Behavioral Simulation for Smart Contracts

Sidi Mohamed Beillahi*
Université de Paris, IRIF, CNRS
Paris, France
beillahi@irif.fr

Gabriela Ciocarlie
SRI International
New York, USA
gabriela.ciocarlie@sri.com

Michael Emmi
SRI International
New York, USA
michael.emmi@gmail.com

Constantin Enea
Université de Paris, IRIF, CNRS, Institut Universitaire de France (IUF)
Paris, France
cenea@irif.fr

Abstract

While smart contracts have the potential to revolutionize many important applications like banking, trade, and supply-chain, their reliable deployment begs for rigorous formal verification. Since most smart contracts are not annotated with formal specifications, general verification of functional properties is impeded.

In this work, we propose an automated approach to verify unannotated smart contracts against specifications ascribed to a few manually-annotated contracts. In particular, we propose a notion of *behavioral refinement*, which implies inheritance of functional properties. Furthermore, we propose an automated approach to inductive proof, by synthesizing simulation relations on the states of related contracts. Empirically, we demonstrate that behavioral simulations can be synthesized automatically for several ubiquitous classes like tokens, auctions, and escrow, thus enabling the verification of unannotated contracts against functional specifications.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Automated reasoning**.

Keywords: Blockchain, Smart contracts, Refinement, Simulation

ACM Reference Format:
Sidi Mohamed Beillahi, Gabriela Ciocarlie, Michael Emmi, and Constantin Enea. 2020. Behavioral Simulation for Smart Contracts.

*This work was conducted during an internship at SRI International.

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386022

In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 17 pages.
https://doi.org/10.1145/3385412.3386022

1 Introduction

Smart contracts are programs that execute on cryptographically-secure distributed ledgers, i.e., *blockchains*, to perform trackable and irreversible transactions. As they offer autonomy for arbitrarily-complex transactions between multiple parties, smart contracts are already powering a sizable economy: applications include banking [4], insurance [2], auction, trade, and supply chain. This rapid adoption has been closely followed by exploitation, including millions of US dollars lost due to vulnerabilities in smart contract code [1, 3, 5].

Formal verification has the potential to mitigate such exploitation significantly. However, scaling verification efforts to a large number of smart contracts is an important challenge. In particular, while specifications that are *specialized* to each individual smart contract are useful for proving customized functional properties [55], *generic* specifications that can be applied to large classes of smart contracts would facilitate verifying contracts en masse. Ideally, the specification for a given class of smart contracts could be written once, and reused for the verification of each contract of that class.

Truly generic specifications must be sufficiently weak so that every correct contract in the given class adheres to its functional properties. Moreover, truly generic specifications must be independent from the state variables of any particular contract, since the state variables of other contracts in the same class generally differ in name, number, and type. Such generic specifications are however unsuited for existing verification tools like solc-verify [36] and VerX [55], which suppose that input contracts are annotated with expressions that refer to state variables, e.g., pre- and post-conditions. This poses a scalability problem since deriving such annotations for each contract from class-wide generic specifications would be a manual labor-intensive process.

In this work, we address this scalability challenge and introduce an approach for verifying *unannotated* smart contracts via automated semantic comparison against *annotated*

smart contracts. Our approach is motivated by the insight that many of the smart contracts instantiated on popular blockchains (e.g., Ethereum [72]) are variations on a relatively small number of canonical contracts and libraries implementing concepts like tokens, ownership, auctions, and escrow. Intuitively, many of these variations obey the principle of *substitutability* [43], meaning that they adhere to the functional properties captured by the annotations of their canonical counterparts.

With a notion of comparison that implies substitutability, we can thus amortize the cost of manually annotating these canonical contracts by verifying a vast number of unannotated contracts. Our notion of *behavioral refinement* relates the input-output behavior of contracts' transactions, i.e., parameters and effects on storage, ignoring internal details like local memory and control flow. By proving that a given contract is a behavioral refinement of another, we guarantee the inheritance of behavioral properties, and in particular that the effects of any sequence of transactions obeys its canonical counterpart's functional properties.

Establishing behavioral refinement for unbounded transaction sequences relies on induction. Akin to inductive invariants for safety properties, proofs of behavioral refinement use induction hypotheses called *simulation relations* [46]. Essentially, a behavioral simulation relation identifies states of two contracts such that initial states are related; the same transaction applied to related states yields related states and identical effects; and related states are *observationally equivalent*, i.e., any function applied to both yields identical values. While simulation relations are known to be incomplete for establishing refinement [45], they have the advantage of guaranteeing inheritance of its canonical counterpart's *hyperproperties* [8, 10]. Hyperproperties [19], described as sets of sets of behaviors (as opposed to standard safety/liveness properties which can be thought of as sets of behaviors), capture many interesting security properties, e.g., the classic noninterference [32], which are relevant in the context of smart contracts [34, 65].

Accordingly, our automated verification approach boils down to the synthesis of behavioral simulation relations in two steps: a passive learning step generates candidate simulation relations, and a deductive verification step checks the validity of each candidate. Candidate generation can be demand-driven according to *counterexample guided inductive synthesis* [61], e.g., initially proposing the trivial simulation relating each pair of contract states, and incrementally proposing candidates which rule out spurious counterexamples from prior validation steps. In this work, we consider a simplification which suffices empirically: generating and validating one single candidate simulation.

To generate candidate simulation relations, we adopt a paradigm of learning from examples [47]. In this work, we consider only *passive* learning, which assumes that the set of examples is fixed a priori, as opposed to being generated on learner demand. In the context of simulation, an example is a pair of states, i.e., one state of each contract: positive examples are pairs of states which must be similar, and negative examples are pairs which must not be similar. To generate examples, we consider sequences of transactions, executed on a blockchain, starting from the initial states of each contract. Intuitively, negative examples correspond to pairs of states which yield distinct observations, and positive examples correspond to pairs of states reached by identical transaction sequences, unless such a pair yields distinct observations, in which case it is a counterexample to simulation. We then leverage off-the-shelf learning algorithms [54] by providing an oracle to evaluate candidate expressions against pairs of states, i.e., by executing such expressions on the blockchain.

To verify candidate simulation relations, we adopt a notion of product programs inspired by relational verification [13]. In particular, we generate an auxiliary *simulation-checking* contract whose verification implies the validity of a given simulation relation. Intuitively, for each function f of the given unannotated contract, the simulation-checking contract provides a function which executes f in lockstep with its canonical contract's counterpart. Besides asserting the equality of effects and return-values, this function includes the candidate simulation as pre- and post-conditions, ultimately implying inductiveness. We verify the simulation-checking contract using an existing verifier [36], which translates Solidity smart contracts to Boogie programs [11], and ultimately to satisfiability modulo theories (SMT) queries.

Empirically, we validate our approach by collecting dozens of Solidity-language smart contracts, identifying canonical contracts for several classes, annotating and verifying these canonical contracts with precise formal specifications, and synthesizing simulation relations from multiple variations of each class. Our implementation can correctly synthesize nontrivial simulation relations for many classes, and integrates off-the-shelf tools for example-guided learning and Solidity verification.

In summary, this work makes the following contributions:

- We demonstrate an application of behavioral simulation to smart contracts (§3-4).
- We develop an algorithm for synthesizing behavioral simulation relations (§5-6).
- We develop a smart contract benchmark suite including variations of identified canonical contracts (§8).
- We evaluate our approach, verifying functional properties for dozens of unannotated smart contracts (§9).

Aside from the aforementioned technical sections, we outline our approach in Section 2, and discuss related work and conclusions in Sections 10 and 11.

2 Overview

In this section, we overview the methodology formalized in Sections 3-6 for synthesizing behavioral simulations. We

illustrate behavioral refinement on a running example (§2.1), describe behavioral simulation for proving refinement (§2.2), and demonstrate synthesis on the running example (§2.3).

2.1 Motivation

We illustrate the concept of behavioral refinement on two contracts implementing an auction (written in the Solidity language of Ethereum), which are partially listed in Figure 1. These excerpts focus on the initialization and the bidding parts of an auction. The contract `RefAuction`[1] plays the role of an annotated canonical implementation of an auction (we omit the exact postconditions for brevity) while `Auction` is a particular variation. We generally refer to canonical implementations as *reference (smart) contracts* while variations like `Auction` are called simply *(smart) contracts*.

The fields of `RefAuction` store information about the beneficiary and the ending time of the auction, the current highest bidder and its bid, and the bids of previous highest bidders (the owners of these bids have the right to reclaim them at any point during the auction – for brevity, this functionality is excluded from these excerpts). While the constructor initializes the beneficiary and the ending time of the auction, the `bid` function allows a participant to pose a new bid which is accepted only if it is bigger than the current highest bid and the timeout did not expire. Otherwise, the `bid` function has no effect on the state of the contract – if the condition inside a `require` statement fails, the invocation is *reverted* and is semantically equivalent to skip. This contract also contains several functions that allow to read its fields, in particular a bid that has been superseded by a higher one (function `PendingReturns`) and the highest bid.

The contract `Auction` is a variation that changes the representation of the auction ending time decomposing it into an auction start time and a bidding duration. The handling of revert conditions in the `bid` function is syntactically distinct, but semantically equivalent to the `require` in `RefAuction`.

Despite syntactic and state representation differences, every sequence of transactions calling methods of `Auction` has the same effect as if they were calling `RefAuction` instead. This relationship can be stated as `Auction` being a *behavioral refinement* of `RefAuction`, i.e., that its behaviors are subsumed by `RefAuction`. We use the term *behavior* to refer to a summary of the inputs and outcomes, e.g., return values, of a sequence of invocations.

