A Solicitous Approach to Smart Contract Verification

RODRIGO OTONI, Università della Svizzera italiana, Switzerland
MATTEO MARESCOTTI[*], Meta, UK
LEONARDO ALT, Ethereum Foundation, Switzerland
PATRICK EUGSTER, Università della Svizzera italiana, Switzerland and Purdue University, USA
ANTTI E. J. HYVÄRINEN, Università della Svizzera italiana, Switzerland
NATASHA SHARYGINA, Università della Svizzera italiana, Switzerland

Smart contracts are tempting targets of attacks, since they often hold and manipulate significant financial assets, are immutable after deployment, and have publicly available source code, with assets estimated in the order of millions of US Dollars being lost in the past due to vulnerabilities. Formal verification is thus a necessity, but smart contracts challenge the existing highly efficient techniques routinely applied in the symbolic verification of software, due to specificities not present in general programming languages. A common feature of existing works in this area is the attempt to reuse off-the-shelf verification tools designed for general programming languages. This reuse can lead to inefficiency and potentially unsound results, since domain translation is required. In this paper we describe a carefully crafted approach that directly models the central aspects of smart contracts natively, going from the contract to its logical representation without intermediary steps. We use the expressive and highly automatable logic of constrained Horn clauses for modeling and we instantiate our approach to the Solidity language. A tool implementing our approach, called Solicitous, was developed and integrated into the SMTChecker module of the Solidity compiler solc. We evaluated our approach on an extensive benchmark set containing 22446 real-world smart contracts deployed on the Ethereum blockchain over a 27 months period. The results show that our approach is able to establish safety of significantly more contracts than comparable, publicly available verification tools, with an order of magnitude increase in the percentage of formally verified contracts.

CCS Concepts: • **Security and privacy → Logic and verification**; *Software and application security*.

Additional Key Words and Phrases: smart contracts, direct modeling, vulnerability detection

1 INTRODUCTION

Distributed ledgers, which underpin the blockchain technology, allow secure transactions between distrusting parties to take place without the need of a supervising authority, such as a bank. Their rise promises to disrupt the established way in which individuals and institutions interact with each other, projecting to providing a faster, cheaper, and more secure way to exchange goods and services. While the use of distributed ledgers was initially restricted to the creation of cryptocurrencies, e.g. Bitcoin [48], as interest in the area grew, blockchain

[*]Work done while at Università della Svizzera italiana.

Authors' addresses: Rodrigo Otoni, Università della Svizzera italiana, Lugano, Switzerland, rodrigo.benedito.otoni@usi.ch; Matteo Marescotti, Meta, London, UK, mmatteo@meta.com; Leonardo Alt, Ethereum Foundation, Zug, Switzerland, leo@ethereum.org; Patrick Eugster, Università della Svizzera italiana, Lugano, Switzerland and Purdue University, West Lafayette, USA, patrick.thomas.eugster@usi.ch; Antti E. J. Hyvärinen, Università della Svizzera italiana, Lugano, Switzerland, antti.hyvaerinen@usi.ch; Natasha Sharygina, Università della Svizzera italiana, Lugano, Switzerland, natasha.sharygina@usi.ch.

2471-2566/2022/9-ART $15.00
https://doi.org/10.1145/3564699

platforms started to also support programs capable of automatically executing contractual agreements written as code, in the form of *smart contracts*.

1.1 The Smart Contracts Technology

Smart contracts are distributed programs designed to manage and enforce contract transactions without relying on trusted authorities, but instead exploiting blockchain platforms to achieve this goal. A prime example of such a platform is Ethereum [14], one of the most used blockchain platforms [33], and the main target for the development of smart contracts. In the case of Ethereum, smart contracts are commonly implemented in high-level languages such as Solidity [23] and Vyper [62], and then compiled to low-level Ethereum virtual machine (EVM) bytecode [68], which is deployed on the blockchain itself. Besides their direct application in the financial sector [19, 54], smart contracts have being used in a variety of other areas, such as healthcare [27, 69], energy management [3, 13, 65], and gaming [46], among others [56].

1.2 Intricacies of Smart Contracts

Concerning their technical aspects, being deployed on blockchains gives smart contracts some peculiar features. Execution-wise, their transactional nature, in which a function execution either finishes successfully or has all its changes reversed, is the main difference in relation to traditional programming languages, such as C/C++ and Java. Regarding the code itself, it is: (i) immutable after deployment, which prevents vulnerability fixes; (ii) publicly visible, allowing potential attackers to search for code exploits; (iii) freely available, meaning that any user can interact with its interface. These features, combined with the fact that smart contracts often hold and manipulate significant financial assets, marks them as likely targets of attacks, with assets estimated in the order of millions of US Dollars having already been lost in the past [5]. The use of formal verification in order to ensure that no vulnerabilities are present in the code before deployment is thus an essential part of smart contracts development.

1.3 Verifying Smart Contracts

Although smart contracts are relatively new, much effort has been put into formally verifying them, with techniques such as static analysis [12, 21, 42, 47, 50, 52] and model checking [1, 38, 43, 66], among others [7, 31, 32, 34, 49, 64], having been applied to this domain. Most approaches, however, involve the use of existing tools, suitable to general software, in the verification of smart contracts. The clear benefit of this is the reuse of established off-the-shelf tools, which can provide much desired stability and efficiency, but an important drawback is the need of a translation from the domain of smart contracts to the tool domain, which adds a new unnecessary layer in the verification framework that is error-prone to develop, requires correctness proofs of its own, and can negatively impact precision and efficiency. The attempt to reuse existing tools is, thus, interesting at first, but not an ideal lasting solution, with the alternative being the development of purpose-built algorithms and tools to handle all features present in the smart contracts domain natively.

1.4 A Direct Modeling Approach

In this work we present a novel approach to the automated verification of smart contracts, called *direct modeling*, which we apply to the Solidity language. Direct modeling means that we generate a set of verification conditions in the target formalism directly from the contract's control-flow graph (CFG), using domain-specific knowledge, and without any intermediary steps. The instantiation of our approach to Solidity targets constrained Horn clauses (CHCs) [8] and is implemented in SOLICITOUS (Solidity contract verification using constrained Horn clauses), which is integrated into the SMTCHECKER [2] module of the Solidity compiler SOLC [22], with this paper describing the specification of SMTCHECKER's CHC model checking engine. We use CHCs for modeling contracts' behaviors because, besides being convenient to model structured computer programs, being used in the verification of languages such as C/C++ [29] and Java [37], CHCs benefit from an active community of researchers interested in their solving [8, 16, 28, 30, 36, 39]. Recent efforts in CHC solving lead to very efficient sequential and parallel solvers [10, 44] that can be directly exploited. The formal models of Solidity smart contracts produced by our approach are accurate, in that they properly encode the semantic traits specific to the smart contracts domain, and are solver-independent, meaning that we do not use solver-specific constructs.

Our aim is the verification of assertions already present in the source code, which we take as the contracts' specifications. This allows developers to not only ensure the absence of common known vulnerabilities, via injection of assertions that block attacking behaviors, prior to carrying out the verification, but to also check functional correctness. Additionally, once the set of states reachable in the CHC model is determined, we necessarily establish either a *contract invariant*, proving that a given property holds after an unbounded number of transactions, or a finite-length *counter-example* (CEX), concretely showing a property violation. Contract invariants are conditions over the contract's variables that always hold after any possible transaction, which can be used by developers to confirm their intents for the code, while CEXs are lists of transactions that lead to an assertion error, which can aid in the fixing of vulnerabilities.

1.5 Contributions

In addition to the detailed description of our direct modeling approach to the automated verification of smart contracts, and its implementation in Solicitous, we report an extensive experimental evaluation using 22446 real-world smart contracts currently deployed on the Ethereum blockchain. We compare Solicitous against three publicly available tools suitable for automated verification of Solidity assertions: SRI's Solc-Verify [31, 32], Microsoft's VeriSol [66], and ConsenSys' Mythril [17]. The results show that Solicitous outperforms the other tools in terms of both precision and efficiency.

To summarize, our contributions are the following:

(1) A precise direct formal modeling of Solidity smart contracts targeting CHCs, that enables efficient fully automated verification using CHC solvers.
(2) An industrial-strength implementation of the modeling approach inside the Solidity compiler.
(3) An extensive evaluation over thousands of real-world deployed contracts, which demonstrates the benefits of our approach in comparison to state-of-the-art techniques.

An earlier version of this work appeared at ISoLA'20 [45]. The present article substantially extends and improves the previous work, in the four-fold following manner:

(I) The modeling approach was enhanced to allow for the capture of the exact order and arguments of function calls, both internal and external, that lead to any assertion error, through the addition of four new rules (cf. $\mathsf{SumId}_{g,id}$, $\mathsf{Call}_{g,id,\rho_{call}}$, $\mathsf{ExtId}_{C,id}$, and $\mathsf{ECall}_{id,\rho_{call}}$).
(II) Solicitous underwent substantial improvements with respect to (a) its language coverage, resulting in fewer over-approximations and thus less inconclusive results, and (b) CEX production, providing users with additional information which enables them to easily reproduce any reported assertion errors.
(III) A more comprehensive evaluation was carried out, involving more than three times the number of contracts used previously and comparing our updated implementation against the newest versions of three state-of-the-art tools.
(IV) Additional explanations were added to every topic covered in the manuscript, including a running example in the form of contract `Auction`, shown in Figure 1, that concretely illustrates all rules in our modeling approach.

1.6 Roadmap

The remainder of this paper is structured as follows. Section 2 presents background information. Section 3 details and illustrates our modeling approach. Section 4 describes the Solicitous tool. Section 5 reports our evaluation results. Section 6 discusses related work. Finally, Section 7 presents conclusions and future work.

2 BACKGROUND

The differences between smart contracts and general programs come not only from smart contracts intended use, but also from their technical aspects, with these differences impacting how verification needs to be carried out. In this section we present the necessary background on smart contracts and the Solidity language, CFGs, and the CHC formalism.

2.1 Smart Contracts

Smart contracts consist of a *storage* and a set of *functions*. The storage is a persistent memory space used to store variables whose values represent the contract state. Functions are the interface by which users interact with the contract. Functions are allowed to access the storage both in read and write modes, and their behavior is defined by their corresponding bytecode instructions, stored persistently in a separate memory residing within the blockchain; the storage management is done via the blockchain platform's primitives, e.g. EVM in Ethereum. The interaction with a contract is performed by calling one of its functions, which can call other functions during its execution. The execution costs are commonly paid via a fee in the platform's native currency. Each individual function call is an *atomic* transaction, i.e. it either executes without raising exceptions, committing its changes, or rolls back completely if an exception occurs, leaving the state unchanged. Contrarily, in standard programming languages all the changes made by a function prior to throwing an exception are preserved.As a concrete example, if a C++ function throws an exception, the changes made to the heap are preserved.

Solidity language. The main high-level language specifically designed for smart contracts targeting EVM bytecode, Solidity, is a Turing-complete language in which a **`contract`** is a structure similar to a class in object-oriented programming languages. Contracts can have data types such as Boolean, integer, array, and map, and declare both public and private functions, depending on whether they can be called directly by the user. Such functions can make use of common programming languages control structures, such as conditionals and loops. Functions and blockchain addresses can be marked as **`payable`**, allowing them to receive funds in *ether*, Ethereum's native currency, with each contract having its own ether balance.

Example. Consider the `Auction` contract, shown in Figure 1, which provides realistic support for an auction. This contract has three state variables (lines 2-4), `bid` and `cash`, of type unsigned integer, and `winner`, of type **`address`**, which is a 20 byte Ethereum address. To manage the auction, `bid` and `cash` store the current highest bid and the amount of currency gathered, respectively, while `winner` stores the address from which the current highest bid was made. One function is present, `offer`, which handles the placing of new bids by users; for simplicity we abstract additional contract features, such as the ability to end the auction and forward the gathered funds to the auctioneer. The `offer` function has two implicit arguments: **`msg.value`**, that stores the amount of funds sent to the function, and **`msg.sender`**, that stores the address from which the function was called. Every new offer is subject to a fee of 10^{15} *wei*[1] (line 7), after which the function checks whether the new bid is greater than the current highest bid (line 8). A **`require`** statement works as a pre-condition in Solidity, usually employed to filter invalid inputs. In `offer`, if the new bid is not large enough the transaction reverts, with the fee payment being rescinded. After validating the new bid, the function returns the previous highest bid, if available, to its owner (lines 9-13), and updates the state variables (lines 14-16). An **`assert`** statement works as a post-condition in Solidity, meaning

[1]wei is the smallest subunit of ether, with 10^{15} wei being worth approximately 2.7 US Dollars at the time of writing.

```
contract Auction {
    uint bid = 0;
    uint cash = 0;
    address payable winner = address(0);

    function offer() public payable {
        uint new_bid = msg.value - 10^15 wei;
        require(bid < new_bid);
        if (winner ≠ address(0)) {
            assert(bid ≤ cash);
            winner.transfer(bid);
            cash = cash - bid;
        }
        bid = new_bid;
        cash = cash + msg.value;
        winner = msg.sender;
    }
    ...
}
```

Fig. 1. Example of a smart contract written in Solidity.

that its expression should never be false in a valid execution, with a violation leading to a `Panic` exception being thrown. In the example, an assertion error happens if the contract does not have enough funds to return the previous highest bid to its owner. Although both the **`require`** and **`assert`** statements stop the function execution and revert the changes made, they do it via different exception types, with the former doing it gracefully, since a failing **`require`** is a valid behavior, and the latter resorting to a `Panic` exception, also thrown, for instance, if a division by zero occurs.

