018, pp. 204–217.

[7] Yoichi Hirai. "Defining the Ethereum Virtual Machine for Interactive Theorem Provers". In: *International Conference on Financial Cryptography and Data Security*. 2017, pp. 520–535.

[8] Lawrence Lessig. "Code is law". In: *The Industry Standard* 18 (1999).

[9] Robin Sierra. "Verification of Ethereum Smart Contracts Written in Vyper". MA thesis. 2019.

[10] Alfred Tarski. "A Decision Method for Elementary Algebra and Geometry". In: *Quantifier Elimination and Cylindrical Algebraic Decomposition*. 1998, pp. 24–84.

[11] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[12] Yi Zhang, Xiaohong Chen, and Deajun Park. *Formal Specification of Constant Product (x*y = k) Market Maker Model and Implementation*. 2018.