Towards Benchmarking of Solidity Verification Tools

Massimo Bartoletti
University of Cagliari, Italy

Fabio Fioravanti
University of Chieti-Pescara, Italy

Giulia Matricardi
University of Chieti-Pescara, Italy

Roberto Pettinau
Technical University of Denmark, Lyngby, Denmark

Franco Sainas
EPFL, Lausanne, Switzerland

Abstract

Formal verification of smart contracts has become a hot topic in academic and industrial research, given the growing value of assets managed by decentralized applications and the consequent incentive for adversaries to tamper with them. Most of the current research on the verification of contracts revolves around Solidity, the main high-level language supported by Ethereum and other leading blockchains. Although bug detection tools for Solidity have been proliferating almost since the inception of Ethereum, only in the last few years we have seen verification tools capable of proving that a contract respects some desirable properties. An open issue is how to evaluate and compare the effectiveness of these tools: indeed, the existing benchmarks for general-purpose programming languages cannot be adapted to Solidity, given substantial differences in the programming model and in the desirable properties. We address this problem by proposing an open benchmark for Solidity verification tools. By exploiting our benchmark, we compare two leading tools, SolCMC and Certora, discussing their completeness, soundness and expressiveness limitations.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Smart contracts, Ethereum, Verification, Blockchain

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.6

Supplementary Material *Software*: `https://github.com/fsainas/contracts-verification-benchmark`, archived at `swh:1:dir:5452a36f0c58435ad4b7dadb7e7563653ffc25b6`

Funding *Massimo Bartoletti*: Partially supported by project SERICS (PE00000014) and PRIN 2022 DeLiCE (F53D23009130001) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

1 Introduction

The rapid growth of decentralized applications based on blockchain technologies have emphasized the importance of ensuring the security of smart contracts – the basic building blocks of these applications. The research on smart contracts security has been proliferating since 2016, leading on the one side to the discovery of a variety of attacks, and on the other side to the development of several tools to detect vulnerabilities of smart contracts before they are deployed. Despite the increasing breadth and precision of these analysis tools, attacks to smart contracts have caused financial losses worth several billions of dollars so far, and are unlikely to be eradicated anytime soon.

A large class of analysis tools for smart contracts are focused on detecting known vulnerability patterns in contracts code. Even though tools of this type can detect many nefarious bugs, statistically the vast majority of the losses due to real-world attacks are caused by logic errors in the contract code, which cannot be prevented by only checking for fixed vulnerability patterns [14]. In this context, a contract can be considered secure when its executions are coherent with some ideal behaviour, even in the presence of adversaries trying to subvert it. Only a few tools support this kind of security analysis, allowing developers to specify the ideal properties the contract is expected to satisfy. In this work we focus on SolCMC and Certora, two leading verification tools for contracts written in Solidity, the main smart contract language for Ethereum and EVM-compatible blockchains. Both tools allow the developer to specify desirable contract properties, and use SMT solvers to verify whether the contract satisfies them, showing a counterexample when detecting a violation. Although both tools have been independently tested by their developers [1, 10], no public comparison exists so far to assess their effectiveness and limitations in practice.

Our long-term goal is a comprehensive, publicly available benchmark to evaluate the effectiveness of verification tools for Solidity contracts. As an initial step towards this goal, in this paper we present a benchmark comprising 323 verification tasks, each one made of a Solidity contract and a property it is expected to satisfy.[1] A crucial component of our benchmark is a manually crafted ground truth of the verification tasks, encompassing multiple versions of each smart contract in order to cover different ways of satisfying or violating its associated properties. To foster the reproducibility of the results, we make available a toolchain that automatises the construction of the verification tasks, their processing with SolCMC and Certora, and the summarisation of the results. Based on these artifacts, we present a preliminary evaluation of SolCMC and Certora, comparing their completeness, soundness, and expressiveness. Finally, we introduce a scoring scheme for Solidity verification tools, which is inspired by schemes used in software verification competitions [7], but taking into account the peculiarities of the smart contracts context.

2 Background and related work

Over the years, several dozens of tools have been developed to analyse Ethereum contracts (see e.g. [26, 23, 21] for systematic surveys). The vast majority of these tools focus on specific types of contract vulnerabilities, such as reentrancy, integer overflow and underflow, mishandled exceptions, transaction ordering dependence, *etc.* [19]. Some tools focus on runtime verification of contracts [4], to force failures when some property violation is detected at run-time. More recent tools give users more control on the properties to be verified, in principle enabling the verification of contract implementations against an ideal, abstract description of their behaviour.