Behavioral refinement is consistent with Liskov's substitutability principle [43], i.e., any contract can be replaced with any of its refinements in any context, as long as a behavior records all the outcomes (effects) which are observable in a context. For the sake of this example, we will focus on return values. Other observable effects which are relevant in a Blockchain environment, e.g., changes on the state of other contracts or Blockchain global variables like the balances of external accounts, are discussed in Section 3.

For instance[2],

$$\texttt{constructor(5, a)} \cdot \texttt{bid(b, 20)} \cdot \texttt{bid(c, 30)} \cdot \texttt{HighestBid()} \Rightarrow 30$$

$$\texttt{constructor(5, a)} \cdot \texttt{bid(b, 20)} \cdot \texttt{bid(c, 10)} \Rightarrow \bot \cdot \texttt{HighestBid()} \Rightarrow 20$$

are two possible behaviors of `Auction` which are also possible when calling methods of `RefAuction` instead (we use the $\bot$ return value to signal a reverted `bid` invocation). More generally, refinement holds because the conditions under which a new bid is accepted are semantically the same even though the two contracts use different representations of the ending time. The constructors of these contracts ensure that the two representations are "consistent" in the sense that

$$\texttt{auctionEnd} = \texttt{auctionStart} + \texttt{biddingTime} \tag{1}$$

which implies that the timing conditions in function `bid` are equivalent. Note that even though `bid` has no return value, using different conditions for accepting a bid would have been "observable" because of the "getter" method that allows to read the highest bid at any point during an execution.

2.2 Behavioral Simulation Relations

Establishing refinement usually relies on an induction argument based on a *(behavioral) simulation relation*, which in our context, is a relation between the states of the two contracts supporting a proof that the reference contract mimics every method invocation of the other contract. The simulation relation supporting such a proof is defined as follows (the fields of `Auction` are prefixed by # to distinguish them from fields of the reference auction having the same name):

$$\begin{aligned} \mathsf{Sim} \stackrel{\text{def}}{=}\ & \texttt{auctionStart} + \texttt{biddingTime} = \texttt{auctionEnd} \\ \wedge\ & \texttt{\#beneficiary} = \texttt{beneficiary} \wedge \texttt{\#highestBidder} = \texttt{highestBidder} \\ \wedge\ & \texttt{\#highestBid} = \texttt{highestBid} \wedge \texttt{\#pendingReturns} = \texttt{pendingReturns} \end{aligned} \tag{2}$$

This states that the fields recording bids and the beneficiary are the same in the two contracts (`pendingReturns` fields are equal when they have the same mappings), while the fields concerning the ending time are related as mentioned above (Equation 1). Environment (global) variables like `now` are assumed to be equal in any two states related by `Sim`. This models the fact that the two contracts refine one another when embedded in the same context (where the environment variables evolve in the same way). To simplify the technical exposition, we omit them from simulation relations. Also, in this section, we ignore the issues related to `bid` being a payable function which transfers Ether.

The initial states of the two contracts (produced after executing the constructor) are obviously related by Sim and also, given any two states (of `Auction` and `RefAuction`, respectively) related by Sim, executing an arbitrary invocation in

[1] Extracted from the documentation page of Solidity [62].

[2] For `bid` invocations, the caller identity and the amount of Ether it sends are written as explicit arguments, and the return value of an invocation (e.g., to `HighestBid()`) is written after $\Rightarrow$. Also, we use small cap letters `a`, `b`, `c` to represent values of type `address`.

```
// @notice invariant ...
contract RefAuction {
  uint public auctionEnd, highestBid;
  address payable public beneficiary;
  address public highestBidder;
  mapping(address => uint) pendingReturns;

  // @notice postcondition ...
  constructor(uint _bidTime, address payable _benefic) public {
    beneficiary = _benefic;
    auctionEnd = now + _bidTime;
  }
  // @notice postcondition ...
  function bid() public payable {
    require(now <= auctionEnd && msg.value > highestBid);
    if (highestBid != 0)
      pendingReturns[highestBidder] += highestBid;
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
  // @notice postcondition ...
  function PendingReturns() public view returns (uint) {
    return pendingReturns[msg.sender];
  }
  // @notice postcondition ...
  function HighestBid() public view returns (uint) {
    return highestBid;
  }
}
```

```
contract Auction {
  uint public auctionStart, biddingTime, highestBid;
  address payable public beneficiary;
  address public highestBidder;
  mapping(address => uint) pendingReturns;

  constructor(uint _bidTime, address payable _benefic) public {
    beneficiary = _benefic;
    auctionStart = now;
    biddingTime = _bidTime;
  }

  function bid() public payable {
    if (now > auctionStart + biddingTime || msg.value <= highestBid)
      revert();
    if (highestBidder != address(0))
      pendingReturns[highestBidder] += highestBid;
    highestBidder = msg.sender;
    highestBid = msg.value;
  }

  function PendingReturns() public view returns (uint) {
    return pendingReturns[msg.sender];
  }

  function HighestBid() public view returns (uint) {
    return highestBid;
  }
}
```

Figure 1. A canonical auction contract (left) and a variation (right), omitting withdrawal, auction-ending, and other view functions. Implicit variables now, `msg.value`, and `msg.sender` yield block timestamps, Ether sent, and callers' addresses.

`Auction` can be mimicked by an invocation in `RefAuction` of the same method with the same arguments and return values. Moreover, the states reached at the end of the two invocations are again related by Sim. The latter enables an extension of this proof to an arbitrary number of invocations.

The existence of this simulation relation implies that `Auction` is a behavioral refinement of `RefAuction`, which implies that it satisfies any property of `RefAuction` characterizing its behaviors. Even more, since the simulation relates the states of the two contracts, it also supports deriving valid-by-construction inductive invariants or pre/post-condition annotations for methods. For instance, an inductive invariant of a reference contract (that holds before and after every method invocation) can be used to define a valid-by-construction inductive invariant for any contract that it simulates. In the context of our running example, the following inductive invariant of the reference auction

$$\mathsf{Inv} \stackrel{\text{def}}{=} \forall i.\mathtt{pendingReturns}[i] \leq \mathtt{highestBid}$$

implies that $\mathsf{Sim} \wedge \mathsf{Inv}$ is an inductive invariant for `Auction`.

2.3 Simulation Relation Synthesis

We propose methodology for synthesizing such simulation relations automatically that consists of two parts: a learning procedure for guessing simulation relation candidates from examples (§2.3.1), and using deductive verification for establishing the validity of the inferred candidates (§2.3.2).

2.3.1 Learning Simulations From Examples.

To generate candidate simulation relations we use a procedure based on learning from examples, where the goal is learning a (first-order) formula that "separates" a set of positive examples from a set of negative examples, i.e., satisfied by all positive examples and falsified by all negative ones. In our context, examples are pairs of states of the contract and reference contract, respectively. The positive examples must be included in any simulation relation while the negative ones must be excluded from any simulation. Classifying examples as positive or negative enables the re-use of any existing learning algorithm that can produce formulas separating between the two, e.g., [27, 28, 54, 57, 59].

The positive examples are pairs of states obtained by executing the *same* sequence of invocations (with the same arguments) from the initial state of both the contract and the reference contract. Such pairs of states are necessarily included in every simulation under the assumption that contracts are deterministic, which roughly, means that the state reached by a contract when executing a sequence of invocations is unique. These two auction contracts satisfy this determinism assumption (this is rather straightforward when the global variable now is assumed to be a constant; otherwise, it is required that any modification of the environment variable now is modeled explicitly as an invocation to a fictitious method – see Section 3 for more details). For instance, the following pair of states is obtained by running $\mathtt{constructor}(2,\mathsf{a}) \cdot \mathtt{bid}(\mathsf{b},10) \cdot \mathtt{bid}(\mathsf{c},20)$ in both contracts

(we write only the keys of pendingReturns that changed with respect to the initial state):

$$\left(\begin{bmatrix}\text{beneficiary} = \text{a}\\ \text{now} = 0\\ \text{auctionStart} = 0\\ \text{biddingTime} = 2\\ \text{highestBid} = 20\\ \text{highestBidder} = \text{c}\\ \text{pendingReturns[b]} = 10\end{bmatrix}, \begin{bmatrix}\text{beneficiary} = \text{a}\\ \text{now} = 0\\ \text{auctionEnd} = 2\\ \\ \text{highestBid} = 20\\ \text{highestBidder} = \text{c}\\ \text{pendingReturns[b]} = 10\end{bmatrix}\right) \tag{3}$$

We generate positive examples by enumerating invocation sequences and producing the pairs of states reached by executing them in the two contracts.

