SolCMC: Solidity Compiler's Model Checker*

Leonardo Alt[1], Martin Blicha[2,3],
Antti E. J. Hyvärinen[2], and Natasha Sharygina[2]

[1] Ethereum Foundation
[2] Università della Svizzera italiana, Lugano, Switzerland
[3] Charles University, Prague, Czech Republic
leo@ethereum.org
{martin.blicha,antti.hyvaerinen,natasha.sharygina}@usi.ch

Abstract. Formally verifying smart contracts is important due to their immutable nature, usual open source licenses, and high financial incentives for exploits. Since 2019 the Ethereum Foundation's Solidity compiler ships with a model checker. The checker, called SolCMC, has two different reasoning engines and tracks closely the development of the Solidity language. We describe SolCMC's architecture and use from the perspective of developers of both smart contracts and tools for software verification, and show how to analyze nontrivial properties of real life contracts in a fully automated manner.

Keywords: Ethereum · Solidity · Symbolic Model Checking · Constrained Horn Clauses · Satisfiability Modulo Theories

1 Introduction

The Ethereum Foundation's compiler for Ethereum platform's most used language Solidity had almost 4 million downloads (3,957,195) over the last 60 days (at the time of submission). Since 2019, this compiler ships with a robust, builtin, easy-to-use, symbolic model checker SolCMC [16], formerly called SMTChecker. SolCMC models a *smart contract*, that is, a program for the Ethereum platform, and its properties as a system of constrained Horn clauses (CHCs) amenable to IC3-style model checking [34]. Since its deployment, SolCMC has increasingly served a dual purpose. On the one hand, smart contract programmers have through it a very visible and easy access to formal verification techniques. On the other hand, perhaps more subtly but no less importantly, the tool serves as a sounding board for developers of Horn solvers. Currently the system interfaces with Spacer [31] and Eldarica [30], making the related techniques available to a large user base. We expect to integrate in SolCMC many other techniques through a similar mechanism. For instance, the tool has a bounded model checking engine for finding bugs by issuing SMT queries to solvers such as z3 [35] and cvc5 [23].

* This work was partially supported by Swiss National Science Foundation grant 200021_185031 and by Czech Science Foundation grant 20-07487S.

Smart contracts running on the Ethereum platform hold and control billions of dollars through their immutable logic, and therefore bugs can lead to massive losses. There are many recent sophisticated tools that increase the security of the Ethereum contract ecosystem by detecting smart contract bugs before they are deployed. However, new and emerging applications from the diverse user base are driving Solidity development at a fast pace and it is difficult to keep tools synchronized with the language. We believe that in the long run, the best way to ensure that a model checker for Solidity is sustainable is by integrating it directly into the compiler distribution, or the *main repository* of the related language tools, as we have done for SolCMC.

The direct integration of the model checker into the compiler has two main advantages. Firstly, we can model precisely and robustly features that are somewhat specific to Solidity and its applications, such as modeling reentrancy callbacks, and the handling of global storage. This makes the model checker capable of synthesizing new contracts that serve as counterexamples for correctness, and computing inductive invariants for the cases where properties hold. Secondly, the short pipeline between the source code and the model allows the presentation of both counterexamples and invariants as compiler warnings and annotations using a vocabulary that is meaningful for the developer.

The goal of SolCMC is to verify properties of programs with minimal user input. Our system supports writing properties as `assert` statements and can in addition automatically check other structural properties such as popping from an empty array and array accesses that are out of bounds, and the lack of underflows, overflows, divisions by zero, and transfers with insufficient balance. Moreover, common Solidity vulnerabilities such as reentrancy mutability and selfdestruct reachability can be verified using test harnesses that make the assertion-based approach more expressive. Thus, the expressiveness of SolCMC allows efficiently obtaining meaningful results for real life contracts in a way that is in practice fully automated. To demonstrate this we analyze the Beacon Chain Deposit Contract that is the base for Ethereum's proof of stake consensus layer, and the OpenZeppelin implementation of the ERC777 token standard.

An extended version of this tool paper including appendices showing detailed experimental results and other analysis is available online in the accompanying artifact, at `https://doi.org/10.5281/zenodo.6512173`.

Related work. Proving correctness and finding bugs in smart contracts is useful in different abstraction targets. The technical details of how smart contracts are encoded by SolCMC are presented in [34]. In this tool paper the emphasis is on orthogonal topics: the usage of options, generation of counterexamples in Solidity-like syntax, interfacing with different Horn solvers, and how contract invariants can be obtained. We also demonstrate the tool's capabilities by analysing two important and complex contracts: the Deposit contract and ERC777.