A prominent tool in this category is SolCMC [2], a symbolic model checker integrated in the Solidity compiler since 2019. Specifying properties in SolCMC requires developers to instrument the contract code with `assert` statements, which are treated as verification targets. Failure of an `assert` means that the desired property is not satisfied by the contract. For example, consider a method `deposit` that receives ETH from any user, recording the sent amount in a `balances` mapping. The property "after a successful `deposit`, the balance entry of `msg.sender` is increased by `msg.value`" can be encoded as the function in Listing 1:

[1] `https://github.com/fsainas/contracts-verification-benchmark`

Listing 1 SolCMC encoding of a safety property.

```
function deposit_user_balance() public payable {
  uint old_user_balance = balances[msg.sender];
  deposit();
  uint new_user_balance = balances[msg.sender];
  assert(new_user_balance == old_user_balance + msg.value);
}
```

This defines a contract invariant that must be true for any reachable contract state: the balance of a user after a successful call is equal to the previous balance plus the deposit. SolCMC transforms the instrumented contract into a set of Constrained Horn Clauses (CHC) [8, 16] which is fed to a CHC satisfiability solver (Spacer [25], integrated in Z3 [17], or Eldarica [22]) to check if any assert can fail. If so, it produces a trace witnessing the violation.

Certora [24, 6] is another leading formal verification tool for Solidity. Unlike SolCMC, it decouples the specification of the properties from the contract code. Properties, written in the Certora Verification Language (CVL), roughly can take the form of `assert` statements ("for all contract runs, the condition holds") or `satisfy` statements ("there exists a run where the condition holds"). For instance, the CVL specification of the `deposit` property seen before is shown in Listing 2. Certora compiles the Solidity contract and its associated properties into a logical formula, and sends it to an SMT solver. Another key difference between Certora and SolCMC is that in SolCMC verification is done locally (since it is part of the Solidity compiler stack), while in Certora it is executed remotely on a cloud service.

Listing 2 Certora encoding of a safety property.

```
rule deposit_user_balance {
  env e; // an arbitrary transaction and context
  address sender = e.msg.sender;
  mathint old_user_balance = getBalanceEntry(sender);
  deposit(e); // calls deposit with context e
  mathint new_user_balance = getBalanceEntry(sender);
  mathint deposit_amount = to_mathint(e.msg.value);
  assert new_user_balance == old_user_balance + deposit_amount;
}
```

Besides SolCMC and Certora, other tools for verifying user-defined properties of Solidity contracts have been proposed (see [2] for a comparison). VerX [29] models properties in a variant of past linear temporal logic. This allows to verify safety properties of contracts, while liveness properties are not expressible. SmartACE [34] verifies properties written in Scribble [15]: contracts are annotated with Scribble annotations (i.e., contract invariants and method postconditions). Scribble transforms the annotated contract into a contract with `assert`s, which are used as verification targets. SmartACE uses local bundle abstractions to reduce the state explosion caused by having to deal with many users interacting with the contract, factorising users into a representative few. It models each contract in LLVM-IR and integrates existing analysers such as SeaHorn and Klee to facilitate verification. Notably, SmartAce has been applied to verify some contracts from the OpenZeppelin library [35].

Comparing verification tools

A primary source of comparison among different verification tools is given by the research papers where these tools were introduced [20, 33, 30]. A problem here is that each comparison is based on an ad-hoc dataset of contracts and properties, which makes it difficult to compare the effectiveness of different tools. The work [3] provides a unifying view of these datasets, by collecting their verification tasks and judgements, and mapping them to a uniform scheme based on the Smart Contract Weakness Classification [19]. A main difference between this dataset and ours is in the nature of the properties in the verification tasks: the ones in [3] are specific vulnerabilities (e.g., reentrancy, overflows, *etc.*), while ours are ideal properties of the analysed contract (e.g., "after calling `foo`, the sender receives 1 ETH").

A few works compare different analysis tools without introducing their own. The work [14] evaluates five tools based on their ability to identify vulnerabilities that have been actually exploited by attacks in the wild. Perhaps surprisingly, the conclusion is that tools that detect specific vulnerability patterns are ineffective against real attacks, being able to counter only $\sim$12% to the economic damage in the considered dataset, while offering no protection against the remaining part of the damage, which exceeds 2 billion dollars. This is a strong motivation for research on analysis techniques and tools that can also detect logic-related bugs, which are the focus of our benchmark. The work [18] proposes a vulnerability classification scheme that extends [19], and evaluates the effectiveness of three bug detection tools. We note that both works [18, 14] focus on tools that detect specific vulnerabilities: at the best of our knowledge, ours is the first comparison between general verification tools for Solidity. Another main difference between our work and [18] is that the comparison in [18] is based on a quantitative evaluation of the tool outcomes (in the form of a confusion matrix), while we also devise a qualitative comparison that explains the reason behind these results, and in particular the causes of unsoundness (false positives) and incompleteness (false negatives).

3 Our benchmark

The benchmark is logically organized in the following components:

- a collection of informal specifications of use cases for smart contracts, each accompanied by a set of desirable properties. We deliberately choose *not* to use a formal language to write the smart contract specifications and the associated properties, since we want to be free to express properties that go beyond those expressible by current verification tools.
- Solidity implementations of the use cases and specifications of their properties in the languages supported by SolCMC and Certora. For each use case we provide multiple Solidity implementations, either respecting the given properties or violating them (in obvious or subtle ways). A *verification task* comprises the implementation of a use case and that of a related property.
- a *ground truth* that assesses, for each verification task, whether the implementation satisfies the associated property or not.

