WIP: Finding Bugs Automatically in Smart Contracts with Parameterized Invariants

Thomas Bernardi* Nurit Dor* Anastasia Fedotov*
Shelly Grossman† Neil Immerman‡ Daniel Jackson§
Alexander Nutz* Lior Oppenheim* Or Pistiner *
Noam Rinetzky† Mooly Sagiv† Marcelo Taube*
John A. Toman* James R. Wilcox*

February 21, 2020

Abstract

This WIP paper describes our experience using formal verification to find bugs in smart contracts. Perhaps surprisingly, the most difficult part of the formal verification process is not the verification itself, but specification: that is, expressing the desired properties of the program. In the domain of smart contracts, we have found that the same invariants apply across many different software versions and platforms. This creates a potential network effect where the cost of formal verification drops as the repertoire of reusable invariants grows. We aim to jumpstart this process by: (i) sharing several invariants we have already identified and found to be useful and (ii) suggesting a specification framework that formalizes these invariants as parameterized Hoare triples. The second point addresses the challenge that differences arise between platforms in the exact form of the invariants. By adopting a common specification language, the community will be able to better communicate knowledge about important invariants between projects, and to collaborate in building tools that support reasoning about such invariants. Finally, we present some preliminary results from applying a tool for automatically checking invariants to some sample contracts.

1 Introduction

Smart contracts are computer programs that implement asset transactions. Unfortunately, smart contracts are hard to get right. Indeed, the Ethereum

*Certora
†Tel Aviv University
‡University of Massachusetts
§MIT

blockchain, which is currently the largest smart contract platform, has experienced several bugs in smart contracts that have led to large financial losses, including the Parity and DAO attacks, [15, 2].

The best way to ensure program correctness is *formal program verification*—the use of math and logic to *prove* that a program behaves appropriately. Indeed, formal verification has been already proposed as a mechanism to secure smart contracts (see, e.g., [7, 14, 17, 11, 1]). However, the literature lacks case studies where formal verification has been applied successfully during contract *development*, and not only after their deployment. The goal of this WIP paper is to demonstrate the value of formal verification in the development process as a mechanism for early bug detection.

Perhaps surprisingly, in our experience, the hardest problem in the formal verification process is stating what properties the smart contract must satisfy: The skill set required to implement an efficient program is very different from the one needed to rigorously formulate its mathematical properties. As a result, most deployed programs today do not have any formal specification, and thus cannot be checked with formal verification techniques.

This paper argues that smart contracts should be formally specified using **reusable** invariants. These invariants could then be evaluated and checked by the community of developers and users. We hope that this community-driven form of specification will create a network effect where invariants developed for one project are reused to secure different versions of the project as well as to secure other, different projects. Our aim is to jumpstart the collective specification process by defining a set of reusable invariants for smart contract correctness which are inspired by major Ethereum projects. These rules are defined informally in Section 2 and formally in Appendix A.

The process of applying a reusable invariant involves three steps. The first step is to state the invariant informally. For example, we might define an invariant Bounded Supply as saying that "the token supply is bounded" or equivalently "infinite minting is not possible." The second step is to formalize the invariant. This is a non-trivial task, because the meanings of the various terms in the informal invariant will differ across contracts. In this case, for example, the informal term "token supply" will be represented in different ways by different contracts, e.g. it could be explicitly stored as a state component, or implicitly computed as the sum of all account balances. We show how to tackle this representation problem using parameterized invariants. The third step is to check the code against the invariant. In this paper, we describe how we applied the Certora Prover to verify several smart contracts against our invariants. Using a prover such as Certora's as a part of a continuous integration process can automatically identify severe bugs during the development process, and ensure they are addressed before the code is released.

Outline. The rest of this paper is organized as follows. Section 2 presents the proposed set of reusable rules. Section 3 demonstrates the reusability of the rules by applying them to identify two bugs in an early testing version of

Invariant	Description	Credit
Bounded Supply	Bounded token supply – No infinite minting	[9]
Aggregated Ledger Integrity	Aggregated values over collections are correctly maintained. For example, "Sufficient Reserve"	
Authorized Operations	Only a superuser can execute an operation (e.g.: mint)	[10]
Proportional Token Distribution	The values of two exchanged tokens are proportional	[5]
Robustness	No radical token value change for small input changes	[6]

Table 1: Examples of useful rules inspired by influential projects in the Ethereum community.