2.2 Control-flow Graphs

A control-flow graph (CFG) is a graph representation of the execution paths of a program, being commonly used for static analysis. The graph's nodes represent *basic blocks*, i.e. sequences of program statements that do not change the control flow of the program, while its edges represent *jumps* modeled after the program's control structures. Programming structures that modify the control flow include conditionals, loops, and function calls, with each edge in a CFG being labeled with a Boolean expression that must be true for the jump to occur.

Given that a smart contract's transaction is rooted at a call to one of its public functions, which can lead to additional function calls, a set of CFGs, each modeling one particular function, can accurately capture the contract's behavior. In Figure 2 we see a representation of the CFG of function `offer` of contract `Auction`. The **`assert`** statements are modeled as *safety blocks*, which contain direct jumps to the exit block guarded by the negation of the asserted expressions. The **`require`** statements are treated as execution constraints, i.e. they do not affect the control flow and only exclude invalid executions, which is their intended purpose. The formal description of CFGs is given in Section 3.1.

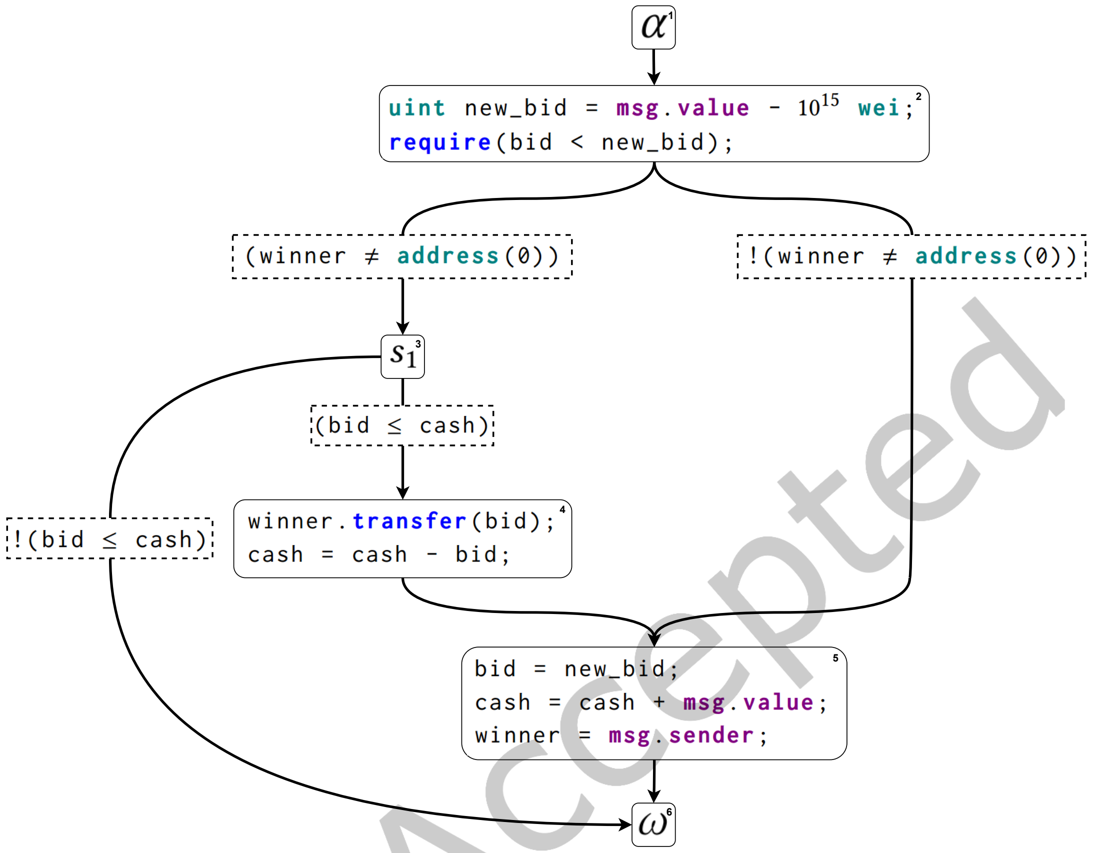

Fig. 2. Graphical representation of the CFG of function offer. Solid and dashed rectangles represent blocks and jumps' conditions, respectively, and α, ω, and s_1 correspond to the entry, exit, and safety blocks.

2.3 Constrained Horn Clauses

We leverage symbolic reasoning in determining the safety of a Solidity program. In this approach the state space of a program consists of the Cartesian product of the domain of the program counter and the domains of the program's variables. The full space is restricted to the space reachable by the program by writing down predicates in some logic that describe the reachable space. With this approach there is no need to explicitly list all the reachable states, e.g. as a set of tuples. The length of the search space representation in a logic is often vastly smaller than the explicit representation. In principle the idea is to compute the states that a program can reach in specific points of interest, given the program's control flow and its basic blocks in the static single assignment (SSA) form. Conceptually the idea is simple: each node in the control flow graph maps to a set of states. The initial states of the program can be obtained from constructors and other initialization. After this, the control flow is traversed, adding new states as we go to the sets associated with the nodes. This traversal continues until a fixed point is reached with respect to the state sets. However, finding logical formulas that represent these states is not easy. To construct the formulas, we apply recently developed methodology that combines well-defined semantics of this intuition with robust, albeit rapidly evolving solving technology.

The existential positive least fixed-point logic (E+LFP) is proven [9] to logically match Hoare logic [35], and is therefore useful for determining partial correctness of programs. Following [8], we use a specialization of E+LFP for verification, namely constrained Horn clauses (CHCs), due to their intuitive syntax in representing transition systems with loops and the efficient decision procedures available for them. We present here a characterization of CHCs based on first-order logic and the fixed-point operator adapted from [9].

Basic notation. To write the precise definition of how CHCs are used to model programs, we first define some notation. Let ψ be a first-order formula over a theory T, with free variables $\boldsymbol{x} = \{x_1, ..., x_n\}$, and $\{\mathcal{P}_1, \dots, \mathcal{P}_m\}$ be a finite set of predicates over $\boldsymbol{x}$ such that no $\mathcal{P}_i$ appears in ψ. A predicate $\mathcal{P}(\boldsymbol{x})$ over a set of variables $\boldsymbol{x}$ is associated with an *interpretation* that states on which values of $\boldsymbol{x}$ the predicate is true. The interpretation can be thought of as a set of tuples of length $|\boldsymbol{x}|$ explicitly stating such values. The satisfiability of $\mathcal{P}_1(\boldsymbol{x}) \land \dots \land \mathcal{P}_m(\boldsymbol{x}) \land \psi(\boldsymbol{x})$ in theory T, with the interpretations of $\mathcal{P}_i$ being $\Delta_{\mathcal{P}_i}$, is denoted by $\bigcup_{i=1}^{m} \Delta_{\mathcal{P}_i} \models_T \mathcal{P}_1(\boldsymbol{x}) \land \dots \land \mathcal{P}_m(\boldsymbol{x}) \land \psi(\boldsymbol{x})$.

Modeling with CHCs. When modeling a program, the predicates $\mathcal{P}_i$ are chosen to represent reachable states in certain key positions. These include the program counter values corresponding to the starts and exits of loops, function call sites, and, depending on the chosen modeling approach, starts and ends of conditional branches. The first-order formulas ψ encode the effect that the program code executed between the positions represented by predicates has on the state. However, the interpretations of the predicates are not explicitly known, but are instead defined implicitly by the program code represented by the formulas ψ and how the predicates are related by these formulas. CHCs provide a way of representing the relations between the program code and predicates, and there are highly engineered implementations of algorithms for determining over-approximations of the interpretations of the predicates for a given system of CHCs.

Given a set of predicates $\mathscr{P}$, a first-order theory T, and a set of variables $\mathcal{V}$, a *system of CHCs* is a set S of clauses of form

$$\mathcal{H}(\boldsymbol{x}) \leftarrow \exists \boldsymbol{y}.\, \mathcal{P}_1(\boldsymbol{y}) \land \dots \land \mathcal{P}_m(\boldsymbol{y}) \land \phi(\boldsymbol{x}, \boldsymbol{y}),\ \text{for } m \geq 0 \qquad \text{(DefClause)}$$

where ϕ is a first-order formula over $\boldsymbol{x}, \boldsymbol{y} \subseteq \mathcal{V}$, with respect to the theory T, $\boldsymbol{x}$ are the distinct variables free in ϕ, $\mathcal{H} \in \mathscr{P}$ is a predicate with arity matching $\boldsymbol{x}$, $\mathcal{P}_i \in \mathscr{P}$ are predicates with arities matching $\boldsymbol{y}$, and no predicate in $\mathscr{P}$ appears in ϕ. For a clause c we write $head(c) = \mathcal{H}$ and $body(c) = \exists \boldsymbol{y}.\, \mathcal{P}_1(\boldsymbol{y}) \land \dots \land \mathcal{P}_m(\boldsymbol{y}) \land \phi(\boldsymbol{x}, \boldsymbol{y})$.

For a concrete example, consider a program code `x = x+1`. To represent the states we need two copies of the program variable `x`, one representing `x`'s value before the execution of the program code and another representing the value after the execution. By convention these are represented by first-order variables x and x', respectively. We then define predicates $\mathcal{P}, \mathcal{Q}$ that hold before and after the execution of the increment. The system of CHCs modeling this fragment would then be $\mathcal{Q}(x') \leftarrow \exists x. \mathcal{P}(x) \land x' = x + 1$.

Precise predicate interpretations from CHCs. The tools that we use in this work do not attempt to determine the precise interpretations of the predicates appearing in the CHC systems, but instead compute, in general, an over-approximation of the models. However, to understand the modeling approach it suffices to consider how to obtain the exact interpretations. To this end, for each predicate $\mathcal{P} \in \mathscr{P}$ we define the transfinite sequence $\Delta^{\alpha}_{\mathcal{P}}$ given by

$$\begin{aligned} \Delta^{0}_{\mathcal{P}} &= \emptyset \\ \Delta^{\alpha+1}_{\mathcal{P}} &= \Delta^{\alpha}_{\mathcal{P}} \cup \{\boldsymbol{a} \mid \textstyle\bigcup_{\mathcal{Q} \in \mathscr{P}} \Delta^{\alpha}_{\mathcal{Q}} \models_T \bigvee_{c \in S, head(c) = \mathcal{P}} body(c)[\boldsymbol{a}/\boldsymbol{x}]\} \qquad \text{(DefPredInt)} \\ \Delta^{\lambda}_{\mathcal{P}} &= \textstyle\bigcup_{\alpha < \lambda} \Delta^{\alpha}_{\mathcal{P}},\ \text{for limit ordinals } \lambda \end{aligned}$$

Since $\Delta^{\alpha}_{\mathcal{P}}$ is monotonic, there is a value for α such that $\Delta^{\alpha}_{\mathcal{P}} = \Delta^{\alpha+1}_{\mathcal{P}}$. We denote the set $\Delta^{\alpha}_{\mathcal{P}}$ with this property by $\Delta_{\mathcal{P}}$.

In our approach to modeling and verifying smart contracts, we are interested in determining whether a bad state is reachable. The bad state is again represented by a CHC with a special head symbol $\bot$ and a body describing the bad state in logic. Determining whether the bad state is reachable reduces then to determining whether the interpretation $\Delta_\bot$ of predicate $\bot \in \mathscr{P}$ is empty.

To continue the above example, we might be interested whether after executing the increment, the value of `x` can be greater than 255. This would be encoded as the CHC $\bot \leftarrow \exists x.x > 255 \land Q(x)$.

Concretely, modern CHC solvers, based on the IC3 [11] algorithm, guarantee that if $\Delta_\bot$ is nonempty then the model of a program violates a safety property, and we are able to map predicate interpretations to a program execution. Conversely, if $\Delta_\bot$ is empty, either the solver does not terminate, or it provides quantifier-free first-order formulas $\eta_{\mathcal{P}}(\boldsymbol{x})$ in T for each $\mathcal{P} \in \mathscr{P}$ that serve as safe inductive invariants in the following sense:

(1) Each $\eta_{\mathcal{P}}$ over-approximates the interpretations $\Delta_{\mathcal{P}}$, that is, $\Delta_{\mathcal{P}} \models_T \mathcal{P}(\boldsymbol{x}) \implies \eta_{\mathcal{P}}(\boldsymbol{x})$.
(2) For each clause $c \in S$ of the form DefClause,
  (a) if $head(c) \neq \bot$, then $\models_T \eta_{\mathcal{P}_1}(\boldsymbol{y}) \land \ldots \land \eta_{\mathcal{P}_m}(\boldsymbol{y}) \land \phi(\boldsymbol{x}, \boldsymbol{y}) \implies \eta_{\mathcal{H}}(\boldsymbol{x})$;
  (b) if $head(c) = \bot$, then $\models_T \neg \left(\eta_{\mathcal{P}_1}(\boldsymbol{y}) \land \ldots \land \eta_{\mathcal{P}_m}(\boldsymbol{y}) \land \phi(\boldsymbol{x}, \boldsymbol{y})\right)$.

