Accurate Smart Contract Verification through Direct Modelling

Matteo Marescotti[1], Rodrigo Otoni[1], Leonardo Alt[2], Patrick Eugster[1], Antti E. J. Hyvärinen[1], and Natasha Sharygina[1]

[1] Università della Svizzera italiana, Lugano, Switzerland
[2] Ethereum Foundation

Abstract. Smart contracts challenge the existing, highly efficient techniques applied in symbolic model checking of software by their unique traits not present in standard programming models. Still, the majority of reported smart contract verification projects either reuse off-the-shelf model checking tools resulting in inefficient and even unsound models, or apply generic solutions that typically require highly-trained human intervention. In this paper, we present the solution adopted in the formal analysis engine of the official Solidity compiler. We focus on the accurate modeling of the central aspects of smart contracts. For that, we specify purpose-built rules defined in the expressive and highly automatable logic of constrained Horn clauses, which are readily supported by an effective solving infrastructure for establishing sound safety proofs or finite-length counterexamples. We evaluated our approach on an extensive set of smart contracts recently deployed in the Ethereum platform. The reported results show that the approach is able to prove correctness and discover bugs in significantly more contracts than comparable publicly available systems.

1 Introduction

Smart contracts are programs designed to manage and enforce contract transactions without relying on trusted parties but instead exploiting the blockchain technology to achieve consensus. The safety of smart contracts is increasingly important: in the past years millions of US Dollars were lost due to bugs [4,6], and currently the smart contracts deployed in the widely used Ethereum platform control increasing amounts of wealth in the order of billions of dollars. This issue is even more pronounced because once deployed in the blockchain, the source code of smart contracts is immutable, complicating the task of fixing errors with new releases.

Ethereum [17] nowadays is the most popular platform for writing smart contracts. High-level languages for implementing smart contracts such as Solidity [3] and Vyper [5] are compiled to the low-level Ethereum Virtual Machine (EVM) that is deployed in the blockchain. In this paper we introduce the *direct modeling* for Solidity smart contracts automatic verification implemented inside the Solidity compiler [18] in collaboration with the Euthereum Foundation. The proposed

direct modelling uses constrained Horn clauses (CHCs) [11] for modelling contract behaviours based on the control-flow. Besides being convenient to model transition systems, CHCs benefit from the current active area of research on their solving. Recent efforts produced several efficient sequential and parallel solvers [33,13] that can be directly exploited. Our algorithm for creating formal models of Solidity smart contracts produces models that are solver-independent (any theorem prover supporting CHCs can be used to solve them), and accurate (the models properly encode the semantic traits specific of smart contracts). Additionally, solving the model automatically provides *contract invariants* that prove unbounded safety, or a finite-length *counterexample* that concretely shows property violation. Contract invariants can also be used by developers to confirm the intents of the code. Specifically, they represent conditions over the contract variables that always hold after any possible transaction. Counterexamples show the interactions with the contract that lead to a violation of the safety properties. This is achieved by providing the list of transactions that produce an assertion error. All these features are implemented in the module (called Solicitous – Solidity contract verification using constrained Horn clauses) of the SMTChecker formal engine inside the official Solidity compiler [18].

In our experiments, our tool solves the CHCs generated with Spacer [29], the IC3 [14] engine of the SMT solver Z3 [35]. We compared Solicitous with SolcVerify [25,24], VeriSol [30] and Mythril [15], and report an extensive experimental evaluation verifying 6138 smart contracts currently deployed on the Ethereum blockchain. We show that Solicitous outperforms the other tools both for proving safety and for discovering bugs. To summarize, this paper provides the following contributions:

- a direct formal modeling of smart contracts using CHCs that enables fully automated verification using generic theorem provers (Sec. 3),
- an industrial implementation inside the Solidity compiler (Sec. 4), and
- an extensive verification experimentation over thousands of real-world contracts which demonstrates the effectiveness of our technique (Sec. 5).

We further discuss related work in Sec. 6, and conclude the paper in Sec. 7. An extended version of this paper including an end-to-end example of the entire verification process, from the Solidity source code to the counterexample, is available at `http://verify.inf.usi.ch/research/fvsc`.

2 Background

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 the corresponding EVM instructions, stored persistently in a separate memory residing within the blockchain. The Ethereum yellow paper [17] provides further details of the semantics of the EVM bytecode. Solidity[3]

[3] Solidity official documentation is available at `https://solidity.readthedocs.io`

is a Turing-Complete language specifically designed for smart contracts targeting EVM. A Solidity contract is a structure similar to a class in object-oriented programming languages. Contracts have data types such as integers, Boolean, array, map, etc. and either external or internal functions depending on whether they can be called directly by the user. Solidity supports control structures that are common in programming languages, such as conditionals and loops.

A control-flow graph (CFG) is a graph representation of the execution paths of a program, and it is commonly used for static analysis. The graph nodes represent *basic blocks*, that is, sequences of program statements that do not change the control flow of the program. Common programming constructs that modify the control flow are branching, loops, and function calls. Moving from one block to the next is a *jump*. Here we consider that the edges in a CFG are labeled with a Boolean expression that must be true for the jump to occur.