Our toolchain processes these data to construct the verification task and runs SolCMC and Certora on each of them.

The way we construct the verification tasks is tool-specific:

- for SolCMC, each property is encoded in Solidity within the associated smart contract. Although, in general, these asserts can be scattered throughout the contract code, in our benchmark we keep the definitions of the properties separated from the contracts, in order to automatize the verification of multiple properties on multiple version of the contract. Accordingly, we provide two ways to write a property:
  - as a function that is added to the contract. This function may assert invariants on the contract state, and may call other contract methods as a property wrapper.
  - as a set of fragments of ghost code that are injected in the contract methods.

In this way, whoever extends the benchmark can write these properties without affecting the behaviour of the original contract, so that the instrumented contract satisfies the considered properties if and only if the original one does. In practice, this can be achieved by preventing ghost code from writing the state of the original contract and from changing its control flow except to signal the violation of the desired property. Future versions of the toolchain will give warnings when detecting potential discrepancies.

- for Certora, we write properties in the Certora Verification Language (CVL) [9]. The syntax of CVL extends Solidity with a set of meta-programming primitives that allow to express complex contract properties. We encode a contract property as a CVL *invariant* when the property involves facts about the state of the smart contract that should be true in any execution, while we encode it as a *rule* when the property concerns the expected behavior of calling one or more contract methods. In general, a rule is a sequence of commands that describe an execution trace of the contract, together with preconditions (`require`) and postconditions. There are two kinds of postconditions: `assert`, which *must* hold for any trace, and `satisfy`, which *can* be satisfied (i.e., it is possible to find a trace that makes them true). Unlike SolCMC, in Certora ghost code can be encoded directly within the properties, without altering the contract being verified.

Scoring the results of the tools

After constructing the verification tasks, the toolchain runs SolCMC (locally) and Certora (remotely) on each of them. The execution outcome on a verification task is summarised and scored according to the schema reported in Table 1. The overall design goal is that a tool that does nothing will have a null score, a tool that provides correct answers when verifying or detecting violation of the properties will have a positive score, and a tool that tricks the user into believing false results (e.g., claiming that a property holds when it is not the case, or viceversa) will have a negative score. We assign a null score in three cases: when the property is not expressible in the tool, when the tool fails to provide an output (e.g., because of aborts, timeouts, or memory exhaustion), and when the tool does not provide a definite answer about the validity of a property. The ratio for assigning the same score here is that it would be easy to make the property expressible by a tool that always diverges.

Our viewpoint is that tools are aimed at certifying that desirable properties are satisfied by a given Solidity implementation. Therefore, our scoring schema privileges soundness over completeness, as a false positive may create much bigger problems to users, as they will be convinced that their contract satisfies a property that in practice does not hold, while a false negative will only make the user doubt of the correctness of the contract.

The basis of our scoring schema is standard: we have two cases (P/N) depending on whether the property in the verification task is satisfied or not, and two cases (T/F) depending on whether the tool answers correctly to the task or not. We slightly deviate from this standard classification, in that we additionally classify the outputs of a tool as *strong* claims (e.g., “the property holds”, “the property is violated”) or *weak* claims (e.g., “the property *might* hold”, “the property *might* be violated”). More specifically, we use the following criteria to distinguish between weak and strong claims of the tools at hand:

- in SolCMC, when verification terminates, the output has one of the following forms:
  - “Assertion violation check is safe!”. We consider this as a strong claim that the tool has verified the property, hence we classify the output as a P!
  - “Assertion violation happens here”. In this case, the tool outputs the line of code where the asserted property is violated, and shows a sequence of method calls that lead to the violation. Hence, we consider this as a strong claim, and classify it as an N!

Table 1 Scoring schema for Solidity verification tools.

Result	Points	Description
ND	0	Property not expressible in the tool
UNK	0	Timeout / Memory exhaustion
TN!	2	Property violated, tool claims violation
TN	1	Property violated, tool conjectures violation
FN!	-8	Property holds, tool claims violation
FN	0	Property holds, tool conjectures violation
TP!	2	Property holds, tool claims correctness
TP	1	Property holds, tool conjectures correctness
FP!	-16	Property violated, tool claims correctness
FP	-1	Property violated, tool conjectures correctness

  - "Assertion violation might happen here". Here the tool has not been able to verify neither the violation, nor the correctness of the property, and (to stay on the safe side) it states that a violation is possible. We classify this output as an N.
- in Certora the classification depends both on the tool output and how the property is modelled in CVL. For an `assert`, the output is classified as P! when Certora returns ok, and N when it rejects the property. We do not put the "!" in the negative case as Certora may reject the property because of an unreachable counterexample, while we put the "!" in the positive case because it has been able to prove that no state (reachable or not) leads to a violation. The case `satisfy` is dealt with dually: we classify as P when Certora returns ok, and N! when it rejects the property.[2] Since our benchmark only admits rules that do not use both `assert` and `satisfy`, this criterion is always applicable.