The definition of negative examples relies on a relation between states which compares return values of read-only methods. As a base case, a negative example is any pair of states that are distinguished by a read-only method, i.e., invoking this method on each of the two states results in different return values. For instance, the following pair of states are distinguished by the HighestBid method:

$$\left(\begin{bmatrix}\text{beneficiary} = \text{a}\\ \text{now} = 0\\ \text{auctionStart} = 0\\ \text{biddingTime} = 2\\ \text{highestBid} = 20\\ \text{highestBidder} = \text{c}\\ \text{pendingReturns[b]} = 10\end{bmatrix}, \begin{bmatrix}\text{beneficiary} = \text{a}\\ \text{now} = 0\\ \text{\_auctionEnd} = 2\\ \\ \text{highestBid} = 30\\ \text{highestBidder} = \text{b}\\ \end{bmatrix}\right) \tag{4}$$

The first state is obtained by calling $\texttt{constructor}(2, \text{a}) \cdot \texttt{bid}(\text{b}, \mathbf{10}) \cdot \texttt{bid}(\text{c}, 20)$ in the Auction contract while the second one is obtained by calling $\texttt{constructor}(2, \text{a}) \cdot \texttt{bid}(\text{b}, \mathbf{30}) \cdot \texttt{bid}(\text{c}, 20)$ in the reference auction. The difference between the two sequences, i.e., the argument to the second bid, is written in bold font (the last bid in the reference auction sequence is not accepted because it is smaller than the previous one). Such pairs of states should be excluded from any simulation relation because otherwise, the reference contract cannot mimic the invocation of that particular read-only method in the other contract. Going further, any pair of states from which executing the *same* sequence of invocations leads to states that are distinguishable by some read-only method is also a negative example (this again relies on the assumption that contracts are deterministic). For instance, the predecessors of the pair of states in (4), reached before making the last bid (i.e., $\texttt{bid}(\text{c}, 20)$), which is the same in both contracts, is such an example:

$$\left(\begin{bmatrix}\text{beneficiary} = \text{a}\\ \text{now} = 0\\ \text{auctionStart} = 0\\ \text{biddingTime} = 2\\ \text{highestBid} = 10\\ \text{highestBidder} = \text{b}\end{bmatrix}, \begin{bmatrix}\text{beneficiary} = \text{a}\\ \text{now} = 0\\ \text{\_auctionEnd} = 2\\ \\ \text{highestBid} = 30\\ \text{highestBidder} = \text{b}\end{bmatrix}\right)$$

As we hinted above, negative examples can also be identified based on invocation sequences, in this case two distinct ones. Therefore, their generation is oblivious to state representations and based on an enumeration of pairs of invocation sequences.

```
contract SimulationCheck is Auction, ReferenceAuction {

  // @notice postcondition Sim
  constructor(uint _biddingTime, address payable _beneficiary)
    Auction(_biddingTime, _beneficiary)
    ReferenceAuction(_biddingTime, _beneficiary) public { }

  // @notice precondition Sim
  // @notice postcondition Sim
  function checkBid() public payable {
    r0 = Auction.bid();
    r1 = ReferenceAuction.bid();
    assert (r0 == r1);
  }
}
```

Figure 2. Validating the simulation relation Sim.

Note that Sim in Equation 2 is indeed a separator between the examples described above.

2.3.2 Verifying Simulation Relations. To verify that a simulation candidate is indeed valid we rely on deductive verification. We generate a *simulation-checking* contract with one function for each of the functions common to the input contracts, invoking each version in turn. Figure 2 lists an excerpt of this contract for our running example. The inheritance mechanism ensures that each state of SimulationCheck is a disjoint union of a state of Auction and RefAuction, respectively. The simulation-checking contract lists the given candidate simulation relation, in this case Sim in Equation 2, as both a pre- and post-condition to each function (as well as a post-condition of the constructor), and asserts that both versions of each function yields the same results. Sim is a valid simulation relation if all the pre/post-conditions and assertions are satisfied by SimulationCheck.

This deductive verification step completes the proof that Auction is a behavioral refinement of RefAuction and that it inherits all its behavioral properties, e.g., a bid is accepted only if it is bigger than every previous bid.

3 Behavioral Refinement

The formalization of behavioral refinement between contracts relies on a simple yet universal model of computation, namely labeled transition systems. A *labeled transition system* (LTS) $A = (Q, \Sigma, s_0, \delta)$ over the possibly-infinite alphabet Σ is a possibly-infinite set Q of states with initial state $s_0 \in Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. The ith symbol of a sequence $\tau \in \Sigma^*$ is denoted τ_i, and ϵ is the empty sequence. An *execution* of A is an alternating sequence of states and transition labels (also called *actions*) $\rho = s_0, a_0, s_1 \ldots a_{k-1}, s_k$ for some $k > 0$ such that $\delta(s_i, a_i, s_{i+1})$ for each $0 \leq i < k$. We write $s_i \xrightarrow{a_i \ldots a_{j-1}}_A s_j$ as shorthand for the subsequence $s_i, a_i, \ldots, s_{j-1}, a_{j-1}, s_j$ of ρ. (in particular $s_i \xrightarrow{\epsilon} s_i$). The projection $\tau|\Gamma$ of a sequence τ is the maximum subsequence of τ over the alphabet Γ. This notation is extended to sets of sequences as usual. A *trace* of A is the projection $\rho|\Sigma$ of an

execution ρ of A. The set of traces of an LTS A is denoted by $T(A)$. An LTS is *deterministic* if for any state s and sequence $\tau \in \Sigma^*$, there is at most one state s' such that $s \xrightarrow{\tau} s'$.

A contract is interpreted as an LTS whose traces represent sequences of invocations to the contract's methods together with their inputs and observable outcomes. A typical example of an observable outcome is the return value, which can be read through invocations from other contracts. Other examples include effects like gas consumption, changes on the state of other contracts, changes on Blockchain global variables like the balances of external accounts, etc. To simplify the technical exposition, we will mostly focus on return values but this is not a limitation. This LTS interpretation is used to formalize and reason about the soundness of our methodology. It is not intended to be constructed explicitly.

Essentially, the states of the LTS are composed of an *internal* part represented as assignments to the contract's fields and the balance of the address at which the contract is deployed, and an *environment* part represented as assignments to environment variables, e.g., `now` in Figure 1, which influence the contract's behavior. The transitions represent invocations to the contract's methods or updates of the environment variables, e.g., increasing the value of `now`[3]. The labels record method names, arguments, and observable outcomes. For uniformity, updates of environment variables are modeled as invocations to some fictitious methods.

Formally, an *invocation label* $m(\vec{u})$ is a method name m along with a vector $\vec{u}$ of argument values. An *operation label* $\ell = m(\vec{u}) \Rightarrow v$ is an invocation label $m(\vec{u})$ along with a return value v. We assume a fixed, but unspecified, domain Vals of argument or return values. Vals includes a distinguished return value $\perp$ associated to invocations that revert. We use $\mathsf{inv}(\ell)$ to refer to the invocation label in an operation label ℓ. This notation is extended to sequences or sets of operation labels as expected. An *interface* Σ is a set of operation labels over a finite set of method names. We use $\Sigma^{\checkmark}$ to denote the subset of Σ that excludes operation labels with $\perp$ as a return value, and $\mathsf{Meths}(\Sigma)$ to denote the method names in Σ.

Definition 3.1. A *(smart) contract* is an LTS $C = (Q, \Sigma, s_0, \delta)$ over an interface Σ.

Example 3.2. Figure 3 pictures a fragment of the LTS interpretation of `RefAuction`. This LTS contains an initial state from where a number of transitions corresponding to constructor invocations are enabled. These different transitions correspond to different sets of arguments passed to the constructor. As mentioned above, a state of this LTS consists of a valuation of all the fields of `RefAuction`, e.g., `beneficiary` and `_auctionEnd`, the balance of the address at which this contract is deployed, written as `balance[this]`, and the environment variable `now`. Invocations that revert, e.g., bidding 10 when the highest bid is 20, marked using the $\perp$ return value, or invocations to read-only methods like `HighestBid` are represented as self-loops. Transitions also represent updates of `now`, e.g., increasing its value by 7.

[3]This LTS can be thought of as a composition between an LTS defining the evolution of the variables controlled by the contract and an LTS defining the evolution of the environment variables (whose states are valuations of these variables). The states of the two LTSs share the valuation of the environment variables (read by the first LTS and updated by the second).

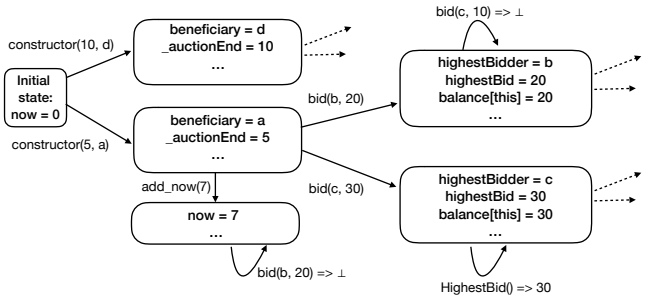

Figure 3. A fragment of the LTS interpretation of `RefAuction`. Boxes represent states and arrows represent transitions. The representation of states emphasizes the fields changed by the incoming transition.

Modeling updates of environment variables as *labeled* LTS transitions is important to ensure that the resulting LTS is deterministic. For instance, assuming that such updates are ϵ transitions in the LTS interpretation of `RefAuction` (Figure 3), the singleton sequence `constructor(5, a)` leads to two distinct states, where `now` = 0 and `now` = 7, respectively. Determinism is important for the soundness of our learning procedure (see Section 5).

Remark 3.1. The notion of contract in Definition 3.1 considers the return value as the only observable outcome of an invocation. This notion can be extended to include other observable effects by enriching the structure of transition labels. For instance, it is quite frequent that the methods of a contract invoke Solidity primitives like `send` for transferring Ether, or methods of other contracts, and possibly even check their return values. An invocation to such a method m can be represented by a transition labeled by $m(\vec{u}) \Rightarrow I, v$ where $\vec{u}$ and v are the arguments and return value of this invocation, and I is a sequence of operation labels corresponding to the "internal" calls made during this invocation (e.g., a call to `send` with its arguments and return value).

The standard refinement relation between two LTSs is defined as the inclusion between the set of traces produced by the two LTSs. For practical reasons, we consider an extension of this notion that allows a contract to refine another even if (1) it has a larger interface (it defines a larger set of methods) or (2) invocations revert more often (this is sound since any update of a reverted invocation is discarded).