Following [8], a set of CHCs is called *satisfiable* if $\Delta_\bot$ is empty, and *unsatisfiable* otherwise.

Note that in presenting the clauses we use some conventions that make reading them easier. First, we omit the existential quantifier since its scope is clear from the arguments of the body for a given clause. Second, we do not write variables that do not appear in the formulas. Third, we omit superfluous equalities, e.g. if an element y_i of $\boldsymbol{y}$ is equated with an element x_j of $\boldsymbol{x}$ in a top-level conjunct of ϕ, we do not write the equality but instead substitute y_i for x_j in the head.

To conclude, this approach maps naturally to modeling programs. Each predicate $\mathcal{P}$ describes the set of reachable states in the points of interest in the program, and correspond to some concrete program counter values. The CHCs encode the flow of control between these points, and their constraints ϕ encode the conditions for control flow and the effects of the SSA executions. The safety properties can be encoded using the special predicate $\bot$ that by convention can only appear as a head of CHCs together with bodies that represent the negations of the safety properties. Finally, the operator Δ is used for accumulating the reachable states in the predicates appearing as heads of the CHCs. We will illustrate the use of CHCs shortly by modeling the contract `Auction` in Figure 1 step by step, starting in Section 3.2.

3 THE MODEL

Our approach to smart contracts verification is based on *direct modeling*, which means that we directly model contracts in the formalism suitable for the back-end reasoning engine used. We use CHCs for modeling contract behaviors, based on their control flow, with our algorithm creating formal models of Solidity smart contracts that are accurate and solver-independent. Accurate models properly encode the semantic traits specific of smart contracts, and solver independence allows any CHC solver to solve them. Additionally, the solving procedure automatically provides either a *contract invariant* or a finite-length *counter-example* (CEX). A contract invariant is a condition over the contract's variables that always holds after any possible transaction, enabling proofs of unbounded safety. A finite-length CEX is a list of transactions that lead to an assertion error, showing a concrete property violation.

This section is divided into seven parts. We first present our formal definition of a smart contract, followed by the details regarding the modeling of contract's functions, static function calls, and dynamic function calls, the overarching algorithm used to create the CHC model of an entire contract, how safety can be checked, and finally how CEXs can be produced.

3.1 Basic Definitions and Notation

We define a contract C as a triplet $\langle \boldsymbol{s}, I(\boldsymbol{s}), F\rangle$, where $\boldsymbol{s}$ is the set of state variables, $I(\boldsymbol{s})$ is the initial state of $\boldsymbol{s}$, and F is the set of all functions in the contract. The disjoint subsets F^+ and F^- of F denote the sets of public and private functions of F. For example, the formal definition of the contract `Auction` in Figure 1 is as follows

$$Auction = \langle \{b, c, w\}, b = 0 \land c = 0 \land w = 0, \{offer(\{v, s\}) \longrightarrow \{\tilde{r}\}\}\rangle$$

with b, c, and w representing the state variables `bid`, `cash`, and `winner`, v and s representing the implicit function arguments **`msg.value`** and **`msg.sender`**, $\tilde{r}$ being a special variable not derived from the source code, used to capture the occurrence of a revert, and $offer \in F^+$. In order to keep the example simple, we refrain from modeling implicit state variables, e.g. `balance` $\in \boldsymbol{s}$, which records the amount of funds held by the contract.

Given a function $f(\boldsymbol{a}) \longrightarrow \boldsymbol{r} \in F$, where $\boldsymbol{a}$ is the set of function arguments, and $\boldsymbol{r}$ is the set of return variables, the CFG of f is the tuple $\langle G, \alpha, \omega, \rho\rangle$. $G = (V, E, \lambda, \mu, S)$ is a node- and edge-labeled directed graph, where V is the set of CFG blocks, $E \subseteq V \times V$ is the set of control flow *jumps*, $\lambda_v \in \lambda$ is, for all $v \in V$, the set that contains the instructions performed by v, $\mu_e \in \mu$ is, for all $e \in E$, the condition under which the jump e is performed, and $S \subseteq V$ is the set of *safety blocks*, with each such block representing a safety property. The CFG blocks $\alpha, \omega \in V$ are respectively the entry and the exit blocks. The CFG of function `offer`, represented as CFG_o, is as follows

$$\begin{aligned}\text{CFG}_o &= \langle G_o, 1, 6, \rho_o\rangle \\ G_o &= (\{1, 2, 3, 4, 5, 6\}, \{\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 2, 5\rangle, \langle 3, 4\rangle, \langle 3, 6\rangle, \langle 4, 5\rangle, \langle 5, 6\rangle\}, \lambda, \mu, \{3\})\end{aligned}$$

with the block identifiers in Figure 2 representing the graph's nodes and the instructions pertaining to λ and μ being as shown in Figure 2, e.g. $\lambda_1 = \lambda_3 = \lambda_6 = \emptyset$ and $\mu_{(1,2)} = \mu_{(4,5)} = \mu_{(5,6)} = \texttt{true}$.

In the encoding of a function f only local variables are manipulated, therefore the labelings λ and μ of each block and jump contain instructions performed only over a set of local variables $\boldsymbol{l}$ of f; changes to state variables are first done in local copies of them, and only committed if f terminates successfully. The injection $\rho : \boldsymbol{s} \cup \boldsymbol{a} \cup \boldsymbol{r} \longrightarrow \boldsymbol{l}$ maps every state variable, function argument, and return variable, to a distinct local variable accessed by the instructions in each block and jump. We extend the mapping notation to sets in the natural way: for a given set of variables $\boldsymbol{z}$, $\rho(\boldsymbol{z}) = \{\rho(x) \mid x \in \boldsymbol{z}\}$. The injection ρ_o present in CFG_o is as follows

$$\rho_o = \{b \longrightarrow l_b, c \longrightarrow l_c, w \longrightarrow l_w, v \longrightarrow l_v, s \longrightarrow l_s, \tilde{r} \longrightarrow l_{\tilde{r}}\}$$

with l_x, $x \in \{b, c, w, v, s\}$, representing the local copy of variable x and $l_{\tilde{r}}$ representing the local copy of the revert variable.

A safety property in the CFG is represented by a safety block. In Solidity, safety properties are specified with the **`assert`** keyword, and their failing during the execution cause the function to revert and return immediately. To achieve this behavior, for every safety block $b \in S$, there exists the jump $e = \langle b, \omega\rangle$, where the condition μ_e is the negation of the property. This ensures a direct jump to the exit block in case the safety property is violated. A jump to the exit block ω from a safety block requires ω to revert the changes made by the function, restoring the state to prior the function's execution. In order to provide ω with the information that a safety property has been broken, λ_b sets $\rho(\tilde{r}) \in \boldsymbol{l}$ to a value that uniquely identifies the violated safety property, with $\tilde{r} \in \boldsymbol{r}$ being a special revert variable not derived from the source code. For the function `offer` we have one safety block, block 3, with $\mu_{(3,4)}$ being the asserted property, `bid` $\leq$ `cash` as shown in Figure 2, and $\mu_{(3,6)}$ being its negation.

3.2 Contract's Functions

Given a contract C with state variables $\boldsymbol{s}$, a function $f(\boldsymbol{a}) \longrightarrow \boldsymbol{r} \in F$, with local variables $\boldsymbol{l}$ and CFG $\langle G, \alpha, \omega, \rho\rangle$, is modeled in the following manner. For each CFG block v, $\mathcal{P}^v_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l})$ is the predicate symbol that represents the states that are reachable in block v, and the SSA formula $\text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$, with $\boldsymbol{l}' = \{x' \mid x \in \boldsymbol{l}\}$, models the behavior of v by formalising in first-order logic the relation between x and x', for each $x \in \boldsymbol{l}$, based on the execution of the instructions in λ_v. For each CFG jump e, the formula $\text{SSA}_{\mu_e}(\boldsymbol{l})$ is the logical condition under which e is taken. The execution of f is defined by three rules.

The *jump rule* models each jump $e = \langle v, u \rangle \in E$, and is expressed by the CHC

$$\mathcal{P}_f^u(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}') \leftarrow \mathcal{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \land \mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \land \mathsf{SSA}_{\mu_e}(\boldsymbol{l}) \tag{$\mathsf{Jump}_{f,e}$}$$

The function `offer` contains seven jumps, as seen in the definition of G_o and illustrated on Figure 2. Applying the rule $\mathsf{Jump}_{f,e}$ to these jumps yield the following CHCs:

$$\mathcal{P}_o^2 \leftarrow \mathcal{P}_o^1 \tag{$\mathsf{Jump}_{o,\langle 1,2\rangle}$}$$
$$\mathcal{P}_o^3 \leftarrow \mathcal{P}_o^2 \land l_{nb} = l_v - 10^{15} \land l_b < l_{nb} \land l_w \neq 0 \tag{$\mathsf{Jump}_{o,\langle 2,3\rangle}$}$$
$$\mathcal{P}_o^5 \leftarrow \mathcal{P}_o^2 \land l_{nb} = l_v - 10^{15} \land l_b < l_{nb} \land \neg(l_w \neq 0) \tag{$\mathsf{Jump}_{o,\langle 2,5\rangle}$}$$
$$\mathcal{P}_o^4 \leftarrow \mathcal{P}_o^3 \land l_b \leq l_c \tag{$\mathsf{Jump}_{o,\langle 3,4\rangle}$}$$
$$\mathcal{P}_o^6 \leftarrow \mathcal{P}_o^3 \land \neg(l_b \leq l_c) \land l_{\tilde{r}} = 3 \tag{$\mathsf{Jump}_{o,\langle 3,6\rangle}$}$$
$$\mathcal{P}_o^5 \leftarrow \mathcal{P}_o^4 \land l_c' = l_c - l_b \tag{$\mathsf{Jump}_{o,\langle 4,5\rangle}$}$$
$$\mathcal{P}_o^6 \leftarrow \mathcal{P}_o^5 \land l_b' = l_{nb} \land l_c' = l_c + l_v \land l_w' = l_s \tag{$\mathsf{Jump}_{o,\langle 5,6\rangle}$}$$

with the signatures of $\mathcal{P}_o^1, \mathcal{P}_o^2, \mathcal{P}_o^3, \mathcal{P}_o^4, \mathcal{P}_o^5$, and $\mathcal{P}_o^6$, which represent the CFG blocks of function `offer`, being $(b, c, w, s, v, l_b, l_c, l_w, l_v, l_s, l_{nb}, l_{\tilde{r}})$. We use l_{nb} to represent the local variable `new_bid`. The **`transfer`** on line 11 of Figure 1 is abstracted out because `balance` $\in \boldsymbol{s}$ is not present. When a primed version of a variable appears in the body of a CHC, such variable is also primed in the head, e.g. in $\mathsf{Jump}_{o,\langle 5,6\rangle}$ we have $\mathcal{P}_o^5(b, c, w, s, v, l_b, l_c, l_w, l_v, l_s, l_{nb}, l_{\tilde{r}})$ representing all states reachable in block 5 and $\mathcal{P}_o^6(b, c, w, s, v, l_b', l_c', l_w', l_v, l_s, l_{nb}, l_{\tilde{r}})$ representing the states reachable in block 6 by jump $\langle 5, 6\rangle$, which contains updates to l_b, l_c and l_w.

To illustrate the application of $\mathsf{Jump}_{f,e}$ to one specific jump, let us consider the CHC $\mathsf{Jump}_{o,\langle 2,3\rangle}$. Its body contains predicate $\mathcal{P}_o^2$ conjoined with SSA_{λ_2}, which is the formula $l_{nb} = l_v - 10^{15} \land l_b < l_{nb}$ modeling the behaviour of block 2, and with $\mathsf{SSA}_{\mu_{\langle 2,3\rangle}}$, which is the formula $l_w \neq 0$ modeling the entry condition of the if-statement of function `offer`. Its head consists of the predicate $\mathcal{P}_o^3$, modeling the states in block 3 reachable through jump $\langle 2, 3\rangle$. For block 3, we can see that it is part of the body of two CHCs, $\mathsf{Jump}_{o,\langle 3,4\rangle}$ and $\mathsf{Jump}_{o,\langle 3,6\rangle}$, the former modeling normal execution and the latter modeling the case in which the assertion fails, with its head being the predicate encoding block 6, the exit block of `offer`.

The *entry rule* sets the local variables equal to the corresponding current values of state variables and passed arguments, and is expressed by the CHC

$$\mathcal{P}_f^\alpha(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \leftarrow \bigwedge_{x \in \boldsymbol{s} \cup \boldsymbol{a}} x = \rho(x) \land \rho(\tilde{r}) = 0 \tag{Entry_f}$$

The variables in $\boldsymbol{s}$ and $\boldsymbol{a}$ are symbolically assigned in Entry_f and never changed throughout the applications of $\mathsf{Jump}_{f,e}$, for any $e \in E$. In case of reverting during execution, these variables provide the necessary information to revert to the state prior to the execution of f. A revert is caused by a jump to ω setting the local variable $\rho(\tilde{r})$ equal to the integer identifier of a safety property that failed, with $\rho(\tilde{r})$ being initially set to zero.