Maker's MCD, which were reported to and confirmed by the Maker team [12]. In Section 4, we apply the Certora prover to identify bugs and verify the reusable invariants in three interesting projects. After concluding the main body of the paper in Section 5, we also present formal definitions of the invariants in Appendix A using parameterized Hoare triples [8].

2 Reusable Smart Contract Invariants

This section (1) describes a curated set of invariants we collected over many interactions with the community while formally verifying smart contracts, and (2) demonstrates the utility of these invariants for finding new subtle bugs in smart contracts. The wording of the informal descriptions of these invariants uses common terminology in the Ethereum community, rendering them intuitive and comprehensible by members of this community. We believe that codifying such invariants, even informally, will make the conversation around smart contract correctness more precise and scalable. Table 1 describes several useful invariants for smart contract correctness, defined informally. Suggested possible formal definitions of these invariants appear in Appendix A.

The invariants capture high-level correctness criteria that rely on the public interface of the contract being checked. Importantly, the invariants are independent of a concrete implementation of the public interface, making a clear distinction between specification and code. Furthermore, the invariants are parameterized in a way that makes them robust against the low-level differences between different implementations. For example, if two contracts capture similar concepts using different interface names, the parameterized invariants would still be applicable to both. The formalized rules are described in detail in Appendix A, using parameterized Hoare triples.

We now describe each invariant in more detail, beginning with Bounded Supply. Let us assume that the token implements a `totalSupply`() function as part

of its interface which returns the current number of tokens.[1] The property requires that the values returned by `totalSupply`() are bounded. This is captured by the following invariant [9]

$$BS(\texttt{totalSupply}, b) \stackrel{\text{def}}{=} \texttt{totalSupply}() \leq b \tag{1}$$

$\texttt{totalSupply}() \leq b$ is an invariant that should hold after every transaction. This property is instantiated with a `totalSupply`() function and a bound b. It requires that throughout the execution, the supply does not exceed b.

The Bounded Supply property applies to all ERC20 tokens. We used the Certora Prover to discover that 8 out of 24 types of tokens checked violate this rule while the remaining 16 satisfy this rule.[2] There were several reasons for the violations: two tokens violated the invariant because the actual bound on the supply depended on a delegated call to an external contract, meaning there could be different bounds at different times depending on values returned by the external code; six tokens violated it due to having assumptions external to the blockchain about limiting the ability to mint.

Aggregated Ledger Integrity is defined as a relationship between a collection of values and their aggregated values, often separately maintained. For example, the `totalSupply()` function of an ERC20 token often refers to the sum over the image of `balanceOf(address)`. Similarly, for applications in decentralized finance (DeFi) such as Compound Finance and Maker, Aggregated Ledger Integrity relates a more intricate relationship between the total amount of collateral stored and the total debt issued. When Aggregated Ledger Integrity is violated, the system's integrity is harmed, and certain operations might fail due to wrongly perceived insufficiency of funds held by the system, or alternatively, might succeed even though there are insufficient funds. That is, legitimate users may fail to withdraw funds that are owned by them, while malicious users may compromise funds that belong to other users.

Robustness is a property that relates two similar executions. The idea is that if the two executions differ slightly in their inputs, then the difference of the effects of the executions is also small. Commonly, the input is the time of the execution. For example, the change of an asset's price over time should be minuscule for short time spans (e.g. minutes). This helps to prevent speculative trading on assets.

Authorized Operations captures which operations should only be executed by users with privileges. For example, the `mint` operation, which can create new tokens from nothing, is often a privileged operation that can be executed only by a designated set of authorized minters. Authorization requirements are also common in complex systems of smart contracts where certain operations should be limited to contracts that belong to the system, and should not be executable by any other user of the blockchain.

[1] This is part of the standard ERC20 interface.

[2] In general the Certora Prover may provide an inconclusive answer, due to the inherent computational difficulty of formal verification, but that did not occur in this case.

Proportional Token Distribution describes a fair exchange of value. One should not be able to exchange an asset worth nothing with a positively valued asset. This is a necessary requirement but is not sufficient to describe all good behaviors of exchanges. If the exchange rate or conversion formula is known, Proportional Token Distribution can be refined to require a more precise condition.