For instance, Figure 4 lists several functions of a contract called `PAX`[4] that allows an owner to mint some set of tokens

[4]A variation of a contract extracted from [64].

```
contract PAX {
  mapping(address => uint256) public balances;
  address public owner;
  constructor() public {
    owner = msg.sender;
  }
  function mint(address to, uint val) public {
    require(msg.sender == owner);
    balances[to] = balances[to] + val;
  }
  function transfer(address to, uint val) public {
    require(val <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - val;
    balances[to] = balances[to] + val;
  }
  function transferOwnership(address _newOwner) public {
    require(msg.sender == owner);
    owner = _newOwner;
  }
}
```

Figure 4. A contract managing a set of tokens.

```
contract ERC20 {
  mapping(address => uint256) public balances;
  function mint(address to, uint val) public {
    balances[to] = balances[to] + val;
  }
  function transfer(address to, uint val) public {
    require(val <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - val;
    balances[to] = balances[to] + val;
  }
}
```

Figure 5. An excerpt from the ERC20 reference contract.

for some specific address (function `mint`), transfer tokens between different addresses (function `transfer`), and change ownership (function `transferOwnership`). Also, Figure 5 lists an excerpt from the ERC20 reference contract in OpenZeppelin [53]. PAX defines the method `transferOwnership` that does not occur in ERC20 and the method `mint` in PAX can revert if it is not called by the owner while its counterpart in ERC20 can not. We consider PAX to be a refinement of ERC20 because any sequence of non-reverted invocations to methods of PAX that exist in both is admitted by ERC20 as well (when looking only at arguments and return values as in the LTS interpretation). This extension of the notion of refinement also allows that a contract is a refinement of several reference contracts. For instance, PAX is also a refinement of the contract `Ownable` from the OpenZeppelin library, which implements an ownership mechanism.

Definition 3.3. A contract C_1 over interface Σ_1 *refines* another contract C_2 over interface Σ_2 when $T(C_1)|\Sigma_2^{\checkmark} \subseteq T(C_2)$.

4 Behavioral Simulations

The standard methodology for proving refinement is based on simulation relations, which are the analog of inductive invariants in proofs of safety. Simulation relations enable an induction scheme to prove inclusion of traces which generally can go forward, from initial states towards end states, or backward, from end states towards initial states. While both types of reasoning, forward or backward, are sound for proving refinement, forward reasoning is easier to automate while being complete for proving refinement of *deterministic* LTSs only [45]. Since smart contracts are most often deterministic, we focus on forward reasoning in this work.

Let $C_1 = (Q_1, \Sigma_1, s_0^1, \delta_1)$ and $C_2 = (Q_2, \Sigma_2, s_0^2, \delta_2)$ be two contracts. A simulation relation R relates states of C_1 and C_2, respectively, in particular their initial states, such that any transition of C_1 from a state q (corresponding to a non-reverted invocation) can be reproduced by C_2 from a state related by R to q (i.e., C_2 has a transition with the same label from the state related by R to q). The end states of the two transitions in C_1 and C_2, respectively, must again be related by R[5]. Formally, a relation[6] $R \subseteq Q_1 \times Q_2$ is called a *(behavioral) simulation* from C_1 to C_2 iff $R(s_0^1, s_0^2)$ and for all $s_1, s_1' \in Q_1$, $a \in \Sigma_1$, and $s_2 \in Q_2$,

$$s_1 \xrightarrow{a}_{C_1} s_1' \wedge R(s_1, s_2) \implies \exists s_2' \in Q_2.\ s_2 \xrightarrow{a|\Sigma_2^{\checkmark}}_{C_2} s_2' \wedge R(s_1', s_2')$$

We adapted the standard definition of a simulation relation to take into account the restriction to non-reverted invocations and that Σ_1 is not necessarily included in Σ_2. Transitions with labels that exist only in C_1 should be mimicked by ϵ (skip) transitions of C_2.

Example 4.1. The relation $\mathsf{Sim}_1 \stackrel{\text{def}}{=}$ `#balances = balances` is a simulation relation from PAX in Figure 4 to ERC20 in Figure 5 (the field `balances` of PAX is prefixed by `#`). This holds because in particular, executing a method of PAX which is not defined by ERC20 does not affect `balances`.

The following statement follows from standard results relating simulation relations and refinement [45].

Theorem 4.2. *If there exists a simulation relation from a contract C_1 to a contract C_2, then C_1 refines C_2. Moreover, if C_1 refines C_2 and C_2 is deterministic, then there exists a simulation relation from C_1 to C_2.*

Theorem 4.2 reduces refinement proofs to synthesizing simulation relations. The next section shows that a simulation relation can be seen as a "separator" between two sets of pairs of states (of the two contracts), analogous to an inductive invariant being a separator between "safe" and "unsafe"

[5] This is a variation called *forward* simulation relation, which corresponds to the forward reasoning mentioned above, from initial states towards end states. In general, proving refinement may also require establishing the existence of a *backward* simulation, which is similar but the preservation of steps is defined in the reverse direction, i.e., for any transition of C_1 *leading* to a state q and any state q' of C_2 related by R to q, there exists a transition of C_2 with the same label leading to q' and starting from a state related by R to the source state of C_1's transition.

[6] For readability, we write binary relations as predicates, e.g., $R(s_1, s_2)$ instead of $(s_1, s_2) \in R$.

states. This enables a learning from examples approach for computing simulation relation candidates.

5 Learning Simulations From Examples

We describe a learning procedure for simulation relations which relies on a classification of pairs of states (of the two contracts) as *positive*, included in every simulation, or *negative*, excluded from every simulation. This classification is based on a notion of *observational distinguishability* between states which holds when two states can be distinguished by return values of read-only methods. We say that an invocation label $m(\vec{u})$ is *read-only* in a contract C when it is enabled in every state, i.e., for any trace $\sigma_1 \cdot \sigma_2 \in T(C)$, there exists $v \in \mathsf{Vals}$ such that $\sigma_1 \cdot (m(\vec{u}) \Rightarrow v) \cdot \sigma_2 \in T(C)$, and it does not enable other invocations, i.e., for any value $v \in \mathsf{Vals}$ and trace $\sigma_1 \cdot (m(\vec{u}) \Rightarrow v) \cdot \sigma_2 \in T(C)$, we have that $\sigma_1 \cdot \sigma_2 \in T(C)$ as well. A method m is *read-only* in a contract C when every invocation label $m(\vec{u})$ is read-only. For instance, the methods `PendingReturns` and `HighestBid` of the contract `RefAuction` in Figure 1 are read-only, while `bid` is not read-only.

Let C_1 and C_2 be two contracts over interfaces Σ_1 and Σ_2, respectively. A method $m \in \mathsf{Meths}(\Sigma_1) \cap \mathsf{Meths}(\Sigma_2)$ is called an *observation method* when it is read-only in both C_1 and C_2. Given a set of observation methods Obs, two states s_1 and s_2 of C_1 and C_2, respectively, are *(observationally) distinguishable* w.r.t. Obs, denoted by $s_1 \bowtie_{\mathsf{Obs}} s_2$, if

$$\exists m \in \mathsf{Obs}, \vec{u} \in \mathsf{Vals}^*, v \in \mathsf{Vals}. \quad s_1 \xrightarrow{m(\vec{u}) \Rightarrow v}_{C_1} s_1 \wedge \neg s_2 \xrightarrow{m(\vec{u}) \Rightarrow v}_{C_2} s_2$$

We will omit the set of methods Obs from the notations when they are not important or understood from the context.

Example 5.1. `PendingReturns` and `HighestBid` in Figure 1 are observation methods for the pair of contracts `Auction` and `RefAuction`. The pair of states in Equation 4 are distinguishable with respect to these two observation methods (`HighestBid` in particular).

The following result shows that any pair of distinguishable states is excluded from any simulation. It follows from an instantiation of the definition of a simulation on transitions corresponding to observation method invocations.

Lemma 5.2. *Let C_1 and C_2 be two contracts and* Obs *a set of observation methods. For any simulation R from C_1 to C_2,*

$$s_1 \bowtie_{\mathsf{Obs}} s_2 \implies \neg R(s_1, s_2)$$

We define two relations P and N over states of C_1 and C_2 representing positive and negative examples for simulation relations, respectively:

$$P(s_1, s_2) : \exists \sigma \in (\Sigma_2^{\checkmark})^*.\ s_0^1 \xrightarrow{\sigma}_{C_1} s_1 \wedge s_0^2 \xrightarrow{\sigma}_{C_2} s_2$$

$$N(s_1, s_2) : \exists s_1', s_2',\ \exists \sigma \in (\Sigma_2^{\checkmark})^*.\ s_1 \xrightarrow{\sigma}_{C_1} s_1' \wedge s_2 \xrightarrow{\sigma}_{C_2} s_2' \wedge s_1' \bowtie s_2'$$

where s_0^1 and s_0^2 are the initial states of C_1 and C_2, respectively. This classification is sound under the assumption that C_2 is deterministic. For negative examples, assuming by contradiction that $(s_1, s_2) \in N$ is included in a simulation relation, the state s_2' reached by C_2 when mimicking some sequence of invocations σ of C_1 should "simulate" the corresponding state s_1' of C_1. However, this cannot be the case since the two states are distinguishable (by Lemma 5.2).

Theorem 5.3. *For any simulation relation R from a contract C_1 to a deterministic contract C_2, we have that:*

$$P \subseteq R \subseteq \neg N.$$

Example 5.4. Positive and negative examples for the pair of contracts `Auction` and `RefAuction` (listed in Figure 1) are given in Equation 3 and Equation 4, respectively.

The reverse of Theorem 5.3 does not hold, i.e., there exist relations R that separate P from N and that are not simulation relations. For instance, if the set of observation methods Obs is empty, then N is also empty. However, not every superset of P satisfies the inductiveness requirement of a simulation relation. This is similar to the fact that not every superset of the reachable set of states in a program is an inductive invariant. This source of incompleteness can be removed by adapting the approach used in the ICE framework [27] for inductive invariant synthesis.