The scoring schema is displayed in Table 1. Coherently with our design choices, we assign FP! the lowest score, because the tool is falsely claiming that a desirable property is true. Strong false negatives FN! (i.e., false alarms) are considered "half as dangerous" than false positives. False weak accepts (FP) have only a mildly negative score, since we treat them as mere conjectures: in fact, when conjecturing a result, the tool is just conveying the fact that it is not convinced of the truth of the opposite result. The asymmetry between FP! and FN! is mimicked on weak judgements by assigning false weak rejects (FN) a null score.

4 Evaluation: SolCMC *vs.* Certora

We discuss in this section what we have learnt by using SolCMC and Certora in the design and application of our benchmark. Here we focus on completeness, soundness and expressiveness of the two tools: their scoring as per Table 1 is presented at the end of the section. As a disclaimer, we note that our evaluation is based on the current versions of the tools[3]. Since they are moving targets, with multiple updates released during the writing of this paper, it is likely that some of the weaknesses discussed below may be fixed in future releases.

Completeness

SolCMC and Certora share some sources of incompleteness, i.e. properties that are true but that the provers do not manage to prove. This is the case e.g. for contracts containing external calls, i.e. calls from the analysed contract to another account [32] (e.g., the Deposit/ERC20

[2] It is possible to limit the set of considered starting states by refining the implementation with a set of invariants. We do this in a best effort manner.

[3] Versions: solc v0.8.24, Eldarica v2.0.9, Z3 v4.12.2, certora-cli v6.3.1

use case). By default, called contracts are considered untrusted by SolCMC and Certora, and accordingly these tools over-approximate their behaviour (even when their code is known). This basically makes the provers fail to verify any property that depends on the behaviour of the called contract, so leading to false negatives. For example, the `assert` in Listing 3 always passes, but SolCMC detects a possible violation (the same happens with Certora). SolCMC has an option to change the default behaviour by considering external calls to be trusted, but a known drawback of this option is a substantial computational overhead. For instance, checking the invalidity of the assertion `c.n()==0` in the contract `C2` of Listing 3 takes ~ 8 minutes using the Z3 solver in our experimental setting, despite the contract being just a few lines of code (by contrast, it takes a few seconds with the untrusted option). In general, even with the default option about untrusted calls, non-termination is not uncommon (see Table 2). This is a general problem related to Z3, which can be easily misled to divergence even with apparently harmless sets of constraints. Even in real-world use cases not specifically crafted to make verification burdensome, computation times sometimes occur to explode unexpectedly.

Listing 3 SolCMC: untrusted external calls.

```
contract C1 {
  uint n;
  constructor() { n=0; }

  function set() external { n=1; }

  // n could be either 0 or 1
}
```

```
contract C2 {
  C1 public c;
  constructor() { c=new C1(); }

  function inv() public view {
    assert(c.n()<=1);
  }
}
```

Many desirable properties of accounts, like e.g. that the balance is updated according to certain rules, are often broken on *contract* accounts. For instance, consider a contract with a method that allows the sender to withdraw funds, and the property "after a successful call to `withdraw`, the balance of `msg.sender` has increased". This property may be violated when the sender is a contract account, which fallbacks on the `withdraw` by giving away all its balance. While properties of this kind do not hold, in general, for contract accounts, they are expected to hold for externally-owned accounts (EOAs), which do not have code (e.g., `withdraw-sender-rcv-EOA` in the bank use case). In general, properties of EOAs are not even expressible, since it is not possible to discriminate EOAs from contract accounts (see the discussion below about expressiveness). A property about EOAs is expressible only if the account under scrutiny is the `msg.sender`. In this case, it is possible to tell that the account is an EOA by comparing it to `tx.origin` (the transaction originator): namely, the two addresses are equal iff `msg.sender` is an EOA. The property above can then be refined as "if `msg.sender` is an EOA, then after a call to `withdraw`, the balance of `msg.sender` has increased". Both SolCMC and Certora fail however to verify that the amended property is satisfied. No alternative encodings of EOAs seem to exist that allow the provers to successfully verify non-trivial properties about them.

Further cases of incompleteness include map invariants (e.g., the sum of a map is preserved), which are unlikely to be proved by both tools, and the over-approximation of the possible environments. E.g., Certora includes the contract in the approximation for `msg.sender`, even if the contract has no calls (e.g., `deposit-contract-balance` in the bank).