Most current tools either analyse the Solidity high level language, similar to SolCMC, or work directly on Ethereum Virtual Machine (EVM) bytecode.

The tools Solc-verify [28] and Verisol [38] verify Solidity properties in an automated way allowing models with unbounded number of transactions by

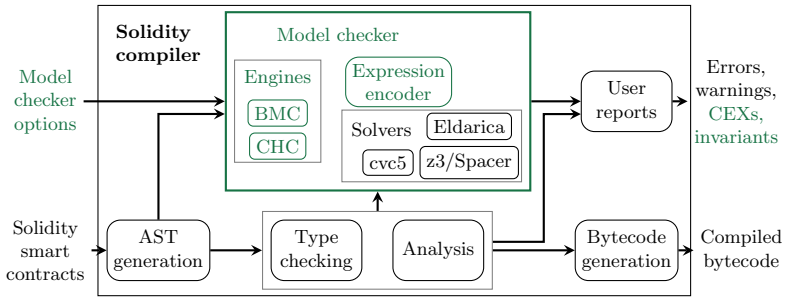

Fig. 1. The Solidity compiler stack with the integrated model checker (in green)

translating Solidity to Boogie [33]. This gives the tools an advantage in engineering resources, but, compared to SolCMC's direct encoding as CHCs, makes producing counterexamples to the user more difficult. Neither of the two tools produce counterexamples or inductive invariants, and the most recent language versions are not supported. SmartACE [39] relies on translation from Solidity to LLVM-IR. This allows for employing multiple analysis tools, but unlike in SolCMC where we use a direct encoding and tighet solver integration, the tools are mostly used as black boxes. EThor [37] also uses Horn clauses but it encodes EVM bytecode, and focuses on specific properties such as reentrancy. The Certora [24] tool relies on invariants to verify EVM bytecode. It is a commercial tool used for smart contract audits and is not publicly available. The $\mathbb{K}$ framework [10] is an assisted theorem prover that provides EVM semantics [29] to analyze EVM bytecode. It is generally able to prove more statements than automated tools, but requires considerable user interaction. HEVM [22] is an implementation of EVM in Haskell that also has a symbolic executor for EVM bytecode. It can prove functional properties but, unlike SolCMC, does not support inductive properties over multiple transactions and loops. HEVM and Echidna [4] also provide fuzzing techniques that help determining whether a candidate assertion is a contract invariant. Slither [14] is a powerful static analyzer that does not provide formal guarantees but can detect many vulnerabilities and dangerous patterns. Act [1] is a declarative specification language for smart contracts that supports three backends: bytecode verification via HEVM, SMT theorems for contract invariants, and a Coq backend that exports Coq definitions of contract state transitions. Finally, the Scribble specification language [13] allows annotating Solidity code and can generate runtime checks for given properties.

2 Solidity Model Checking

The high level overview of the compilation process is depicted in Fig. 1, with the model checker module emphasized. When enabled, Solidity model checking becomes another pass over the source code in the normal compilation process that starts after parsing and Abstract Syntax Tree (AST) generation. If there

Table 1. SolCMC verification targets

Arithmetic	Structural
arithmetic underflow/overflow, division by zero, insufficient transfer balance	assertions, popping empty array, out of bounds index access

were no errors, the compiler produces the optimized bytecode together with any warnings, such as counterexamples found by the model checker.

This paper concentrates on SolCMC's unbounded model checker based on CHCs. The tool also has a BMC engine that generates SMT queries and links against cvc5 [23] and z3 [35].

2.1 The CHC Verification Engine

SolCMC encodes a smart contract as a system of constrained Horn clauses, based on [34]. The checker supports loops, multi-transaction computation paths, contract invariants, tracking contract balances throughout their lifetimes, and precise multi-contract calls. If the analyzed contract calls external functions unsafely, the model checker also synthesizes malicious external actors and represents them as reentrant calls.

The Horn queries are dispatched to a Horn solver. The encoding requires the solver to support nonlinear Horn clauses and at least the SMT theories for Linear Integer Arithmetic (LIA), Arrays, and the tuples subset of Algebraic Datatypes (ADT). Furthermore, nonlinear integer arithmetic and bitwise operations, if present, are encoded in the respective theories NIA and BV. To the best of our knowledge only Spacer [31] and Eldarica [30] satisfy those requirements. SolCMC has a tight integration with Spacer via its C++ API, whereas Eldarica is integrated using the compiler's SMT callback [21], and is currently accessible via solc-js [15], the JavaScript wrapper of the compiler's WebAssembly binary.

