

# Solvent: liquidity verification of smart contracts

Massimo Bartoletti<sup>1</sup>, Angelo Ferrando<sup>2</sup>, Enrico Lipparini<sup>3</sup>, Vadim Malvone<sup>4</sup>

<sup>1</sup> Università degli Studi di Cagliari, Cagliari, Italy

<sup>2</sup> Università degli Studi di Modena e Reggio Emilia, Modena, Italy

<sup>3</sup> Università degli Studi di Genova, Genova, Italy

<sup>4</sup> Télécom Paris, Palaiseau, France

**Abstract.** Smart contracts are programs executed by blockchains networks to regulate the exchange of crypto-assets between untrusted users. Due to their immutability, public accessibility and high value at stake, smart contracts are an attractive target for attackers, as evidenced by a long history of security incidents. This has been a driving factor for the application of formal methods to Ethereum, the leading smart contract platform, and Solidity, its main smart contract language, which have become the target of dozens of verification tools with varying objectives. A current limitation of these 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 decentralized applications that implement financial ecosystems on top of public blockchains. These applications are based on so-called smart contracts, i.e. programs that are run by blockchain networks and automatize the exchange of crypto-assets between users, without relying on trusted authorities. It is estimated that more than 100 billions of dollars worth of crypto-assets are today controlled by smart contracts [14]. 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 [12].

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 [20]. 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 [12], which are outside the scope of most vulnerability detection tools.

The main tools that support the verification of the contracts’ business logic have serious limitations concerning soundness, completeness and expressivity [8]. The reasons of these limitations are in part inherent to the target languages of the tools (Solidity and its compiled bytecode), which have some design choices that are problematic for verification. The well-known reentrancy issue, which is an ongoing cause of problems since the infamous DAO attack [5], is only one of the causes: in general, the glitches of high-level abstractions over the low-level target (e.g., gas costs [15], vulnerable tokens [28]) are a common cause of bugs. Second, existing verification tools for Solidity have limited support for properties asking that, whenever certain states are reached, users can perform some actions that lead to a desirable asset exchange [24]. Properties of this kind, which are neither safety nor liveness properties and are often dubbed *liquidity* [26,9,21], are crucial in the smart contracts setting: typically, one would like to know if, after having deposited some crypto-assets in a contract, they will be able to redeem a given function of the deposited amount. Not being able to verify these properties is a serious limitation: indeed, several real-world attacks exploited liquidity weaknesses, allowing attackers to steal or freeze crypto-assets [3].

To address these issues we propose Solvent<sup>5</sup>, a tool that verifies liquidity properties of smart contracts: in particular, Solvent can practically express and verify properties of the exchanges of assets between users. To overcome the above-mentioned limitations due to Solidity, Solvent operates on a version of Solidity purified from the main semantical quirks of the full language, and extended with special constructs to express hashed timelock protocols, which are commonly used in contracts involving user-chosen secrets (e.g., in gambling games and blind auctions). Solvent takes as input a contract and a set of user-defined liquidity properties, and translates them into constraints in the SMT-LIB standard [7], reducing the verification problem to a symbolic model checking one. Then, techniques such as bounded model checking and predicate abstraction are employed, relying on Z3 [13] 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 out of the scope of industrial verification tools operating on the full Solidity, like e.g. SolCMC [4] and Certora [18], as well as academic tools [19,16,23,22,25,27] (due to space constraints, we compare Solvent with these tools in the Appendix). Solvent provides developers with useful feedback, by detecting logical errors that would otherwise remain unnoticed. In particular, when Solvent detects that a property is violated, it produces a witness of the violation in terms of a concrete execution trace leading the contract to a state from which the desired state change (possibly involving transfers of crypto-assets) is not realisable.

---

<sup>5</sup> <https://github.com/AngeloFerrando/Solvent>

**Contributions** The main contributions of this tool demonstration 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](#));
- a concrete demonstration of the tool applicability, showing subtle bugs in existing smart contracts that cause crypto-assets to get frozen forever.

