Solvent: liquidity verification of smart contracts

Massimo Bartoletti[1], Angelo Ferrando[2], Enrico Lipparini[(✉)1,3], Vadim Malvone[4]

[1] Università degli Studi di Cagliari, Cagliari, Italy
[2] Università degli Studi di Modena e Reggio Emilia, Modena, Italy
[3] Università degli Studi di Genova, Genova, Italy
[4] Télécom Paris, Palaiseau, France

Abstract. Smart contracts are an attractive target for attackers, as evidenced by a long history of security incidents. A current limitation of smart contract verification tools is that they are not really effective in expressing and verifying liquidity properties regarding the exchange of crypto-assets: for example, is it true that in every reachable state a user can fire a sequence of transactions to withdraw a given amount of crypto-assets? We propose Solvent, a tool aimed at verifying these kinds of properties, which are beyond the reach of existing verification tools for Solidity. We evaluate the effectiveness and performance of Solvent through a common benchmark of smart contracts.

1 Introduction

In recent years we have seen a steady rise of smart contracts that implement financial ecosystems on top of public blockchains, and control today more than 100 billions of dollars worth of crypto-assets [17]. The peculiarities of the setting (i.e., the absence of intermediaries, the immutability of code after deployment, the quirks in smart contract languages) make smart contracts an appealing target for attackers, as bugs might be exploited to steal crypto-assets or just cause disruption. This is witnessed by a long history of attacks, which overall caused losses exceeding 6 billions of dollars [15].

Formal methods provide an ideal defense against these attacks, since they enable the creation of tools to detect bugs in smart contracts before they are deployed. Indeed, smart contracts verification tools, often based on formal methods, have been mushrooming in the last few years: for Ethereum alone — largely the main smart contract platform — dozens of tools exist today [26]. Still, the actual effectiveness of these tools in countering real-world attacks is debatable: indeed, attacks to smart contracts have continued to proliferate, refining their strategies from exploits of known vulnerability patterns to sophisticated attacks to the contracts' business logics. As a matter of fact, the vast majority of the losses due to real-world attacks are caused by logic errors in the contract code [15], which are outside the scope of most vulnerability detection tools. In particular, several real-world attacks were based on *liquidity* weaknesses of smart contracts, which were exploited by attackers to steal or freeze crypto-assets [3]. Liquidity

(also called *solvency* [25] or *enabledness* [32]) expresses the ideal behaviour of contracts in terms of the exchange of crypto-assets [37,12,27]: users want to be guaranteed that, whenever certain states are reached, they can always perform some actions that lead to a desirable assets transfer. There are two key points in this notion: the user wants to be able to constrain *who* can perform these actions, and *how many* actions are needed [32].

The current tools that support the verification of Solidity cannot verify general liquidity properties. This is due both to the design choices of Solidity, their target language, and to the complex logical structure of such properties. The difficulty of verifying Solidity is caused by several glitches of high-level abstractions over the low-level target (e.g. reentrancy [5], gas costs [20], and non-native tokens [39]). Indeed, it has been observed that existing tools already face challenges (in terms of soundness and completeness) in the verification of even simpler properties [11]. We believe that effectively verifying liquidity requires first to purify Solidity from its main semantical quirks. Once we have done this, we must deal with the peculiar logical structure of liquidity properties. Indeed, liquidity cannot be expressed in terms of safety or liveness, and bounded liveness cannot model liquidity either (see Section 2). To deal with liquidity, verification techniques that go beyond safety and (bounded) liveness are therefore necessary.

Contributions. We propose Solvent, a tool that verifies liquidity properties of smart contracts. Solvent takes as input a contract, written in a purified version of Solidity, and a set of user-defined liquidity properties. The tool translates them into SMT constraints [9], reducing the verification problem to an SMT-based symbolic model checking one [8]. Then, techniques such as bounded model checking and predicate abstraction are employed, relying on Z3 [16] and cvc5 [6] as back-end SMT solvers. Experiments on a benchmark of real-world smart contracts show that Solvent can efficiently verify relevant liquidity properties of their behaviour. These properties are currently out of the scope of industrial verification tools operating on the full Solidity, like e.g. SolCMC [4] and Certora [23], as well as academic tools [24,21,31,28,34,38] (we discuss such tools in Section 5). Solvent provides developers with useful feedback, by detecting logical errors that would otherwise remain unnoticed. In particular, when Solvent detects a property violation, it produces a concrete execution trace that leads the contract to a state from which the desired asset exchange is unrealisable.

Summing up, the main contributions of the paper include:

- a fully automated encoding of an expressive subset of Solidity and of liquidity properties of smart contracts into SMT constraints;
- a toolchain to perform bounded model checking and predicate abstraction using Z3 and cvc5 as off-the-shelf SMT solvers, producing counterexamples (that are actually replayable in Ethereum) when the property is violated;
- a thorough evaluation of the tool effectiveness and performance on a benchmark of real-world smart contracts, which we extend with relevant liquidity properties (available on the tool github repository [10]);
- a concrete demonstration of the tool applicability, showing subtle bugs in existing smart contracts that cause crypto-assets to get frozen forever.

2 On liquidity

In this section, we clarify our notion of liquidity, and we support our claim that it cannot be expressed in terms of safety or (bounded) liveness. Recall that safety has the form "*p always holds*", liveness has the form "*p eventually holds*", and bounded liveness "*in at most m steps p holds*". Liquidity, instead, has the form "*for certain users a, there always exists a sequence of at most m actions that a can perform to make p hold*" (where m is a parameter).