Listing 4 Unsoundness in SolCMC: *selfdestruct*.

```
contract CallWrapper is
    ReentrancyGuard {
  function callwrap(address called)
       public nonReentrant {
    called.call("");
  }
  ...
}
```

```
// SolCMC invariant
function inv(address a) public {
  uint b = address(this).balance;
  callwrap(a);

  // contract balance is preserved
  assert(b==address(this).balance);
}
```

Soundness

Analysing the results of our benchmark, we have spotted a few sources of unsoundness (i.e., the property does not hold but the tool falsely claims it is true) for both SolCMC and Certora. In SolCMC, false positives may happen when reasoning about the contract balance in case of external calls and reentrancy guards, as shown in Listing 4 (see the `bal` property in the CallWrapper use case). The contract `CallWrapper` has a single method, which performs a low-level call to an arbitrary address; the `nonReentrant` modifier by OpenZeppelin ensures that this call is non-reentrant. Now, consider the property: "the contract balance is preserved by `callwrap`". Apparently, it might seem to hold, because the contract has no `payable` nor `receive` methods. However, there are other asynchronous events that can make the contract balance increase: e.g., it can receive ETH from a *coinbase* transaction (i.e., the first transaction in a block, which collects the block reward), or from a *selfdestruct* (i.e., an action performed by a contract to destroy itself and transfer the remaining ETH to another account) [31]. Therefore, the property does *not* hold, since the address called by `callwrap` can be a contract that triggers a *selfdestruct*. SolCMC here produces a false positive, claiming that the invariant `inv` is always satisfied. We conjecture that this output derives from an under-approximation of SolCMC, which believes that the absence of reentrant methods implies that the call cannot affect the contract state, including the balance. Note instead that, when removing the `nonReentrant` modifier, SolCMC correctly detects that the invariant may be violated. Certora instead correctly classifies the property as false, but it produces a false positive on an extension of `CallWrapper` with a variable `s` that can be updated by the method `set`, and the property "`s` is preserved by `callwrap`" (see Listing 5 and the `stor` property in the CallWrapper use case). This property is false, since the account called by `callwrap` can perform a reentrant call to `set`. Certora fails to understand that a call to an address may lead to additional code execution, including a further call to one of the methods of the contract. Hence, Certora claims that the property holds, hereby being unsound [13].

Listing 5 Unsoundness in Certora: untracked reentrant calls.

```
contract CallWrapper {
  uint s;
  ...
  function set(uint snew) public {
    s = snew;
  }
}
```

```
rule P(address a) {
  env e;
  uint s0 = currentContract.s;
  callwrap(e, a);
  uint s1 = currentContract.s;
  assert s1 == s0;
}
```

Certora has some further documented under-approximations that may lead to unsoundness [11, 12]. When dealing with invariants, Certora checks that the invariant is preserved after the execution of each contract method, neglecting the effect of *selfdestruct* or *coinbase* transactions. These transactions may increase the ETH balance of the analysed contract, which has no way of preventing this unexpected incoming ETH: therefore, if the invariant depends on the contract balance, it may be broken at any time. As an example, consider the contract `DoNothing` in Listing 6 , which just saves its initial balance in a variable `bal0`, and does nothing afterwards. The associated CVL property is an invariant checking that the contract balance is always equal to the stored initial balance. Certora claims that the invariant holds, since it holds at creation and it is preserved after the execution of each method (`balanceOf`). This is unsound, since e.g. a *coinbase* transaction can send ETH to the contract, making the actual balance exceed `bal0`.

Listing 6 Unsoundness in Certora: untracked *selfdestruct* and *coinbase* transactions.

```
contract DoNothing {
  uint bal0;
  constructor() {
    bal0 = address(this).balance;
  }
  function balanceOf(address a)
      public view returns (uint) {
    return a.balance;
  }
}
```

```
// Certora invariant specification

invariant inv()
  balanceOf(currentContract)
  ==
  currentContract.bal0;
```

Besides the artificial example in Listing 5, false positives may occur in real-world use cases when reasoning about the state after a low-level call. For instance, in our benchmark this is the case for a simple bank contract allowing users to deposit and withdraw ETH, and the property requiring that after a successful `withdraw`, the balance entry of the sender is decreased of the right amount. Apart from these glitches, both SolCMC Certora perform comparably well regarding false positives on our benchmark (see Table 2).

Expressiveness limitations

One of the main categories of properties that cannot be encoded in SolCMC are liveness properties (e.g., `wd-fin-before` in the vault use case). For a minimal example, consider Listing 7 and the property "`foo` never reverts". Intuitively, we can call `foo` inside of a `try-catch` statement, making sure that the method does not revert by checking that the `catch` is not reachable. However, this encoding is unsound, since the invariant is satisfied by some implementations of `foo` that actually revert. Specifically, this is the case of implementations of `foo` that never revert when the method is called by the contract itself, like the one in Listing 7, left. The invariant passes, but if foo is called from any account different from `Liveness`, then `foo` reverts. This is not the only way to trick SolCMC into verifying a false liveness property, as any assumption made by having the contract call itself can be used (e.g., reentrancy). In general, when expressing a property in SolCMC, any external call made in the invariant does not faithfully capture the intended property, as it does not model a call made by an arbitrary user, but only by the contract itself. The same problem exists when we use a low-level call instead of an external call within a `try-catch`. Attempting to encode the liveness property as the success of the invariant itself does not work either: any command in the body of the invariant cannot ensure that a certain line of it is always reached.