The model checker generates verification targets automatically for the conditions listed in Table 1. In particular a smart contract developer can combine *assertions* with *test harnesses* (see, e.g., Sec. 4) to specify complex behavior. The Solidity language has the statements `require` and `assert`, which SolCMC uses to capture developer intent: Conditions inside `require` statements are considered assumptions, and `assert` statements should be true for every execution. The model checker then treats every `assert` as a verification target and attempts to either prove it by finding an invariant, or give a counterexample for its correctness.

2.2 Horn Encoding

SolCMC's CHC encoding is based on the imperative encoding of [25], and is presented in detail in [34]. Horn logic is a popular formalism for expressing reachability problems. It is equivalent to the *existential positive fix-point logic* [26], and provides a convenient syntax for the use of existentially quantified predicates that in our encoding represent reachable states and effects of transactions. The

Solidity AST first gets transformed into a Control Flow Graph (CFG). CFG nodes have corresponding CHC predicates, and edges are encoded as Horn rules with constraints created from the Single Static Assignment (SSA) form of the statements and expressions of the CFG block. Below we give an overview of the encoding that highlights the critical parts.

The encoding consists of three types of predicates that represent reachable states or possible transitions: *function bodies* (B_f) and *summaries* (S_f) represent the effect of function calls to f; *interfaces* (I_C) represent the states a contract C can reach after initialization and each transaction; and *nondeterministic interfaces* (N_C) encode the effects the environment may have to a contract C. We use the following variables in the encoding: e, an integer error flag. Each verification target has a positive unique error id; 0 is reserved for no errors. a, the contract address. **abi**, a tuple of Solidity's ABI functions. **cr**, a tuple of Solidity's cryptographic functions: `keccak256`, `sha256`, `ripemd160`, and `ecrecover`. Both **abi** and **cr** are constant in the encoding. They are passed through the rules to ensure consistency everywhere. **tx**, a tuple of the transaction data, e.g., message sender, data, block number, etc. **st**, the blockchain state, a tuple containing the balances and storage for every contract. Balances are represented by an array mapping addresses to their balances. Each contract has a storage tuple that contains the state variables of that contract. **x**, the program state, input, output and local variables in the scope of that node. When necessary, we refer to the state variables as **s**. For **x** and **st** we use primes to denote the effect of rules on these variables.

Function bodies encode constructors, deployment procedures, and function summaries. For example, the contract **contract** `Acc` { **uint8** `x = 0;` **function** `acc(uint8 y)` **external** { `x += y;` } } gets encoded into the rules

$$e = 0 \land \mathbf{st} = \mathbf{st}' \land x = x' \land y = y' \land 0 \leq y' \leq 255 \land 0 \leq x' \leq 255$$
$$\implies \mathbf{B}_{\mathtt{acc}}(e, a, \mathbf{abi}, \mathbf{cr}, \mathbf{tx}, \mathbf{st}, x, y, \mathbf{st}', x', y')$$

stating that the function can always be called, its execution starts with no error, the initial variables have the current values, and the program variables' types are constrained;

$$\mathbf{B}_{\mathtt{acc}}(e, a, \mathbf{abi}, \mathbf{cr}, \mathbf{tx}, \mathbf{st}, x, y, \mathbf{st}', x', y') \land (x' + y' > 255)$$
$$\implies \mathbf{S}_{\mathtt{acc}}(1, a, \mathbf{abi}, \mathbf{cr}, \mathbf{tx}, \mathbf{st}, x, y, \mathbf{st}', x', y')$$

stating that an overflow in summation is an error, with label 1; and

$$\mathbf{B}_{\mathtt{acc}}(e, a, \mathbf{abi}, \mathbf{cr}, \mathbf{tx}, \mathbf{st}, x, y, \mathbf{st}', x', y') \land (x'' = x' + y) \land (x'' \leq 255)$$
$$\implies \mathbf{S}_{\mathtt{acc}}(e, a, \mathbf{abi}, \mathbf{cr}, \mathbf{tx}, \mathbf{st}, x, y, \mathbf{st}', x'', y),$$

which exits the function with no error and updates the contract state variable x.