The CHC modeling the entry of function `offer`, produced by rule Entry_f, is the following:

$$\mathcal{P}_o^1 \leftarrow b = l_b \land c = l_c \land w = l_w \land s = l_s \land v = l_v \land l_{\tilde{r}} = 0 \tag{Entry_o}$$

The *function summary* of a given function is the relation between its input and output, derived from all possible function executions. Let $\mathcal{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r})$ be the predicate symbol representing the function summary of an execution of f. In this context, the input is represented by the state variables prior to the execution, $\boldsymbol{s}$, and the function arguments, $\boldsymbol{a}$, while the output is represented by the state variables after the execution, $\boldsymbol{s}'$, and the return values,

$\boldsymbol{r}$. The *summary rule* is expressed by the CHC

$$\mathcal{S}_f(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}',\boldsymbol{r}) \leftarrow \mathcal{P}_f^{\omega}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{l}) \wedge \underbrace{\big(\rho(\tilde{r}) \neq 0 \implies \bigwedge_{x\in \boldsymbol{s}} x' = x\big)}_{revert} \wedge \underbrace{\big(\rho(\tilde{r}) = 0 \implies \bigwedge_{x\in \boldsymbol{s}} x' = \rho(x)\big)}_{commit} \wedge \underbrace{\bigwedge_{x\in \boldsymbol{r}} x = \rho(x)}_{return} \tag{Sum$_f$}$$

Rule Sum$_f$ contains three constraints, *revert* and *commit*, which are mutually exclusive, and *return*. The *revert* constraint ensures that an execution is reverted when ω is reached having the local variable corresponding to $\tilde{r}$ set to the identifier of a safety property. The *commit* constraint stores the local copy of the state variables in $\boldsymbol{s}'$, modeling a commit of the computed values. The *return* constraint equates the return variables $\boldsymbol{r}$ with their corresponding local variables.

The CHC modeling the summary of function `offer`, produced by rule Sum$_f$, is as follows

$$\begin{aligned}\mathcal{S}_o \leftarrow \mathcal{P}_o^6 &\wedge \big(l_{\tilde{r}} \neq 0 \implies b' = b \wedge c' = c \wedge w' = w\big)\\ &\wedge \big(l_{\tilde{r}} = 0 \implies b' = l_b \wedge c' = l_c \wedge w' = l_w\big) \wedge \tilde{r} = l_{\tilde{r}}\end{aligned} \tag{Sum$_o$}$$

with the signature of $\mathcal{S}_o$ being $(b, c, w, v, s, b', c', w', \tilde{r})$.

Definition 3.1 (Function Model). Given a contract function f, the set of CHCs modeling f, Π_f, consists of applications of the jump rule Jump$_{f,e}$, for each control flow jump e of f, and the entry and summary rules for f, Entry$_f$ and Sum$_f$.

For function `offer` we have the set

$$\begin{aligned}\Pi_o = \{&\mathsf{Jump}_{o,\langle 1,2\rangle}, \mathsf{Jump}_{o,\langle 2,3\rangle}, \mathsf{Jump}_{o,\langle 2,5\rangle}, \mathsf{Jump}_{o,\langle 3,4\rangle}, \mathsf{Jump}_{o,\langle 3,6\rangle}, \mathsf{Jump}_{o,\langle 4,5\rangle}, \mathsf{Jump}_{o,\langle 5,6\rangle}\} \cup\\ &\{\mathsf{Entry}_o, \mathsf{Sum}_o\}\end{aligned}$$

3.3 Function Calls

Consider functions f and g, which need not be distinct, represented by CFGs G_f and G_g. A function call is performed by a block v in G_f if its labeling λ_v contains the call instruction to G_g. At runtime, the execution of the CFG block v is performed by executing the CFG block α of G_g, which constitutes the start of an execution of G_g. When ω of G_g is reached, the transaction represented by the execution of G_g is finalized by committing any changes to the state variables. The execution is then resumed from v, mapping the return variables of g to the corresponding local variables of f, and updating the local variables of f representing state variables to match the new values resulting from the commit just performed by the concluded transaction. When the execution terminates, ρ is used to commit the changes performed locally in the model to the state variables.

Consider a control flow jump $e = \langle v, u\rangle$, where λ_v contains a function call to $g(\boldsymbol{a}_g)$ returning variables $\boldsymbol{r}_g$. The summary of g is used to synchronize the local variables of f with the new state committed by g after its execution terminates. To precisely represent distinct calls to the same function, in order to accurately record where an exception occurs, if it does so, we create uniquely tagged summaries for each call and use them in the definition of their respective SSA formulas.

The tagged summaries are created by the *summary id rule*, expressed by the CHC

$$\mathcal{S}_{g^{id}}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}',\boldsymbol{r}) \leftarrow \mathcal{S}_g(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}',\boldsymbol{r}) \tag{SumId$_{g,id}$}$$

The first tagged summary of function `offer` would be given by the following CHC:

$$\mathcal{S}_{o^1} \leftarrow \mathcal{S}_o \tag{SumId$_{o,1}$}$$

with additional tagged summaries being possible through CHCs SumId$_{o,n}$, $n \in \{2, 3, ...\}$.

Given a freshly tagged summary, $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ of $\mathsf{Jump}_{f,e}$, containing a function call, is defined as

$$\mathcal{S}_{g^{id}}(\boldsymbol{s}', \boldsymbol{a}_g, \boldsymbol{s}'', \boldsymbol{r}_g) \wedge \underbrace{\bigwedge_{x \in \boldsymbol{a}_g \cup \boldsymbol{r}_g} x = \rho_{call}(x)}_{\textit{arguments and returns passing}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{s}} \left(x' = \rho(x) \wedge x'' = \rho(x)'\right)}_{\textit{state set and update}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{l} \setminus \boldsymbol{l}_{call}} x' = x}_{\textit{untouched locals}} \qquad (\mathsf{Call}_{g,id,\rho_{call}})$$

The tagged summary maps s' to s'', instead of s to s', and appends the function's name to its arguments and return variables, in order to avoid a clash with the representation of the variables in the callee. The mapping $\rho_{call} : \boldsymbol{a}_g \rightarrow \boldsymbol{l}, \boldsymbol{r}_g \rightarrow \boldsymbol{l}'$ is specific to this call and maps both the arguments of g to $\boldsymbol{l}$ and the return variables of g to $\boldsymbol{l}'$. The set of local variables that can be affected by the call is $\boldsymbol{l}_{call} = \rho_{call}(\boldsymbol{r}_g) \cup \rho(\boldsymbol{s})$. The arguments are passed by value, thus local variables $\rho_{call}(\boldsymbol{a}_g)$ provided as arguments to function g are not affected by g's execution.

The SSA defined in $\mathsf{Call}_{g,id,\rho_{call}}$ constrains its function summary in a three-fold manner. First, the *arguments and returns passing* conjunction uses ρ_{call} to match arguments and return variables to the respective local variables of the caller. Second, the *state set and update* conjunction ensures that the local variables in $\boldsymbol{l}'$ that represent state variables are updated according to the execution g. Note that in case a revert occurs in the execution of g, $\rho(x) \equiv \rho(x)'$, meaning the state has not changed, and $\rho_{call}(\tilde{r_g}) \in \boldsymbol{l}$ is set according to the revert, which allows the modeling of revert propagation or catching by the caller. Lastly, for each local variable not in $\boldsymbol{l}_{call}$, the *untouched locals* conjunction equates its primed and non-primed versions, modeling that its value is not affected by the execution of g. Since all variables in $\boldsymbol{l}'$ are constrained, $\mathsf{Call}_{g,id,\rho_{call}}$ models a deterministic execution of the callee. By combining rules $\mathsf{Jump}_{f,e}$ and $\mathsf{Call}_{g,id,\rho_{call}}$, the resulting CHC is nonlinear, i.e. it contains more than one predicate in its body, in this case the predicates $\mathcal{P}_f^v$ and $\mathcal{S}_{g^{id}}$.

If the contract `Auction` contained a function `refundFee`, which allowed selected bidders to have their fees refunded, a call to it by some other function in `Auction` would be defined by the SSA

$$\begin{aligned} &\mathcal{S}_{rf^1} \wedge v_{rf} = l_v \wedge s_{rf} = l_s \wedge \tilde{r}_{rf} = l'_{\tilde{r}} \\ &\quad \wedge (b' = l_b \wedge b'' = l'_b) \wedge (c' = l_c \wedge c'' = l'_c) \wedge (w' = l_w \wedge w'' = l'_w) \\ &\quad \wedge l'_v = l_v \wedge l'_s = l_s \end{aligned} \qquad (\mathsf{Call}_{rf,1,\rho_{call_{rf1}}})$$

with the summary $\mathcal{S}_{rf^1}$ given by the CHC

$$\mathcal{S}_{rf^1} \leftarrow \mathcal{S}_{rf} \qquad (\mathsf{SumId}_{rf,1})$$

The predicates' signatures in $\mathsf{Call}_{rf,1,\rho_{call_{rf1}}}$ and $\mathsf{SumId}_{rf,1}$ are $(b', c', w', v_{rf}, s_{rf}, b'', c'', w'', \tilde{r}_{rf})$ and $(b, c, w, s, v, b', c', w', \tilde{r})$. Identifier 1 is assumed to tag only this particular call to `refundFee`, and `refundFee` has neither explicit arguments and return variables nor local variables in its body.

3.4 Contract's External Behavior

A transaction of a given a contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$ consists of the execution of a single public function of C. Contract transactions can therefore be modeled by the summaries of every function $f \in F^+$, with each summary providing the relation between the state variables before and after the execution of its associated function. The *external behavior* of C encompasses all possible interactions between it and an external contract, being defined as the transitive closure of C's transactions. This way we capture the relation between the state variables before and after an arbitrary number of calls to any of the contract's public functions, in any order; to capture whether an assertion error occurs in such calls, the revert variable $\tilde{r}$ is also recorded. We define the predicate $\mathcal{E}_C$ that models the external behavior of C inductively. The *external base rule* is expressed by the CHC

$$\mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}, 0) \leftarrow \top \qquad (\mathsf{ExtBase}_C)$$

and the *external inductive rule* is expressed, for each $f \in F^+$, by the CHC

$$\mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}'', \tilde{r}'') \leftarrow \mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}', \tilde{r}') \wedge \mathcal{S}_f(\boldsymbol{s}', \boldsymbol{a}, \boldsymbol{s}'', \boldsymbol{r}) \wedge \tilde{r}' = 0 \wedge \tilde{r}'' = \tilde{r} \qquad (\mathsf{ExtInd}_{C,f})$$

The external behaviour of contract `Auction`, assuming it contains the function `offer` as well as the function `refundFee` described at the end of Section 3.3, is given by the following CHCs:

$$\begin{aligned}
&\mathcal{E}_A \leftarrow \top && (\mathsf{ExtBase}_A)\\
&\mathcal{E}_A \leftarrow \mathcal{E}_A \wedge \mathcal{S}_o \wedge \tilde{r}' = 0 \wedge \tilde{r}'' = \tilde{r} && (\mathsf{ExtInd}_{A,o})\\
&\mathcal{E}_A \leftarrow \mathcal{E}_A \wedge \mathcal{S}_{rf} \wedge \tilde{r}' = 0 \wedge \tilde{r}'' = \tilde{r} && (\mathsf{ExtInd}_{A,rf})
\end{aligned}$$

with the signature of $\mathcal{E}_A$ being $(b^p, c^p, w^p, b^q, c^q, w^q, \tilde{r}^q)$, p and q being n primes, $n \in \{0, 1, 2\}$, depending of the rule being applied, and the signature of $\mathcal{S}_o$ and $\mathcal{S}_{rf}$ being $(b', c', w', s, v, b'', c'', w'', \tilde{r})$.

Predicate $\mathcal{E}_C$ can be used to model calls to functions of an external contract D, whose source code is unknown, capturing all possible interactions between C and D. Every control flow jump $\langle v, u\rangle$ of C in which block v contains a call to a function with unknown code is modeled by a uniquely tagged $\mathcal{E}_C$, instead of the called function's summary. The *external id rule* is given by the CHC

$$\mathcal{E}_{C^{id}}(\boldsymbol{s}, \boldsymbol{s}'', \tilde{r}'') \leftarrow \mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}'', \tilde{r}'') \qquad (\mathsf{ExtId}_{C,id})$$

The first tagged predicate modeling the external behaviour of contract `Auction` is given by the following CHC:

$$\mathcal{E}_{A^1} \leftarrow \mathcal{E}_A \qquad (\mathsf{ExtId}_{A,1})$$

with $\mathcal{E}_{A^1}$ having the same signature as $\mathcal{E}_A$.