The interaction with a contract is performed by calling one of its external functions. During a function execution both external and internal functions can be called. Each individual function call is an *atomic* transaction, i.e., it either executes without exceptions committing the changes, or rolls back completely if an exception occurs, leaving the state unchanged. Contrarily, in standard programming languages all the changes made in the heap by a function prior to throwing an exception are preserved.

In [12] the *Existential Positive Least Fixed-Point logic* (E+LFP) is proven to logically match Hoare logic [27] and is therefore useful for determining partial correctness of programs. Following [11], in this work we use a specialization of E+LFP called *constrained Horn clauses* (CHC) due to the intuitive syntax in representing transition systems with loops, and the efficient decision procedures available for them. We give here a characterisation of CHC based on first-order logic and the fixed-point operator adapted from [12]. Let ψ be a first-order formula over a theory T with free variables $\vec{x}$, and a finite set $\{P_1, \ldots, P_n\}$ be predicates over $\vec{x}$ not appearing in ψ. We denote by $\bigcup_{i=1}^{n}\{\Delta_{P_i}\} \models_T \psi(\vec{x}) \wedge P_1(\vec{x}) \wedge \ldots \wedge P_n(\vec{x})$ the satisfiability of $\psi(\vec{x}) \wedge P_1(\vec{x}) \wedge \ldots \wedge P_n(\vec{x})$ in theory T when the interpretations of P_i are Δ_{P_i}.

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

$$H(\vec{x}) \leftarrow \exists \vec{y}.\, \phi(\vec{x}, \vec{y}) \wedge P_1(\vec{y}) \wedge \ldots \wedge P_m(\vec{y}) \text{ for } m \geq 0 \tag{1}$$

where ϕ is a first-order formula over $\vec{x}, \vec{y} \subseteq \mathcal{V}$ with respect to the theory T; $\vec{x}$ is the tuple of distinct variables free in ϕ; $H \in \mathcal{P}$ a predicate with arity matching $\vec{x}$; $P_i \in \mathcal{P}$ predicates with arities matching $\vec{y}$; and no predicate in $\mathcal{P}$ appears in ϕ. For a clause c we write $head(c) = H$ and $body(c) = \exists \vec{y}.\, \phi(\vec{x}, \vec{y}) \wedge P_1(\vec{y}) \wedge \ldots \wedge P_m(\vec{y})$. For each predicate $P \in \mathcal{P}$ we define the transfinite sequence Δ_P^α given by

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

Since the sequence Δ_P^α is monotonic, there is a value for α such that $\Delta_P^\alpha = \Delta_P^{\alpha+1} = \Delta_P$.

In the context of modeling and verification, in this paper we are in addition interested in determining whether the $\Delta_\bot$ of the predicate $\bot \in \mathcal{P}$ is empty. In particular the CHC solver we use guarantees that if $\Delta_\bot$ is nonempty then the model of a program violates a safety property and the solver is able to map the construction to an execution. Conversely, if $\Delta_\bot$ is empty, the solver either does not terminate, or provides quantifier-free first-order formulas $\psi_P(\vec{x})$ in T for each $P \in \mathcal{P}$ that serve as safe inductive invariants in the following sense. First, each ψ_P over-approximate the interpretations Δ_P, that is, $\{\Delta_P\} \models_T P(\vec{x}) \implies \psi_P(\vec{x})$. Second, for each clause $c \in S$ of the form (1) where $head(c) \neq \bot$, $\models_T \phi(\vec{x}, \vec{y}) \wedge \psi_{P_1}(\vec{y}) \wedge \ldots \wedge \psi_{P_m}(\vec{y}) \implies \psi_H(\vec{x})$. Third, if $head(c) = \bot$, then $\models_T \neg(\phi(\vec{x}, \vec{y}) \wedge \psi_{P_1}(\vec{y}) \wedge \ldots \wedge \psi_{P_m}(\vec{y}))$. We use the terminology from [11] and call a set of CHCs *satisfiable* if $\Delta_\bot$ is empty, and *unsatisfiable* otherwise.

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 that do not appear in the formulas. Third, we often omit superfluous equalities: if an element y_i of $\vec{y}$ is equated with an element x_j of $\vec{x}$ in a top-level conjunct of ϕ, we do not write the equality but instead substitute y_i for x_j in the head.

3 The Model

We define a contract C with the 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 respectively the sets of external and internal functions of F. Given a function $f(\boldsymbol{a}) \to \boldsymbol{r} \in F$, 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*; λ_v is the set that contains, for all $v \in V$, the set of instructions performed by v; μ_e 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*, each representing a safety property. During the execution of f only local variables are manipulated. Therefore the labelings λ and μ, respectively, of each block and jump, are instructions performed only over a set of local variables $\boldsymbol{l}$ of f. The CFG blocks $\alpha, \omega \in V$ are respectively the entry block and the exit block. The injection $\rho : \boldsymbol{s} \cup \boldsymbol{a} \cup \boldsymbol{r} \to \boldsymbol{l}$ 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 function notation to sets in the natural way: for a given set of variables $\vec{z}$, $\rho(\vec{z}) = \{\rho(x) \mid x \in \vec{z}\}$.