To illustrate the differences among these notions, consider a user opening a bank account. The user wants to be guaranteed that, if they deposit money in the bank, then they will always be able to withdraw their money by performing a *reasonable amount of actions*. They do not want to *always* withdraw their money (safety), or to just *eventually* withdraw their money (liveness). For example, requiring liveness but not liquidity would allow the bank to give a guarantee such as "*any client will withdraw their money after the bank has been visited overall 1 million times after the deposit*", making the liveness property "*eventually I will withdraw*" true (assuming the fairness condition that visits to the bank happen infinitely often). However, the more desirable liquidity property "*I am always able to perform at most 3 actions that make me withdraw*" would not hold. Bounded liveness would not be an appropriate requirement either. Indeed, assume that the bank guarantees that "*the user always receives their money before 3 actions have been made*". This would be undesirable, since it says that *any* sequence of 3 actions, performed by *any* user, would trigger the withdrawal.

In LTL, we have that safety properties have the form "Gp", and liveness properties have the form "GFp". Liquidity properties, however, cannot be expressed in LTL, as we need to existentially quantify over the paths ("*there exists a path* where the user performs at most m actions to make p hold"). In CTL, the closest formulation would be "$AG\ EX\ p \vee AG\ EX\ EX\ p \vee \cdots \vee AG\ (EX)^m\ p$". This, however, still does not capture the strategic aspect of liquidity, which requires that the sequence that makes p hold consists of actions made by certain users, and therefore we need to impose conditions over transition variables.

3 Verifying liquidity properties with Solvent

We demonstrate our tool through a simple example and provide some highlights on how it works. Solvent operates in two steps: (1) given as input a smart contract and a set of liquidity properties, it encodes the contract and the properties into constraints in the SMT-LIB standard [7]; (2) then, it issues satisfiability queries to an SMT solver to detect if the required properties are violated. If so, it produces a counterexample, in the form of a sequence of transactions leading to a state where a required property cannot be satisfied.

To illustrate Solvent, we consider in Listing 1 a simple crowdfunding contract. The contract is akin to a class in OO programming, with attributes that define its state and methods (triggered by blockchain transactions) that update it. The user who fires the transaction (denoted by `msg.sender`) can transfer some

```
contract Crowdfund {
  int immutable end_donate; // last block number for donations
  int immutable target;     // threshold for successful campaign
  address immutable owner;  // beneficiary of the campaign
  mapping (address => int) donors; // records users' donations
  bool target_reached;      // initialized to false

  constructor(address o, int e, int t) {
    owner = o; end_donate = e; target = t
  }
  function donate() payable {
    require (block.number<=end_donate);
    donors[msg.sender] = donors[msg.sender] + msg.value;
    if (balance>=target) { target_reached = true }
  }
  function wdOwner() {
    require (block.number>end_donate && target_reached);
    owner.trasfer(balance) // send contract balance to owner
  }
  function wdDonor() { // SUBTLE BUG HERE
    require (block.number>end_donate && balance<target);
    msg.sender.transfer(donors[msg.sender]);
    donors[msg.sender] = 0
  }
}
property donor_wd {
  Forall xa [ !target_reached && block.number>end_donate
    -> Exists tx [1,xa] // 1 = max tx length, xa = tx sender
    [ <tx>xa.balance    // balance of xa after tx is performed
      >= xa.balance + donors[xa] ] ]
}
```

Listing 1: A crowdfunding contract (with a subtle bug) and a liquidity property.

amount (`msg.value`) of cryptocurrency to the contract along with the call. The constructor specifies the owner of the crowdfunding campaign, the deadline for donations, and the target amount. The method `donate` allows anyone to donate any amount before the deadline; `wdOwner` allows the owner to redeem the whole contract balance if the campaign target has been reached and the deadline has expired; finally, `wdDonor` allows donors to withdraw their donations after the deadline, if the campaign target has not been reached.

Solvent encodes each method into an SMT constraint that, given transaction variables, ties next-state variables to current-state variables, using auxiliary variables to represent intermediate internal states. Each `require` is encoded as an if-then-else, where, if the condition fails, the method reverts, i.e. next-state variables coincide with current-state variables.

A crucial property of crowdfunding contracts is that donors can redeem their donations after the deadline whenever the target is not reached. We specify this property as `donor_wd` in Listing 1. This reads as follows: for all users `xa`, if the target has not been reached and the deadline has passed, then there exists a sequence `tx` of transactions of length 1 signed by `xa` such that, in the state reached after executing `tx`, the balance of `xa` is increased by `st.donors[xa]`. Solvent detects that this property is violated. This is correct, although surprising,

because of a subtle bug in `wdDonor`. There, the `require` ensures that donors can withdraw only if the contract balance is less than the target. This would seem correct, since `donate` is the only method that can receive ETH (as stated by the `payable` tag). The quirk is that contracts can receive ETH even when there are no `payable` methods, through block rewards, which can send ETH to any address, or *selfdestruct*, which transfer the remaining ETH in a contract to an address at their choice [1]. Notably, an attacker could exploit a *selfdestruct* to freeze all the funds in the contract, preventing donors from withdrawing!

To translate `donor_wd` into an SMT constraint, Solvent considers its negation, and introduces a new existentially quantified variable for `xa`, and new universally quantified variables for all transaction variables in the sequence `tx` and for all next-state variables. Then, it reduces to checking whether there exists an `xa` for which, if the antecedent holds, for all transaction variables and next-state variables, either the transactions invalidly tie current and next-state, or the required consequent does not hold. If this formula is *un*satisfiable, then the property holds; otherwise, a counterexample is given. E.g., for `donors_wd`, the counterexample is the following sequence of transitions:

```
[1] constructor(2,0,2)   msg.sender=address(4)   msg.value=0
[2] donate()             msg.sender=address(4)   msg.value=1
[3] selfdestruct()       msg.sender=address(0)   msg.value=1
```

Here, the last transition represents a call to an adversarial contract, with a method invoking *selfdestruct* on `Crowdfund`.[5] This can be easily translated into a concrete Proof-of-Concept (PoC), leading the contract to a state where `donors_wd` cannot be satisfied.

To fix the contract, we replace the condition `balance<target` in `wdDonor` with `!target_reached`. With this fix, Solvent correctly detects that `donor_wd` holds.