Interface rules. The *interface CFG node* is an artificial node that represents the idle state of a contract. This node is crucial to the encoding when modelling transactions, querying error flags, committing state changes, generating

counterexamples, and translating inductive contract invariants. It is reachable at the beginning and end of every transaction. Transactions may revert due to invalid inputs or program logic, in which case all state changes are rolled back. The interface node may contain state changes if the transaction did not revert. Each contract `C` has a predicate $\mathbf{I}_{\mathtt{C}}$, whose parameters are a, $\mathbf{abi}, \mathbf{cr}$, $\mathbf{st}$ and the state variables $\mathbf{s}$ of the contract. The rules only change $e, \mathbf{st}$ and $\mathbf{s}$, and for better readability we use ellipsis $(\ldots)$ to denote the unchanged parameters. One rule is added per contract linking the deployment procedure to the interface: $\mathbf{D}_{\mathtt{C}}(\ldots) \implies \mathbf{I}_{\mathtt{C}}(\ldots)$. For each external function `f` in the contract `C`, we add the *query rule* and the *update rule*

$$\mathbf{I}_{\mathtt{C}}(\ldots, \mathbf{st}, \mathbf{s}, \ldots) \wedge \mathbf{S}_{\mathtt{f}}(e, \ldots, \mathbf{st}, \mathbf{s}, \ldots, \mathbf{st}', \mathbf{s}', \ldots) \wedge e > 0 \implies \mathbf{Err}_{\mathtt{f}}(e)$$

$$\mathbf{I}_{\mathtt{C}}(\ldots, \mathbf{st}, \mathbf{s}, \ldots) \wedge \mathbf{S}_{\mathtt{f}}(e, \ldots, \mathbf{st}, \mathbf{s}, \ldots, \mathbf{st}', \mathbf{s}', \ldots) \wedge e = 0 \implies \mathbf{I}_{\mathtt{C}}(\ldots, \mathbf{st}', \mathbf{s}', \ldots).$$

The Horn query given to the solver then asks whether $\mathbf{Err}_{\mathtt{f}}(e)$ is reachable, for each error label e. In this modelling, if the property is safe, inductive invariants chosen by the solver as an interpretation for the predicates $\mathbf{I}_{\mathtt{C}}$ represent the invariants for contracts `C`.

Nondeterministic interface rules. The *nondeterministic interface CFG node* is an artificial node that represents every possible behavior of the contract from an external point of view, in an unbounded number of transactions. This node is essential to model calls that the contract makes to external unknown contracts, as well as reentrancy if present. The predicate that represents this node has the same parameters as the interface predicate, but with the error flag and an extra set of program variables and blockchain state, in order to model possible errors and state changes. For every contract `C` the encoding adds the base case rule $\mathbf{N}_{\mathtt{C}}(0, \ldots, \mathbf{st}, \mathbf{s}, \mathbf{st}, \mathbf{s})$ which performs no state changes. Then for every external function `f` in the contract the encoding adds the inductive rule $\mathbf{N}(0, \ldots, \mathbf{st}, \mathbf{s}, \mathbf{st}', \mathbf{s}') \wedge \mathbf{S}_{\mathtt{f}}(e, \ldots, \mathbf{st}', \mathbf{s}', \mathbf{st}'', \mathbf{s}'') \implies \mathbf{N}(e, \ldots, \mathbf{st}, \mathbf{s}, \mathbf{st}'', \mathbf{s}'')$. These rules allow us to encode an external call to unknown code using a single constraint $\mathbf{N}(e, \ldots, \mathbf{st}, \mathbf{s}, \mathbf{st}', \mathbf{s}')$ which models every reachable state change in the contract, in any unbounded number of transactions. If a property is unsafe, these rules force the solver to synthesize the behavior of the adversarial contract. Otherwise, the interpretation of such predicate gives us inductive reentrancy properties that are true for every external call to unknown code in the contract.

3 User Features

As SolCMC is shipped inside the Solidity compiler, it is available for the users whenever and wherever they interact with the compiler. There are currently three major ways the compiler is used: 1. Interfacing with the WebAssembly release through official JavaScript bindings; 2. Interfacing with a binary release on command line; 3. Using web based IDEs, such as Remix [12]. Option 3 is the most accessible, but currently allows only limited configuration of the model checker

through `pragma` statements in source code. Options 1 and 2 both allow extensive configuration, but in addition 1 enables the *SMT callback* feature needed, e.g., for Eldarica. In 2 the options can be provided either on the command line or in JSON [19], whereas 1 accepts only JSON using the JavaScript wrapper [15].