## 2 Tool demo

In this section 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 SMT constraints; (2) then, it issues satisfiability queries to SMT solvers (in our setting, Z3 and cvc5) 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 ETH to the contract along with the call (denoted by `msg.value`). The constructor specifies the owner of the crowdfunding campaign, a deadline for donations, and the target (in ETH, the Ethereum native cryptocurrency). The method `donate` allows anyone to donate any ETH 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

```

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 (target_reached && block.number > end_donate);
        owner.transfer(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]
        [ <tx>balance[xa] >= balance[xa] + donors[xa] ] ]
}

```

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

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 *unsatisfiable*, then the property holds; otherwise, a counterexample is produced. E.g., for `donors_wd`, the counterexample is the following sequence of transactions, which lead to a state where the property cannot be satisfied:

|     |                               |                                    |                          |
|-----|-------------------------------|------------------------------------|--------------------------|
| [1] | <code>constructor(0,2)</code> | <code>msg.sender=address(4)</code> | <code>msg.value=0</code> |
| [2] | <code>donate()</code>         | <code>msg.sender=address(4)</code> | <code>msg.value=1</code> |
| [3] | <code>coinbase()</code>       | <code>msg.sender=address(0)</code> | <code>msg.value=1</code> |

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.

### 3 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 benchmark is focussed on current verification tools for Solidity, which have limited support for 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 1000s of CPU time. A subset of the results are shown in Table 1 (see [github](#) for the full results). We mark each property as: “ $\mathbf{X}(N)$ ”, if the solver finds a trace that violates the property (with  $N$  being the length of the shortest trace leading to a violation); “ $\checkmark$ ”, if it proves that the property holds in all possible states; “ $\checkmark(N)$ ”, 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 one, 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.<sup>6</sup>

*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  $\checkmark(N)$ , 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  $\mathbf{X}$ ).

<sup>6</sup> 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 [17].

| Contract           | Property                 | Liquid? | cvc5   |      | Z3     |       |
|--------------------|--------------------------|---------|--------|------|--------|-------|
|                    |                          |         | Result | Time | Result | Time  |
| Auction            | seller_wd                | ✓       | ✓(10)  | —    | ✓(12)  | —     |
|                    | old_winner_wd            | ✓       | ✓(27)  | —    | ✓(12)  | —     |
|                    | no_frozen_funds          | ✗       | ✗(3)   | 2.62 | ✗(3)   | 2.59  |
| Bank               | deposit_not_revert       | ✓       | ✓      | 4.88 | ✓      | 23.34 |
|                    | withdraw_not_revert      | ✓       | ✓(7)   | —    | ✓(9)   | —     |
| Bet                | any_timeout_join         | ✓       | ✓(19)  | —    | ✓(34)  | —     |
|                    | oracle_win               | ✓       | ✓(16)  | —    | ✓(16)  | —     |
|                    | any_timeout_win          | ✓       | ✓(89)  | —    | ✓(20)  | —     |
|                    | no_frozen_funds          | ✗       | ✗(2)   | 1.90 | ✗(2)   | 2.01  |
| Crowdfund<br>(bug) | owner_wd                 | ✓       | ✓      | 4.65 | ✓      | 22.97 |
|                    | donor_wd                 | ✗       | ✗(3)   | 3.10 | ✗(3)   | 42.80 |
|                    | no_frozen_funds          | ✗       | ✗(1)   | 0.99 | ✗(1)   | 0.90  |
| Escrow             | arbiter_wd_fee           | ✓       | ✓(17)  | —    | ✓(22)  | —     |
|                    | buyerorseller_wd_deposit | ✓       | ✓(22)  | —    | ✓(87)  | —     |
|                    | anyone_wd                | ✗       | ✗(2)   | 1.77 | ✗(2)   | 1.96  |
|                    | no_frozen_funds          | ✗       | ✗(3)   | 2.92 | ✗(3)   | 2.75  |
| HTLC               | owner_wd                 | ✓       | ✓(15)  | —    | ✓(21)  | —     |
|                    | verifier_wd_timeout      | ✓       | ✓(19)  | —    | ✓(21)  | —     |
|                    | no_frozen_funds          | ✓       | ✓      | 4.96 | ✓      | 29.42 |
| Lottery            | one_player_win           | ✓       | ✓(11)  | —    | ✓(15)  | —     |
|                    | p1_redeem_nojoin         | ✓       | ✓(26)  | —    | ✓(28)  | —     |
|                    | p1_redeem_noreveal       | ✓       | ✓(19)  | —    | ✓(17)  | —     |
|                    | p2_redeem_noreveal       | ✓       | ✓(20)  | —    | ✓(21)  | —     |
| Payment splitter   | anyone_wd_ge             | ✓       | ✓(2)   | —    | ✓(1)   | —     |
|                    | anyone_wd_releasable     | ✓       | ✓(2)   | —    | ✓(1)   | —     |
|                    | anyone_wd                | ✗       | ✗(1)   | 1.03 | ✗(1)   | 0.84  |
| Vault              | fin_owner                | ✓       | ✓(17)  | —    | ✓(19)  | —     |
|                    | canc_recovery            | ✓       | ✓(52)  | —    | ✓(21)  | —     |
|                    | wd_fin_owner             | ✗       | ✗(1)   | 1.34 | ✗(1)   | 1.18  |
| Vesting wallet     | owner_wd_expired         | ✓       | ✓      | 6.46 | ?      | T/O   |
|                    | owner_wd_started         | ✓       | ?      | T/O  | ?      | T/O   |
|                    | owner_wd_uncond          | ✗       | ?      | T/O  | ✗(1)   | 1.14  |
|                    | owner_wd_beforestart     | ✗       | ?      | T/O  | ✗(1)   | 0.96  |
|                    | owner_wd_empty           | ✗       | ?      | T/O  | ✗(1)   | 0.96  |
|                    | 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  $\checkmark(N+1)$ .

