Analyzing Safety of Smart Contracts

Mohan Dhawan

IBM Research

Blockchain is the design pattern that underpins the Bitcoin cryptocurrency [23]. However, its use of consensus to validate interaction amongst participant nodes is a key enabler for applications that require mutually distrusting peers to conduct business without the need for a trusted intermediary. One such use is to enable a smart contract, which programatically encodes rules to reflect any kind of multi-party interaction. With over $1.4 billion invested in blockchain last year [1], and the increasing trend towards autonomous applications, smart contracts are fast becoming the preferred mechanism to implement financial instruments (e.g., currencies, derivatives, wallets, etc.) and applications such as decentralized gambling.

While the faithful execution of a smart contract is enforced by the blockchain's consensus protocol, it remains the prerogative of the participating entities to (i) verify the smart contract's *correctness*, i.e., the syntactic implementation follows the best practices, and (ii) validate its *fairness*, i.e., the code adheres to the agreed upon higher-level business logic for interaction. While manual auditing of contracts for correctness is possible to an extent, it still remains laborious and error prone. Automatic formal auditing, on the other hand, requires specialized tools and logic. The problem is exacerbated by the fact that smart contracts, unlike other distributed systems code, are immutable and hard to patch in case of bugs, irrespective of the money they hold. For example, investors in TheDAO [10] lost cryptocurrency worth around $50 million because of a bug in the code that allowed an attacker to repeatedly siphon off money [11]. In this paper, we tackle the problem of formal verification of smart contracts, since reasoning about their correctness and fairness is critical before their deployment.

The smart contract in Fig. 1 advertises a 15% profit payout to any investor. However, the contract has both correctness and fairness issues. First, the arithmetic operation in line 6 can potentially overflow, which is a correctness bug. Second, the variable `index` never increments within the loop, and thus the payout is made to just one investor. Finally, the `break` statement exits the loop after payment to the first investor, who is the contract owner. Thus, the contract does not payback any other investor. The last two bugs result in fairness issues.

Most prior art in the area of smart contracts deals with security and/or privacy concerns in designing them [17, 18, 20, 25]. There is, however, little work that analyzes smart contracts for vulnerabilities [12, 14, 21]. Oyente [21] uses symbolic execution for bug detection at the bytecode level, but it is neither sound nor complete. Thus, it can result in several false alarms even in trivial contracts, as we observed and communicated to Oyente's developers [6]. Since it is very hard to recreate the intent from the bytecode alone (due to loss of contextual information such as types, reuse of same bytecode for different function calls, etc.) several fairness and correctness issues, including integer overflow/underflow amongst others, are thus completely ignored by

```
(1) while (Balance > (depositors[index].Amount * 115/100)
                  && index<Total_Investors) {
(2)   if(depositors[index].Amount!=0)) {
(3)     payment = depositors[index].Amount * 115/100;
(4)     depositors[index].EtherAddress.send(payment);
(5)     Balance -= payment;
(6)     Total_Paid_Out += payment;
(7)     depositors[index].Amount=0; //remove investor
(8)   } break;
(9) }
```

Fig. 1: An unfair contract (adapted from [8]).

Oyente. Further, it conservatively handles loops [1] [7, 9] (with a bound of one) resulting in under approximation of loop behavior, and thus fails to detect the two fairness bugs in Fig. 1.

Bhargavan *et al.* [14] propose a framework to formally verify smart contracts written in a subset of Solidity using F*, which leaves out important constructs, such as loops. Considering the 22,493 contracts that we analyzed, around 93% contained loops. Thus, their tool will operate on a fraction of publicly available contracts, which is also corroborated by their results; they could evaluate only 46 out of 396 contracts. While use of F* may enable reasoning about most correctness and fairness properties, the authors suggest that such reasoning may require manual proofs. Although it is unclear when such situations may arise. In contrast, we establish that completely automated verification enables analysis of published contracts at a much larger scale. Why3 [12] is an experimental tool for formal verification of Solidity contracts, which is under active development and supports only a small subset of the entire syntax [4]. Further, Solidity to Why3 translation has not yet been tested and thus cannot be trusted [3].

We present the design and implementation of ZEUS—a practical framework for automatic formal verification of smart contracts using abstract interpretation and symbolic model checking. ZEUS takes as input the smart contracts written in high-level languages and leverages user assistance to help generate the correctness and/or fairness criteria in a XACML-styled template [13]. It translates these contracts and the policy specification into a low-level intermediate representation (IR), such as LLVM bitcode [19], encoding the execution semantics to correctly reason about the contract behavior. It then performs static analysis atop the IR to determine the points at which the verification predicates (as specified in the policy) must be asserted. Finally, ZEUS feeds the modified IR to a verification engine that leverages constrained horn clauses (CHCs) [15, 16, 22] to quickly ascertain the safety of the smart contract.