A safety property in the CFG is represented by a safety block. In Solidity, safety properties are specified with the `assert` keyword. Safety properties failing during the execution cause the function to revert and return immediately. To achieve this behaviour, 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 by restoring the state prior the function's execution. In order to provide ω with the information that a safety property has been broken, λ_b sets the special variable $\tilde{r} \in \boldsymbol{l}$ to a value that uniquely identifies the violated safety property.

Consider functions f and f' (which can be the same), represented by CFGs G and G' respectively. Function calls are performed by a block v in G whose labeling λ_v contains the call instruction to G'. At runtime, the execution of the CFG block v is performed by executing the CFG block α of G'. When ω of G' is executed, the transaction represented by the execution of G' is finalized by committing any changes to the state variables. The execution is then resumed from v, mapping the return variables of f' to the expected 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.

3.1 Model of a Contract Function

This section presents the rules for creating the CHC model of a function $f(\boldsymbol{a})$ of a contract having state variables $\boldsymbol{s}$, returning variables $\boldsymbol{r}$, and manipulating local variables $\boldsymbol{l}$.

The CHCs are constructed given the control flow graph $\langle G, \alpha, \omega, \rho \rangle$ of the function f, where $G = (V, E, \lambda, \mu)$. For each CFG block v, the *Static Single Assignment (SSA) formula* $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l'})$, where $\boldsymbol{l'} = \{x' \mid x \in \boldsymbol{l}\}$, models the behavior of v by formalizing in logic the relation between x and x' for each $x \in \boldsymbol{l}$, based on the execution of the instructions in λ_v. The formula $\mathsf{SSA}_{\mu_e}(\boldsymbol{l})$ of each jump e is the logical condition under which e is taken. For each CFG block $v \in V$, $\mathcal{P}^v_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l})$ is a predicate symbol representing the states that are reachable in the block v. The set of rules representing the execution of f is defined as follows. For each jump $e = \langle v, u \rangle \in E$, the *jump rule* of e is the CHC

$$\mathcal{P}^u_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l'}) \leftarrow \mathcal{P}^v_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \wedge \mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l'}) \wedge \mathsf{SSA}_{\mu_e}(\boldsymbol{l}). \qquad (\mathsf{Jump}_{f,e})$$

The *entry rule* sets the local variables equal to the corresponding current values of state variables and passed arguments.

$$\mathcal{P}^\alpha_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \leftarrow \bigwedge_{x \in \boldsymbol{s} \cup \boldsymbol{a}} x = \rho(x) \wedge \rho(\tilde{r}) = 0. \qquad (\mathsf{Entry}_f)$$

The variables in $\boldsymbol{s}$ and $\boldsymbol{a}$ are symbolically assigned in (Entry_f) and never changed throughout the jump rules $(\mathsf{Jump}_{f,e})$ of 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 a jump to ω setting the local variable $\rho(\tilde{r})$ equal to the integer identifier of a safety property that failed. Initially, $\rho(\tilde{r})$ is set to zero. Let $\mathcal{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'}, \boldsymbol{r})$ be the predicate symbol representing the *function summary* of the execution of f. The function summary expresses the relation between the input and the output of an execution of the function. In this context the input is represented by the function arguments $\boldsymbol{a}$ and state

variables $\boldsymbol{s}$ prior execution, and the output is represented by the return values $\boldsymbol{r}$ and the state variables $\boldsymbol{s}'$ after the execution. The *summary rule* of f is the CHC

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

The *revert* constraints in (Sum_f) 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. Conversely, the mutually exclusive *commit* constraints store the local copy of the state in $\boldsymbol{s}'$, modeling a commit of the computed values. The *return* constraints equate the return variables $\boldsymbol{r}$ with the corresponding local variables.

Definition 1. *Given a contract function f, the set of CHC Π_f modeling f is the set consisting of the jump rule of e ($\mathsf{Jump}_{f,e}$) for each control flow jump e of f, and the entry and summary rules from f (Entry_f) and (Sum_f).*

3.2 Function Calls

Let $e = \langle v, u \rangle$ be a control flow jump 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 after g's execution terminates. Therefore, $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ is defined as

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

where $\rho_{call} : \boldsymbol{a}_g \to \boldsymbol{l}, \boldsymbol{r}_g \to \boldsymbol{l}'$ is the mapping specific for this call that maps both arguments of g to $\boldsymbol{l}$ according to how they are passed, and the return variables of g to $\boldsymbol{l}'$ according to how they are assigned; $\boldsymbol{l}_{call} = \rho_{call}(\boldsymbol{r}_g) \cup \rho(\boldsymbol{s})$ is the set of local variables that can be affected by the call. We assume arguments are passed by value. Therefore local variables $\rho_{call}(\boldsymbol{a}_g)$ corresponding to the arguments of g are not affected by the execution of the block. The *argument and return passing* uses ρ_{call} to match arguments and return variables to the respective local variables of the caller. The *state set and update* conjunction makes sure that the local variables in $\boldsymbol{l}'$ representing the state variables get updated according to the execution of the just-ended transaction. For each local variable not in $\boldsymbol{l}_{call}$, the *untouched locals* constraint equates its primed and non-primed versions, modeling that its value is not affected by the block execution, and therefore remains unchanged after the jump. Note that the primed version of the local variables in $\boldsymbol{l}_{call}$ are set in the former constraints according to the

effects of the call. This ensures that all variables in $\boldsymbol{l}'$, which are passed to the predicate $\mathcal{P}^u_f$, are constrained, modeling a deterministic execution. By applying ($\mathsf{Jump}_{f,e}$), the resulting CHC is non-linear because it contains the two predicates $\mathcal{P}^v_f$ and $\mathcal{S}_g$.