Note that the fixed contract still has a liquidity vulnerability: if the target is not reached, donors can redeem their donations, but any extra funds possibly existing at contract creation, or received through block rewards or *selfdestruct* actions, will be frozen in the contract. To fix this vulnerability, we first need to quantify these extra funds: we do this by adding a variable `tot_donations` that we update in `donate` and in `wdDonor`, so that the extra budget is now given by `balance-tot_donations`. We then add a method that allows anyone to transfer any extra budget to the owner after the deadline. Solvent verifies that the revised contract still enjoys `donor_wd`, and transfers any extra budget to the owner, i.e.:

```
property no_frozen_funds {
  Forall xa [ balance>tot_donations && block.number>end_donate
  -> Exists tx [1, xa]
  [ (<tx>balance[owner] >= balance[owner]
                           + (balance - tot_donations)) ] ]   }
```

4 Evaluation

We test our tool over a common benchmark for Solidity verification [2], which includes a representative set of real-world contracts and properties. Since this

[5] Note that *selfdestruct* is still active [18].

benchmark is focussed on current verification tools for Solidity, which do not deal with general liquidity properties, we extend it with relevant properties of this kind for each contract (see the github page). Overall, we end up with 107 verification tasks, which we manually check for the ground truth.

Setup. We run Solvent on each verification task on a 3GHz 64-bit Intel Xeon Gold 6136 CPU and a GNU/Linux OS (x86_64-linux) with 64 GB of RAM, with either cvc5 (v. 1.1.3-dev.152.701cd63ef) or Z3 (v. 4.13.0) as a back-end. The run-time limit for each verification task is 400s of CPU time. A subset of the results are shown in Table 1 (see github for the full results). We mark each property as: "✗(N)", if the solver finds a trace that violates the property (with N being the length of the shortest trace leading to a violation); "✓", if it proves that the property holds in all possible states; "✓(N)", if it proves that the property holds for every trace of length at most N; and "?" if it timeouts.

Results. First, we note that both solvers never return an inconsistent answer. For all non-liquid properties, except two, at least one of the solvers is able to find a counterexample. When a counterexample is found, the result is returned quite quickly, and the trace is quite short. For liquid properties, the solvers are able to prove the property only for some instances. Still, in most cases, they manage to verify the property up-to traces of significant length. Two contracts ("Payment splitter" and "Vesting wallet") are significantly tough for both solvers. This is not surprising though, as they both present non-linear behaviour, thus requiring to solve SMT formulas in the theory of Nonlinear Integer Arithmetic, which is undecidable and notoriously hard to deal with in practice for SMT solvers.[6]

Discussion. The results show that our tool is particularly good at finding counterexamples. They are witnessed by a sequence of transactions that can be replayed in the actual Ethereum, leading to a state from which the desired outcome is unreachable. On the other hand, when Solvent states that a property holds, there is no guarantee that the property is preserved "as-is". For instance, reentrancy vulnerabilities (which are abstracted away in our symbolic semantics), can falsify the property. Nonetheless, the output of Solvent guarantees that no conceptual error has been made in the business logic of the contract. Even when Solvent outputs ✓(N), the larger the N the more relevant the information given to users: indeed, empirically we observed that in the benchmark [2], property violations are already observable after short traces. Remarkably, we spot that several contracts in the benchmark [2] have liquidity vulnerabilities, i.e., crypto-assets remain frozen in the contract (in Table 1, where `no_frozen_funds` is ✗).

5 Related work

The Ethereum ecosystem includes several bug detection tools that can spot *specific* forms of liquidity vulnerabilities in smart contracts. In that setting,

[6] Strategies to overcome the obstacles posed by NIA have been recently discussed, for the specific case of formulas coming from the verification of smart contracts, in [22].

Contract	Property	Liquid?	cvc5		Z3	
			Result	Time	Result	Time
Auction	`no_frozen_funds`	✗	✗(3)	1.10	✗(3)	1.15
	`seller_wd`	✓	✓(10)	—	✓(9)	—
	`old_winner_wd`	✓	✓(21)	—	✓(9)	—
Bank	`deposit_not_revert`	✓	✓	2.04	✓	19.79
	`withdraw_not_revert`	✓	✓(7)	—	✓(8)	—
Bet	`any_timeout_join`	✓	✓(17)	—	✓(12)	—
	`oracle_win`	✓	✓(14)	—	✓(7)	—
	`any_timeout_win`	✓	✓(39)	—	✓(9)	—
	`no_frozen_funds`	✗	✗(2)	0.70	✗(2)	0.74
Crowdfund (bug)	`owner_wd`	✓	✓	2.31	✓	26.64
	`donor_wd`	✗	✗(3)	1.38	✗(3)	7.78
Crowdfund (fix2)	`owner_wd`	✓	✓	2.34	✓	29.34
	`donor_wd`	✓	✓(7)	—	✓(6)	—
	`no_frozen_funds`	✓	✓(9)	—	✓(8)	—
Escrow	`arbiter_wd_fee`	✓	✓(17)	—	✓(10)	—
	`buyerorseller_wd_deposit`	✓	✓(16)	—	✓(41)	—
	`anyone_wd`	✗	✗(2)	0.70	✗(2)	0.76
	`no_frozen_funds`	✗	✗(3)	1.17	✗(3)	1.23
HTLC	`owner_wd`	✓	✓(16)	—	✓(9)	—
	`verifier_wd_timeout`	✓	✓(17)	—	✓(9)	—
	`no_frozen_funds`	✓	✓	2.37	✓	54.84
Lottery (A.3)	`one_player_win`	✓	✓(13)	—	✓(9)	—
	`p1_redeem_nojoin`	✓	✓(23)	—	✓(8)	—
	`p1_redeem_noreveal`	✓	✓(16)	—	✓(9)	—
	`p2_redeem_noreveal`	✓	✓(17)	—	✓(8)	—
Payment splitter	`anyone_wd_ge`	✓	✓(2)	—	✓(1)	—
	`anyone_wd_releasable`	✓	✓(2)	—	✓(1)	—
	`anyone_wd`	✗	✗(1)	0.40	✗(1)	0.38
Vault	`fin_owner`	✓	✓(14)	—	✓(11)	—
	`canc_recovery`	✓	✓(20)	—	✓(10)	—
	`wd_fin_owner`	✗	✗(1)	0.45	✗(1)	0.44
Vesting wallet	`owner_wd_expired`	✓	✓	2.16	?	T/O
	`owner_wd_started`	✗	?	T/O	?	T/O
	`owner_wd_uncond`	✗	?	T/O	✗(1)	0.36
	`owner_wd_beforestart`	✗	?	T/O	✗(1)	0.35
	`owner_wd_empty`	✗	?	T/O	✗(1)	0.35
	`owner_wd_released`	✗	?	T/O	?	T/O