In 1 and 2 several parameters are available to the user for better control when trying to prove complex properties. We list here some examples, using the command line options (without the leading `--`). The JSON descriptions are named similarly. The model checking engine — BMC, CHC or both — is selected with the option `model-checker-engine`. Individual verification targets can be chosen with `model-checker-targets`, and a per-target verification timeout (in ms) can be set with `model-checker-timeout`. By default, all unproved verification targets are given in a single message after execution. More details are available by specifying `model-checker-show-unproved`. Option `model-checker-contracts` provides a way to choose the contracts to verify. Typically the user specifies only the contract they wish to deploy. Inherited and library contracts are included automatically, avoiding verifying every contract as the main one. Some options affect the encoding. For example, integer division and modulo operations can be encoded with the SMT function symbols `div` and `mod` or by SolCMC's own encoding using linear arithmetic and slack variables. Depending on the backend one is often preferred to the other. The default is the latter, the former is set by `model-checker-div-mod-no-slacks`.

Solidity provides the NatSpec [20] format for rich documentation. An annotation `/// @custom:smtchecker abstract-function-nondet` instructs SolCMC to abstract a function nondeterministically. Abstracting functions as an Uninterpreted Function [32] is under development.

Counterexamples and inductive invariants. When a verification target is disproved, SolCMC provides a readable counterexample describing how to reach the bug. In addition to the line of code where the verification target is breached, the counterexample states the trace of transactions and function calls leading to the failure along with concrete values substituted for the arguments, and the values of the state variables at the point of failure. When necessary, the trace includes also synthesized reentrant calls that trigger the failure.

Similarly, when SolCMC proves a verification target, the user may instrument the checker to provide safe inductive invariants. The invariants can, for instance, be used as an additional proof that the verification target holds. Technically the invariants are interpretations for the predicates in the CHC system and are presented in a human readable Solidity-like syntax. Similarly to counterexamples, the invariants are given also for predicates guaranteeing correctness under reentrancy. The extended version of this paper contains a short example illustrating the counterexamples and inductive invariants. It also presents more complex examples of both features, which were obtained from our experiments with the ERC777 token standard.

4 Real World Experiments

In this section we analyse two real world smart contract systems using SolCMC. Both contracts are massively important and highly nontrivial for automated tools due to their use of complex features, loops, and the need to produce nontrivial inductive invariants. While only the main results are stated in this section, we want to emphasize that the results were achieved after an extensive, albeit mechanical, experimentation on the two backend solvers (Spacer and Eldarica) and a range of parameters. To us the fact that they were successfully analysed using an automatic method is a strong proof of the combined power of our encoding approach and the backend solvers.

4.1 CHC solver options

The options we pass to the underlying CHC solvers Spacer and Eldarica may make the difference between a quick solving and divergences. For Spacer, we use the options `rewriter.pull_cheap_ite=true` which pulls if-then-else terms to the top level when it can be done cheaply, `fp.spacer.q3.use_qgen=true` which enables the quantified lemma generalizer, `fp.spacer.mbqi=false` which disables the model-based quantifier instantiation, and `fp.spacer.ground_pobs=false` which grounds proof obligations using values from a model. For Eldarica, we have found the adjustment of the predicate abstraction to be useful: `-abstract:off` disables abstraction, `-abstract:term` uses term abstraction, and `-abstract:oct` uses the octal abstraction.

4.2 Deposit Contract

The Ethereum 2.0 (Eth2) [9] Deposit Contract [3,2] is a smart contract that runs on Ethereum 1.0 collecting deposits from accounts that wish to be validators on Eth2. By the time of submission of this paper more than 9,100,194 ETH were held by the Deposit Contract, the equivalent of tens of billions USD in recent rates. Besides the financial incentive, this contract's functionality is essential to the progress of the protocol. The contract was formally verified before deployment [36] and further proved safe [27] with considerable amount of manual work. Despite having relatively few lines of code (less than 200), the contract remains a challenge for automated tools, because of its use of many complex constructs at the same time, such as ABI encoding functions, loops, dynamic types, and hash functions.

As part of the logic of the `deposit` function, a new entry is created in a Merkle tree for the caller. The contract asserts that such an entry can always be found, expressed as an `assert(false)` in a program location reachable only if such an entry is not found (line 162 in [2]). Using SolCMC this problem can be encoded into a 1.4MB Horn logic file containing 127 rules, which uses the SMT theories for Arrays, ADTs, NIA, and BV. After a syntactical change, Eldarica can show the property safe automatically in 22.4 seconds, while Spacer times out after 1 hour (see the extended version for details). The change is necessary to avoid bit-vector reasoning and consists of replacing the test `if ((size & 1)`

`== 1)` with a semantically equivalent form `if ((size % 2) == 1)` on lines 88 and 153 in [2].