3.3 Contract's External Behaviour

Given a contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$, a contract transaction is the execution of a public function. A single contract transaction is therefore modelled by the summaries of every function f in F^+, each proving the relation between state variables $\boldsymbol{s}, \boldsymbol{s}'$ before and after a transaction performed by calling f. The *external behaviour* of the contract is defined as the transitive closure of contract transactions, modelling an arbitrary number of calls to any public function, in any order. The external behaviour provides the relation between state variables before and after any possible interaction with the contract performed by an external contract.

We define the predicate $\mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}')$ that models the external behavior of C inductively, where the base case is the CHC

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

and the inductive steps are, for each function f in F^+, the CHCs

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

The external behaviour of C can be used to model calls to a function of an external contracts D which source code in unknown before runtime. In this way, any possible transaction resulting from D interaction during runtime is considered. Every control flow jump $\langle v, u \rangle$ in C, where the block v contains a call to a function that is unknown before runtime, is modelled using $\mathcal{E}_C$ in place of the called function summary. The resulting $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ is built similarly to ($\mathsf{Call}_{g,\rho_{call}}$), with the difference of omitting the *argument and return passing* constraints. The local variables in ρ_{call} are unconstrained in order to nondeterministically model any possible values returned by the unknown function. Specifically, the resulting $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ is

$$\mathcal{E}_C(\boldsymbol{s}', \boldsymbol{s}'') \wedge \underbrace{\bigwedge_{x \in \boldsymbol{s}} \left(x' = \rho(x) \wedge x'' = \rho(x)'\right)}_{\textit{state set and update}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{l} \setminus \boldsymbol{l}_{call}} x' = x}_{\textit{untouched locals}}. \tag{$\mathsf{ECall}_{\rho_{call}}$}$$

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. A counterexample for such 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.

Input : A contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$.
Output : The set of CHC Π_C.
Initially: $\Pi_C = \{(\mathsf{Init}_{\mathcal{C}}), (\mathsf{ExtBase}_{\mathcal{C}})\}$.

1 **foreach** $f = \langle G, \alpha, \omega, \epsilon, \rho \rangle \in F$ **do**
2 Let $\boldsymbol{a}, \boldsymbol{r}, \boldsymbol{l}$ respectively the arguments, returns and local variables of f.
3 Let $\Pi_f \coloneqq \{(\mathsf{Entry}_f), (\mathsf{Sum}_f)\}$
4 Let $G = (V, E, \lambda, \mu)$
5 **foreach** $e = \langle v, w \rangle \in E$ **do**
6 **if** *v contains a call to $g(\boldsymbol{a}_g) \to \boldsymbol{r}_g$* **then**
7 Create ρ_{call} from λ_v
8 **if** (Sum_g) *is known* **then** $\mathsf{SSA}_{\lambda_v} \coloneqq (\mathsf{Call}_{g,\rho_{call}})$;
9 **else** $\mathsf{SSA}_{\lambda_v} \coloneqq (\mathsf{ECall}_{\rho_{call}})$;
10 **else**
11 $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \coloneqq \mathrm{Model}(\lambda_v)$
12 **end**
13 $\mathsf{SSA}_{\mu_e} \coloneqq \mathrm{Model}(\mu_e)$
14 $\Pi_f \coloneqq \Pi_f \cup \{(\mathsf{Jump}_{f,e})\}$
15 **end**
16 $\Pi_C \coloneqq \Pi_C \cup \Pi_f$
17 **if** $f \in F^+$ **then**
18 $\Pi_C \coloneqq \Pi_C \cup \{(\mathsf{ExtInd}_{\mathcal{C},f}), (\mathsf{RootTr}_{\mathcal{C},f})\}$
19 **end**
20 **end**

Algorithm 1: The algorithm to construct Π_C.

3.4 Checking Contract Safety

Let $\mathcal{C}(\boldsymbol{s})$ be the predicate representing the reachable values for the contract. The initial state is modeled by the CHC

$$\mathcal{C}(\boldsymbol{s}) \leftarrow I(\boldsymbol{s}). \tag{$\mathsf{Init}_{\mathcal{C}}$}$$

Every transition performed by a call to a public function is modeled by the *root transition rule*. For each public function $f \in F^+$,

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

Definition 2. *Given a contract C, the set of CHC Π_C modeling any possible behavior of C is defined as the union of the initial rule* $(\mathsf{Init}_{\mathcal{C}})$*, the external base case rule* $(\mathsf{ExtBase}_{\mathcal{C}})$*, all the rules Π_f of every function $f \in F$, and for each public function $f \in F^+$ the root transition rule* $(\mathsf{RootTr}_{\mathcal{C},f})$ *and the external inductive rule* $(\mathsf{ExtInd}_{\mathcal{C},f})$*.*