Table 1: Solvent benchmark (subset; execution times are in seconds). For $\checkmark(N)$ properties we write "—" when the prover timeouts before proving $\checkmark(N+1)$.

these vulnerabilities are often referred to as "Locked Ether", which are roughly described as the absence of a mechanism to withdraw Ether from the contract. The tools capable of detecting Locked Ether bugs include Slither [19], SmartCheck [35], Maian [30], Securify2 [37], ConFuzzius [36] and sFuzz [29]. Each of these tools has its own internal encoding of the Locked Ether property, making it difficult to compare them [33]. There are two main differences between these tools and ours. First, while these tools can only spot that a predefined hard-coded Locked Ether property is violated, Solvent can verify custom liquidity properties, defined by our specification language. In particular, Solvent can verify properties that express *who* can withdraw *how much*, and under *which conditions*; not just that Ether does not get frozen in the contract. Second, bug detection tools only focus on property violations, while Solvent (a formal verifi-

cation tool) is capable of verifying that a liquidity property is satisfied (always, or up to a certain bound on the number of transactions).

Other state-of-the-art formal verification tools for Solidity do not support general liquidity properties. SolCMC, the prover shipped with the Solidity compiler, as well as other verification tools such as Zeus [24], solc-verify [21], VerX [31] and SmartACE [38], only support safety properties, while other tools, such as Certora [23], VeriSolid [28], and SmartPulse [34], can also verify liveness properties. However, as discussed in Section 2, this is not enough for liquidity properties (see Appendix A for a more in-depth discussion).

6 Conclusions and Future work

Solvent has already proven useful to spot general liquidity vulnerabilities, which are beyond the reach of current tools. Still, there is space for improvements.

First, Solvent uses off-the-shelf SMT solvers, relying on bounded model checking to find counterexamples, and on predicate abstraction to prove that the property holds. An alternative approach that we plan to investigate is to leverage advanced techniques (e.g., abstraction-refinement, k-induction, etc.) used by modern infinite-state symbolic model checkers.

Second, we plan to extend our Solidity fragment to narrow the gap with the actual Solidity (see Appendix A for a discussion of the main differences). In particular, we note that concretising our `transfer` into contract-to-contract calls, as in Solidity, would require to take failures and reentrancy into account. E.g., consider a contract with a method:

```
foo() { owner.transfer(1); msg.sender.transfer(1); }
```

and a property requiring that, whenever the contract balance is at least 2, any user can increase their balance by 1. In our Solidity fragment, this property is true, since both transfers will succeed under the given hypotheses. This is not the case in the concrete Solidity: e.g., `owner` could be a contract address, whose fallback method could fail under certain conditions. In this case, the second transfer in `foo` would not happen, and so the property would be violated. Since considering each transfer as potentially failing would make most contracts illiquid, a possible approach would be to allow queries to specify which transfers or addresses to be considered trusted.

Other future work include the automatic transformation of the counterexamples given by Solvent into an executable PoC (e.g., in the HardHat tool), and the formalisation of liquidity in a suitable logic.

Acknowledgments. Work partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU, and by PRIN 2022 PNRR project DeLiCE (F53D23009130001).

References

1. SMTChecker and formal verification: contract balance. https://docs.soliditylang.org/en/v0.8.24/smtchecker.html#contract-balance (2023)
2. An open benchmark for evaluating smart contracts verification tools. https://github.com/fsainas/contracts-verification-benchmark (2024)
3. Alois, J.: Ethereum Parity hack may impact ETH 500,000 or $146 million. https://www.crowdfundinsider.com/2017/11/124200-ethereum-parity-hack-may-impact-eth-500000-146-million/ (2017), accessed on April 9, 2024
4. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity compiler's model checker. In: Computer Aided Verification. LNCS, vol. 13371, pp. 325–338. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_16
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Principles of Security and Trust (POST). LNCS, vol. 10204, pp. 164–186. Springer (2017). https://doi.org/10.1007/978-3-662-54455-6_8, http://dx.doi.org/10.1007/978-3-662-54455-6_8
6. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
7. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at https://smtlib.cs.uiowa.edu/language.shtml
8. Barrett, C., Tinelli, C.: Satisfiability Modulo Theories, pp. 305–343. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11, https://doi.org/10.1007/978-3-319-10575-8_11
9. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability (2021). https://doi.org/10.3233/978-1-58603-929-5-825
10. Bartoletti, M., Ferrando, A., Lipparini, E., Malvone, V.: Solvent: liquidity verification of smart contracts, https://github.com/AngeloFerrando/Solvent
11. Bartoletti, M., Fioravanti, F., Matricardi, G., Pettinau, R., Sainas, F.: Towards benchmarking of Solidity verification tools. In: Formal Methods in Blockchain (FMBC). OASIcs, vol. 118, pp. 6:1–6:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/OASICS.FMBC.2024.6
12. Bartoletti, M., Lande, S., Murgia, M., Zunino, R.: Verifying liquidity of recursive Bitcoin contracts. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/LMCS-18(1:22)2022
13. Bliudze, S., Cimatti, A., Jaber, M., Mover, S., Roveri, M., Saab, W., Wang, Q.: Formal verification of infinite-state BIP models. In: Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 9364, pp. 326–343. Springer (2015). https://doi.org/10.1007/978-3-319-24953-7_25
14. Chahoki, A.Z., Roveri, M., Amyot, D., Mylopoulos, J.: Revisiting formal verification in VeriSolid: An analysis and enhancements. In: Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis. CEUR Workshop Proceedings, vol. 3629, pp. 55–60. CEUR-WS.org (2023)
15. Chaliasos, S., Charalambous, M.A., Zhou, L., Galanopoulou, R., Gervais, A., Mitropoulos, D., Livshits, B.: Smart contract and DeFi security: Insights from

