Learning Contract Invariants Using Reinforcement Learning

Junrui Liu*
junrui@cs.ucsb.edu
UC Santa Barbara
Veridise

Yanju Chen*
yanju@cs.ucsb.edu
UC Santa Barbara
Veridise

Bryan Tan
bryan@veridise.com
Veridise

Işıl Dillig
isil@cs.utexas.edu
University of Texas at Austin
Veridise

Yu Feng
yufeng@cs.ucsb.edu
UC Santa Barbara
Veridise

ABSTRACT

Due to the popularity of smart contracts in the modern financial ecosystem, there has been growing interest in formally verifying their correctness and security properties. Most existing techniques in this space focus on common vulnerabilities like arithmetic overflows and perform verification by leveraging *contract invariants* (i.e., logical formulas hold at transaction boundaries). In this paper, we propose a new technique, based on deep reinforcement learning, for automatically learning contract invariants that are useful for proving arithmetic safety. Our method incorporates an off-line training phase in which the verifier uses its own verification attempts to learn a *policy* for contract invariant generation. This learned (neural) policy is then used at verification time to predict likely invariants that are also useful for proving arithmetic safety. We implemented this idea in a tool called Cider and incorporated it into an existing verifier (based on refinement type checking) for proving arithmetic safety. Our evaluation shows that Cider improves both the quality of the inferred invariants as well as inference time, leading to faster verification and hardened contracts with fewer run-time assertions.

ACM Reference Format:
Junrui Liu, Yanju Chen, Bryan Tan, Işıl Dillig, and Yu Feng. 2022. Learning Contract Invariants Using Reinforcement Learning. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As decentralized cryptocurrencies grow in popularity, ensuring the security of smart contracts is increasingly becoming a pressing concern. As a result, there has been a whirlwind of research interest in finding security vulnerabilities of programs written in Solidity, which is currently the most popular programming language for smart contract development. Because many serious bugs are caused by common vulnerabilities like reentrancy and arithmetic overflows, much of the research in this space focuses on detecting such common vulnerabilities [2, 12, 24, 27].

Although existing research has attacked verification of smart contracts from different angles, a common theme among several threads of research is that they rely on a so-called *contract invariant*, which is a logical formula over *storage variables*[1] that always holds at transaction boundaries [2, 17, 18, 25, 29]. For example, when generating verification conditions for a given method, the contract invariant can be assumed as part of the pre-condition. Existing work has shown that inference of contract invariants is important for successful verification and elimination of false positives reported by the analyzer [12, 24, 27].

Motivated by the importance of contract invariants in smart contract analysis, this paper presents a new technique for inferring such contract invariants. Our method is based on machine learning and primarily targets contract invariants that are useful for discharging arithmetic overflows, one of the most common and nefarious security vulnerabilities in Solidity programs. The use of machine learning in this context allows our approach to consider a rich class of contract invariants, while also scaling to real-world Solidity programs. Since the candidate invariants predicted by the learnt ML model are independently verified, our method is guaranteed to return valid contract invariants.

As in other verification contexts, a central problem to applying machine learning in this space is the paucity of labeled training data in the form of program/invariant pairs. Inspired by prior work [3, 21, 22], we address this problem using off-line reinforcement learning (RL), wherein the method learns from the underlying verifier's failed and successful verification attempts. By starting with a random initial policy and updating it based on feedback from the verifier, we can gradually learn a model that is capable of predicting contract invariants that are useful for proving arithmetic safety.

In this paper, we show how to formulate the contract invariant generation problem as a Markov Decision Process (MDP) (a necessary pre-condition for applying RL to any problem) and use the policy gradient algorithm to learn a (neural) *policy* that induces a probability distribution over candidate invariants. Notably, we design our reward function to take into account not only whether a formula is a *valid* contract invariant but also to consider how *useful* it is for discharging the generated verification conditions. To make the problem of learning tractable, we propose an *arithmetic*

*Both authors contributed equally to the paper

[1]A storage variable is one that is stored on the Blockchain.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

dependency graph (ADG) abstraction of smart contracts and utilize a graph neural network (GNN) architecture to *encode* this abstraction into a vector. Our neural policy also incorporates a *decoder* network that can be used to map vectors in $\mathbb{R}^n$ to program-specific terms that should appear as part of the contract invariant.

In addition to formulating the contract invariant inference problem as a deep reinforcement learning problem, another contribution of this paper is to use the learned neural policy to implement the type inference backend of SolType, a refinement type system designed for checking for arithmetic safety. Our approach uses the learned policy to perform backtracking search over candidate invariants in decreasing order of likelihood and tries to find a valid type assignment that can be used to discharge as many arithmetic overflows as possible.

We have implemented the proposed approach in a tool called Cider and compare it against two baselines for contract invariant inference. One of these baselines utilizes a CHC solver [1] and the other one is an instantiation of monomial predicate abstraction [7, 10]. We evaluate the effectiveness of these techniques both in terms of the quality of the inferred invariants as well as inference time. Our results show that our approach reduces both verification time as well the number of required run-time checks.