Theorem 5.3 implies that there exists no simulation relation when the set of positive and negative examples intersect.

Corollary 5.5. *If $P \cap N \neq \emptyset$, then there exists no simulation from C_1 to C_2, provided that C_2 is deterministic.*

For deterministic contracts where the return value of an invocation in a given state is unique, positive and negative examples can be represented precisely using *invocation sequences*. This enables a procedure for enumerating such examples which consists in enumerating (pairs of) invocation sequences and which is oblivious to state representations.

A contract C is *return-value deterministic* if it is deterministic and for any method m, arguments $\vec{u}$, and admitted trace $\sigma \in T(C)$, there is a single label $m(\vec{u}) \Rightarrow v$ such that $\sigma \cdot (m(\vec{u}) \Rightarrow v) \in T(C)$. Determinism does not imply uniqueness of return values. For instance, an extension of `Auction` (Figure 1) with a read-only method `foo` that returns a *random* value, computed using `block.difficulty` for instance, remains deterministic. The following result shows that states of return-value deterministic contracts can be represented precisely using invocation sequences.

Lemma 5.6. *For any return-value deterministic contract C, the following holds:*

$$\forall s, s'.\ s_0 \xrightarrow{\sigma} s \wedge s_0 \xrightarrow{\sigma'} s' \wedge inv(\sigma) = inv(\sigma') \implies s = s'$$

Based on Lemma 5.6, each positive example can be represented by a single invocation sequence (the pair of states being reproducible by running this sequence of invocations in both contracts) and each negative example can be represented by two invocation sequences, each sequence representing a state in one of the two contracts. Also, checking that a pair of states is a negative example reduces to checking whether by running the same (possibly-empty) sequence of invocations on the two states, irrespectively of the return values, leads to two states which are distinguishable. This is sound under the return-value determinism assumption.

The classification of simulation examples we presented above makes it possible to leverage off-the-shelf learning algorithms that compute formulas that are satisfied by positive examples and falsified by negative ones, e.g., [27, 28, 54, 57, 59], up to a bounded enumeration of such examples. The problem of checking whether such a formula is a valid simulation relation is discussed in the next section.

6 Verifying Simulations

We reduce the problem of verifying that a simulation candidate is valid to checking that it is an inductive invariant for a composition of contracts, which is formalized using a slight variation of the standard product construction for their LTS interpretations. Therefore, the *product* $C_1 \times C_2$ of two contracts is defined as follows: the states are pairs of states of C_1 and C_2, respectively, and a state (s_1, s_2) can perform a transition labeled by $a \in \Sigma_1^{\checkmark}$ to one of the following states:

- (s_1', s_2') if s_1 and s_2 can perform a transition labeled by a to s_1' and s_2', respectively
- (s_1', s_2) if $a \notin \Sigma_2^{\checkmark}$ and s_1 can perform a transition labeled by a to s_1', and
- a fail state $\lightning$ if $a \in \Sigma_2^{\checkmark}$ and only s_1 can perform an a transition.

The second case is required for simulation relations towards reference contracts that have a smaller interface while the last case makes it possible to detect invalid simulation candidates. Note also that $C_1 \times C_2$ excludes transitions corresponding to reverted invocations of C_1. An *inductive invariant* for a contract $C = (Q, \Sigma, s_0, \delta)$ is a set of states I such that (1) $s_0 \in I$ and (2) if $s \in I$ and $s \xrightarrow{a} s'$, for some symbol a, then $s' \in I$. The following theorem shows that any inductive invariant of the product (that does not contain the fail state) is also a simulation relation. The reverse holds when C_2 is deterministic.

Theorem 6.1. *Let C_1 and C_2 be two contracts. If R is an inductive invariant for $C_1 \times C_2$ such that $\lightning \notin R$, then R is simulation from C_1 to C_2. Moreover, if C_2 is deterministic and R is a simulation from C_1 to C_2, then R is an inductive invariant for $C_1 \times C_2$ and $\lightning \notin R$.*

```
contract A {
  ...
  function foo(uint x) public view returns (uint) { require(x>42); ... }
  function bar() public view returns (uint) { ... }
}
contract B {
  ...
  function foo(uint x) public view returns (uint) { ... }
}
contract AxB is A, B {
  ...
  function sync_foo(uint x) public {
    r0 = A.foo(x);
    r1 = B.foo(x);
    require(r0 != ⊥);
    assert (r0 == r1);
  }
  function sync_bar() public {
    A.bar();
  }
}
```

Figure 6. A contract AxB representing the product of the LTS interpretations of two contracts A and B.

In the following, we discuss a concrete instantiation of the results above that relies on source code instead of LTS interpretations. The most important point is defining a contract that represents the product of the LTS interpretations of two contracts. As hinted in Section 2.3.2, such a contract can be defined using the inheritance mechanism of Solidity. The more subtle issues are related to enforcing transitions with the same label, since the label includes an invocation *and* a return value, and dealing with reverted invocations and methods that are defined in only one of the two contracts.

We explain these issues using the contracts A and B in Figure 6, where B is intended to simulate A (their fields are omitted). The method foo is defined in both contracts, but A's version contains a require that may revert certain invocations, while the method bar is defined only in contract A. Note that methods defined only in B can be ignored while checking whether it simulates another contract.

The contract AxB is used to represent the product of the LTS interpretations of A and B. Since foo is defined in both contracts, the method sync_foo represents synchronous invocations of foo in A and B while also ensuring equality of return values, unless foo fails in A. Transitions of the product corresponding to bar invocations are represented using the method sync_bar. If AxB verifies the assertion, then its LTS interpretation restricted to invocations of sync_foo and sync_bar is the product of the LTS interpretations of A and B. Note that AxB can fail the assertion although B is deterministic and it simulates A. This is possible when A is *not* return-value deterministic, e.g., foo can return two values in both A and B when executed from the initial state.

By Theorem 6.1, if a relation R between states of `A` and `B` is an inductive invariant of `AxB` restricted to `sync_foo` and `sync_bar` (it holds before and after every invocation) and `AxB` verifies the assertion, then R is simulation from `A` to `B`.

Remark 6.1. This construction can be extended to handle certain specificities of Solidity. For instance, to deal with payable functions like `bid` in the auction contracts from Figure 1, it is sufficient to introduce a ghost variable in the reference contract that tracks the value of the balance (i.e., adding the amount in `msg.value`). Then, a simulation relation relates this ghost variable and the balance of the simulated contract instead of the two balances. Also, to establish the fact that a reference contract invocation makes the same "external " calls (to Solidity primitives like `send` or to other contracts) as the invocation of the contract it simulates (see Remark 3.1) we rely on auxiliary variables that record the sequence of calls with their arguments in each of the two invocations. We then *assert* the equality between these auxiliary variables and *assume* that these calls have the same return values. This models the fact that the two contracts refine one another when placed in the same context where the environment produces the same responses.

7 Implementation

In this section we describe an implementation of our methodology for Solidity smart contracts. Our implementation consists of four main components: an *example generator*, an example-guided *synthesizer*, a *blockchain oracle*, and a deductive *verifier*. As input, our implementation requires a pair of Solidity smart contracts with overlapping function signatures, and parameters to limit example generation, including the sets of values to use for transaction parameters, and the number of contract states to explore.

Given these inputs, the example generator provides the synthesizer with positive and negative examples, where each example corresponds to a pair of contract states. In turn, the synthesizer provides the verifier with a candidate simulation relation separating positive and negative examples. Since examples correspond to contract states on an Ethereum blockchain, the synthesizer relies on an oracle to evaluate expressions on examples. Finally, the verifier validates candidate simulation relations.

While this simple scheme sufficed for our empirical study, in principle, the selection of input parameters could be automated in a refinement loop from spurious verifier counterexamples, i.e., following *counterexample guided inductive synthesis* [61]. Furthermore, although we assume the annotated contract against which the given unannotated contract is compared is identified a priori, in principle this identification might be performed, e.g., via machine learning classifiers.

7.1 Example Generation

Our example generator executes transaction sequences on the Ganache [26] personal blockchain for Ethereum using the Web3 Ethereum JavaScript API [71] and Solidity compiler [63]. Given limits transaction parameters, e.g., small sets of integer and address values, the example generator systematically explores every transaction sequence in lexicographic order up to the given threshold on the number of contract states. For each state we record as observations the return values for each read-only (view) function over the given parameter limits. In case the states reached in some transaction sequence yield different observations, we return the transaction sequence and observations as a counterexample refuting simulation. Otherwise, the example generator yields positive and negative examples according to Section 5.

Two notable issues that the example generator must overcome are potential nondeterminism, e.g., due to account creation and transaction block mining, and controlling transaction parameters, e.g., the message *sender* parameter. While the former can be managed via parameters to Ganache, the latter required instantiating auxiliary contracts at various addresses to invoke target functions - effectively setting the sender to the auxiliary contract's address.

7.2 Synthesis

Our synthesizer component extends the Precondition Inference Engine (PIE) [54], a tool which learns a set of *features*, i.e., atomic predicates, (and a Boolean combination of these features) separating positive and negative examples by enumerating candidate features of increasing complexity. We extend PIE along two principle axes. First, we extend its grammar to include types and operations to handle Solidity language features like addresses, arrays, and maps. Second, instead of concrete examples on which to evaluate candidate features, we make examples *symbolic*, and delegate evaluation of features on examples to a blockchain oracle.