Algorithm 1 gives an overall view of the modeling technique. Given as input a smart contract C, the algorithm returns the set Π_C of CHCs modeling C. Initially, Π_C consists only of the initial rule of C. Then, the loop from line 1 to 20

iterates over each contract function f, gradually producing the respective set Π_f that is finally merged with Π_C in line 16. The internal loop from line 5 to 15 iterates over every edge $\langle v, w \rangle$ of the CFG of f. The case where v is a block representing a function call is handled in lines 6 to 9, using either the summary of the called function or the external predicate. Otherwise, a formal model representing the block execution is generated in line 10, and used in the jump rule.

Definition 3 (Safety Rule). *The safety rule Σ_f for the CHC model of a public function f is $\bot \leftarrow \mathcal{C}(\boldsymbol{s}) \wedge \mathcal{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \wedge \tilde{r} \neq 0$. The safety rule of a contract C is the set Σ_C of the safety rules of every public function of C.*

The safety rule ensures that a function f is safe, in the sense that every possible transaction of f does not revert, i.e. produce assertion violations. A contract C is safe if and only if the set $\Pi_C \cup \Sigma_C$ is satisfiable.

3.5 Counterexample Generation

The *refutation*, or proof of unsatisfiability, for $\Pi_C \cup \Sigma_C$ proves that a specific safety query in Σ_C can not be satisfied, i.e., $\Delta_\bot$ is non-empty. While our solving methodology can show satisfiability over unbounded executions through the use of over-approximation, we can only represent finite counterexamples. This, of course, is not a practical limitation since in real programs we are only interested in bugs that manifest themselves after a finite number of steps. While the description of how a counter-example is constructed in our solver is outside of the scope of this paper, we give here a short overview of the refutations themselves.

A refutation is a tree-shaped structure obtained by an unwinding of clauses. The nodes of the refutation are labeled with clauses. The root v_0 of the tree is labeled with a clause with $\bot$ as head. For each predicate P in the body of a clause c, we create a child labeled with a unique clause c' such that $\mathit{head}(c') = P$. The leaves of the tree are labeled with clauses with no predicates in the body. Let $v_0, \ldots, v_k$ be a path from the root to a leaf, labeled with clauses $c_0, \ldots, c_k$. Given a clause c of form (1), let $\mathit{body}_\phi(c)$ denote the constraint ϕ of c. Then in a refutation for all such paths it must hold that

$$\models_T \mathit{body}_\phi(c_0)(\vec{x_0}, \vec{x_1}) \wedge \mathit{body}_\phi(c_1)(\vec{x_1}, \vec{x_2}) \wedge \ldots \wedge \mathit{body}_\phi(c_k)(\vec{x_{k-1}}, \vec{x_k}). \quad (2)$$

A counterexample corresponds then to a first-order structure satisfying (2) as follows: The counterexample generation traverses the entire refutation tree and considers only the nodes that refer to the initial state rule ($\mathsf{Init}_\mathcal{C}$), the root transaction rule ($\mathsf{RootTr}_{\mathcal{C},f}$), or the safety rule. The breath-first search results in a list of nodes that has the safety rule as first element (the root), a possibly empty list of elements representing root transaction rules, and finally a leaf representing an initial rule. The first-order structure satisfying (2) is used to produce a model of the initial state for the counterexample setup. Then, each following node represents the result of a transaction whose children model (i) the contract state prior the transaction, and (ii) a function call with given arguments that results in a new state. The last transaction involves a call to the function

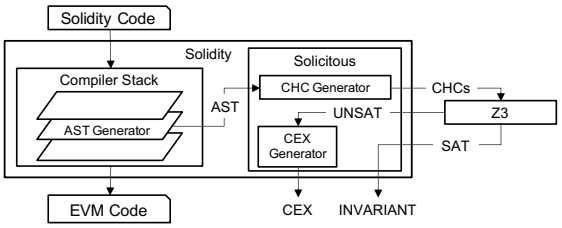

Fig. 1. Solicitous module inside the Solc compiler.

$\hat{f}$ that resulted in a revert. The arguments of each such function are then used to produce a trace of function calls which serves as the counterexample.

4 Implementation

Our approach is being implemented in collaboration with the engineers from the Ethereum Foundation, inside the SMTChecker component [2,8] of the Solidity compiler [18]. Specifically, the implementation of our work consists of the CHC model checking engine of SMTChecker, called Solicitous.

The Solicitous functionality can be enabled in the compilation by providing the corresponding *pragma* directive in the source file. Once enabled, the compiler provides the main Abstract Syntax Tree (AST) to Solicitous that generates the CHC model of the contract following Alg. 1. The CHC model is then provided to the engine Spacer [29] of the SMT solver Z3 [35] for solving. In case an assertion failure is detected, Solicitous can provide a transaction trace as a witness to the failure, which can easily be checked by the developer. An overview of Solicitous and Solidity can be seen in Fig. 1.