ZEUS leverages three key observations to be both sound and scalable. First, while the blockchain has execution akin to a concurrent system with task-based semantics, a transaction comprises of just one call chain starting from a publicly visible function in the smart contract. This observation helps significantly reduce the state space exploration for verifying most properties. Also, data dependence across transactions, such as read/write hazards among persistent state variables, requires analyzing $O(n^2)$ pairs of transaction interleavings. Second, smart contracts are both control- and data-driven. Thus, modeling contracts using abstract interpretation along with symbolic

[1] Oyente is under active development and future releases could add more features and reduce the false alarms.

model checking allows ZEUS to soundly reason about program behavior. Abstract interpretation computes loop and function summaries over data domains, which are then used during the model checking phase that now operates upon a reduced state space. Lastly, CHCs provide a suitable mechanism to represent verification conditions, which can be discharged efficiently by SMT solvers.

ZEUS also benefits greatly from verification atop LLVM bitcode. Not only does this allow ZEUS to leverage an industry strength tool-chain for analysis, it enables ZEUS to plug in any verifier that operates upon the standardized (and formally verified [26]) LLVM bitcode. Use of LLVM bitcode also helps ZEUS to support verification of smart contracts for different blockchain platforms, including Ethereum [2] and Hyperledger Fabric [5] (or Fabric), written in diverse high-level languages, such as C#, GO and JAVA. Note that most high-level languages have mature source code to LLVM bitcode translators already available. We leverage LLVM's rich API set to develop the first Solidity to LLVM bitcode translator, which faithfully implements execution semantics for majority of the Solidity syntax for verification. Furthermore, use of LLVM passes allow ZEUS to separate translation from implementation of verification checks.

This paper makes the following contributions:

(**1**) We classify several new and previously known issues but unstudied in the context of smart contracts and show that they can potentially lead to loss of money.

(**2**) We present a formal abstraction of Solidity's execution semantics for verifying smart contracts using a combination of abstract interpretation and symbolic model checking.

(**3**) We present the design and implementation of ZEUS, a symbolic model checking framework for verification of correctness and fairness policies. We build the first Solidity to LLVM bitcode translator and provide a program analysis module that automatically inserts verification conditions given a policy specification. We also provide abstraction strategies to correctly model Solidity's execution semantics to ensure soundness. Further, we build an interactive predicate extraction tool to make it easy to specify policies for multi-party interactions in smart contracts.

(**4**) We present the first large scale source code analysis of Solidity-based smart contracts. Our evaluation with 22,493 Solidity smart contracts (of which 1524 were unique) indicates that about 94.6% of them (with a net worth of over $0.5 billion) are vulnerable to one or more correctness issues. However, we do not investigate the practical exploitability of these bugs. Additionally, we selected several representative contracts and applied contract-specific fairness criteria.

(**5**) ZEUS is sound (with zero false negatives) and outperforms Oyente for contracts in our data set, with low false positive rate and an order of magnitude improvement in time for analysis.

(**6**) We show ZEUS's generic applicability by leveraging it to verify smart contracts for the Fabric blockchain. We also demonstrate the ease of applying ZEUS to a verifier of choice by using SMACK [24] for verification.

References

1. Blockchain investment in 2016. https://www.cryptocoinsnews.com/pwc-expert-1-4-billion-invested-blockchain-2016/.

2. Ethereum. https://www.ethereum.org/.
3. Formal Verification and Ethereum. https://ethereum.stackexchange.com/questions/11092/what-is-formal-verification-and-why-is-it-important-for-smart-contracts.
4. Formal Verification for Solidity Contracts. https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts.
5. Hyperledger Fabric. https://hyperledger.org/projects/fabric.
6. Inian Parameshwaran. Personal Communication.
7. Loi Luu. Personal Communication.
8. Multiply your ether. https://etherscan.io/address/0xc357a046c5c13bb4e6d918a208b8b4a0ab2f2efd#code.
9. Oyente: An Analysis Tool for Smart Contracts. https://git.io/vFAlX.
10. The DAO. https://en.wikipedia.org/wiki/The_DAO_(organization).
11. The DAO is kind of a mess. https://www.wired.com/2016/06/biggest-crowdfunding-project-ever-dao-mess/.
12. Why3. http://why3.lri.fr/.
13. XACML. https://tools.ietf.org/html/rfc7061.
14. K. Bhargavan et al. Formal Verification of Smart Contracts: Short Paper. In *PLAS '16*.
15. N. Bjørner et al. Program Verification as Satisfiability Modulo Theories. In *SMT '12*.
16. A. Gurfinkel et al. The SeaHorn Verification Framework. In *CAV '15*.
17. A. Juels et al. The Ring of Gyges: Investigating the Future of Criminal Smart Contracts. In *CCS '16*.
18. A. E. Kosba et al. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *S&P '16*.
19. C. Lattner et al. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*.
20. L. Luu et al. Demystifying Incentives in the Consensus Computer. In *CCS '15*.
21. L. Luu et al. Making Smart Contracts Smarter. In *CCS '16*.
22. K. L. McMillan. Interpolants and Symbolic Model Checking. In *VMCAI 2007*.
23. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System.
24. Z. Rakamarić et al. SMACK: Decoupling Source Language Details from Verifier Implementations. In *CAV '14*.
25. F. Zhang et al. Town Crier: An Authenticated Data Feed for Smart Contracts. In *CCS '16*.
26. J. Zhao et al. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL '12*.