Even restricting to safety, expressing properties that involve two or more method calls in sequence is tricky, and in particular it cannot be done by using invariants, only. E.g., the invariant in Listing 8 (right) is a seemingly reasonable (but unsound) encoding of the property "`bar` cannot be called twice in a row" (see also the property `wd-twice` in the vault).

Listing 7 Completeness in SolCMC: a wrong encoding of a liveness property.

```
contract Liveness {
  function foo() {
    require(msg.sender==
            address(this));
  }
  ...
}
```

```
// wrong encoding of "foo succeeds"

function inv() public {
  try this.foo() {} catch {
    assert(false);
  }
}
```

Consider e.g. the implementation of `bar` in the contract `Sequence`: here, two consecutive calls to `bar` are possible whenever the callers are distinct: hence, the property is violated. Notice instead that the invariant is satisfied, because the sender of both calls to `bar` is the same, since coincides with the sender of `inv`. A sound encoding is still possible, but at the cost of instrumenting the contract with ghost code, which although supported by our toolchain, is a complex and error-prone operation in general. By contrast, Certora can express smoothly this kind of properties as CVL rules, whenever the number of calls in the sequence is fixed. Properties involving *unbounded* sequences are instead not expressible even in Certora (see e.g. the property `always-wd-all-many` in the tokenless bank use case).

Listing 8 Completeness in SolCMC: multiple sequential calls.

```
contract Sequence {
  address last;
  function foo() { ... }
  function bar() {
    require(msg.sender!=last);
    last = msg.sender;
  }
  ...
}
```

```
function inv() public {
  bar();
  bar();
  assert(false);
}
```

The property specification language supported by Certora is quite powerful, allowing us to express most of the properties in our benchmark. Still, there are interesting classes of properties that cannot be expressed. This is the case e.g. for properties of the form "for all reachable states, some user can do something that eventually produces some desirable effect". For example, consider the `Deposit` contract in Listing 9. The contract allows anyone to withdraw any fraction of its balance through the method `pay`, unless the variable `frozen` is true. Now, `frozen` is controlled by the contract owner through the method `freeze`: hence, the owner at any point can freeze the contract balance, preventing anyone from withdrawing. A desirable property contracts, in general, is that the funds stored in the contract cannot be frozen forever, a property often referred to as *liquidity* [33, 5, 27]. A tentative formalization of the liquidity property is the CVL rule in Listing 9. The rule is satisfied if there exists some starting state such that, for all `sender` address and value `v`, `sender` can fire a transaction `pay(v)` that increases their balance by `v`. This however 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`). Note that an alternative formalization where the `satisfy` is replaced by an `assert` does not work too. In this case, we would require that, for all reachable states, a transaction `pay(v)` is never reverted, *for all* choices of the amount `v`. 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 case it would be easy to fix the property by requiring that the transaction is not reverted for all values `v` less than the contract balance and when `frozen` is false, in general for liquidity we would like to know if there *exist* parameters that make the desirable property true, which is not expressible in CVL.

Listing 9 Expressiveness of Certora: along all paths, eventually.

```
contract Deposit {
  address owner;
  bool frozen;
  constructor () payable {
    owner = msg.sender;
  }
  function freeze() public {
    require (msg.sender == owner);
    frozen = true;
  }
  function pay(uint v) public {
    require(!frozen);
    (bool succ,) = msg.sender.call{
        value: v}("");
    require(succ);
  }
}
```

```
// Certora rule specification

rule P(address sender, uint v) {
  // sender initial balance
  mathint b0 = bal(sender);
  env e;

  require e.msg.sender == sender;

  pay(e, v);

  // sender balance after pay(v)
  mathint b1 = bal(sender);

  // looking for a positive example
  satisfy(b1 == b0 + v);
}
```

Another category of properties that are not easily expressible, are those that reason about transfers of funds from the contract. For example, consider the property "the method `pay` calls the sender, transferring 10 wei from the contract to it", which is trivially satisfied by the contract in Listing 10 (see also `arbitrate-send` in the escrow use case). It might seem reasonable to express this property as a simple check on the balance of the contract and of the sender, as the invariant in Listing 10 (right). This however would not be correct: in fact, a contract that sends 10 wei to a middle man who forwards the sum to the sender would satisfy the invariant but violate the property. Indeed, we do not believe that properties of this kind are expressible in SolCMC. In CVL instead we can express a property very similar to the above, in which the required called contract is not the sender, but a specific account. This is possible with the use of hooks (this would require using hooks). It is unclear whether the exact property above is expressible in Certora.

As mentioned before, neither SolCMC nor Certora can, in general, express properties that are specific to EOAs. This derives from the fact that EOAs are not always discernible from contract accounts [28]. Other classes of properties that seem beyond the expressiveness boundaries of existing Solidity verification tools are those about game-theoretic interactions between users and adversaries, and fairness and probabilistic properties (see, respectively, the properties `rkey-no-wd` and `okey-rkey-private-wd` in the vault use case).