3 A Tale of Two Bugs

This section demonstrates the reusability of the rules in Table 1 by applying them to an early testing release of MakerDAO's Multi-Collateral Dai (MCD) [13]. These bugs have been independently identified by auditors and bug bounty participants [12][3].

3.1 The Bounded Supply Property

Figure 1 and Figure 2 show a slightly simplified snippet of the MCD code (for presentation purposes). The code implements an auction where the prize has decreasing value, and the payment stays constant. A bidder that reduces the prize is set as a tentative winner. In case no one is challenging the last bid by further reducing the prize, it is possible to close the auction after either one hour passed since the last successful bid, or when the auction's time elapsed. Upon closing the auction, the system mints an amount of tokens equal to the prize, and transfers it to the winner.

A violation of Bounded Supply by this code is shown in Table 2. The trace demonstrates that the supply can reach 2^{256} tokens. We start with a small supply equal to 100. The malicious user can then trigger the creation of a new auction using the `new` function. Since the auction starts with an unrealistic initial prize of $2^{256} - 1$, any number below that can be given as argument to `bid`. If indeed the bid `b` is the smallest number seen so far, the sender of the `bid` transaction becomes the current, tentative winner of the auction. Therefore, the attacker can set the bid to $2^{256} - 100$. Expiry is set to 1 hour after the last bid, so in the case that (1) the invocation of `bid` happened at some time, say 2 hours since invocation of `new`; and (2) no one else will reduce this bid further after the invocation of `bid`, then the attacker can invoke `close` at the earliest possible time—3 hours and 1 second—to fulfill the auction and mint $2^{256} - 100$ tokens, equal to the latest prize. Thus, the supply of tokens will increase from 100 to 2^{256}.

The last steps of the trace, that consist of the invocation of `close` and the subsequent call to `mint`, were found by checking the Bounded Supply invariant using the Certora Prover tool, identifying `close` as an operation that potentially mints, and finding a starting state for `close`, for which the supply increases to 2^{256}. A snippet from the actual output of Certora Prover on MakerDAO's MCD code is shown in Figure 3. It summarizes the results of checking Bounded Supply

[3]We thank the Maker team for several productive conversations about these topics.

```
contract Auction {
  function new(uint id, uint payment) authorized {
     require(auction[id].end_time == 0);//check id in not occupied
     auction[id] = Auction(2**256-1,payment,this,0,now+1 day);
                  // arguments: prize, payment, winner, bid_expiry, end_time
  }
  function bid(uint id, uint b) {
     require(b <auction[id].prize); // prize can only decrease
     // new winner pays by repaying last winner
     collateral.transferFrom(msg.sender,auction[id].winner,auction[id].payment);
     // update new winner with new prize
     auction[id].prize = b;
     auction[id].winner = msg.sender;
     auction[id].bid_expiry = now + 1 hour;
  }
  function close(uint id) {
     require(auction[id].bid_expiry != 0
          && (auction[id].bid_expiry < now || auction[id].end_time < now));
     Token.mint(auction[id].winner, auction[id].prize);
     collateral.transfer(AuctionRewardsDestination, auction[id].payment);
     delete auction[id];
  }
}
```

Figure 1: The Action contract of the test MCD. Implementations of `transfer`, `transferFrom` are standard and not given here.

on each of the public methods of MCD's *Flopper* auction contract. The violating method in the MCD code is called `deal` and it is the equivalent to `close` in our simplified example. The figure also shows the concrete supply values in the counterexample produced by the tool, which are `0x1fff` initially and `0xf..f` finally.

3.2 Aggregate Ledger Integrity

The code snippet in Figure 1, inspired by MCD's auction contracts, contains yet another bug—one that affects the integrity of the auction contract. In order to facilitate payment for the auction prize, every tentative winner transfers in advance an amount of collateral tokens equal to the fixed payment price (`payment`). The integrity of the auction is thus described with the following formula:

$$\sum_i \texttt{auction}[i].\texttt{payment} = \texttt{collateral.balanceOf}(\texttt{this})$$

```
contract Token {
  function mint(address who, uint amount) authorized {
      balances[who] = safeAdd(balances[who],amount);
      supply = safeAdd(supply,amount);
  }
}
```

Figure 2: A standard token contract snippet used in MCD for both `token` and `collateral`. The implementation of `balanceOf` is standard, and not given here.