The emphasis of this paper is in the modelling of the control flow of Solidity contracts. The control flow corresponds to AST nodes related to language constructs such as loops and conditional branches. Visiting these nodes triggers the creation of the corresponding clauses as described in Sec. 3. In addition, the AST nodes corresponding to Solidity expressions result in accumulating the constraint ϕ of the clauses. Each expression node introduces a new SMT variable of the type of the expression. As an implementation detail, the unique identifiers the compiler assigns to AST nodes are used for guaranteeing unique names for these variables.

Solidity offers two special types of functions: *modifiers* and *constructors*. Modifiers represent pieces of code that envelope a function body. Therefore, modifiers' definitions depend on the functions they envelope, and they are not encoded separately but instead in-lined to the functions. Constructors define the initialization procedure executed at deployment time of a contract. The constructor modeling is prepended by providing the initialization $I(\boldsymbol{s})$ where variables are either zeroed or given their explicit initial values. In contracts that inherit base classes, the inheritance order is obtained by the Solidity compiler using the C3

linearization [9]. In addition, each constructor is executed exactly once. In our implementation, the entire deployment procedure, which might include the inheritance linearization and state variable initialization, is in-lined into a single constructor function.

Solicitous currently supports a working subset of the Solidity language, including the complex control flow and arithmetic operators (except exponentiation), integers of all available sizes, Boolean variables, arrays, mappings access and assignment, 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 and the addition of the remaining language features is a goal of the Ethereum Foundation, and the supported subset of language is therefore expected to grow.

5 Experiments

We evaluate the precision and language coverage of Solicitous on a set of real-world contracts from a 17 month period, between the block 7 million, mined 2nd of January 2019 and the block 10 million, mined on 4th of May 2020. We took all contracts in that period that are written in Solidity v0.5 and v0.6, and are available through the Etherscan block explorer [1]. The benchmarks are available at `https://scm.ti-edu.ch/repogit/verify-solidity-contracts.git`.

We queried 1147850 addresses and obtained 136802 contract sources, of which 27887 are unique: 367 v0.6, 10301 v0.5, and 17219 of previous versions. We run the tools only on contracts containing assertions. However, we checked also assertions that were commented out. We believe that commented assertions are of special interest because developers might have removed them before deployment in order to reduce gas cost, believing them to always hold. In total, we obtained 6061 v0.5 contracts including 11076 assertions (the V5 benchmark set), and 77 v0.6 contracts including 163 assertions (the V6 benchmark set).

We compare Solicitous[4] against three other tools: Solc-Verify [25,24] and VeriSol [30] that verify Solidity source code, and Mythril [15] that verifies EVM bytecode. Mythril differs from the other tools in that it is a purely bounded checking engine of three transactions. Unlike Solicitous, Solc-Verify and VeriSol, Mythril does not produce safe inductive invariants, and contracts Mythril reports safe can be considered safe only up to three transactions after contract deployment. In this sense Mythril can report only unsafe results, and only if a counterexample within three transactions exists. It is also hard to make claims about the validity of its counterexamples, as Mythril authors do not provide any scientific publication that explains their technique. Despite its limitations, Mythril is well known in the smart contracts community for having the best support for language features. In our comparative analysis, Mythril serves as a gold standard for the language support metric. To the best of our knowledge these

[4] Available at `https://github.com/usi-verification-and-security/solc`

tools are the only ones with which an automated comparison is possible.[5] Both Solc-Verify and VeriSol support only Solidity v0.5, thus for the comparative analysis using V5 we use a legacy version of Solicitous supporting v0.5 that has no support for counterexample generation. Solc-Verify, VeriSol and legacy Solicitous are sound but over-approximative. Specifically, while safe results are justified in these tools by an inductive invariant that proves safety, the tools do not justify unsafe results: in particular they do not provide an execution that would serve as a counterexample for the validity of an assertion. Therefore we distinguish between 'not safe' and 'unsafe', using the former when no or spurious counterexample is produced and the latter when a concrete counterexample proves a real bug. We separately evaluate the current Solicitous implementation using V6 to assess the concrete counter-example generation for proving unsafe results.

5.1 Counterexample Generation

The overall results for the V6 benchmark set is shown in Table 1. We run Solicitous with two different types of encodings where integer arithmetic is encoded both without and with modularity. The former allows arbitrarily large values, while the latter models overflow and underflow precisely. Mythril reports 13 safe contracts up to three transactions. Solicitous performs the best over this benchmark set, not only guaranteeing a good number of contracts to be safe, but also supporting the language features present in most contracts. The counterexamples of the 7 unsafe contracts reported by Solicitous were all checked to be concrete with the Ethereum evaluator HEVM [19]. Every counterexample leads to a runtime exception. Despite the small number of benchmarks due to Solidity v0.6 being very recent at the time of writing, our results show that Solicitous is capable of generating valuable witnesses of assertion failures that can help developers to prevent vulnerabilities.

In addition to its standard execution, in which a potential assertion failure is reported by mentioning its line number in the source file, Solicitous is also capable of generating concrete counterexamples to prove that the result is unsafe and not spuriously reported not safe due to the over-approximations of unsupported features. Unlike the fixed-size bounded approach of VeriSol and Mythril, Solicitous generates counterexamples of arbitrary length, reporting assertion failures that can happen at any point in the lifecycle of a contract.