tool evaluations and practitioner surveys. In: IEEE/ACM International Conference on Software Engineering (ICSE). pp. 60:1–60:13. ACM (2024). https://doi.org/10.1145/3597503.3623302, https://doi.org/10.1145/3597503.3623302
16. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Defillama. https://defillama.com/, accessed on April 9, 2024
18. EIP-4758: Deactivate SELFDESTRUCT. https://eips.ethereum.org/EIPS/eip-4758, accessed on June 15, 2024
19. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: Workshop on Emerging Trends in Software Engineering for Blockchain, (WETSEB@ICSE). pp. 8–15 (2019). https://doi.org/10.1109/WETSEB.2019.00008
20. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: MadMax: analyzing the out-of-gas world of smart contracts. Commun. ACM **63**(10), 87–95 (2020). https://doi.org/10.1145/3416262
21. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for Solidity smart contracts. In: Verified Software. Theories, Tools, and Experiments (VSTTE). LNCS, vol. 12031, pp. 161–179. Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_11
22. Hozzová, P., Bendík, J., Nutz, A., Rodeh, Y.: Overapproximation of non-linear integer arithmetic for smart contract verification. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 94, pp. 257–269 (2023). https://doi.org/10.29007/h4p7
23. Jackson, D., Nandi, C., Sagiv, M.: Certora technology white paper. https://docs.certora.com/en/latest/docs/whitepaper/index.html (2022)
24. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Network and Distributed System Security Symposium (NDSS). The Internet Society (2018)
25. Kirstein, U.: Formal verification helps finding insolvency bugs — Balancer V2 bug report. https://medium.com/certora/formal-verification-helps-finding-insolvency-bugs-balancer-v2-bug-report-1f53ee7dd4d0, accessed on August 30, 2024
26. Kushwaha, S.S., Joshi, S., Singh, D., Kaur, M., Lee, H.N.: Ethereum smart contract analysis tools: A systematic review. IEEE Access **10**, 57037–57062 (2022). https://doi.org/10.1109/ACCESS.2022.3169902
27. Laneve, C.: Liquidity analysis in resource-aware programming. J. Log. Algebraic Methods Program. **135**, 100889 (2023). https://doi.org/10.1016/J.JLAMP.2023.100889
28. Nelaturu, K., Mavridou, A., Stachtiari, E., Veneris, A.G., Laszka, A.: Correct-by-design interacting smart contracts and a systematic approach for verifying ERC20 and ERC721 contracts with VeriSolid. IEEE Trans. Dependable Secur. Comput. **20**(4), 3110–3127 (2023). https://doi.org/10.1109/TDSC.2022.3200840
29. Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y., Minh, Q.T.: sFuzz: an efficient adaptive fuzzer for Solidity smart contracts. In: International Conference on Software Engineering (ICSE). pp. 778–788. ACM (2020). https://doi.org/10.1145/3377811.3380334
30. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Annual Computer Security Applications

Conference (ACSAC). pp. 653–663. ACM (2018). https://doi.org/10.1145/3274694.3274743
31. Permenev, A., Dimitrov, D.K., Tsankov, P., Drachsler-Cohen, D., Vechev, M.T.: VerX: Safety verification of smart contracts. In: IEEE Symposium on Security and Privacy. pp. 1661–1677. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00024
32. Schiffl, J., Beckert, B.: A practical notion of liveness in smart contract applications. In: Formal Methods in Blockchain (FMBC). OASIcs, vol. 118, pp. 8:1–8:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/OASICS.FMBC.2024.8
33. Sendner, C., Petzi, L., Stang, J., Dmitrienko, A.: Large-scale study of vulnerability scanners for Ethereum smart contracts. In: IEEE European Symposium on Security and Privacy (S&P). pp. 220–220. IEEE (2024). https://doi.org/10.1109/SP54263.2024.00230
34. Stephens, J., Ferles, K., Mariano, B., Lahiri, S.K., Dillig, I.: SmartPulse: Automated checking of temporal properties in smart contracts. In: IEEE Symposium on Security and Privacy. pp. 555–571. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00085
35. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: SmartCheck: Static analysis of Ethereum smart contracts. In: Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 9–16. ACM (2018). https://doi.org/10.1145/3194113.3194115
36. Torres, C.F., Iannillo, A.K., Gervais, A., State, R.: ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In: IEEE European Symposium on Security and Privacy (EuroS&P). pp. 103–119. IEEE (2021). https://doi.org/10.1109/EUROSP51992.2021.00018
37. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 67–82. ACM (2018). https://doi.org/10.1145/3243734.3243780
38. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Verifying Solidity smart contracts via communication abstraction in SmartACE. In: Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 13182, pp. 425–449. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_21
39. Xia, P., Wang, H., Gao, B., Su, W., Yu, Z., Luo, X., Zhang, C., Xiao, X., Xu, G.: Trade or trick?: Detecting and characterizing scam tokens on Uniswap decentralized exchange. In: ACM SIGMETRICS/IFIP Performance. pp. 23–24. ACM (2022). https://doi.org/10.1145/3489048.3522636

A Supplementary material

Because of space constraints, we include in this Appendix the following supplementary material:

- in Appendix A.1 we give a brief overview of Solidity, and discuss the main differences with the fragment supported by Solvent;
- in Appendix A.2 we discuss related work on bug detection and formal verification tools for smart contracts. In particular, we provide a thoroughly discussion of the differences between Solvent and the two main verification tools for Solidity, i.e. SolCMC and Certora, highlighting why verification of liquidity properties is currently out of their scope. We also discuss other verification tools, such as VeriSolid [28] and SmartPulse [34], that can verify liveness properties but not liquidity ones.
- in Appendix A.3 we evaluate Solvent on a more complex use case, i.e. a fair 2-players lottery. Implementing this use case is quite error-prone, since the contract must implement a commit-reveal protocol that prescribes punishments whenever a player behaves dishonestly, e.g. by refusing to perform some required action. The contract must ensure that, even in these cases, an honest player has at least the same payoff that she would have by interacting with another honest player. Proving relevant liquidity properties of this contract seems far beyond the capabilities of existing verification tools. To overcome these limitations, we provide Solvent with an extended syntax to express hashed timelock protocols, similarly to what is done in other smart contract languages (e.g., in Tezos).

A.1 Background on Solidity, and relation to Solvent's fragment

Solidity was one of the first contract languages to be introduced, and it is currently the main high-level smart contract language for Ethereum and other blockchains that support the Ethereum Virtual Machine (EVM). Solidity code is compiled into EVM bytecode, and then executed by the blockchains nodes.

Solidity follows the account-based stateful model: namely, the ownership of crypto-assets of both users and contracts is recorded into accounts; furthermore, contracts can store data (i.e., state variables) in their account. Accounts are partitioned into Externally Owned Accounts (EOAs), which are controlled by users through private keys, and contract accounts. Every account is uniquely identified by an *address*.

Transactions are sent from EOAs to contract accounts to trigger state transitions in contracts; these transitions possibly involve transfers of crypto-currency from the caller EOA to the called contract, and from the latter to other accounts (both EOAs and contracts). Each transaction is signed by a single EOA, denoted as `msg.sender`, and can transfer units of ETH from the caller EOA to the contract. The amount of transferred ETH is denoted by `msg.value`. Contracts have state variables and methods that can update them, as exemplified by the crowdfund contract in Section 3.

The main differences between the languages supported by Solvent and the actual Solidity are the following:

- Solidity features contract-to-contract calls, while Solvent only features EOA-to-contract calls. Contract calls in Ethereum are quite burdensome, since the callee can in turn call any other contract (including the original caller), paving the way to so-called reentrancy attacks [5]. While it would be possible to extend our verification technique to take contract-to-contract calls into account, in the current version of Solvent we have opted to drop this feature, since existing verification tools for Solidity already provide effective defence against reentrancy attacks.
- In Solidity, transfers of crypto-currency from a contract to another account are encoded as contract-to-contract calls, while in Solvent we assume that the transferred amount actually arrives at the destination address, i.e. that this address is either an EOA or a contract account that does not perform a further call. This assumption simplifies stating properties about the funds transferred from a contract (see Appendix A.2), and is used in other verification tools such as VerX [31], where it is called *effective external callback freedom* assumption.
- Solidity features (possibly unbounded) loops and a complex gas mechanism (specified at the EVM level) to avoid divergent computations and reward blockchain nodes for processing transactions. Although these features are not present in Solvent, thereby limiting its expressiveness, the benchmark in Table 1 shows that Solvent is still expressive enough for a wide range of applications. Furthermore, we note that unbounded loops are discouraged even in Solidity, since they may be exploited by attackers to make a contract become stuck because an iteration exceeds the block gas limit.
- Solvent features a special syntax to express *hashed timelock protocols*, where a user first store the hash of a chosen secret in the contract, and then reveal the secret, making the contract check that the hash of the revealed secret corresponds to the stored hash. This is a common pattern, used e.g. in gambling games and blind auctions, but where existing verification tools for Solidity fail to give the expected results (see Appendix A.2). We exemplify this feature to design a 2-players lottery in Appendix A.3, showing that Solvent manages to provide developers with a useful feedback.

A.2 Verification of liquidity in other tools

We now discuss more concretely why Certora, SmartPulse and VeriSolid cannot express general liquidity properties.

First, we discuss Certora, one of the leading formal verification tools for Solidity. Consider the `Freezable` contract in Listing 2. The contract allows anyone to withdraw part of its balance through the method `pay`, unless the variable `frozen` is true. This variable is controlled by the `owner` through the method `freeze`: hence, the `owner` at any time can freeze the contract balance, preventing anyone from withdrawing. A desirable property of the `Freezable` contract, and of smart contracts in general, is that crypto-assets cannot be frozen forever.

The rules in Listing 3 are tentative specifications of the liquidity property is the Certora Verification Language (CVL). We claim that neither rule correctly encodes the intended liquidity property:

- The rule `liq_satisfy` is satisfied if there exists some starting state such that, for some `sender` address and some `v`, `sender` can fire a transaction `pay(v)` that increases its balance by `v`. This is not a correct way to encode our liquidity property: indeed, Certora says that the property is satisfied, since there *exists* a trace that makes the condition in the `satisfy` statement true: this is the trace where the `owner` has not set `frozen` yet.
- The rule `liq_assert`, which is identical to `liq_satisfy` but for the `satisfy` statement that replaces the `assert`, is satisfied if, for all reachable states, for all `sender` and *for all* values `v`, a transaction `pay(v)` is never reverted. Also this rule does not correctly specify the intended liquidity property: Certora would correctly state that the property is false, because there are some values `v` that make the transaction fail (e.g., when `v` exceeds the contract balance).

Although in this simple example we could fix the rule `liq_assert` by requiring that the transaction is not reverted for all values `v` less than the contract balance and when `frozen` is false, in general we would like to know if there *exist* parameters that make the desirable property true, which is not expressible in CVL. Actually, in Solvent we can express exactly this property, as shown in Listing 4.