Contract	**Time**	**State (partial)**	**Next operation**
Auction	0	supply=100,auction[1]={0}	`new`(1, 5)
Auction	2 hours	supply=100,auction[1]={end_time=1 day, prize=2^{256}}	`bid`(1,2^{256}-100,sender=John)
Auction	3 hours + 1 second	supply=100,auction[1]={end_time=1 day, prize=2^{256}-100, bid_expiry=3 hours, winner=John}	`close`(1)
Token	3 hours + 1 second	same as above	`mint`(John,2^{256}-100)
Final state:		supply=2^{256},auction[1]={0}	

Table 2: A trace which shows a bug in the Maker MCD test version allowing infinite minting.

deny(address)	👍	1		
gemBalanceOf(address)	👍	1		
gemTotalSupply()	👍	1		
wards(address)	👍	1		
deal(uint256)	👎	2	Assert message: "Cannot increase to MAX_UINT256" Arguments values: [f=deal(uint256)]	_supply = 0x1fff f = deal(uint256) arg = 217 supply_ = 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Figure 3: Output from the Certora-Prover tool on MakerDAO's *Flopper* contract.

where `this` is the *address* (identifier) of the auction contract. The formula above is an instance of Aggregated Ledger Integrity. It relates between the holdings of the auction contract in its collateral token (represented by the right-hand side of the formula), and the sum of all payments sent by auction participants.

Every time a new winner is chosen, the new winner repays the previous winner back with `payment` collateral tokens. When the auction is closed, the received finalized payment is transferred to another contract that manages the rewards, and the auction structure is nullified. However, this invariant is not guaranteed to hold when a new auction is started. While the initiator can choose the value for the new auction's `payment` using the `payment` parameter of the `new` function, it is not guaranteed that the amount of `payment` collateral tokens was actually transferred to the auction contract. The `new` function does not perform a transfer from the sender's collateral to the benefit of the auction contract.

In the example presented, the integrity violation could lead to many exploit scenarios. An attacker can use it to leak collateral tokens out of the auction contract. Also, since the auction contract sends out the rewards after closing an auction, it is also possible to render many auctions non-operational by fulfilling an auction with a large bid that was never transferred to the auction.

We were able to identify this bug in MCD's *Flapper* auction contract. The `kick` method, equivalent to `new` in our example, did not guarantee that the new auction's bid is transferred to the auction contract. The root cause was that in *Flapper* auctions, initial bids were expected to be zero, therefore satisfying the invariant even though no transfer operation occurred. However, anyone could call `kick`, even with a non-zero initial bid.

Interestingly, this bug also manifests as a violation of the Authorized Operations invariant. Nevertheless, the ability to find this bug with Aggregated Ledger Integrity instead, attests to the usefulness of higher-level specifications that can give insight about the correctness of the contract as a whole, as opposed to each function individually.

4 Preliminary Verification Experience

We have used the Certora Prover to check the invariants from Table 1 on several leading projects using Ethereum smart contracts. The results are shown in Table 3. Since formal verification is in general undecidable, the Certora Prover may fail to verify an invariant or identify a violation. Hence, the usability of the tool hinges on its ability to verify (or falsify) properties of interest for realistic contracts. We note that the Certora Prover terminated with a decisive answer on all the cases described here.

Project	Application	Invariants					Verification Time
		BS	ALI	R	AO	PTD	
Compound Finance	DeFi	⊠	☹	☹	☺	☹	1 hour
MakerDao	DeFi	☹	☹	☺	☹	☹	15 minutes
Celo	Payments	☺	☺	☺	☹	⊠	5 hours

Table 3: Applicability of different invariants (denoted by their initials) to influential projects in the Ethereum community. ☺ indicates a invariant is relevant to a contract and the Certora Prover was able to formally verify that the code satisfies this invariant on all inputs. ☹ indicates a invariant is relevant to a contract and the Certora Prover was able to uncover an input which violates the invariant which was confirmed to be a bug with the developers. ⊠ indicates that we are not sure if this invariant is applicable to these contracts.