As an optimization we provide the synthesizer with a set of *seed features* generated from the given pair of contracts. Intuitively, the seed features correspond to equalities between terms over the respective contracts' fields that are likely to hold. For instance, when contracts each have a read-only (view) function f which evaluates terms t_1 and t_2, respectively, we generate the equality $t_1 = t_2$. While this is not generally feasible for view functions with complex control flow, it is useful in practice, since many view functions have simple bodies, e.g., a single return statement.

7.3 Verification

Our verifier consists of the reduction from simulation checking to deductive verification, described in Section 6, along with the solc-verify verifier [36], which in turn reduces Solidity contract verification to Boogie verification (and ultimately

Contract	Source	Variations
Auction	Ethereum	3
CrowdSale (token offering)	OpenZepplin	3
ERC 20 (token)	OpenZepplin	5
ERC 165 (interface detection)	OpenZepplin	4
ERC 721 (non-fungible token)	OpenZepplin	3
Escrow (payment)	OpenZepplin	3
Gambling	dice2.win	1
LifeCycle (life cycle)	OpenZepplin	5
Lottery	Etherscan	1
MultiSigWallet	ConsenSys	1
Ownable (ownership)	OpenZepplin	5
Roles (access control)	OpenZepplin	5
Voting	Etherscan	2
	Total	41

Figure 7. Collection of smart contracts.

SMT solving). We have contributed only GitHub issues and feature requests to solc-verify.

Besides the nuances described in Section 6 relating to effects on global state, e.g., balances, verifying the simulation-checking contract with solc-verify involved one key nuance regarding modular verification. In particular, while invocations to the annotated reference contract can be verified *modularly*, i.e., using only its pre- and post-conditions, invocations to the unannotated contract are verified *inline*, i.e., explicitly reasoning about the statements in its implementation. Besides helping to virtualize potentially-conflicting global effects between the two invocations, such modularity generally improves tractability.

8 Case Study of Solidity Smart Contracts

In this section we outline our case study of Solidity smart contracts, including collection methodology, a partial taxonomy, and an analysis of syntactic similarities. Our starting points for sourcing canonical contracts included the Solidity documentation [63], the Etherscan block explorer and analytics platform [22], the State of the DApps curated directory for decentralized applications [52], and the OpenZepplin contracts library [53].

A first observation is that a vast number of contracts on the, e.g., Ethereum blockchain are variations on a relatively-small number of canonical contacts like those listed in the first column in Figure 7. We found that more than half of the 47 398 contracts extracted from the Ethereum blockchain and studied in [21], which cover each of the eighteen Ethereum application categories from State of the DApps, contain keywords associated with these canonical contracts. This finding seems consistent with common practice, since standardization mechanisms such as Ethereum Request for Comment (ERC) are widely used.

In order to use these canonical contracts as targets for our verification methodology, we manually annotated them with full functional specifications, and verified the annotations with solc-verify [36].

To source contract variations, we collected contracts from Etherscan, as well as popular Blockchain platforms including Moloch Ventures [70], 0xcert [7], Sirin Labs [60], Bit Nation [16], and Crypto Kitties [20]. Overall we collected a set of 43 unannotated contracts, 41 of these contracts are categorized in Figure 7 based on which canonical contract they implement. The remaining two contracts are referred to as *multi-contracts* as they simultaneously implement multiple canonical contracts. The collected contracts can be found at [14].

Finally, to assess the need for the automated synthesis procedure described in Section 5, we considered weaker syntactic approaches with varying degrees of sophistication. For example, simply considering the conjunction of equalities between fields with the same names could work for simple single-field contracts, like ownership. In case contracts renamed fields, some field-name similarity heuristic would be required. For contracts with multiple fields, more sophisticated field-matching heuristics would be required, and so on. Then there are contracts whose simulation involves arithmetic expressions, further complicating heuristics. While the current generation of smart contracts we've studied are relatively simple, future generations could render such heuristics fairly useless. As noted in Section 11, our approach is relatively complete, and, as demonstrated in Section 9, capable of synthesizing simulations for many non-trivial contracts.

9 Experimental Evaluation

In this section we outline an empirical study of our automated verification approach applied to the Solidity smart contracts described in Section 8 using the implementation described in Section 7. We are able to run our tool on all contracts from Figure 7 except MultiSigWallet and Gambling, which require generating non-primitive transaction parameters, including addresses of deployed token contracts and components of cryptographic signatures.

The overview of Figure 8 summarizes our results, listing the generated simulation relations (omitting atomic terms) and verification outcomes. Each row, labeled $c \times n$ corresponds to n unannotated contracts compared against one canonical annotated contract c, e.g., auction $\times$ 3 corresponds to 3 distinct unannotated auction contracts compared with one canonical auction contract. (The rows labeled multi-$i \times 3$ are exceptions; in these cases we consider one unannotated contract compared against 3 distinct canonical contracts, corresponding to cases of multiple inheritance/interfaces.) In all but 3 cases we are able to generate plausible candidate simulation relations, and in all but 3 cases we are able to verify these relations – see Section 9.1.

In the "simulation relations" column, we list the learned simulation relations in prefix notation, omitting atomic terms,

i.e., contract fields and constants. In the verified column, we list the number of canonical-and-unannotated-contract pairs for which a candidate simulation relation was:

- computed and verified, e.g., $\mathsf{T} \times 3$ in the auction row indicates success for 3 contract pairs;
- computed but not verified, e.g., $\mathsf{F} \times 1$ in the crowdsale row indicates a candidate simulation relation our implementation did not verify; and
- not computed, e.g., $\perp \times 2$ in the erc20 row indicates 2 pairs for which our implementation did not compute plausible candidate relations.

Our approach synthesizes simulation relations which are notably simpler than the inductive invariants which would be required to verify the functional properties of unannotated contracts by other means. For example, the inductive invariants for typical auction contracts would require disjunctions over auction phases, e.g., active vs. completed, while simulation relations between typical auction contracts need only conjunctions of equalities (see Figure 8). Previous works on relational verification make the same observation [13, 23].

For each phase we summarize runtimes, in seconds. Distributions with mean μ, standard deviation σ, and population count n are represented as $\mu \pm \sigma : k$, where σ is omitted when 0, and k is omitted when equal to the subject count n of the row labeled $c \times n$. Among the three phases, synthesis generally takes much longer, e.g., minutes, than example generation, e.g., seconds, and verification, e.g., one second.

9.1 Cases Where Simulation Was Not Proved

Our implementation only failed to compute candidate simulation relations in 3 cases. However, each failure is due to the discovery of genuine counterexamples to simulation (and refinement). Counterexamples arise in 2 out of 5 ERC-20 variations and in the multi-1 contract which simultaneously implements three canonical contracts: ERC-20, Ownable, and Pausable.

The first counterexample arises due to the *transferFrom* function of ERC-20. The canonical contract subtracts the transferred amount from the sender balance before adding it to the receiver balance, reverting when the subtraction underflows, while the variation contract does the reverse. Thus after executing the following transactions:

$$a_1\text{: approve}(a_2, 2);\ a_2\text{: transferFrom}(a_1, a_1, 2)$$

the function allowance(a_1, a_2) returns 2 in the first case, since a_2's allowance has not decreased, but 0 in the second. Since the *transferFrom* function is present in the ERC-20 token standard [6] this counterexample corresponds to a vulnerability of the unannotated contract.

The remaining two cases arise from ERC-20's *decreaseAllowance* function. While the canonical contract reverts the transaction if the requested decrease is greater than the current allowance, the variation contract simply sets the allowance to zero without reverting the transaction. Thus after executing the following transactions:

$$a_1\text{: increaseAllowance}(a_2, 1)$$
$$a_1\text{: decreaseAllowance}(a_2, 2)$$

the function allowance(a_1, a_2) returns 1 in the first case, but 0 in the second. Note that the *decreaseAllowance* function is not present in the ERC-20 token standard but only in the OpenZeppelin implementation that we use as the canonical ERC-20 contract.

Our implementation is limited since it does not automatically generate loop and contract invariants for verifying candidate simulation relations. Generally speaking, loop invariants on otherwise-unannotated contracts are necessary for methods with loops; contract invariants can be required in cases where the unannotated-contract state invariants are not implied by the combination of canonical-contract state invariants (which are given) and candidate simulation relations (which are computed by our synthesizer). While our experiments never required loop invariants, contract invariants were required in one case, to characterize fields of the unannotated contracts which have no direct correspondence to canonical-contract fields. In particular, one of whitelisted's unannotated contracts maintains a length field equal to the number of elements in an array; the corresponding canonical contract has no such length field. Such relationships hold equally in all positive and negative examples since examples only include reachable contract states. In contrast, invariant-generation for individual contracts would distinguish a contract's reachable and unreachable states. We consider generating contract and loop invariants orthogonal to simulation relations, and standard techniques exist [55].

9.2 Example Generation Phase

For the example generation phase we count blockchain transactions executed, transaction sequences (traces), states encountered, and positive and negative examples. Our implementation usually learns simulation relations from a relatively small set of examples: 100 examples usually suffice, up to 450 in the worst cases. Another observation is that the total number of positive and negative examples is several times the number of explored states. This happens because negative examples arise not only from observationally-inequivalent states encountered among the executed transaction sequences, but also inductively from prefixes of longer negative examples – see Lemma 5.6. The 3 cases where no examples were generated correspond to genuine counterexamples to simulation.