5.2 Comparative Analysis

To get a better understanding of Solicitous performance on a larger benchmark set, we evaluated the 0.5 version of Solicitous, Solc-Verify, and VeriSol on V5. The results are shown in Table 2. Safe contracts are those for which all the assertions in the code are proved safe by safe inductive invariants. Not safe contracts have at least one assertion that is not proven safe. The timeout of each individual

[5] We considered two other tools for the comparison, namely Zeus [28] and SAFEVM [7], but Zeus is not publicly available and SAFEVM only supports Solidity v0.4.

Table 1. Experimental results for the V6 benchmark set. INT and MOD stand for integer and modulo arithimetics. SOL and M respectively stand for Solicitous and Mythril. Verified shows the percentage of contracts with either Safe or Unsafe result.

	INT	MOD	
	SOL	SOL	M
Safe	32	27	–
Unsafe	7	7	1
Timeout	5	9	63
Error	33	34	0
Verified	50%	44%	18%

Table 2. Experimental results for the V5 benchmark set. INT and MOD stand for integer and modulo arithimetics. SOL, SV, VS, and M respectively stand for Solicitous, Solc-Verify, VeriSol and Mythril. The Verified row shows the percentage of contracts reported either Safe or Not safe. The best result in each category is highlighted. * These numbers refer to unsafe reports proved by a concrete counterexample.

	INT			MOD			
	SOL	SV	VS	SOL	SV	VS	M
Safe	**1720**	778	135	**1681**	54	117	–
Not safe	142	572	298 (46*)	93	515	198 (31*)	23*
Timeout	586	89	**37**	678	**56**	130	5426
Error	3613	4622	5591	3609	5436	5616	**33**
Verified	**30%**	22%	7%	**29%**	9%	5%	9%

verification run is 60 seconds. Verification tasks halted for various types of errors are counted in the error row.

Solicitous reports the largest amount of safe inductive invariants for both arithmetic encodings. Regarding the not safe results, Solicitous can indistinguishably produce spurious and concrete results depending on whether unsupported features are present or not, since they are modelled as non-deterministic operations in order to preserve soundness. Similarly, Solc-Verify introduces over-approximations during its translation to Boogie that produce the same effect. VeriSol presents the same issue, however if no invariant is found it performs a further step creating a bounded model of length four. If the bounded check reports unsafe, VeriSol produces a concrete counterexample. In summary, VeriSol can prove an assertion unsafe only if it can fail within four transactions after contract deployment. The unsafe reports proved by a concrete counterexample are shown with an asterisk in Table 2.

The table also provides a comparison against Mythril. Due the tool limitations, the number of contracts reported safe (579) is not reported in Table 2. Our

experiments show that Solicitous is the tool that guarantees the largest amount of contracts to be safe, and that it is also the one able to verify the largest amount of contracts in general. Regarding the coverage of language features, using the amount of errors as a proxy metric, we see that Mythril possesses the best support. Solicitous is closer to it than Solc-Verify or VeriSol. Given the positive results, aligned with the practical nature of the benchmarks set used, Solicitous stands as a valuable tool for Solidity developers.

6 Related Work

There is much interest in formally verifying Ethereum smart contracts, and several tools rely on different techniques to verify either Solidity or Vyper source code, or EVM bytecode. Oyente [31] is one of the pioneers in this field, and uses symbolic execution of EVM bytecode to find common vulnerabilities. Mythril [15] is a security tool based on control-flow analysis and concolic execution of EVM, supporting analysis of assertions up to a fixed bound of transactions. MAIAN [36] is also bounded in the number of transactions and searches EVM bytecode for three specific types of vulnerabilities. Securify [39] encodes EVM bytecode into Datalog to analyze programs, targeting specific types of bugs encoded as data patterns. VerX [37] verifies temporal properties written using a specification language for a particular class of contracts referred as *effectively external callback free*. It requires user intervention when the automatic inference of abstraction predicates fails. The tool is not publicly available. Manticore [34] has a symbolic execution engine for EVM that uses SMT to systematically explore the state space of the contract by repeatedly executing *symbolic transactions*. KEVM [26] is a formal specification of the EVM semantics written in the K-framework [38]. It provides an assisted theorem prover and a specification language for further analysis, including reachability. Similarly, KVyper [22] and KSolidity [21] are the Vyper and Solidity semantics expressed over the the K-framework. KLAB [16] provides a specification language tailored for smart contracts that compiles to general K properties and a framework for proof debugging and counterexample analysis based on KEVM. SAFEVM [7] verifies EVM code produced by Solidity 0.4 through an intermediate translation to C that can be checked with three different backend C-verifiers. Zeus [28] translates Solidity into LLVM bitcode which is fed to the SeaHorn [23] model checker. A subset of Solidity not including loops is verified after a translation to F* [10]. Why3 [40] has also been used to verify translated Solidity programs. However, Why3 does not support many of the Solidity constructs and is no longer developed. Slither [20] translates Solidity to its own intermediate SSA language and performs bounded checks for several vulnerability classes. More recently, the tools Solc-Verify [25] from SRI and VeriSol [30] from Microsoft verify Solidity contracts using the language Boogie as intermediate representation. The estimation of gas consumption in order to cope with gas-related vulnerabilities is considered in [32].