Scoring

We display in Table 2 the results obtained by running SolCMC and Certora on the verification tasks in our benchmark. As expected, Certora is able to express more properties than SolCMC; despite that, SolCMC's score is close to Certora's, which is penalized by its seemingly better behaviour with respect to soundness, the causes of which are exactly those discussed before in Listings 5 and 6. Regarding the two CHC solvers used by SolCMC, we note that Z3 has several UNK entries more than Eldarica due to its propensity to timeout. This explains the difference in score between Z3 and Eldarica: the latter performs worse because of its

Listing 10 Expressiveness of SolCMC: a wrong attempt of reasoning about received ETH.

```
contract Deposit {
  // pay 10 wei to the sender
  function pay() public {
    (bool succ,) = msg.sender.call
                   {value: 10}("");
    require(succ);
  }
}
```

```
function invariant() public {
  uint s0 = msg.sender.balance;
  uint c0 = address(this).balance;

  pay();

  uint s1 = msg.sender.balance;
  uint c1 = address(this).balance;
  assert(s1==s0+10 && c1==c0-10);
}
```

tendency to terminate with the wrong answer (getting -16 for the FP!), whereas Z3 just diverges (getting 0 for the UNK). Besides that, we do not note significant differences between these two CHC solvers. We highlight the absence of weak false positives (FP) in our results: SolCMC never claims that a property holds unless it is sure of it, while Certora only does so when the properties are encoded using `satisfy` statements (which have not been used in our current use cases).

Table 2 Scoring of SolCMC and Certora (323 verification tasks).

	ND	UNK	TP(!)	TN(!)	FP(!)	FN(!)	Score
Certora	60	0	97(97)	94(0)	7(7)	65(0)	176
SolCMC (Z3)	153	22	86(86)	49(48)	2(2)	11(9)	165
SolCMC (Eldarica)	153	5	88(88)	54(52)	7(7)	16(10)	90

5 Conclusions

We have discussed an ongoing collaborative effort towards the construction of a benchmark for comparing formal verification tools for Solidity. Once completed, the benchmark will serve as a valuable resource for developers, providing guidance in choosing the appropriate tool based on project requirements and contract complexity. In the meanwhile, we have given a first qualitative evaluation of SolCMC and Certora by discussing their soundness, completeness and expressiveness limitations based on our experience on developing the benchmark.

Constructing the ground truth was perhaps the most difficult task we encountered while developing the benchmark. In particular, while it is often easy to tell that an intentionally bugged contract violates a property, ensuring that a property is satisfied is error-prone. Indeed, Solidity has a few semantical quirks that make manual reasoning about contracts quite burdensome (reentrancy, in particular, may be very tricky). For this reason, it could make sense to simplify the construction of the ground truth by reducing the number of verification tasks for each use case: for instance, we could provide just a contract version that satisfies all the desirable properties, and just one version witnessing the violation of each property. While we are not able to certify the correctness of our ground truth beyond any reasonable doubt, we expect that the open-source nature of our benchmark can foster the collaboration with the community of experts. To consolidate our ground truth (at least, for negative properties), we plan to add, in future versions of the benchmark, references to actual contracts in the Ethereum testnet that violate the property. In this regard, we note that the counterexamples outputted by SolCMC and Certora when they find a violation are rarely informative about its causes. Incorporating other datasets of ground truth, like e.g. [3], into ours, would also allow us to extend the comparison between SolCMC and Certora, at least for the kind of specific contract weaknesses considered in these datasets.

In the choice of the use cases in our benchmark, one of our primary criteria was simplicity: this is clearly motivated by the need of manually constructing the ground truth. One meaningful criterion to extend the benchmark could be to find use cases that depend on corner cases of the semantics of Solidity, to test which approximations are made in these cases by the verification tools. Extending the benchmark with more complex use cases, including more convoluted ways to violate properties, and experimenting it on other verification tools beyond SolCMC and Certora are also viable directions for future work.

References

1 Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. Solcmc: Cav 2022 artifact. `https://github.com/leonardoalt/cav_2022_artifact/tree/main`, 2022.

2 Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. Solcmc: Solidity compiler's model checker. In *Computer Aided Verification (CAV)*, volume 13371 of *LNCS*, pages 325–338. Springer, 2022. `doi:10.1007/978-3-031-13185-1_16`.

3 Monika Di Angelo and Gernot Salzer. Consolidation of ground truth sets for weakness detection in smart contracts. In *Financial Cryptography Workshops*, volume 13953 of *LNCS*, pages 439–455. Springer, 2023. `doi:10.1007/978-3-031-48806-1_28`.

4 Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: Contract-Larva and open challenges beyond. In *Runtime Verification*, volume 11237 of *LNCS*, pages 113–137. Springer, 2018. `doi:10.1007/978-3-030-03769-7_8`.

5 Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. Verifying liquidity of recursive Bitcoin contracts. *Log. Methods Comput. Sci.*, 18(1), 2022. `doi:10.46298/LMCS-18(1:22)2022`.