## 4 Conclusions and Future work

Solvent has already proven useful to spot non-trivial liquidity vulnerabilities in smart contracts, which are beyond the reach of mainstream verification tools for Solidity. Still, there is space for improvements. First, Solvent makes use of 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 is to leverage more advanced techniques (such as abstraction-refinement, k-induction, etc.) used by modern infinite-state symbolic model checkers. Further future work involves extending the Solvent fragment to narrow the gap with the full Solidity language. This could amount to extend the subset of Solidity supported by the tool to include e.g. contract-to-contract calls (see the Appendix for further extensions), and to automatically transform the transaction sequences outputted by Solvent into executable Ethereum tests in the [Ganache](#) tool.

*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., Blich, 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](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](https://doi.org/10.1007/978-3-662-54455-6_8), [http://dx.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](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. Bartoletti, M., Fioravanti, F., Matricardi, G., Pettinau, R., Sainas, F.: Towards benchmarking of Solidity verification tools. In: Formal Methods in Blockchain (FMBC) (2024), to appear
9. 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](https://doi.org/10.46298/LMCS-18(1:22)2022)
10. 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](https://doi.org/10.1007/978-3-319-24953-7_25)
11. 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)
12. 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>
13. 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](https://doi.org/10.1007/978-3-540-78800-3_24)
  14. Defillama. <https://defillama.com/>, accessed on April 9, 2024
  15. 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>
  16. 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](https://doi.org/10.1007/978-3-030-41600-3_11)
  17. 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>
  18. Jackson, D., Nandi, C., Sagiv, M.: Certora technology white paper. <https://docs.certora.com/en/latest/docs/whitepaper/index.html> (2022)
  19. 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)
  20. 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>
  21. 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>
  22. Nelaturu, K., Mavridou, A., Stachtari, 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>
  23. 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>
  24. Schiffl, J., Beckert, B.: A practical notion of liveness in smart contract applications. In: Formal Methods in Blockchain (FMBC) (2024), to appear
  25. 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>
  26. 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>
  27. Wesley, S., Christakis, M., Navas, J.A., Treffer, R.J., Wüstholtz, V., Gurfinkel, A.: Verifying Solidity smart contracts via communication abstraction in SmartACE. In: Verification, Model Checking, and Abstract In-