The SSA formula modeling the external call in block v, $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$, is similar to $\mathsf{Call}_{g,id,\rho_{call}}$, with two differences: the predicate used is $\mathcal{E}_{C^{id}}$, instead of $\mathcal{S}_{g^{id}}$, and the *arguments and returns passing* constraint is reduced to the update of $\rho(\tilde{r})'$. The local variables in $\rho_{call}(\boldsymbol{r}_g) \setminus \{\rho(\tilde{r})'\}$ are unconstrained in order to nondeterministically model any possible values returned by the unknown source code. The definition of $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ is

$$\mathcal{E}_{C^{id}}(\boldsymbol{s}', \boldsymbol{s}'', \tilde{r}'') \wedge \underbrace{\rho(\tilde{r})' = \tilde{r}''}_{\textit{revert recording}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{s}} \left(x' = \rho(x) \wedge x'' = \rho(x)'\right)}_{\textit{state set and update}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{l} \setminus \boldsymbol{l}_{call}} x' = x}_{\textit{untouched locals}} \qquad (\mathsf{ECall}_{id,\rho_{call}})$$

If the function `refundFee` contained a call to an external function `unknown`, such a call would be defined by the following SSA formula:

$$\begin{aligned}
&\mathcal{E}_{A^1} \wedge l'_{\tilde{r}} = \tilde{r}'' && (\mathsf{ECall}_{1,\rho_{call_{u1}}})\\
&\wedge (b' = l_b \wedge b'' = l'_b) \wedge (c' = l_c \wedge c'' = l'_c) \wedge (w' = l_w \wedge w'' = l'_w)\\
&\wedge l'_v = l_v \wedge l'_s = l_s
\end{aligned}$$

If a safety proof for this model can be obtained, then it is not possible to construct an external contract that can violate assertions in C by any sequence of reentrant calls. The existence of a CEX for this model implies that there exists a contract that can be designed specifically for violating one or more assertions, by calling one or more public functions in a particular order and returning specific values. The unique tag on $\mathcal{E}_{C^{id}}$ allows us to record which external function call led to the assertion failure, in case a CEX is produced, which is important because calls made at different points of a function's body can have different effects, in case state variables are being manipulated.

3.5 Contract's Complete Behavior

Given a contract C, let $C(\boldsymbol{s})$ be the predicate representing the reachable values for the contract's state variables. The contract's initial state is modeled by the *initialization rule*, expressed by the CHC

$$C(\boldsymbol{s}) \leftarrow I(\boldsymbol{s}) \tag{Init_C}$$

Every root transaction with $f \in F^+$ is modeled by the *root transaction rule*, expressed by the CHC

$$C(\boldsymbol{s}') \leftarrow C(\boldsymbol{s}) \land \mathcal{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \land \tilde{r} = 0 \tag{$\mathsf{RootTr}_{C,f}$}$$

For the contract `Auction` and function `offer` shown in Figure 1, we have

$$A \leftarrow b = 0 \land c = 0 \land w = 0 \tag{Init_A}$$

$$A' \leftarrow A \land \mathcal{S}_o \land \tilde{r} = 0 \tag{$\mathsf{RootTr}_{A,o}$}$$

Predicate A's signature is (b, c, w), with A' standing for $A(b', c', w')$; $\mathcal{S}_o$'s signature is as Sum_o's.

Definition 3.2 (Contract Model). Given a contract C, the set of CHCs modeling all possible behaviors of C, Π_C, consists of applications of the initialization rule Init_C and the external base rule $\mathsf{ExtBase}_C$, together with the set of CHCs Π_f of every function $f \in F$ and, for each public function $f \in F^+$, the external inductive rule $\mathsf{ExtInd}_{C,f}$ and the root transaction rule $\mathsf{RootTr}_{C,f}$.

The complete CHC model of contract `Auction` is

$$\Pi_A = \{\mathsf{Init}_A, \mathsf{ExtBase}_A\} \cup \Pi_o \cup \{\mathsf{ExtInd}_{A,o}, \mathsf{RootTr}_{A,o}\}$$

Our modeling technique is summarized in Algorithm 1. Given as input a smart contract C, the algorithm returns the set Π_C of CHCs modeling C. Initially, Π_C contains only applications of rules Init_C and $\mathsf{ExtBase}_C$. The loop from line 1 to 26 iterates over every contract function f, constructing their respective Π_f sets, which are merged with Π_C in line 22, and applying rules $\mathsf{ExtInd}_{C,f}$ and $\mathsf{RootTr}_{C,f}$, in line 24, if the function is public. The internal loop from line 6 to 21 iterates over every edge $e = \langle v, w \rangle$ of the CFG of f, in order to apply rule $\mathsf{Jump}_{f,e}$. For the modeling of block v, the case in which it represents a function call is handled in lines 7 to 15, using either the summary of the called function or the predicate of an external call, while if no function call is present in v, the formal model representing its execution is generated in line 17.

3.6 Checking Contract Safety

The safety of a public function $f \in F^+$, Σ_f, is modeled by the *safety rule*, expressed by the CHC

$$\bot \leftarrow C(\boldsymbol{s}) \land \mathcal{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \land \tilde{r} \neq 0 \tag{$\mathsf{Safety}_{C,f}$}$$

Definition 3.3 (Contract Safety). Given a contract C, its safety condition consists of the set Σ_C containing the safety rules of every public function $f \in F^+$.

If $\Delta_\bot$ is empty for a given function f, then no transaction of f can result in an assertion error. A given contract C is safe if, and only if, the set of CHCs $\Pi_C \cup \Sigma_C$ is satisfiable. If an assertion error can be reached during the execution f, then there exist an interpretation of C and $\mathcal{S}_f$ that evaluates the body of the safety rule to *true*, making the set of CHCs unsatisfiable. Conversely, if the set $\Pi_C \cup \Sigma_C$ is satisfiable, then there exists a first-order formula $\eta(\boldsymbol{s})$ such that $\Delta_C \models_T C(\boldsymbol{s}) \implies \eta(\boldsymbol{s})$. The formula $\eta(\boldsymbol{s})$, called a *safe inductive invariant*, over-approximates the states reachable through any sequence of transactions and proves inductively that Σ_C always holds. Any formula $\iota(\boldsymbol{s})$ implied by a safe inductive invariant is a contract invariant, i.e. it is true in every reachable contract state.

```
Input     : contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$
Output    : set of CHCs $\Pi_C$
Initially : $\Pi_C \leftarrow \{\mathsf{Init}_C, \mathsf{ExtBase}_C\}$
1  foreach $f = \langle G, \alpha, \omega, \rho \rangle \in F$ with $G = \langle V, E, \lambda, \mu, S \rangle$ do
2      $\boldsymbol{a} \leftarrow$ arguments of $f$
3      $\boldsymbol{r} \leftarrow$ return variables of $f$
4      $\boldsymbol{l} \leftarrow$ local variables of $f$
5      $\Pi_f \leftarrow \{\mathsf{Entry}_f, \mathsf{Sum}_f\}$
6      foreach $e = \langle v, w \rangle \in E$ do
7          if $v$ contains a call to $g(\boldsymbol{a}_g) \rightarrow \boldsymbol{r}_g$ then
8              create $\rho_{call}$ from $\lambda_v$                    // maps arguments and returns of the call
9              if $\mathsf{Sum}_g$ is known then
10                 $\Pi_f \leftarrow \Pi_f \cup \{\mathsf{SumId}_{g,id}\}$            // adds a freshly tagged summary
11                 $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \leftarrow \mathsf{Call}_{g,id,\rho_{call}}$
12             else
13                 $\Pi_f \leftarrow \Pi_f \cup \{\mathsf{ExtId}_{C,id}\}$            // adds a freshly tagged external call
14                 $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \leftarrow \mathsf{ECall}_{id,\rho_{call}}$
15             end
16         else
17             $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \leftarrow \mathrm{model}(\lambda_v)$     // models according to the instructions of v
18         end
19         $\mathsf{SSA}_{\mu_e} \leftarrow \mathrm{model}(\mu_e)$                // models according to the conditions of e
20         $\Pi_f \leftarrow \Pi_f \cup \{\mathsf{Jump}_{f,e}\}$
21     end
22     $\Pi_C \leftarrow \Pi_C \cup \Pi_f$
23     if $f \in F^+$ then
24         $\Pi_C \leftarrow \Pi_C \cup \{\mathsf{ExtInd}_{C,f}, \mathsf{RootTr}_{C,f}\}$
25     end
26 end
```

Algorithm 1: The algorithm to construct Π_C.

For the code shown in Figure 1, having the predicates' signatures as in $\mathsf{RootTr}_{A,o}$, the safety rule of function `offer`, Σ_o, and the safety condition of contract `Auction`, Σ_A, are respectively

$$\bot \leftarrow A \wedge \mathcal{S}_o \wedge \tilde{r} \neq 0 \qquad (\Sigma_o)$$
$$\Sigma_A = \{\Sigma_o\}$$

Contract `Auction` has a potential vulnerability that can allow malicious users to siphon funds out of it, in a two step attack using function `offer`. First, the attacker makes a bid that is smaller than the bidding fee, e.g. by calling `offer` with `msg.value` = 0, leading to an underflow during the calculation of `new_bid` (line 7), which will be set to $2^{256} - 10^{15} +$ `msg.value`, assuming wrapping, i.e. overflow and underflow, is enabled; the type of `uint` is unsigned integer of 256 bits. This allows the attacker to set themselves as the winner, while sending no funds to the contract. Second, the attacker calls `offer` again, this time sending more funds than previously, but still less than the bidding fee, causing another underflow. Since the attacker is the current winner at this point, the function will refund them their previous bid (lines 9-13). The `assert` ensuring that a refund cannot be larger than the amount of funds gathered by the auction so far (line 10) prevents this attack.

A common practice in the domain of smart contracts is to add assertions during development, with the goal of catching unintended behaviors, but to remove them prior to deployment, in order to reduce deployment and execution costs. In the case of contract `Auction`, if the vulnerability was not caught prior to the removal of the `assert`, this would leave it open to the attack described, depending on the semantics of integer arithmetics adopted. Since the attack relies on underflows happening, the satisfiability of the set $\Pi_A \cup \Sigma_A$ varies by language version. The newest Solidity version, v0.8, treats overflow and underflow as invalid operations by default, which automatically revert, making the attack impossible. Using v0.8 default semantics when generating the SSA formulas will thus make the set $\Pi_A \cup \Sigma_A$ satisfiable[2]. The Solidity versions preceding v0.8, which account for the overwhelming majority of deployed contracts, and that can be expected to be used for a significant number of contracts deployed in the near future as well, since developers tend to migrate slowly to newer versions, have, however, wrapping behavior enabled, allowing the siphoning of funds.

By integrating our verification approach into the development process, developers can catch not only this simple vulnerability, but all vulnerabilities that can be specified using assertions, including documented exploits, and also check for functional properties. The checking procedure can easily be applied during development, since both the construction of the set of CHCs $\Pi_C \cup \Sigma_C$, for a given contract C, and the CHC solving, are fully automated, as detailed in Section 4.

3.7 Counter-Example Production

The *refutation*, or proof of unsatisfiability, of $\Pi_C \cup \Sigma_C$ proves that a specific safety rule in Σ_C cannot be satisfied, i.e. $\Delta_\perp$ is nonempty. While our solving methodology can show satisfiability over unbounded executions, through the use of over-approximation, we can only represent finite refutations. This is, of course, not a practical limitation, since we are only interested in vulnerabilities that manifest themselves after a finite number of steps. The description of how a refutation is constructed by the CHC solver is outside of the scope of this paper; instead we give here an overview of the refutations themselves and of how they are used in the production of CEXs.

A refutation is a tree-shaped structure obtained by an unwinding of clauses. Its nodes are labeled with clauses, with its root, v_0, being labeled with a clause having $\perp$ as head. For each predicate P in the body of clause c of a given refutation node, we create for said node a child labeled with a unique clause c', with $head(c') = P$. The leaves of the tree are labeled with clauses containing no predicates in their body. In a refutation, for all paths $v_0, \ldots, v_k$ from the root to a leaf, labeled with clauses $c_1, \ldots, c_k$, it must hold that

$$\models_T \; body_\phi(c_1)(\boldsymbol{x}_0, \boldsymbol{x}_1) \wedge body_\phi(c_2)(\boldsymbol{x}_1, \boldsymbol{x}_2) \wedge \ldots \wedge body_\phi(c_k)(\boldsymbol{x}_{k-1}, \boldsymbol{x}_k) \qquad \text{(DefRefPath)}$$

with $body_\phi(c)$ denoting the constraint ϕ of a given clause c in the general form DefClause.

A CEX is produced from a refutation by traversing the tree and listing the nodes that refer to the initialization rule Init_C, the root transaction rule $\mathsf{RootTr}_{C,f}$, and the safety rule $\mathsf{Safety}_{C,f}$. Due to the refutation's structure, traversing its leftmost path gives us a list of nodes in which the first element is labeled with a safety rule, followed by zero or more elements labeled with root transaction rules, and the last element is a leaf labeled as an initialization rule. The conjunction of the clauses in the obtained list satisfies DefRefPath and represents a trace of transactions that leads to an assertion error, with the children of each node modeling the contract state prior the transaction and the function call with specific arguments that results in the new state.

Assuming underflow as a valid operation, $\Pi_A \cup \Sigma_A$ is unsatisfiable, and thus leads to a refutation, shown in Figure 3. In order to produce the CEX, the tree is traversed, providing us with the list $\langle \Sigma_o, \mathsf{RootTr}_{A,o}, \mathsf{Init}_A \rangle$. The initial transaction in the CEX trace is the one from deployment, given by node Init_A, which sets `bid = 0`, `cash = 0`,

[2]Solidity v0.8 allows wrapping behaviour via the `unchecked` command, which enables the semantics used in older versions.

and `winner = 0`, and is followed by the function calls modeled by the two other nodes. Node $\mathrm{RootTr}_{A,o}$ models the result of the first call to `offer`, with **`msg.value`** `= 0` and **`msg.sender`** `= 0xA1`. Node Σ_o models the results of the second call to `offer`, with **`msg.value`** `= 1` and **`msg.sender`** `= 0xA2`, which results in an assertion error.