4.3 ERC777

ERC777 [6] is a token standard that offers extra features compared to the ERC20 [5] standard. Besides the usual transfer and allowance features, ERC777 mainly adds account operators and transfer hooks which allow smart contracts to react to sending and receiving tokens. This is similar to the native feature of reacting to receiving Ether. In this experiment we analyze the OpenZeppelin implementation [11] of ERC777. This contract is an interesting benchmark for automated tools not only because of its importance, but also because it is a rather large smart contract system with 1200 lines of Solidity code, in 8 files, and it uses complex high level constructs such as assembly blocks, heavy inheritance, strings, arrays, nested mappings, loops, hash functions, and makes external calls to unknown code. The implementation follows the specification precisely, and does not guarantee a basic safety property related to tokens: *The total supply of tokens should not change during a transfer.*

Compared to the usual ERC20 token `transfer` that simply decreases and increases the balances of the two accounts involved in the transfer, the ERC777 `transfer` function may call unknown contracts to notify them that they are sending/receiving tokens. The logic in these external contracts is completely arbitrary and unknown to the token contract. For example, they could make a reentrant call to one of the nine ERC777 token mutable functions from its external interface.

Since the analyzed ERC777 implementation is agnostic on how tokens are initially allocated, no tokens are distributed in the base implementation at deployment. Therefore, to study the property, we write the following test harness [7] that uses the ERC777 token implemented by OpenZeppelin.

```
import "<path>/ERC777.sol";

contract Harness is ERC777 {
 constructor(
   address[] memory defOps_,
   uint amt_
 ) ERC777("ERC777", "E7", defOps_){
   _mint(msg.sender, amt_, "", "");
 }
```

```
 function transfer(address r, uint a)
   public override returns (bool) {
   uint prev = totalSupply();
   bool res = ERC777.transfer(r, a);
   uint post = totalSupply();
   assert(prev == post);
   return res;
 }
}
```

First, we allocate `amt_` tokens to the creator of the contract, in order to have tokens circulating. Then, we override the `transfer` function, where our `transfer` function simply wraps the one from the `ERC777` contract, asserting that the property we want to verify is true after the original transfer.

The resulting Horn encoding is 15 MB large and contains 545 rules. The property can be shown unsafe by Eldarica in all its configurations, the quickest taking slightly less than 3 minutes, including generating the counterexample (see the extended version for details). All Spacer's configurations time out after 1 hour. Since the property is unsafe, SolCMC also provides the full transaction

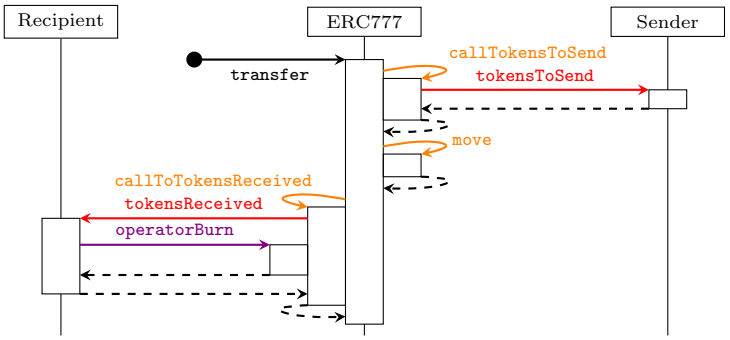

Fig. 2. Transaction trace that violates the safety property in `transfer`

trace required to reach the assertion failure. The transaction trace is visualized in Fig 2 in the form of a sequence diagram, where solid arrows represent function calls and dashed arrows represent the return of the execution control. The full output of the tool can be found in the extended version.

The diagram shows the transaction trace from the call to `transfer` of ERC777 (after our wrapper contract has been created and its `transfer` was called). `transfer` performs 3 internal function calls (in orange): 1) `_callTokensToSend` performs the external call to notify the sender; 2) `_move` moves the tokens from the sender to the recipient; 3) `_callTokensReceived` notifies the recipient. The external calls to unknown code are shown in red. The transaction trace also contains the synthesized behaviour for the recipient (in purple). It is a reentrant call to `operatorBurn` in the ERC777 token contract itself, where some of the tokens of the recipient contract will be burned. At the end of the execution of `transfer`, the assertion is no longer true. The total supply of tokens after the call is not the same as the total supply before the call, as some tokens were burned during the transaction.