- terpretation (VMCAI). LNCS, vol. 13182, pp. 425–449. Springer (2022). [https://doi.org/10.1007/978-3-030-94583-1\\_21](https://doi.org/10.1007/978-3-030-94583-1_21)
28. 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 the verification of smart contracts. In particular, we draw a detailed comparison between Solvent and the two main verification tools for Solidity, i.e. SolCMC and Certora, highlighting their weaknesses in the verification of liquidity properties. We also compare Solvent with other verification tools that can address liveness properties, such as VeriSolid [\[22\]](#) and SmartPulse [\[25\]](#).
- 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 [2](#).

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 [23], 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 Related work

Current mainstream verification tools for Solidity are quite limited when it comes to liquidity properties. SolCMC, the prover shipped with the Solidity compiler, and other verification tools such as Zeus [19], solc-verify [16], VerX [23] and SmartACE [27], support safety properties, and therefore they cannot express any of the liquidity properties considered in this paper.

Other tools such as Certora [18], VeriSolid [22], and SmartPulse [25], can reason about liveness properties, i.e. properties of the form “for all reachable states, *eventually* some desirable outcome will be obtained”. While liveness properties are closely related to liquidity properties, and in some cases can be used to express similar concepts, they cannot express most of the properties considered in this paper (and relevant in practice): namely, they cannot express properties of

the form “for all reachable states, some user can do something that *immediately* produces some desirable effect”.

First, we analyse Certora, one of the leading formal verification tools for Solidity. To illustrate the expressiveness limitations of the tool, 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 in 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 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.

```

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> balance[xa] == balance[xa] + balance ]
    ]
}

```

Listing 4: Encoding of a liquidity property in Solvent.

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<sup>7</sup>.

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<sup>8</sup>. 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 [8].

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 [25] and VeriSolid [22] seem capable to deal with liveness properties, which are similar (though not equivalent) to liquidity.

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. Indeed, in order to express the concept “there exists a path that leads to a desirable outcome”, one needs to use the **E** temporal operator of CTL. 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 2) a target property could be “if the target is not reached, then a donor gets its 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

<sup>7</sup> 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.

<sup>8</sup> <https://docs.openzeppelin.com/contracts/4.x/api/utils#Address>

```

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>balance[rcv] == balance[rcv] + balance
        ]
    ]
}

```

Listing 7: Liveness of transfers in Solvent.



solver infers the (minimal) length of the sequence, the called methods, and their actual parameters in order to produce the desired state change. This allows for a more succinct representation of liveness properties compared to SmartPulse.

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 [10]. The liveness properties specified in [22] are not accompanied by fairness assumptions (unlike SmartPulse), but in principle this seems doable without reworking the verification techniques.<sup>9</sup> It is not clear whether the liveness properties expressed in CTL can encompass the liquidity properties addressed by Solvent: indeed, to express liquidity we mix universal and existential quantification on variables (e.g., “for all users, there exists a sequence of transactions”), which seems beyond the scope of plain CTL.

### 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 liveness 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 liveness 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`;

<sup>9</sup> A recent extension of VeriSolid include some forms of fairness constraints [11].

```

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!balance
        } else {
            player2!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!balance;
    state = 3 // next = end
  }
  function redeem2_noreveal() {
    require (state==2 && block.number >= end_reveal);
    player2!balance;
    state = 3 // next = end
  }
  function redeem1_noreveal() {
    require (state==4 && block.number >= end_reveal+100);
    player1!balance;
    state = 3 // next = end
  }
  function empty() {
    require (state == 3);
    msg.sender!balance
  }
}

```

Listing 9: A lottery contract (part 2).

- 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).

```

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

```

Listing 10: Ideal properties of the lottery contract.