4 IMPLEMENTATION

Our approach is implemented inside the SMTCHECKER [2] component of the Solidity compiler SOLC[3] [22], as part of an ongoing collaboration with the engineers of the Ethereum Foundation. The implementation, called SOLICITOUS (Solidity contract verification using constrained Horn clauses), consists of the CHC model checking engine of SMTCHECKER.

4.1 Toolchain Integration

The SOLICITOUS functionality can be enabled by simply adding **`pragma experimental`** `SMTChecker` directly into the source file, prior to the compilation. If enabled, the compiler provides the contract's CFG to SOLICITOUS, which produces the CHC model $\Pi_C \cup \Sigma_C$, following Algorithm 1 for Π_C and $\mathrm{Safety}_{C,f}$ for Σ_C. The CHC model is then provided to SPACER [40], the IC3 [11] engine of the Z3 [18] solver, for checking. In case an assertion error is detected, SOLICITOUS provides a transaction trace as a witness to the error, which can be easily validated by the developer. An overview of SOLICITOUS can be seen in Figure 4.

4.2 Modeling Details

The modeling of control flow structures such as conditionals and loops is not restricted by rule $\mathrm{Jump}_{f,e}$, following the topology of the CFG provided. The types of all variables in the CHC model directly reflects their source code equivalents, with addresses being treated as uninterpreted symbols and the unique names of the variables being derived from the CFG structure.

In addition to general functions, Solidity has two special function-like structures: *modifiers* and *constructors*. A modifier is a code fragment that envelops a number of selected functions' bodies. In SOLICITOUS, modifiers are not modeled separately, but are instead inlined to the functions they modify. A constructor contains the initialization procedure executed during the deployment of a contract. Constructors are used in the definition of $I(\mathbf{s})$, where variables are given either an explicit initial value or the default initial value of their type. In contracts that inherit base classes, the inheritance order is obtained by the Solidity compiler using the C3 linearization [6], with each constructor being executed exactly once. In SOLICITOUS, the entire deployment procedure, which might include state variables' initialization and inheritance linearization, is inlined into a single **`constructor`** function.

4.3 Current Scope

To the best of our knowledge, no verification tool targeting Solidity supports the complete range of language features, with all tools working on a best-effort basis. SOLICITOUS currently supports a large working subset of Solidity, including the complex control flow and arithmetic operators, excluding exponentiation, integers of all available sizes, Boolean variables, arrays, mappings' assignment and access, and inheritance. Strings and structs are currently not supported, and their occurrences in ϕ are replaced by nondeterministic operations in order to maintain soundness. Continuous support in terms of both language features and versions is a goal of the Ethereum Foundation, with the supported subset of language expected to grow in the future.

[3]Available at https://github.com/ethereum/solidity/releases.

5 EXPERIMENTS

We performed large-scale experiments to evaluate both the precision and efficiency of our approach, as well as the current support of language features offered by our implementation. The evaluation was done in a fully automated fashion [61], enabling easy replication by interested third parties.

5.1 Benchmarks

Our benchmarks consist of real-world contracts deployed on the Ethereum blockchain over a period of 27 months. We gathered all contract deployment transactions present from block 7 million, mined on the 2nd of January 2019, to block 12 million, mined on the 8th of March 2021, and queried the Etherscan block explorer [20] for their respective source codes. We obtained the source code of 224186 contracts, of which 73614 are unique: 355 v0.8, 2617 v0.7, 16981 v0.6, 25306 v0.5, and 28355 of versions older than 0.5.

In our experiments we used only contracts containing `assert` statements, which were selected in two complementary ways. The first way was to simply select contracts already containing asserts, with these contracts being used as they were deployed. The second way involved contracts containing only asserts that were commented, with these contracts having their asserts uncommented. Commented asserts are of interest because developers might have commented them before deployment in order to reduce deployment and execution costs, believing them to always hold. In total, we used 22446 contracts in our evaluation: 38 v0.8, including 61 asserts, 870 v0.7, including 1110 asserts, 9136 v0.6, including 11114 asserts, and 12402 v0.5, including 20912 asserts; older versions were not included due to lack of tool support. All benchmarks are publicly available[4].

5.2 Approaches Compared Against

We compare Solicitous against three publicly available tools suitable for automated verification of Solidity assertions: SRI's Solc-Verify [31, 32] and Microsoft's VeriSol [66], that verify Solidity source code, and ConsenSys' Mythril [17], that verifies EVM bytecode. To the best of our knowledge these are the only tools with which an automated comparison is possible. We considered two other tools for comparison, namely Zeus [38] and SAFEVM [1], but Zeus is not publicly available and SAFEVM only supports Solidity v0.4.

Of the selected tools, Solicitous, Solc-Verify, and VeriSol can produce safe inductive invariants, and thus establish contract safety. Mythril, however, is a purely bounded checking engine, being only capable of verifying up to a set number of transactions after contract deployment; by default a depth of three transactions is considered. Given this, Mythril can never ensure safety, only report unsafe or inconclusive results. Despite these limitations, Mythril is well known in the smart contracts community for having good support for language features, and we choose it to serve as the gold standard for the language support metric. Regarding CEX production, Solicitous can produce CEXs of arbitrary length, reporting assertion errors that can happen at any point during the lifecycle of a contract, while VeriSol and Mythril can produce CEXs only up to given length, and Solc-Verify does not produce CEXs at all; VeriSol has a hybrid approach, first attempting to establish unbounded safety, and if that is not successful it performs a bounded check.

The features implemented in Solicitous vary by language version, with the description in Section 4.3 reflecting the current status of the v0.8 implementation. The support for language features is smaller for previous versions, with the v0.5 and v0.6 implementations not producing CEXs. This variation between versions is assumed to also hold for the other tools. Solicitous supports the full range of Solidity versions from v0.5 to v0.8, as does Mythril, but Solc-Verify is restricted to v0.5, v0.7, and v0.8, while VeriSol only supports v0.5. We relied on Z3 4.8.10 as the back end for all tools and set their encoding to modular arithmetic mode in order to properly capture the behaviour of arithmetic types.

[4]Available by cloning the git repository https://scm.ti-edu.ch/repogit/verify-solidity-contracts.git.

Table 1. Summary of experimental results. The best results are highlighted.

	Solicitous (SOL)	Solc-Verify (SV)	VeriSol (VS)	Mythril (M)
Benchmarks	**22446**	13310	12402	**22446**
Safe	**6651**	103	201	0
Unsafe	8	0	9	**21**
Inconclusive	2970	5601	230	728
Timeout	9058	3163	4032	19511
Tool crash	3759	4443	7930	2186
Verified	∼**29%**	∼0.7%	∼1%	∼0.09%

5.3 Comparative Analysis

The summary of our results can be seen in Table 1, with the number of benchmarks available for each tool being derived from the Solidity versions it supports. A breakdown of results per Solidity version can be seen in Table 2. Safe contracts are those for which all asserts are proved safe by safe inductive invariants. Unsafe contracts are those for which a CEX was produced. Inconclusive results arise when the tool fails to either establish safety or produce a CEX. The timeout for each individual tool execution is 60 seconds. The difference between an inconclusive result and a timeout is that in the former case the tool terminates successfully but is unable to classify the contract, while in the latter case the process running the tool is killed upon reaching the time limit; in the occurrence of a timeout a developer can decide to run the tool for longer, which is not applicable for inconclusive results, arising due to fundamental limitations of the tool's approach. Crashes happen when language elements not handled properly by the tool are present in the contract, e.g. inlined assembly code. A contract is considered verified if it is classified as either safe or unsafe.

Solicitous was able to guarantee one order of magnitude more contracts to be safe, in comparison with Solc-Verify and VeriSol. In terms of catching assertion errors, Solicitous was the most performant tool for the versions in which it produces CEXs, v0.7 and v0.8, with Mythril having the best result overall, probably due to the large percentage of instances of versions 0.5 and 0.6. The large number of benchmarks that lead to an inconclusive result or a timeout, among all tools, indicates the highly nontrivial nature of smart contracts verification. Regarding tool crashes, Mythril has shown itself to be the more stable tool, as expected, with Solicitous having the least amount of crashes among the tools capable of unbounded verification.

In order to compare the performance of the tools we gathered the runtimes of the executions that produced safe inductive invariants. The results are summarized in Figure 5. Solicitous was able to verify more than 4000 contracts in less than 10 seconds, and more than 6000 in less than 30 seconds, which highlights its applicability for contract developers. Solc-Verify's runtimes are also distributed throughout the time axis, with most of the 103 contracts classified as safe being done so in less than 40 seconds. VeriSol achieved all its 201 safe results in less than 10 seconds. The fact that the majority of the contracts were classified as safe in less than half the allocated time indicates that, for all tools, practical results can be achieved with small timeouts. We can also conclude that increasing the timeout can be beneficial for both Solicitous and Solc-Verify, but may not be for VeriSol. Given the positive results, aligned with the practical nature of the benchmarks, Solicitous stands as a valuable tool for Solidity developers.

Table 2. Experimental results detailed per Solidity version. SOL, SV, VS, and M stand, respectively, for Solicitous, Solc-Verify, VeriSol, and Mythril. A dash means that the corresponding tool does not support the specific Solidity version. The best results are highlighted.

(a) Version 0.5, totaling 12402 instances.

	SOL	SV	VS	M
Safe	**3689**	95	201	0
Unsafe	0	0	9	**13**
Inconclusive	2383	5327	230	279
Timeout	4101	3045	4032	9935
Tool crash	2229	3935	7930	2175
Verified	~**29%**	~0.7%	~1%	~0.1%

(b) Version 0.6, totaling 9136 instances.

	SOL	SV	VS	M
Safe	**2877**	-	-	0
Unsafe	0	-	-	**7**
Inconclusive	583	-	-	413
Timeout	4159	-	-	8706
Tool crash	1517	-	-	10
Verified	~**31%**	-	-	~0.07%

(c) Version 0.7, totaling 870 instances.

	SOL	SV	VS	M
Safe	**72**	8	-	0
Unsafe	**7**	0	-	1
Inconclusive	4	268	-	35
Timeout	775	118	-	833
Tool crash	12	476	-	1
Verified	~**9%**	~0.9%	-	~0.1%

(d) Version 0.8, totaling 38 instances.

	SOL	SV	VS	M
Safe	**13**	0	-	0
Unsafe	**1**	0	-	0
Inconclusive	0	6	-	1
Timeout	23	0	-	37
Tool crash	1	32	-	0
Verified	~**36%**	0%	-	0%

5.4 Manual Inspection and Vulnerabilities Found

To understand the types of vulnerabilities found by each tool and the contracts in which they occur, we manually inspected all thirty-eight contracts that were classified as unsafe in our experiments. In addition, we also inspected all v0.8 contracts classified as safe by Solicitous, in order to understand how complex are the contracts for which Solicitous can ensure safety.

Of the eight contracts classified as unsafe by Solicitous, five are governance contracts based on ERC20, one is a token exchange contract based on ERC20 and ERC165, one is token sale contract, and one is a voting contract. These contracts have 584 lines of code on average, with the smallest having 97 lines of code and the largest having 1325 lines of code. The vulnerabilities found are caused by reentrant calls, affecting one contract, overflow, also affecting one contract, and unexpected inputs, affecting the remaining six contracts. Regarding the assertion failures caused by unexpected inputs, one is simply due to using `assert` for input validation instead of `require`, one is a call by the contract owner to a function containing the `selfdestruct` statement, which is guarded by `assert(balance > 0)`, with the implicit assumption being that the owner will transfer all funds from the contract before making such a call, and four are asserting properties on values returned from calls to external contracts. The vulnerabilities caused by unexpected inputs, although less severe, are still problematic, since they lead to execution fees not being refunded depending on the Solidity version being used.

Of the nine contracts classified as unsafe by VeriSol, seven are governance contracts based on ERC20, one is a token sale contract, and one is a simple token transfer contract. These contracts have 229 lines of code on average, with the smallest having 37 lines of code and the largest having 397 lines of code. The vulnerabilities found are caused by overflow, affecting five contracts, unexpected inputs, affecting two contracts, and asserting a `transfer` statement, also affecting two contracts. Both assertions failures caused by unexpected inputs are due to using `assert` for input validation instead of `require`. The transferring of funds can fail for various reasons pertaining to the target address and should thus be avoided.

Of the twenty-one contracts classified as unsafe by MYTHRIL, four are governance contracts based on ERC20, four are token sale contracts, two are token time lock contracts, four are wallet management contracts, two are airdrop contracts, three are logging contracts, which store hashes associated with timestamps, one is a signature contract, and one is a simple contract that delegates calls to a specified address. These contracts have 390 lines of code on average, with the smallest having 23 lines of code and the largest having 990 lines of code. The vulnerabilities found are caused by overflow, affecting thirteen contracts, unexpected inputs, affecting seven contracts, and unexpected branch execution, affecting one contract. The assertion failures caused by unexpected inputs are due to validation of functions' inputs, two occurrences, and owner privileges checking, five occurrences. The assertion failure caused by an unexpected branch execution consists of an `assert(false)` statement in a part of the code that should not be reachable.

Of the thirteen v0.8 contracts classified as safe by SOLICITOUS, nine are governance contracts based on ERC20, two are token exchange contracts based on ERC20, ERC165, and ERC1363, one is a betting contract, and one is a vesting contract. These contracts have 565 lines of code on average, with the smallest having 233 lines of code and the largest having 721 lines of code. An interesting point is that three contracts contain comments stating that they were independently audited.