Given the number of mutable external functions of ERC777 and their complexity, we consider the discovery of the counterexample to be quite an achievement. We ascribe the success to the combined power of the CHC encoding and the Horn solver.

One way to guarantee that our property holds is to disallow reentrancy throughout the contract using a mutex. After changing the `ERC777` library [8], we ran the tool again on our test harness. Spacer timed out, but Eldarica was able to prove that the restricted system is safe in all its configurations, the fastest one finishing in 26.2 seconds, including the generation of the inductive invariants for every predicate. SolCMC now reports back the reentrancy property `<errorCode> = 0` given as part of the proof (the property is presented here in a simplified manner, see the extended version for details). The inductive property states that no external call performed by the analyzed contract can lead to an error. This shows that the reentrant path can no longer be taken.

4.4 Discussion

While producing the above analysis of the real life contracts, we experimented with two backend solvers Spacer and Eldarica, and a range of parameters for them. This phase (documented in the extended version of this paper) was critical in producing the results, because Eldarica and Spacer excel in different domains and parameter selection has a major impact on both verification success and run time. In both cases above Eldarica performed clearly better than Spacer. This seems to be because Eldarica handles abstract data types better than Spacer. This conclusion is backed by experimental evidence. We ran SolCMC using both Spacer and Eldarica on the SolCMC regression test suite consisting of 1098 solidity files [17] and 3688 Horn queries [18]. The experiment shows that while the solvers give overall similar results, in two categories that make heavy use of ADTs, Eldarica is consistently able to solve more benchmarks than Spacer. For lack of space, the detailed analysis is given in the extended version.

Our encoding uses tuples to encode data that makes sense to be bundled together. Moreover, arrays of tuples are used to emulate Uninterpreted Functions (UFs) to abstract injective functions such as cryptographic primitives. This is necessary due to UFs not being syntactically allowed in predicates of Horn instances. While this increases the complexity of the problem, we have chosen this path to reduce encoding complexity, considering that a pre processing step may be available in the future to flatten such tuples and arrays.

5 Conclusions and Future Work

This paper presents the model checker SolCMC that ships with the Ethereum Foundation's compiler for the Solidity language. We believe that the automated and usable tool has the potential to link a high volume of Solidity developers with the community working on tools for formal verification. The tool is stable, and, having been integrated into the compiler, tracks closely the quickly developing language.

We advocate for a *direct encoding approach* where the same AST gets compiled both into EVM bytecode and into a verification model in SMT-LIB2 or the format used in the CHC competition. In our experience this makes it more natural to model features specific to Solidity and Ethereum smart contracts as well as for generating usable counterexamples and inductive invariants in comparison to producing first a language-agnostic intermediate verification representation that is then processed for reasoning engines.

We argue for the ease of use of the tool by showing nontrivial properties of real life contracts. The experiments also identify interesting future development opportunities in the current CHC formalism. We show how the formalism's limitations can be worked around using abstract data types, and discuss their impact on tool efficiency.