7 Conclusions

We presented a formal technique for modeling smart contracts using CHCs. The constructed models (i) formally capture semantic features specific to smart contracts, (ii) enable fully-automated verification of safety properties, and (iii) are suitable for exploiting generic theorem provers in the task of analysis and contract invariants generation. We implemented our technique for the Solidity language and demonstrated its effectiveness through an extensive experimentation involving 6138 contracts specifying 11239 safety properties. Based on these experiments we believe that our technique represents an effective, highly promising avenue for smart contract verification.

Acknowledgements. The authors would like to thank Enrique Fynn and Fernando Pedone for their kind assistance in providing us with the addresses for the deployed Ethereum contracts used in the experiments. This work is partially supported by the SNSF grant 200021_185031 and by the ERC grant FP7-617805.

References

1. Etherscan, `https://etherscan.io`
2. Smtchecker documentation, `https://solidity.readthedocs.io/en/v0.6.6/security-considerations.html#formal-verification`
3. Solidity documentation, `https://solidity.readthedocs.io`
4. theDAO, `https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413`
5. Vyper documentation, `https://vyper.readthedocs.io`
6. Parity security alert (2017), `https://www.parity.io/security-alert-2/`
7. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: SAFEVM: a safety verifier for ethereum smart contracts. In: Proc. ISSTA 2019. pp. 386–389
8. Alt, L., Reitwiessner, C.: SMT-based verification of solidity smart contracts. In: Proc ISoLA 2018. pp. 376–388. Springer, Cham
9. Barrett, K., Cassels, B., Haahr, P., Moon, D.A., Playford, K., Withington, P.T.: A monotonic superclass linearization for dylan. In: Proc. OOPSLA 1996. pp. 69–82
10. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Proc. PLAS 2016. pp. 91–96
11. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II. pp. 24–51 (2015)
12. Blass, A., Gurevich, Y.: Existential fixed-point logic. In: Computation Theory and Logic, In Memory of Dieter Rödding. pp. 20–36 (1987)
13. Blicha, M., Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: A cooperative parallelization approach for property-directed k-induction. In: Proc. VMCAI 2020. LNCS, vol. 11990, pp. 270–292. Springer
14. Bradley, A.R.: Sat-based model checking without unrolling. In: Proc. VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer
15. ConsenSys: Mythril (2018), `github.com/ConsenSys/mythril`
16. Erfurt, D., Lundfall, M., Hildenbrandt, E., Livnev, L.: Klab (2020), `https://github.com/dapphub/klab`

17. Ethereum Foundation: Ethereum: A secure decentralised generalised transaction ledger (2018), ethereum.github.io/yellowpaper/paper.pdf
18. Ethereum Foundation: Solidity compiler (2018), github.com/ethereum/solidity
19. Ethereum Foundation: HEVM Ethereum evaluator (2020), https://github.com/dapphub/dapptools/tree/master/src/hevm
20. Feist, J., Grieco, G., Groce, A.: Slither: A Static Analysis Framework For Smart Contracts. arXiv e-prints arXiv:1908.09878 (Aug 2019)
21. Framework, K.: Solidity semantics (2018), https://github.com/kframework/solidity-semantics
22. Framework, K.: Vyper semantics (2018), https://github.com/kframework/vyper-semantics
23. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Proc. CAV 2015. pp. 343–361. Springer
24. Hajdu, Á., Jovanovic, D.: SMT-friendly formalization of the solidity memory model. In: Proc. ESOP 2020. LNCS, vol. 12075, pp. 224–250. Springer
25. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. CoRR abs/1907.04262 (2019)
26. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In: Proc. CSF 2018. pp. 204–217
27. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
28. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Proc. NDSS 2018. The Internet Society
29. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods System Design. 48(3), 175–205 (2016)
30. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for azure blockchain. CoRR abs/1812.08829 (2018)
31. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making Smart Contracts Smarter. In: Proc. CCS 2016. pp. 254–269. CCS '16, ACM
32. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: Proc. ISoLA 2018. LNCS, vol. 11247, pp. 450–465. Springer (2018)
33. Marescotti, M., Gurfinkel, A., Hyvärinen, A.E.J., Sharygina, N.: Designing parallel PDR. In: Stewart, D., Weissenbacher, G. (eds.) Proc. FMCAD 2017. pp. 156–163. IEEE
34. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. CoRR abs/1907.03890 (2019)
35. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS 2008. LNCS, vol. 4963, pp. 337 – 340. Springer
36. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. CoRR abs/1802.06038 (2018)
37. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: safety verification of smart contracts. In: Proc. IEEE SSP 2020, to appear
38. Rosu, G., Serbanuta, T.F.: An Overview of the K Semantic Framework. The Journal of Logic and Algebraic Programming 79(6), 397 – 434 (2010)
39. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical Security Analysis of Smart Contracts. In: Proc. CCS 2018. pp. 67–82. ACM
40. Why3: Why3 (2018), why3.lri.fr