5 Conclusion

An early and influential paper in formal methods [4] argued that verification was doomed because, amongst other things, it would never be feasible to specify the complex quirks of real systems, and that, even if it was, fully automated verification would never be feasible. Our experience suggests that, at least in the domain of smart contracts, neither of these objections holds. In fact, as we have argued, smart contracts seem to be an especially promising target for formal methods, since many different contracts obey the same set of invariants. We are optimistic that the resulting network effect will make formal verification a cost-effective way to detect bugs in contracts that might otherwise have led to disastrous exploits.

References

[1] L. Alt and C. Reitwießner. SMT-based verification of solidity smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, pages 376–388, 2018.

[2] V. Buterin. Critical update re: Dao vulnerability. `https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/`, 2016. [Online; accessed 2-July-2017].

[3] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, 2012.

[4] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

[5] J. Flatow. Fair exchange of value. Personal Communication, 2019.

[6] G. Hayes. Robustness in smart currencies. Personal Communication, 2018.

[7] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. M. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 204–217, 2018.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[9] S. Islam. Bounded minting in smart contracts. Personal Communication, 2019.

[10] S. Islam. Privileged operations. Personal Communication, 2019.

[11] S. Lahiri. Verisol: A formal verifier for solidity based smart contracts. `https://www.microsoft.com/en-us/research/project/verisol-a-formal-verifier-for-solidity-based-smart-contracts/`, 2019. [Online; accessed November 8].

[12] MakerDAO. Mcd security roadmap update: October 2019. `https://blog.makerdao.com/mcd-security-roadmap-update-october-2019/`, October 2019. [Online; accessed 8-November-2019].

[13] MakerDAO. Multi collateral dai git repository, committed august 6, 2019. `https://github.com/makerdao/dss/tree/a283a48a22e302b3ec264792114da40ad1e855db`, 2019. [Online; acceesed November 8].

[14] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *IEEE S&P 2020*, 2020. To appear.

[15] S. Petrov. Another parity wallet hack explained. `https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c`, 2017. [Online; accesed 2-November-2019].

[16] M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 57–69, 2016. Extended version available from `https://www.cs.utexas.edu/~isil/chl-extended.pdf`.

[17] The Certora team. The certora prover. `http://www.certora.com`, 2019. [Online; accessed November 8].

Rule	Parameters	Definition
BS	$supply, b, d, op$	$\{S = supply()\}\ op\ \{supply() - S \geq d \Rightarrow supply() \leq b\}$
ALI	sum, map	$sum() = \sum_x map(x)$
AO	$op, auth$	$\{!auth(U)\}\ op(U)\ \{\textbf{fail}\}$
PTD	$op, token, token'$	$\{T = token() \wedge T' = token'()\}\ op\ \{T = token() \Leftrightarrow T' = token'()\}$
R	$op, in, out, \delta, \epsilon$	$[\lvert in_2() - in_1()\rvert \leq \delta]\ op(in)\ [\lvert out_2() - out_1()\rvert \leq \epsilon]$

Table 4: Formal definitions of the rules in Table 1.

A Formalizing the Customized Rules

Table 4 formally defines the rules from Table 1 using Hoare logic [8]. We write $\{P\}\ C\ \{Q\}$ where P and Q are Boolean conditions and C is a sequence of instructions to denote that if the execution of C on input states satisfying P, terminates then the resulting state must satisfy Q. This can be written as

if P **then** {
    C;
    assert Q;
}

Upper case letters denote Ghost variables which are universally quantified. We use pure functions to observe parts of the state in the condition.

The Bounded Supply rule weakens the rule from Equation (1) by allowing the supply to grow beyond bound b, provided that once the bound is breached, each operation increases the supply with less than d units. Note that if $d = 0$ then the new rule is equivalent to the one given in Equation (1). This captures policies of some Decentralized financial applications allowing controlled increase of supply.

The Aggregated Ledger Integrity property is a global invariant which need to be maintained by every operation.

Authorized Operations is a security requirement on operations which can only be executed by a superuser. It must revert if the address is not a superuser.

The Proportional Token Distribution property relates the values of two related tokens *token* and *token′*. It requires that *op* either changes both or neither. It may be possible to strengthen this rule, requiring a numerical correlation between the two tokens.

The most intricate property is robustness which requires reasoning about 2 traces. Here we use Cartesian Hoare Logic [16] to compare two executions of an operation to guarantee that small input change implies small output change. Here the precondition relates the values of two inputs observed by the *in* function on two executions and the postcondition requires correlates the *out* values. This is a special case of the robustness property [3].