Another kind of properties that are not easily expressible are those that concerning transfers of crypto-currency from the contract. For example, consider the contract `Transfer` in Listing 5. The method `withdraw` allows anyone to transfer any fraction of the contract balance to the `rcv` address. Properly formalising this simple property is surprisingly burdensome.

A first attempt would be to express the property is through the `invariant` in Listing 6, which asserts a constraint on the balances of the contract and of `rcv` before and after a `withdraw`. This however would not be a correct formalisation, for multiple reasons. First, a contract that sends 10 units of crypto-currency to an intermediary who forwards the funds to the address `rcv` would violate the intended property but possibly satisfy the invariant in Listing 6. Second, in general the invariant might not hold, as detected by both SolCMC and Certora. Actually, if `rcv` is a contract address, the transfer could trigger another call that in turns transfers the crypto-currency elsewhere, thus breaking the invariant[7].

A case where we are certain that the `withdraw(v)` will successfully increase the recipient's balance by `v` units of crypto-currency is when `rcv` is an EOA. However, even in this simple case expressing and verifying the correct transfer property is problematic. First, there is no general way for a contract to discriminate between an EOA and a contract address[8]. Second, even in the cases where it is possible to determine that an address is an EOA, existing verification tools such as SolCMC and Certora do not manage to verify that the property holds [11].

[7] The actual feasibility of this further transfer also depends on the amount of gas units transferred to `rcv` and consumed by its fallback function. However, these gas costs are usually not taken into account by verifiers.

[8] https://docs.openzeppelin.com/contracts/4.x/api/utils#Address

```
contract Freezable {
  address immutable owner;
  bool frozen;

  constructor () payable {
    owner = msg.sender;
  }

  function freeze() external {
    require (msg.sender == owner);
    frozen = true;
  }

  function withdraw(int amount) external {
    require(!frozen);
    msg.sender.transfer(amount);
  }
}
```

Listing 2: A freezable deposit contract.

```
// Certora specification
rule liq_satisfy(address sender, uint v) {
  mathint b0 = bal(sender); // sender initial balance
  env e;
  require e.msg.sender == sender;
  withdraw(e, v);
  mathint b1 = bal(sender); // sender balance after pay(v)
  satisfy(b1 == b0 + v);    // looking for a positive example
}

rule liq_assert(address sender, uint v) {
  mathint b0 = bal(sender); // sender initial balance
  env e;
  require e.msg.sender == sender;
  withdraw(e, v);
  mathint b1 = bal(sender); // sender balance after pay(v)
  assert(b1 == b0 + v);     // looking for a negative example
}
```

Listing 3: Wrong encodings of a liquidity property in Certora.

```
// Solvent specification
property liq {
  Forall xa [
    !frozen
    -> Exists tx [1, xa]
      [ <tx>xa.balance == xa.balance + balance ]
  ]
}
```

Listing 4: Encoding of a liquidity property in Solvent.

In Solvent, we encode the property as in Listing 7, by specifying a constraint on the balances of the contract and of the address `rcv`. Solvent correctly detects that the property holds. The underlying assumption here is that all the

addresses to which a contract transfers crypto-currency behave as EOAs, i.e. they cannot perform internal calls to send the received crypto-currency to some other address. Note that the `transfer` command should rule internal calls, since the amount of gas forwarded to the recipient is not sufficient to pay the gas to complete the execution of an internal call. The other underlying assumption is that the recipient is not rejecting inbound transfers of crypto-currency (this would be possible, e.g., by crafting a recipient contract with a fallback function that always fails). Since there is no rational reason to implement this behaviour, we assume that contracts never deliberately refuse to receive crypto-currency. Furthermore, this would be pointless, since e.g. there is no way to prevent a *selfdestruct* transaction to transfer crypto-currency to an address.

Besides Certora, only SmartPulse [34] and VeriSolid [28] seem capable to deal with liveness properties. SmartPulse targets directly Solidity code, and it has a property specification language based on Linear Temporal Logic (LTL). This makes it possible to express liveness properties, but not liquidity properties, that are out of the reach of LTL (see Section 2). Therefore, the properties supported by SmartPulse and Solvent have uncomparable expressiveness. To be usable in practice, liveness properties must be accompanied by a *fairness assumption*, i.e. another LTL formula that specifies the traces where a user performs some required action in order to reach the desired state. For example, in our crowdfunding contract (Section 3) a target property could be "if the target is not reached, then a donor gets their money back", and the associated fairness assumption could be "the donor performs the action `wdDonor`". A main difference between Solvent and SmartPulse is that, while in SmartPulse the designer of the property must anticipate, in the fairness assumption, the sequence of transactions that a user must perform in order to reach a certain state change, in Solvent this sequence is inferred by the the SMT solver. In particular, the solver infers the (minimal) length of the sequence, the called methods, and their actual parameters in order to produce the desired state change.

VeriSolid takes as input a Solidity contract and its properties expressed in Computation Tree Logic (CTL), transforms the contract into an equivalent Abstract State Machine (ASM), and verifies the properties against the ASM using tools in the BIP toolchain, such as the nuXmv symbolic model checker [13]. The liveness properties specified in [28] are not accompanied by fairness assumptions (unlike SmartPulse), but in principle this seems doable without reworking the verification techniques.[9] We have already noted that the liveness properties expressed in CTL cannot encompass the general liquidity properties addressed by Solvent. We further note that to express liquidity we mix universal and existential quantification on variables (e.g., "for all users, there exists a sequence of transactions made by the user").