6 Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Neil Immerman, Daniel Jackson, Alexander Nutz, Lior Oppenheim, Or Pistiner, Noam Rinetzky, Mooly Sagiv, Marcelo Taube, John A. Toman, and James R. Wilcox. WIP: Finding bugs automatically in smart contracts with parameterized invariants. `https://groups.csail.mit.edu/sdg/pubs/2020/sbc2020.pdf`, 2020.

7 Dirk Beyer. Competition on software verification and witness validation: SV-COMP 2023. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 13994 of *LNCS*, pages 495–522. Springer, 2023. `doi:10.1007/978-3-031-30820-8_29`.

8 N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation* (II), volume 9300 of *LNCS*, pages 24–51. Springer, 2015. `doi:10.1007/978-3-319-23534-9_2`.

9 Certora. The Certora Verification Language. `https://docs.certora.com/en/latest/docs/cvl/index.html`, 2022.

10 Certora. Formal verification of OpenZeppelin (may-june 2022). `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/certora/reports/2022-05.pdf`, 2022.

11 Certora. Certora prover documentation: invariants. `https://docs.certora.com/en/latest/docs/cvl/invariants.html`, 2023.

12 Certora. Certora prover documentation: Prover approximations. `https://docs.certora.com/en/latest/docs/prover/approx/index.html`, 2023.

13 Certora report: CallWrapper. `https://prover.certora.com/output/95211/e265c818d176463cbaa6f53e4d0fe394?anonymousKey=7d67af6ba7eedc4f099a133a04f4d36953f377dc`, 2024.

14 Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. Smart contract and DeFi security: Insights

from tool evaluations and practitioner surveys. In *ICSE*, pages 60:1–60:13, 2024. doi:10.1145/3597503.3623302.

15 Consensys. Write smart contract specifications using Scribble. https://consensys.io/diligence/scribble/, 2023.

16 E. De Angelis, F. Fioravanti, J. P. Gallagher, M. V. Hermenegildo, A. Pettorossi, and M. Proietti. Analysis and transformation of constrained Horn clauses for program verification. *Theory Pract. Log. Program.*, 22(6):974–1042, 2022. doi:10.1017/S1471068421000211.

17 L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

18 Bruno Dias, Naghmeh Ivaki, and Nuno Laranjeiro. An empirical evaluation of the effectiveness of smart contract verification tools. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 17–26, 2021. doi:10.1109/PRDC53464.2021.00013.

19 Enterprise Ethereum Alliance. Smart contract weakness classification (SWC). https://swcregistry.io/, 2020.

20 Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE/ACM, 2019. doi:10.1109/WETSEB.2019.00008.

21 Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. A survey on formal verification for Solidity smart contracts. In *Australasian Computer Science Week Multiconference*, pages 3:1–3:10. ACM, 2021. doi:10.1145/3437378.3437879.

22 Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–7, 2018. doi:10.23919/FMCAD.2018.8603013.

23 Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. Security threat mitigation for smart contracts: A comprehensive survey. *ACM Comput. Surv.*, 55(14s):326:1–326:37, 2023. doi:10.1145/3593293.

24 Daniel Jackson, Chandrakana Nandi, and Mooly Sagiv. Certora technology white paper. https://docs.certora.com/en/latest/docs/whitepaper/index.html, 2022.

25 Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based Model Checking for Recursive Programs. *Formal Methods in System Design*, pages 175–225, 2016. doi:10.1007/s10703-016-0249-4.

26 Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062, 2022. doi:10.1109/ACCESS.2022.3169902.

27 Cosimo Laneve. Liquidity analysis in resource-aware programming. *J. Log. Algebraic Methods Program.*, 135:100889, 2023. doi:10.1016/J.JLAMP.2023.100889.

28 OpenZeppelin. Utilities / address. https://docs.openzeppelin.com/contracts/4.x/api/utils#Address, 2024.

29 Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. VerX: Safety verification of smart contracts. In *IEEE Symposium on Security and Privacy*, pages 1661–1677. IEEE, 2020. doi:10.1109/SP40000.2020.00024.

30 Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and provably sound static analysis of Ethereum smart contracts. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 621–640. ACM, 2020. doi:10.1145/3372297.3417250.

31 The Solidity Authors. SMTChecker and formal verification: contract balance. https://docs.soliditylang.org/en/v0.8.24/smtchecker.html#contract-balance, 2023.

32 The Solidity Authors. SMTChecker and formal verification: untrusted calls and reentrancy. https://docs.soliditylang.org/en/latest/smtchecker.html#trusted-external-calls, 2023.

33 Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *ACM*

SIGSAC Conference on Computer and Communications Security (CCS), pages 67–82. ACM, 2018. `doi:10.1145/3243734.3243780`.

34 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz, and Arie Gurfinkel. Verifying Solidity smart contracts via communication abstraction in SmartACE. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 13182 of *LNCS*, pages 425–449. Springer, 2022. `doi:10.1007/978-3-030-94583-1_21`.

35 Scott Wesley and Valentin Wüstholz. Verify OpenZeppelin. `https://github.com/contract-ace/verify-openzeppelin`, 2022.