6 RELATED WORK

There is much interest in formally verifying smart contracts. With Ethereum being one of the most used platforms for smart contract applications currently, most verification works target either EVM bytecode, which is deployed directly on the blockchain, or Solidity source code, which is compiled to EVM bytecode. Verification approaches vary in both their scope and the manner in which they are carried out. The scope ranges from the checking of specific vulnerabilities, usually selected from among high-profile documented exploits, to the use of different forms of specification languages, be they code assertions, design patterns, or behavioral descriptions. The manner goes from fully automated verification, aimed at developers, to manually intensive checking, usually intended to be used as auditing aid.

6.1 Automated Verification of Specific Vulnerabilities

OYENTE [42] is one of the pioneers in the field, using symbolic execution of EVM bytecode, underpinned by satisfiability modulo theories (SMT) solving, to check for common pre-defined vulnerabilities. MAIAN [50] uses symbolic execution to check for specific types of trace vulnerabilities in EVM bytecode, i.e. vulnerabilities that manifest themselves over many transactions, and also relies on SMT solving. MANTICORE [47] is based on a symbolic execution engine for EVM that searches for pre-defined vulnerabilities using SMT solving. SLITHER [21] is a static analysis framework for the verification of Solidity contracts containing several vulnerability detectors based on bounded model checking. ETHAINTER [12] is a static analysis tool focused on tainted information detection on EVM bytecode, it relies on Datalog and can catch vulnerabilities such as free access to a contract's `selfdestruct` instruction. The issue of smart contract gas consumption, i.e. the payment of fees for the execution of contracts, and its related vulnerabilities, is considered in [43], where gas consumption estimation is tackled via SMT solving. An approach aimed at preventing vulnerabilities from being introduced in the first is place is that of SCILLA [58], an alternative programming language based on System F [53] built with contract safety as a principal concern. SCILLA's type system prevents a number of runtime vulnerabilities from being implemented and the language is accompanied by a framework for static analyses capable of checking specific properties, including the estimation of gas consumption. In contrast to the previously mentioned works, however, Scilla does not target Ethereum, being used instead in the Zilliqa platform [63].

Compared to our approach, these techniques are restricted in terms of the scope of their verification. Solicitous differs from them by not constraining the verification to predefined properties, allowing developers instead to check any properties that can be expressed via code assertions.

6.2 Automated Verification using Specification Languages

A verification framework based on F* [60] as an intermediary language, enabling the translation of both EVM and Solidity code to F* and the subsequent checking of error patterns via SMT solving, is proposed in [7]. It allows for the definition of different patterns in F*, but lacks support for many important language features, such as loops. Securify [64] encodes EVM bytecode into Datalog and checks for defined bytecode patterns, with a set of patterns being pre-defined, and a specification language being provided for the definition of additional ones. EthBMC [26] is a bounded model checker based on SMT solving targeting EVM bytecode. It can define a precise model of a contract's memory and is capable of checking a set of relevant properties, including access to the `selfdestruct` instruction. Extension of the checking capabilities is possible by encoding additional properties as constraints to be checked. Zeus [38] is a framework to check the correctness and fairness of smart contracts targeting the Ethereum or Fabric [4] blockchain platforms, with a fairness specification language being defined, from which assertions are injected into the source code prior to verification. It translates high-level source code into LLVM [41] bitcode, from which CHCs are generated and discharged to a solver, using either the SeaHorn [29] or SMACK [15] model checkers. SAFEVM [1] translates EVM bytecode to C, and then uses C verifiers to check properties such as invalid array access and division by zero, as well as assertion violations. eThor [57] is a static analysis tool targeting EVM bytecode. It abstracts the bytecode into CHCs via its HoRSt framework and uses reachability checking to verify the absence of vulnerabilities such as reentrancy. It can also check assertion failures by verifying the reachability of the INVALID EVM instruction. SmartPulse [59] allows for the checking of safety and liveness properties in Solidity contracts. The desired properties need to be specified in the SmartLTL language and be provided to SmartPulse together with the contract's source code and a model of the environment in which it is expected to operate. SmartPulse instruments the contract's code based on the properties specified, translates it to the Boogie intermediary language, and then performs verification based on counter-example guided abstraction refinement (CEGAR).

The three tools used for comparison in our evaluation also fall in this category. Solc-Verify [31, 32] translates Solidity source code to the Boogie intermediary language and then generates verification conditions that can be discharged to SMT solvers. In addition to checking for issues such as overflow and underflow, and assertion violations, it also provides support for annotations that can be added to the code to complement assertions. VeriSol [66] also translates Solidity source code to the Boogie intermediary language, checking assertion violations first in an unbounded manner and, if that does not yield a result, in a bounded fashion. Mythril [17] is a tool capable of symbolic execution of EVM bytecode to check assertion violations and some specific properties. It is potentially unsound, since it relies solely on bounded analysis over a given number of transactions, leaving undisclosed bugs that happen only after extended use of the contract. In contrast to our approach, these techniques tend to rely on existing frameworks, e.g. Boogie, and provide weaker guarantees, e.g. Mythril.

The fundamental problem that all cited tools attempt to tackle, that of verifying that a contract complies with a user-made specification, is also the target of Solicitous. The difference here comes from how the problem is approached. The cited works all rely on indirect representation in one way or another, even when CHCs are involved, as is the case of Zeus and eThor, while Solicitous models the contracts directly in the formalism suitable for solving.

6.3 Manually Supported Verification

K [55] is a semantic framework that has specific support for EVM [34], as well as Solidity [25] and Vyper [24], but is time-consuming and difficult for non-expert users to interact with, since it relies on manual intervention for verification. Deductive verification using Why3 is proposed in [49]. This approach involves crafting and verifying smart contracts in Why3's language, WhyML, and then compiling them to EVM, but it was evaluated only on a single case study. VerX [52] verifies functional properties of Solidity contracts written using its own specification language, it uses SMT solving and has a certain degree of push-button automation, but may require user input during the verification. The cited tools differ fundamentally from Solicitous by requiring human intervention during the verification process, in addition to the specification of properties. Compared to our approach, these techniques have the obvious drawback of not being fully automated, with their target audience comprising mainly of highly specialized users, and should thus be considered orthogonal.

7 CONCLUSIONS

In this article we presented a novel approach to the automated verification of smart contracts, called direct modeling, which allows us to bypass intermediary steps commonly found in current verification approaches. Our approach is instantiated to the Solidity language and targets the CHC formalism, leading to CHC models that (a) formally capture the semantic features specific of smart contracts, (b) enable efficient fully automated verification of contracts' properties, and (c) can directly exploit powerful CHC solvers for the production of both safe inductive invariants and CEXs. We implemented our approach in Solicitous, the CHC model checking engine of the Solidity compiler's formal verification module SMTChecker. An extensive evaluation involving 22446 real-world contracts specifying in total 33197 properties was performed, comparing Solicitous against three state-of-the-art tools. The results obtained demonstrate the benefits of our approach, with an order of magnitude improvement in the percentage of verified contracts. In light of our evaluation, we believe that our approach represents an effective and highly promising avenue for the verification of smart contracts.

Interesting directions for future research include (i) the further enhancement of the modeling, (ii) validation of the verification, and (iii) instantiation to other languages. For the first direction, enhancements could be made by considering both known external code during the verification, e.g. a call to a previously deployed contract written by the same development team, in order to increase precision, and the gas consumption of the contract's functions, to predict their execution fees. For the second direction, we envision lightweight correctness certificates that can be used to independently and automatically validate the safety of smart contracts, with their practical goal being to provide assurances to third parties about the contracts they are interacting with. Concretely, such certificates would validate results of unsatisfiability for CHC solvers, in a similar manner to what is already done with solvers for Boolean satisfiability (SAT) [67] and SMT [51]. For the third direction, our approach could be instantiated to other languages for smart contracts development, in order to investigate its generality and extend its practical application.

ACKNOWLEDGMENTS

Rodrigo Otoni's work was supported by the Swiss National Science Foundation grant 200021_197353 and by the European Research Council grant FP7-617805. Matteo Marescotti and Antti E. J. Hyvärinen's work at Università della Svizzera italiana, Switzerland, was supported by the Swiss National Science Foundation grant 200021_185031. Patrick Eugster's work was supported in parts by the European Research Council grant FP7-617805. The authors would like to thank Enrique Fynn and Fernando Pedone for their kind assistance in gathering the initial batch of Ethereum addresses for the contracts used in our experiments.

REFERENCES

[1] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2019. SAFEVM: A Safety Verifier for Ethereum Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 386–389.

[2] Leonardo Alt and Christian Reitwiessner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods*. 376–388.

[3] Merlinda Andoni, Valentin Robu, David Flynn, Simone Abram, Dale Geach, David Jenkins, Peter McCallum, and Andrew Peacock. 2019. Blockchain technology in the energy sector: A systematic review of challenges and opportunities. *Renewable and Sustainable Energy Reviews* 100 (2019), 143–174.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference*. Article 30, 15 pages.

[5] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust*. 164–186.

[6] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A Monotonic Superclass Linearization for Dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 69–82.

[7] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. 91–96.

[8] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. *Fields of Logic and Computation II. Lecture Notes in Computer Science* 9300 (2015), 24–51.

[9] Andreas Blass and Yuri Gurevich. 1987. Existential Fixed-Point Logic. In *Computation Theory and Logic*. 20–36.

[10] Martin Blicha, Antti E. J. Hyvärinen, Matteo Marescotti, and Natasha Sharygina. 2020. A Cooperative Parallelization Approach for Property-Directed k-Induction. In *Proceedings of the 21st International Conference on Verification, Model Checking, and Abstract Interpretation*. 270–292.

[11] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. 70–87.

[12] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[13] Vlada Brilliantova and Thomas Wolfgang Thurner. 2019. Blockchain and the future of energy. *Technology in Society* 57 (2019), 38–45.

[14] Vitalik Buterin. 2013. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper.

[15] Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamarić, and Michael Emmi. 2016. SMACK Software Verification Toolchain. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*. 589–592.

[16] CHC competition. 2021. Report on the 2021 edition. https://chc-comp.github.io/2021/presentation.pdf.

[17] ConsenSys. 2021. Mythril. https://github.com/ConsenSys/mythril.

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

[19] Benjamin Egelund-Müller, Martin Elsman, Fritz Henglein, and Omri Ross. 2017. Automated Execution of Financial Contracts on Blockchains. *Business & Information Systems Engineering* 59, 6 (2017), 457–467.

[20] Etherscan. 2021. Etherscan - The Ethereum Blockchain Explorer. https://etherscan.io/.

[21] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework For Smart Contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. 8–15.

[22] Ethereum Foundation. 2021. Solidity Compiler. https://github.com/ethereum/solidity.

[23] Ethereum Foundation. 2021. Solidity Documentation. https://solidity.readthedocs.io.

[24] K Framework. 2018. KVyper: Semantics of Vyper in K. https://github.com/kframework/vyper-semantics.

[25] K Framework. 2019. KSolidity: Semantics of Solidity in K. https://github.com/kframework/solidity-semantics.

[26] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *Proceedings of the 29th USENIX Security Symposium*. 2757–2774.

[27] William J. Gordon and Christian Catalini. 2018. Blockchain Technology for Healthcare: Facilitating the Transition to Patient-Driven Interoperability. *Computational and Structural Biotechnology Journal* 16 (2018), 224–230.

[28] Arie Gurfinkel and Nikolaj Bjørner. 2019. The Science, Art, and Magic of Constrained Horn Clauses. In *Proceedings of the 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 6–10.
[29] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Proceedings of the 27th International Conference on Computer Aided Verification*. 343–361.
[30] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2018. Quantifiers on Demand. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis*. 248–266.
[31] Ákos Hajdu and Dejan Jovanovic. 2020. SMT-Friendly Formalization of the Solidity Memory Model. In *Proceedings of the 29th European Symposium on Programming*. 224–250.
[32] Ákos Hajdu and Dejan Jovanovic. 2020. Solc-Verify: A Modular Verifier for Solidity Smart Contracts. In *Proceedings of the 11th International Conference on Verified Software. Theories, Tools, and Experiments*. 161–179.
[33] Hedgeweek. 2020. Ethereum was the most traded cryptocurrency in Q3 2020. https://www.hedgeweek.com/2020/11/11/292088/ethereum-was-most-traded-cryptocurrency-q3-2020-11m-average-daily-transactions.
[34] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium*. 204–217.
[35] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.
[36] Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*. 157–171.
[37] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *Proceedings of the 28th International Conference on Computer Aided Verification*. 352–358.
[38] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. 15 pages.
[39] Anvesh Komuravelli, Nikolaj Bjørner, Arie Gurfinkel, and Kenneth L. McMillan. 2015. Compositional Verification of Procedural Programs Using Horn Clauses over Integers and Arrays. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*. 89–96.
[40] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods System Design* 48, 3 (2016), 175–205.
[41] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. 12 pages.
[42] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 254–269.
[43] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. 2018. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods*. 450–465.
[44] Matteo Marescotti, Arie Gurfinkel, Antti E. J. Hyvärinen, and Natasha Sharygina. 2017. Designing parallel PDR. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*. 156–163.
[45] Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. 2020. Accurate Smart Contract Verification Through Direct Modelling. In *Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods*. 178–194.
[46] Tian Min, Hanyi Wang, Yaoze Guo, and Wei Cai. 2019. Blockchain Games: A Survey. In *Proceedings of the 2019 IEEE Conference on Games*. 1–8.
[47] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 1186–1189.
[48] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf.
[49] Zeinab Nehaï and François Bobot. 2020. Deductive Proof of Industrial Smart Contracts Using Why3. In *Formal Methods 2019 International Workshops*. 299–311.
[50] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663.
[51] Rodrigo Otoni, Martin Blicha, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. 2021. Theory-Specific Proof Steps Witnessing Correctness of SMT Executions. In *Proceedings of the 58th ACM/IEEE Design Automation Conference*. 541–546.
[52] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*. 1661–1677.
[53] John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*. 408–423.