9.3 Synthesis Phase

For the synthesis phase we count the fields and seed features given to the synthesizer, non-atomic terms in the generated simulation relation, and the number of queries to

Overview

contracts	simulation relations	verified
auction × 3	(∧(∧(∧(∧(∧(=)(=))(=))(=))(=))(= (+)))×3	T×3
crowdsale × 3	(∧(∧(∧(∧(=)(=))(=))(=))(=))×3	F×1, T×2
erc165 × 4	(=)×4	T×4
erc20 × 5	(∧(=)(=))×3	T×3, ⊥×2
erc721 × 3	(∧(∧(∧(=)(=))(=))(=))×3	T×3
escrow × 3	(∧(=)(=))×3	T×3
finalizable × 2	(∧(=)(=))×2	T×2
lottery × 1	(∧(=)(=))×1	F×1
multi-1 × 3	(∧(=)(=))×1, (=)×1	T×2, ⊥×1
multi-2 × 3	(∧(=)(=))×2, (=)×1	T×3
ownable × 4	(=)×4	T×4
pausable × 3	(∧(=)(=))×3	T×3
signer-role × 2	(=)×2	T×2
voting × 2	(∧(=)(=))×2	T×2
whitelisted × 3	(∧(=)(=))×3	F×1, T×2

Example Generation

contracts	transactions	traces	states	positive	negative	time
auction × 3	174	47	78	47	370	33±1.6
crowdsale × 3	99.3±11.5	25	34	12.7±8.1	330	20±1
erc165 × 4	26	7	10	7	36	5.7±0.1
erc20 × 5	287.6±391.3	21±3.5: **3**	38±6.9: **3**	12.6±11.8	60±54.8	52.5±59.3
erc721 × 3	154	39	74	31.7±12.7	100	121.6±1.1
escrow × 3	76	19	34	19	100	10.2±0.1
finalizable × 2	28	7	10	6	28	4.4±0.1
lottery × 1	176	33	62	33	370	20.7
multi-1 × 3	212±322.2	7: **2**	10: **2**	3±2.6	14±14	53±66.2
multi-2 × 3	50±41.6	13±10.4	22±20.8	11±12.2	47.3±46.1	24.8±19.6
ownable × 4	26	7	10	5	28	5±0.1
pausable × 3	26	7	10	4	14	4.5±0.1
signer-role × 2	26	7	10	5	30	6.6±0.1
voting × 2	34	11	10	11	84	13.1±0.9
whitelisted × 3	66±8	17±2	30±4	15±1	118±39	10.8±0.9

Synthesis

contracts	fields	seeds	terms	queries	time
auction × 3	15	5	7	10914	2561.8±11.3
crowdsale × 3	12	4.7±0.6	5	659.3±556.6	345.3±135.7
erc165 × 4	4	1	1	43	20.3±0.1
erc20 × 5	7.6±1.3	1.8±0.4	2: **3**	121±3.5: **3**	69.2±4.1: **3**
erc721 × 3	18	4	4	131.7±12.7	104.4±10
escrow × 3	6	1	2	299±1.7	72.5±0.3
finalizable × 2	4	1	2	92	19.4±0.2
lottery × 1	6	1	2	840	353.7
multi-1 × 3	15±1	1	1.5±0.7: **2**	46.5±19.1: **2**	26.7±2.1: **2**
multi-2 × 3	13±1	1.3±0.6	1.7±0.6	74±46.8	44.5±35.5
ownable × 4	2	1	1	33	15.7±0.2
pausable × 3	4	1	2	54	13.6
signer-role × 2	2	0	1	70	21.3±0.1
voting × 2	6±2.8	0	2	4037.5±4635.1	324±303
whitelisted × 3	4.3±0.6	0.7±0.6	2	2114±3105.9	172.6±152.5

Verification

contracts	lines of code	verified fns.	unverified	time
auction × 3	481.3±11.6	11	0	1.9±0.1
crowdsale × 3	527.7±31.6	19.7±0.6	0.3±0.6	2.1±0.2: **2**
erc165 × 4	115.3±15.9	3	0	0.8
erc20 × 5	431±124.7: **3**	11.7±1.5: **3**	0: **3**	1.2: **3**
erc721 × 3	684.7±7.1	10	0	1.3
escrow × 3	227.7±11	7	0	1.1±0.1
finalizable × 2	146.5±20.5	5	0	0.8
lottery × 1	224	-	-	-
multi-1 × 3	408±5.7: **2**	5.5±0.7: **2**	0: **2**	1.2±0.2: **2**
multi-2 × 3	582±46.9	6.7±2.1	0	1.1±0.1
ownable × 4	185.3±25.2	4.5±0.6	0	0.8
pausable × 3	206.3±3.2	5	0	0.8
signer-role × 2	159.5±2.1	6	0	0.9
voting × 2	167±9.9	5	0	0.9
whitelisted × 3	177±5.3	5.7±0.6	0.3±0.6	1±0.1: **2**

Figure 8. Summary of results. **Overview**: generated simulation relations (atomic terms omitted) and verification outcomes; **Example Generation**: counting blockchain transactions executed, transaction sequences (traces), states, and generated examples; **Synthesis**: counting contract fields and seed features passed as input, non-atomic terms in generated simulations, and blockchain-oracle queries; and **Verification**: counting lines of Solidity code, verified functions, and unverified functions.

the blockchain oracle for evaluating new features against examples. Note that the seed features were automatically generated as explained in Section 7.2. The primary factors to overall runtime, which is roughly proportional to the number of oracle queries, are the number and sizes of generated terms.

Despite similarities between varying canonical contract refinements, a naive syntactic strategy of listing equalities, i.e., that used to generate seed features, would not suffice (cf. §8). In most cases, the synthesizer is forced to generate terms that are not seed features while enumerating a relatively small number of candidates (column "queries").

9.4 Verification Phase

For the verification phase we count the lines of Solidity code, verified functions, and unverified functions. While verification succeeds in most cases, current limitations in solc-verify yield a few failures. The first two cases were caused by skipping and reporting parsing errors for functions which have Solidity features that are not supported by the tool. The last case requires establishing a contract invariant (see §9.1), yet we do not currently apply invariant-generation to individual contracts. Note that for the auction contract, the function which allows previous highest bidders to reclaim their bids invokes the Solidity `send` function to transfer ether. Thus, to prove that this function preserves the candidate simulation relation, we apply the technique described in Remark 6.1 where we use shadow variables to record the status of the invocations of `send`.

10 Related Work

Analysis of Smart Contracts. A number of systems have been proposed for detecting vulnerabilities in smart contracts. These systems are based on static analysis, e.g. [33, 40, 66, 69], symbolic execution engines, e.g. [37, 41, 44, 51, 67], or dynamic analysis, e.g., [35]. The systems based on static analysis are designed to expose certain coding patterns that are prone to critical bugs and cannot establish full functional correctness. In contrast, our work makes it possible to establish behavioral simulations towards verified contracts which implies full functional correctness. The systems based on symbolic execution or dynamic analysis are incomplete and can only establish correctness for *bounded* executions.

Functional Verification of Smart Contracts. Several previous works have developed methodologies for proving full functional correctness of smart contracts using theorem provers like Coq, F*, and Isabelle/HOL, e.g., [9, 15, 34, 39, 58], SMT solvers, e.g, [36, 42], or predicate abstraction [55]. These works rely on user-provided functional specifications while our work, by establishing behavioral simulations, makes it possible to verify contracts for which such specifications do not exist (as long as the simulations relate them with verified contracts).

Computing Refinement Relations Between Finite-State Systems. The complexity of computing simulation relations between *finite-state* systems has been addressed quite extensively in the literature, e.g. [17, 18, 29, 30, 38, 56]. Some of these works extend to infinite-state systems as long as they have *finite* similarity quotient which intuitively, means that they are simulated by a finite-state system. This is not the case for smart contracts which store infinite-domain inputs in their state, e.g., the auction bids of Figure 1.

Synthesizing "Small-Step" Simulation Relations. An established approach for proving the validity of compiler optimizations consists in synthesizing simulation relations from source to optimized programs, e.g., [12, 31, 48–50, 68]. These simulation relations concern traces of a small-step operational semantics of the two programs while our approach computes *behavioral* simulations which relate programs in terms of operation sequences, ignoring local memory and control-flow. Moreover, the simulation relations are synthesized at compile time during the construction of the optimized program. A reduction of simulation relation synthesis to solving a set of Horn clauses has been investigated in [24, 25]. This reduction has been evaluated only for validating compiler optimizations and applying it to smart contracts would require modeling Solidity semantics with Horn clauses, which is non-trivial.

Learning-Based Synthesis of Preconditions or Inductive Invariants. Learning from examples has been used to synthesize preconditions or inductive invariants that imply a *user-provided* specification, e.g, [27, 28, 54, 57, 59]. Our work addresses the verification problem when such specifications are lacking. The learning procedures defined in these works are however re-usable in our context. Our implementation leverages the one defined by Padhi et al. [54].

11 Conclusion

Towards verifying unannotated smart contracts against precise functional specifications, we have proposed a notion of behavioral refinement, along with an automated simulation-based proof methodology. As noted in Section 5-6, our method is complete modulo three (unavoidable) sources of incompleteness: deductive verification, simulation for proving trace refinement, and learning from a bounded set of examples.

For verifying candidate simulation relations, our current implementation assumes manually-provided loop invariants, and, in some cases (see §9.1), contract invariants. This manual effort could likely be automated for many contracts of interest using standard invariant-generation techniques, e.g., [55]. Regardless, we consider the cost of any such manual effort to be offset by a significant benefit: the inheritance of arbitrary specifications established by the corresponding canonical contract(s). This includes hyperproperties like noninterference [32, 34, 65], because we use simulation relations instead of arbitrary trace refinement relations. The incompleteness due to simulation relations is thus also counterbalanced by the preservation of a larger class of specifications.

In the future we might extend our approach along a few dimensions. First, we might eliminate the need to provide example-generation parameters, e.g., using verifier counterexamples to drive example generation. Second, we might automate the identification of canonical contracts against which to consider refinements, e.g., using machine-learning classifiers. Finally, we might relax compatibility requirements on function signatures, e.g., to allow simulation among similar functions with varying parameter types.