[9] A recent extension of VeriSolid include some forms of fairness constraints [14].

```
contract Transfer {
  address payable immutable rcv;

  constructor() payable {
    rcv = payable(msg.sender);
  }

  function withdraw(uint v) public {
    require(v<=address(this).balance);
    rcv.transfer(v);
  }
}
```

Listing 5: A simple deposit contract.

```
// Invariant to be processed by the SolCMC verifier
function invariant(uint v) public {
  uint s0 = rcv.balance;
  uint c0 = address(this).balance;

  withdraw(v);

  uint s1 = rcv.balance;
  uint c1 = address(this).balance;

  assert(s1==s0+v && c1==c0-v);
}
```

Listing 6: A wrong attempt to reason about transfers in SolCMC.

```
// Solvent specification
property liquidity_live {
  Forall xa
  [
    true
    ->
    Exists tx [1, xa]
    [
      <tx>rcv.balance == rcv.balance + balance
    ]
  ]
}
```

Listing 7: Liveness of transfers in Solvent.

A.3 Use case: a 2-players lottery

Consider a lottery where 2 players bet 1 ETH each, and the winner — who is chosen fairly between the two players — redeems the whole pot. Since smart contract are deterministic and external sources of randomness (e.g., random number oracles) might be biased, to achieve fairness we follow a commit-reveal-punish protocol, where both players first commit to the secret hash, then reveal their secrets (which must be preimages of the committed hashes), and finally the winner is computed as a fair function of the secrets.

We show below an implementation of the lottery protocol in Solvent; we then apply our tool to verify some relevant liquidity properties. Intuitively, the protocol followed by honest players is the following:

1. `player1` joins the lottery by paying 1 ETH and committing to a secret;
2. `player2` joins the lottery by paying 1 ETH and committing to another secret;
3. if `player2` has not joined, `player1` can redeem her bet after block `end_commit`;
4. once both secrets have been committed, `player1` reveals the first secret;
5. if `player1` has not revealed, `player2` can redeem both players' bets after block `end_reveal`;
6. once `player1` has revealed, `player2` reveals the secret;
7. if `player2` has not revealed, `player1` can redeem both players' bets after block `end_reveal+100`;
8. once both secrets have been revealed, the winner, who is determined as a function of the two revealed secrets, can redeem the whole pot of 2 ETH.

We implement the lottery protocol in Listings 8 and 9. The expected liquidity properties of the contract, formalised in Listing 10, are the following:

- `p1_redeem_nojoin`: in state 1, `player1` can redeem at least her bet after the block `end_commit`;
- `p2_redeem_noreveal`: in state 2, `player2` can redeem at least both players' bets after the block `end_reveal`;
- `anyone_liquid3`: in state 3, anyone can withdraw the whole contract balance;
- `p1_redeem_noreveal`: in state 4, `player1` can redeem at least both players' bets after the block `end_reveal`;
- `one_player_win`: in state 5, either `player1` or `player2` can redeem at least both players' bets.

Solvent correctly manages to verify that all these properties hold (up-to a given bound of transactions).

```
contract Lottery {
  address player1
  address player2
  int immutable end_commit  // last round to join
  int immutable end_reveal  // last round to reveal
  hash hashlock1
  hash hashlock2
  secret secret1
  secret secret2
  int state

  constructor(int tc, int tr) {
    require (tc < tr);
    end_commit = tc;
    end_reveal = tr;
    state = 0 // next = join1
  }
  function join1(address a1, hash h1) payable {
    require (state==0 && msg.value==1);
    player1 = a1;
    hashlock1 = h1;
    state = 1 // next = join2 or redeem1_nojoin
  }
  function join2(address a2, hash h2) payable {
    require (state==1 && msg.value==1);
    player2 = a2;
    hashlock2 = h2;
    state = 2 // next = reveal1
  }
  function reveal1(secret s1) {
    require (state==2 && block.number >= end_commit);
    require (sha256(s1) == hashlock1);
    secret1 = s1;
    state = 4 // next = reveal2 or redeem2_noreveal
  }
  function reveal2(secret s2) {
    require (state==4 && block.number >= end_reveal + 100);
    require (sha256(s2) == hashlock2);
    secret2 = s2;
    state = 5 // next = win
  }
  function win() {
    require (state==5);
    if ((length(secret1) + length(secret2)) % 2 == 0) {
        player1.transfer(balance)
    } else {
        player2.transfer(balance)
    };
    state = 3 // next = end
  }
  ...
```

Listing 8: A lottery contract (part 1).

```
contract Lottery {
  ...
  function redeem1_nojoin() {
    require (state==1 && block.number>=end_commit);
    player1.transfer(balance);
    state = 3 // next = end
  }
  function redeem2_noreveal() {
    require (state==2 && block.number >= end_reveal);
    player2.transfer(balance);
    state = 3 // next = end
  }
  function redeem1_noreveal() {
    require (state==4 && block.number >= end_reveal+100);
    player1.transfer(balance);
    state = 3 // next = end
  }
  function empty() {
    require (state == 3);
    msg.sender.transfer(balance)
  }
}
```

Listing 9: A lottery contract (part 2).

```
property player1_can_redeem_nojoin {
  Forall xa [
    st.state == 1 && st.block.number >= st.end_commit
    -> Exists tx [1, xa] [
      (<tx>player1.balance >= player1.balance + 1)
    ]
  ]
}
property player2_can_redeem_noreveal {
  Forall xa [
    st.state == 2 && st.block.number >= st.end_reveal
    -> Exists tx [1, xa] [
      (<tx>player2.balance >= player2.balance + 2)
    ]
  ]
}
property anyone_liquid3_live {
  Forall xa [
    st.state == 3
    -> Exists tx [1, xa] [
      (<tx>xa.balance >= xa.balance + balance)
    ]
  ]
}
property player1_can_redeem_noreveal {
  Forall xa [
    st.state == 4 && st.block.number >= st.end_reveal+100
    -> Exists tx [1, xa] [
      (<tx>player1.balance >= player1.balance + 2)
    ]
  ]
}
property one_player_win {
  Forall xa [
    state == 5
    -> Exists tx [1, xa] [
      (<tx>player1.balance >= player1.balance + 2) ||
      (<tx>player2.balance >= player2.balance + 2))
    ]
  ]
}
```

Listing 10: Ideal properties of the lottery contract.