References

1. Act 0.1 released. https://fv.ethereum.org/2021/08/31/act-0.1/, accessed: 2022-01-21
2. Deposit Contract deployed on Ethereum mainnet. https://etherscan.io/address/0x00000000219ab540356cbb839cbe05303d7705fa#code, accessed: 2022-01-21
3. Deposit Contract specification and source code. https://github.com/ethereum/consensus-specs/blob/master/specs/phase0/deposit-contract.md, accessed: 2022-01-21
4. Echidna source code and documentation. https://github.com/crytic/echidna/, accessed: 2022-01-21
5. ERC20 documentation. https://eips.ethereum.org/EIPS/eip-20, accessed: 2022-01-21
6. ERC777 documentation. https://eips.ethereum.org/EIPS/eip-777, accessed: 2022-01-21
7. ERC777 Property Wrapper contract. https://github.com/leonardoalt/openzeppelin-contracts/blob/master/contracts/token/ERC777/ERC777PropertyUnsafe.sol, accessed: 2022-01-21
8. ERC777 using a mutex to prevent reentrancy.. https://github.com/leonardoalt/openzeppelin-contracts/blob/master/contracts/token/ERC777/ERC777Mutex.sol, accessed: 2022-01-21
9. Ethereum Consensus Layer specification. https://github.com/ethereum/consensus-specs, accessed: 2022-01-21
10. K framework. https://kframework.org, accessed: 2022-01-21
11. Openzeppelin Solidity implementation of the ERC777 standard. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC777/ERC777.sol, accessed: 2022-01-21
12. Remix IDE. https://remix.ethereum.org, acessed: 2022-01-13
13. Scribble documentation. https://docs.scribble.codes/language/introduction, accessed: 2022-01-21
14. Slither source code and documentation. https://github.com/crytic/slither, accessed: 2022-01-21
15. solc-js documentation. https://github.com/ethereum/solc-js, accessed: 2022-01-21
16. SolCMC documentation. https://docs.soliditylang.org/en/latest/smtchecker.html, accessed: 2022-01-21
17. SolCMC tests. https://github.com/ethereum/solidity/tree/develop/test/libsolidity/smtCheckerTests, accessed: 2022-01-21
18. SolCMC tests' Horn queries. https://github.com/leonardoalt/chc_benchmarks_solidity, accessed: 2022-01-21
19. Solidity compiler input and output JSON description. https://docs.soliditylang.org/en/v0.8.11/using-the-compiler.html#compiler-input-and-output-json-description, accessed: 2022-01-21
20. Solidity NatSpec Format. https://docs.soliditylang.org/en/v0.8.11/natspec-format.html, accessed: 2022-01-21
21. Solidity's SMT callback documentation. https://github.com/ethereum/solc-js#example-usage-with-smtsolver-callback, accessed: 2022-01-21
22. Symbolic execution for hevm. https://fv.ethereum.org/2020/07/28/symbolic-hevm-release/, accessed: 2022-01-21

23. 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: Proc. TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer (2022)
24. Bernardi, T.P., Dor, N., Fedotov, A.N., Grossman, S., Immerman, N., Jackson, D., Nutz, A., Oppenheim, L., Pistiner, O., Rinetzky, N., Sagiv, M., Taube, M., Toman, J., Wilcox, J.R.: WIP: Finding bugs automatically in smart contracts with parameterized invariants (2020), `https://www.certora.com/pubs/sbc2020.pdf`, accessed: 2022-01-21
25. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. LNCS, vol. 9300, pp. 24–51. Springer (2015)
26. Blass, A., Gurevich, Y.: Existential fixed-point logic. In: Computation Theory and Logic, In Memory of Dieter Rödding. LNCS, vol. 270, pp. 20–36. Springer (1987)
27. Cassez, F.: Verification of the incremental Merkle tree algorithm with Dafny. In: Proc. FM 2021. LNCS, vol. 13047, pp. 445–462. Springer (2021)
28. Hajdu, Á., Jovanović, D.: solc-verify: A modular verifier for solidity smart contracts. In: Proc. VSTTE 2019. LNCS, vol. 12031, pp. 161–179. Springer (2019)
29. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In: Proc. CSF 2018. pp. 204–217. IEEE Computer Society (2018)
30. Hojjat, H., Rümmer, P.: The ELDARICA Horn solver. In: Proc. FMCAD 2018. pp. 1–7. IEEE (2018)
31. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (2016)
32. Kroening, D., Strichman, O.: Equality logic and uninterpreted functions. In: Decision Procedures: An Algorithmic Point of View, pp. 77–95. Springer, Berlin, Heidelberg (2016)
33. Leino, K.R.M.: This is Boogie 2 (june 2008), `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`
34. Marescotti, M., Otoni, R., Alt, L., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Accurate smart contract verification through direct modelling. In: Proc. ISoLA 2020. vol. 12478, pp. 178–194. Springer (2020)
35. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)
36. Park, D., Zhang, Y., Rosu, G.: End-to-end formal verification of ethereum 2.0 deposit smart contract. In: Proc. CAV 2020. LNCS, vol. 12224, pp. 151–164. Springer (2020)
37. Schneidewind, C., Grishchenko, I., Scherer, M., Maffei, M.: EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts, p. 621–640. ACM (2020)
38. Wang, Y., Lahiri, S., Chen, S., Pan, R., Dillig, I., Born, C., Naseer, I., Ferles, K.: Formal verification of workflow policies for smart contracts in Azure blockchain. In: Proc. VSTTE 2019. LNCS, vol. 12031, pp. 87–106. Springer (2019)
39. Wesley, S., Christakis, M., Navas, J.A., Trefler, R., Wüstholz, V., Gurfinkel, A.: Verifying solidity smart contracts via communication abstraction in SmartACE. In: Proc. VMCAI 2022. LNCS, vol. 13182, pp. 425–449. Springer (2022)