Acknowledgments

This work is partly supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 678177).

References

[1] 2016. theDAO. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413
[2] 2017. Blockchain is empowering the future of insurance. https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/
[3] 2017. An In-Depth Look at the Parity Multisig Bug. http://hackxingdistributed.com/2017/07/22/deep-dive-parity-bug
[4] 2017. Northern Trust uses blockchain for private equity record- keeping. http://www.reuters.com/article/nthern-trust-ibm-blockchain-idUSL1N1G61TX.
[5] 2017. Parity security alert. https://www.parity.io/security-alert-2/
[6] 2020. ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20
[7] 0xcert. 2019. https://github.com/0xcert
[8] Rajeev Alur, Pavol Cerný, and Steve Zdancewic. 2006. Preserving Secrecy Under Refinement. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II (Lecture Notes in Computer Science)*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.), Vol. 4052. Springer, 107–118. https://doi.org/10.1007/11787006_10
[9] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 66–77. https://doi.org/10.1145/3167084
[10] Hagit Attiya and Constantin Enea. 2019. Putting Strong Linearizability in Context: Preserving Hyperproperties in Programsthat Use Concurrent Objects. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary (LIPIcs)*,

Jukka Suomela (Ed.), Vol. 146. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:17. https://doi.org/10.4230/LIPIcs.DISC.2019.2

[11] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Vol. 4111. Springer, 364–387. https://doi.org/10.1007/11804192_17

[12] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore D. Zuck. 2005. TVOC: A Translation Validator for Optimizing Compilers. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings (Lecture Notes in Computer Science)*, Kousha Etessami and Sriram K. Rajamani (Eds.), Vol. 3576. Springer, 291–295. https://doi.org/10.1007/11513988_29

[13] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings (Lecture Notes in Computer Science)*, Michael J. Butler and Wolfram Schulte (Eds.), Vol. 6664. Springer, 200–214. https://doi.org/10.1007/978-3-642-21437-0_17

[14] Smart Contracts Benchmark. 2019. https://github.com/beillahi/smart-contract-simulation-data

[15] 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: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, Toby C. Murray and Deian Stefan (Eds.). ACM, 91–96. https://doi.org/10.1145/2993600.2993611

[16] BitNation. 2019. https://github.com/Bit-Nation

[17] Doron Bustan and Orna Grumberg. 2003. Simulation-based minimazation. *ACM Trans. Comput. Log.* 4, 2 (2003), 181–206. https://doi.org/10.1145/635499.635502

[18] Gérard Cécé. 2017. Foundation for a series of efficient simulation algorithms. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005069

[19] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210. https://doi.org/10.3233/JCS-2009-0393

[20] Awesome CryptoKitties. 2019. https://github.com/cryptocopycats

[21] Thomas Durieux, Joao F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *International Conference on Software Engineering (ICSE 2020)*.

[22] Etherscan. 2019. https://etherscan.io Retrieved November 19th, 2019.

[23] Azadeh Farzan and Anthony Vandikas. 2019. Automated Hypersafety Verification. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11561. Springer, 200–218. https://doi.org/10.1007/978-3-030-25540-4_11

[24] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2015. Automated Discovery of Simulation Between Programs. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings (Lecture Notes in Computer Science)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.), Vol. 9450. Springer, 606–621. https://doi.org/10.1007/978-3-662-48899-7_42

[25] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2016. Property Directed Equivalence via Abstract Simulation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9780. Springer, 433–453. https://doi.org/10.1007/978-3-319-41540-6_24

[26] Ganache. 2019. https://www.trufflesuite.com/docs/ganache/overview

[27] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

[28] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 499–512. https://doi.org/10.1145/2837614.2837664

[29] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. 2003. From Bisimulation to Simulation: Coarsest Partition Problems. *J. Autom. Reasoning* 31, 1 (2003), 73–103. https://doi.org/10.1023/A:1027328830731

[30] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. 2015. Rank and simulation: the well-founded case. *J. Log. Comput.* 25, 6 (2015), 1331–1349. https://doi.org/10.1093/logcom/ext066

[31] Rigel Gjomemo, Kedar S. Namjoshi, Phu H. Phung, V. N. Venkatakrishnan, and Lenore D. Zuck. 2015. From Verification to Optimizations. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings (Lecture Notes in Computer Science)*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.), Vol. 8931. Springer, 300–317. https://doi.org/10.1007/978-3-662-46081-8_17

[32] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 11–20. https://doi.org/10.1109/SP.1982.10014

[33] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL* 2, OOPSLA (2018), 116:1–116:27. https://doi.org/10.1145/3276486

[34] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Lujo Bauer and Ralf Küsters (Eds.), Vol. 10804. Springer, 243–269. https://doi.org/10.1007/978-3-319-89722-6_10

[35] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28. https://doi.org/10.1145/3158136

[36] Ákos Hajdu and Dejan Jovanovic. 2019. solc-verify: A Modular Verifier for Solidity Smart Contracts. *CoRR* abs/1907.04262 (2019). arXiv:1907.04262 http://arxiv.org/abs/1907.04262

[37] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 531–548. https://doi.org/10.1145/3319535.3363230

[38] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee,*

Wisconsin, USA, 23-25 October 1995. IEEE Computer Society, 453–462. https://doi.org/10.1109/SFCS.1995.492576

[39] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.), Vol. 10323. Springer, 520–535. https://doi.org/10.1007/978-3-319-70278-0_33

[40] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[41] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 1317–1333. https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[42] Shuvendu K. Lahiri, Shuo Chen, Yuepeng Wang, and Isil Dillig. 2018. Formal Specification and Verification of Smart Contracts for Azure Blockchain. *CoRR* abs/1812.08829 (2018). arXiv:1812.08829 http://arxiv.org/abs/1812.08829

[43] Barbara Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1811–1841. https://doi.org/10.1145/197320.197383

[44] 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, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 254–269. https://doi.org/10.1145/2976749.2978309

[45] Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233. https://doi.org/10.1006/inco.1995.1134

[46] Robin Milner. 1989. *Communication and concurrency*. Prentice Hall.

[47] Tom M. Mitchell. 1997. *Machine learning*. McGraw-Hill. http://www.worldcat.org/oclc/61321007

[48] Kedar S. Namjoshi, Giacomo Tagliabue, and Lenore D. Zuck. 2013. A Witnessing Compiler: A Proof of Concept. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings (Lecture Notes in Computer Science)*, Axel Legay and Saddek Bensalem (Eds.), Vol. 8174. Springer, 340–345. https://doi.org/10.1007/978-3-642-40787-1_22

[49] Kedar S. Namjoshi and Lenore D. Zuck. 2013. Witnessing Program Transformations. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings (Lecture Notes in Computer Science)*, Francesco Logozzo and Manuel Fähndrich (Eds.), Vol. 7935. Springer, 304–323. https://doi.org/10.1007/978-3-642-38856-9_17

[50] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 83–94. https://doi.org/10.1145/349299.349314

[51] Ivica Nikolic, 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, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 653–663. https://doi.org/10.1145/3274694.3274743

[52] State of the DApps. 2019. https://www.stateofthedapps.com

[53] OpenZeppelin. 2019. https://github.com/OpenZeppelin

[54] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 42–56. https://doi.org/10.1145/2908080.2908099

[55] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Chohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE Symposium on Security and Privacy (SP 2020)*.

[56] Francesco Ranzato and Francesco Tapparo. 2010. An efficient simulation algorithm based on abstract interpretation. *Inf. Comput.* 208, 1 (2010), 1–22. https://doi.org/10.1016/j.ic.2009.06.002

[57] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. 2008. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, Barbara G. Ryder and Andreas Zeller (Eds.). ACM, 295–306. https://doi.org/10.1145/1390630.1390666

[58] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Temporal Properties of Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV (Lecture Notes in Computer Science)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 11247. Springer, 323–338. https://doi.org/10.1007/978-3-030-03427-6_25

[59] Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2012. Interpolants as Classifiers. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, 71–87. https://doi.org/10.1007/978-3-642-31424-7_11

[60] Sirin-labs. 2019. https://github.com/sirin-labs

[61] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, John Paul Shen and Margaret Martonosi (Eds.). ACM, 404–415. https://doi.org/10.1145/1168857.1168907

[62] Solidity. 2019. https://solidity.readthedocs.io/en/v0.4.24/solidity-by-example.html#blind-auction Retrieved November 19th, 2019.

[63] the Contract-Oriented Programming Language Solidity. 2019. https://solidity.readthedocs.io/en/v0.5.0/

[64] Paxos Standard ERC20 stablecoin PAX. 2019. https://github.com/paxosglobal/pax-contracts/blob/3d50aa32c4691c46d2bf3f8150fff270849e8dbe/contracts/PAXImplementation.sol Retrieved November 19th, 2019.

[65] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. 2019. zkay: Specifying and Enforcing Data Privacy in Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1759–1776. https://doi.org/10.1145/3319535.3363222

[66] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*. ACM, 9–16. http://ieeexplore.ieee.org/document/8445052

[67] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC*

2018, San Juan, PR, USA, December 03-07, 2018. ACM, 664–676. https://doi.org/10.1145/3274694.3274737

[68] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 295–305.

[69] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 67–82.

[70] Moloch Ventures. 2019. https://github.com/MolochVentures

[71] web3.js Ethereum JavaScript API. 2019. https://web3js.readthedocs.io/en/v1.2.4/

[72] Gavin Wood. 2016. Ethereum: a Secure Decentralised Generalised Transaction Ledger.