[54] Alfonso D. D. M. Rius and Eamonn Gashier. 2020. Smart Derivatives: On-Chain Forwards for Digital Assets. In *Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods*. 195–211.

[55] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[56] S. Rouhani and R. Deters. 2019. Security, Performance, and Applications of Smart Contracts: A Systematic Survey. *IEEE Access* 7 (2019), 50759–50779.

[57] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 621–640.

[58] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 30 pages.

[59] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy*. 555–571.

[60] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 256–270.

[61] Ole Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (2011), 42–47.

[62] Vyper Team. 2021. Vyper Documentation. https://vyper.readthedocs.io.

[63] Zilliqa Team. 2017. Zilliqa Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf.

[64] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[65] Xiaonan Wang, Wentao Yang, Sana Noor, Chang Chen, Miao Guo, and Koen H. van Dam. 2019. Blockchain-based smart contract for energy demand management. *Energy Procedia* 158 (2019), 2719–2724.

[66] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2020. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *Proceedings of the 11th International Conference on Verified Software. Theories, Tools, and Experiments*. 87–106.

[67] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing*. 422–429.

[68] Gavin Wood. 2021. Ethereum: A Secure Decentralised Generalised Transaction Ledger (Petersburg Version 41c1837). https://ethereum.github.io/yellowpaper/paper.pdf.

[69] Peng Zhang, Jules White, Douglas C. Schmidt, Gunther Lenz, and S. Trent Rosenbloom. 2018. FHIRChain: Applying Blockchain to Securely and Scalably Share Clinical Data. *Computational and Structural Biotechnology Journal* 16 (2018), 267–278.

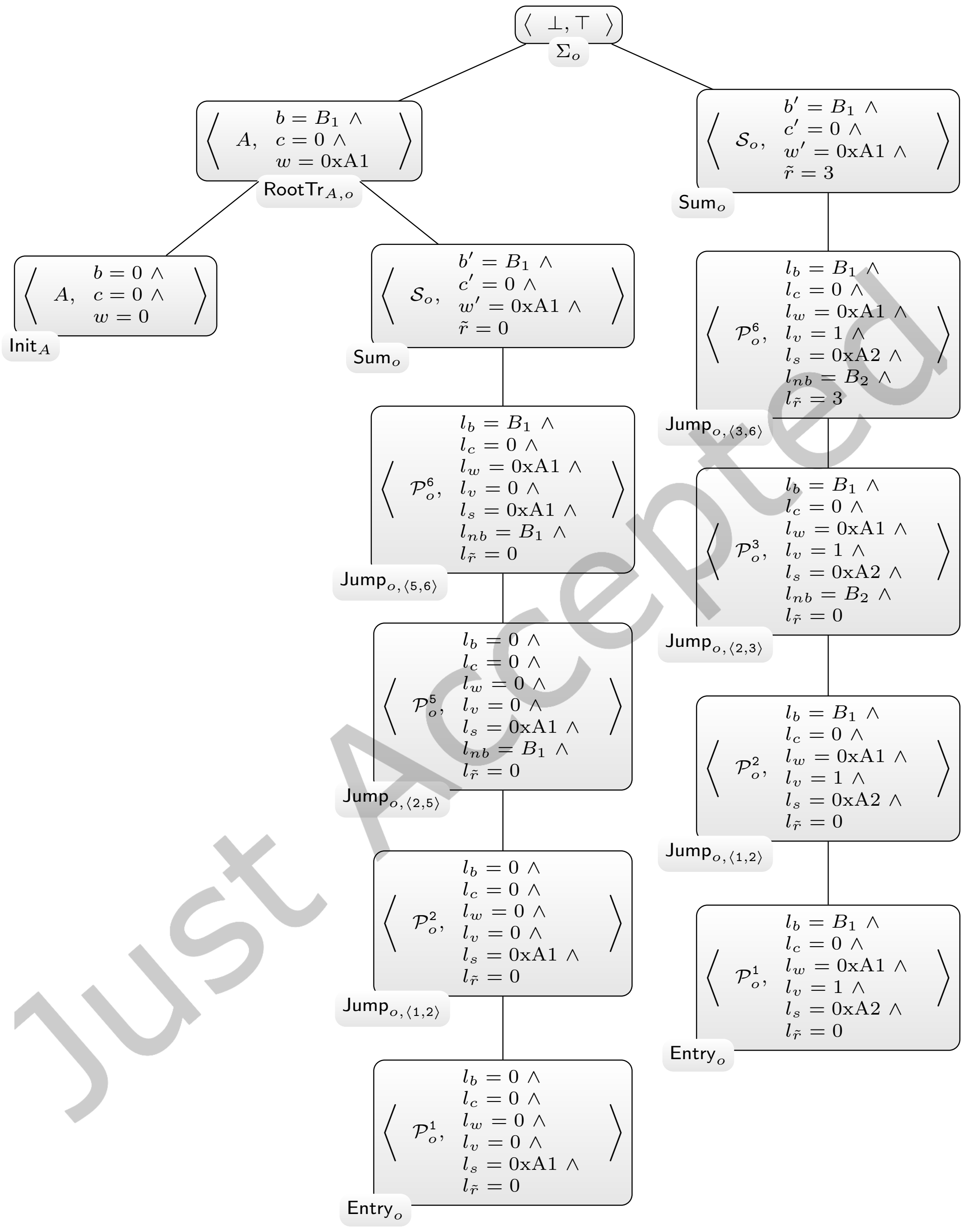
⟨ ⊥, ⊤ ⟩
Σ_o
⟨ A, $b = B_1 \wedge c = 0 \wedge w = 0\text{xA1}$ ⟩
$\mathsf{RootTr}_{A,o}$
⟨ A, $b = 0 \wedge c = 0 \wedge w = 0$ ⟩
Init_A
⟨ $\mathcal{S}_o$, $b' = B_1 \wedge c' = 0 \wedge w' = 0\text{xA1} \wedge \tilde{r} = 0$ ⟩
Sum_o
⟨ $\mathcal{P}_o^6$, $l_b = B_1 \wedge l_c = 0 \wedge l_w = 0\text{xA1} \wedge l_v = 0 \wedge l_s = 0\text{xA1} \wedge l_{nb} = B_1 \wedge l_{\tilde{r}} = 0$ ⟩
$\mathsf{Jump}_{o,\langle 5,6\rangle}$
⟨ $\mathcal{P}_o^5$, $l_b = 0 \wedge l_c = 0 \wedge l_w = 0 \wedge l_v = 0 \wedge l_s = 0\text{xA1} \wedge l_{nb} = B_1 \wedge l_{\tilde{r}} = 0$ ⟩
$\mathsf{Jump}_{o,\langle 2,5\rangle}$
⟨ $\mathcal{P}_o^2$, $l_b = 0 \wedge l_c = 0 \wedge l_w = 0 \wedge l_v = 0 \wedge l_s = 0\text{xA1} \wedge l_{\tilde{r}} = 0$ ⟩
$\mathsf{Jump}_{o,\langle 1,2\rangle}$
⟨ $\mathcal{P}_o^1$, $l_b = 0 \wedge l_c = 0 \wedge l_w = 0 \wedge l_v = 0 \wedge l_s = 0\text{xA1} \wedge l_{\tilde{r}} = 0$ ⟩
Entry_o
⟨ $\mathcal{S}_o$, $b' = B_1 \wedge c' = 0 \wedge w' = 0\text{xA1} \wedge \tilde{r} = 3$ ⟩
Sum_o
⟨ $\mathcal{P}_o^6$, $l_b = B_1 \wedge l_c = 0 \wedge l_w = 0\text{xA1} \wedge l_v = 1 \wedge l_s = 0\text{xA2} \wedge l_{nb} = B_2 \wedge l_{\tilde{r}} = 3$ ⟩
$\mathsf{Jump}_{o,\langle 3,6\rangle}$
⟨ $\mathcal{P}_o^3$, $l_b = B_1 \wedge l_c = 0 \wedge l_w = 0\text{xA1} \wedge l_v = 1 \wedge l_s = 0\text{xA2} \wedge l_{nb} = B_2 \wedge l_{\tilde{r}} = 0$ ⟩
$\mathsf{Jump}_{o,\langle 2,3\rangle}$
⟨ $\mathcal{P}_o^2$, $l_b = B_1 \wedge l_c = 0 \wedge l_w = 0\text{xA1} \wedge l_v = 1 \wedge l_s = 0\text{xA2} \wedge l_{\tilde{r}} = 0$ ⟩
$\mathsf{Jump}_{o,\langle 1,2\rangle}$
⟨ $\mathcal{P}_o^1$, $l_b = B_1 \wedge l_c = 0 \wedge l_w = 0\text{xA1} \wedge l_v = 1 \wedge l_s = 0\text{xA2} \wedge l_{\tilde{r}} = 0$ ⟩
Entry_o

Fig. 3. Refutation tree for the set $\Pi_A \cup \Sigma_A$. Besides its label, each node is annotated with a tuple containing the head of the referenced clause and the current value of relevant variables. The values 0xA1 and 0xA2 represent two Ethereum addresses, $B_1 = 2^{256} - 10^{15}$, and $B_2 = 2^{256} - 10^{15} + 1$.

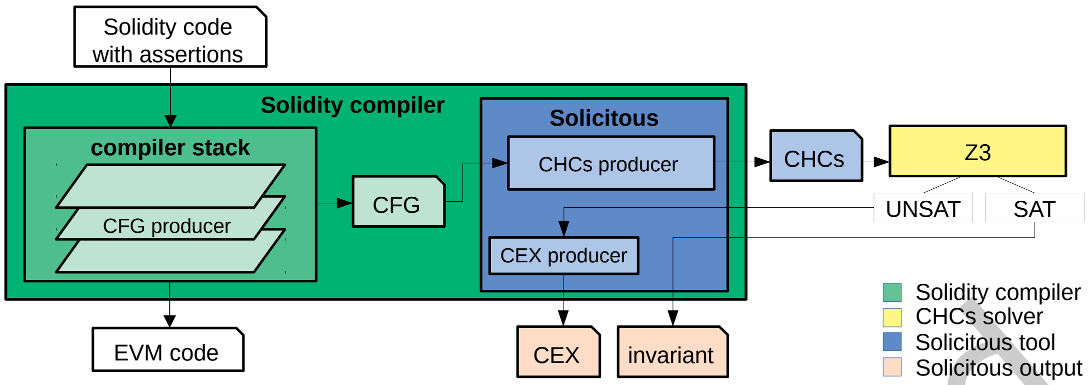

Fig. 4. SOLICITOUS module inside the Solidity compiler SOLC.

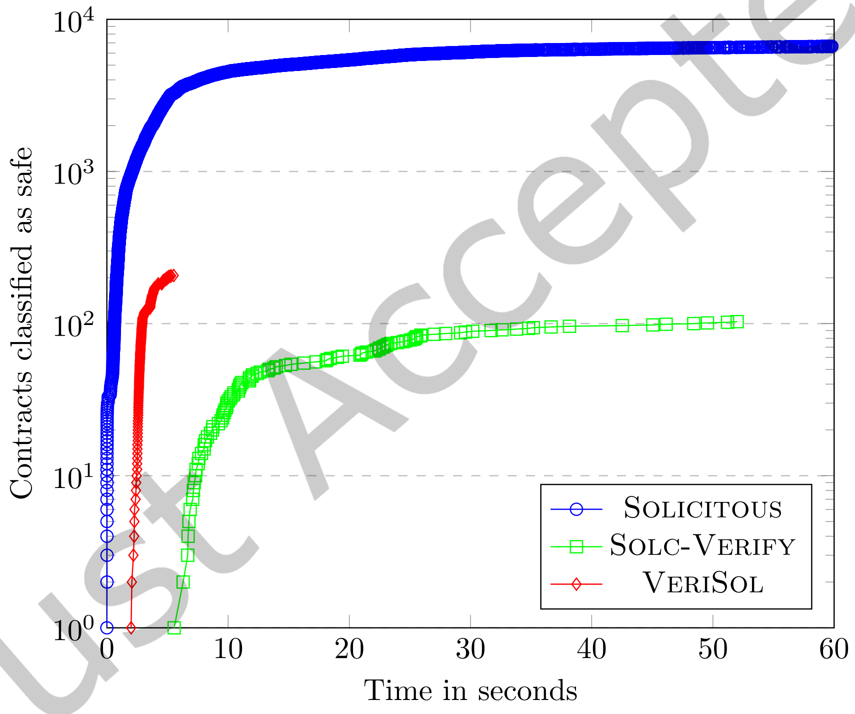

Fig. 5. Comparison of number of contracts classified as safe by allocated time.