To summarize, the contributions of this paper include:

- a formulation of the contract invariant generation problem as a Markov Decision Process
- a technique for finding an optimal policy for this MDP, incorporating a new program abstraction and an encoder-decoder-style neural policy
- an implementation of the proposed approach in a tool called Cider, which is incorporated as the type inference engine of an existing tool, and its evaluation in the context of arithmetic overflow checking

2 BACKGROUND ON CONTRACT INVARIANTS AND SOLTYPE

Because our technique builds on prior work [27] for discharging arithmetic overflows using a refinement type system, this section provides background on contract invariants as well as aspects of SolType [27] that are relevant to our proposed technique.

2.1 Contract Invariants for Overflow Checking

To gain some intuition about contract invariants and why they are necessary for overflow checking, consider the Solidity contract shown in Figure 1. This contract, called `ExampleToken`, uses a storage variable called *balances* that maps the address of each account to their balance. This contract also uses another variable called *totalSupply*, which tracks the number of available tokens. As standard, functions *mint* and *burn* produce and consume tokens respectively and update storage variables *balances* and *totalSupply* accordingly.

While the code shown in Figure 1 is free of arithmetic overflow errors, proving arithmetic safety by considering each function *in isolation* is actually infeasible. For instance, consider the addition at line 20. Since there is no overflow check guarding the update to *balances* the addition operation at line appears, at first glance, to be potentially unsafe. To see why it is safe, we need to consider the *relationship* between *totalSupply* and *balances*: Because *totalSuppy*

```
1  contract ExampleToken {
2    uint initialSupply = 100000000 * (10 ** 18);
3    uint totalSupply;
4    mapping(address => uint) balances;

6    ExampleToken() {
7      totalSupply = initialSupply;
8      balances[msg.sender] = totalSupply;
9    }

11   function burn(uint _value) {
12     require(_value <= balances[msg.sender]);
13     balances[msg.sender] = balances[msg.sender] - _value;
14     totalSupply = totalSupply - _value;
15   }

17   function mint(uint _to, uint _value) {
18     require(totalSupply + _value >= totalSupply);
19     totalSupply = totalSupply + _value;
20     balances[_to] = balances[_to] + _value;
21   }
22 }
```

Figure 1: An example Solidity contract.

is equal to the sum of all values stored in *balances*, the `require` annotation at line 18 (which performs a run-time check) also ensures the safety of the addition operation at line 20. However, without knowing this *global invariant*, we cannot prove the safety of line 20 by analyzing function `mint` *in isolation*.

Thus, as this example illustrates, discharging arithmetic overflows often requires having a so-called *contract invariant*, which is a logical formula that holds at the beginning of each transaction. For instance, for our running example, the following contract invariant is useful for proving overflow safety:

$$\sum_i balances[i] = totalSupply. \tag{1}$$

2.2 Refinement Type System for Solidity

SolType is a refinement type system for Solidity that is designed to prove arithmetic safety of smart contracts. The key observation behind SolType is that, in order to be useful for discharging overflows, the type system needs to relate values of scalar variables to aggregations over complex data structures (including deeply nested mappings). Based on this observation, type refinements (i.e., logical qualifiers) in SolType provide a rich vocabulary for performing aggregations over Solidity data structures. Specifically, a refinement type in SolType is of the form $\{\nu : T \mid \Phi\}$ where T is a base type and Φ is a logical formula that belongs to the grammar shown in Figure 2. Note that terms in this type system allow flattening mappings (`Flatten`), performing projections onto specific fields of a struct stored in mappings ($\texttt{Fld}_f$), and summing over all elements in a mapping (Sum).

Using SolType, we can prove the arithmetic safety of our running example from Figure 1 by expressing the contract invariant from Equation 1 as the following type annotation on *balances*:

$$balances : \{\mathsf{Map}(\mathsf{UInt}) \mid \mathsf{Sum}(\nu) = totalSupply\}. \tag{2}$$

Given just this type annotation, the underlying type system can then prove the safety of all arithmetic operations in Figure 1.

$\square \in \{\ ,\ =\}$			comparison operation
$\ \in \{\ ,\ ,\ ,\ \bullet\}$			arithmetic operation
Φ	::=		**boolean formula**
	|	true | false	boolean constants
	|	$A \square A$	comparisons
	|	$\neg \Phi$	negation
	|	$\Phi \wedge \Phi$	conjunction
M	::=		**mapping expression**
	|	x	mapping variable
	|	Flatten(M)	flattening
	|	Fld$_f(M)$	field projection
A	::=		**arithmetic expression**
	|	n	integer constants
	|	x	variable
	|	$A_1 \quad A_2$	binary operation
	|	$A_1[A_2]$	data structure access
	|	$.x$	struct field selector
	|	Sum(M)	sum of mapping
	|	MaxInt	maximum machine integer constant

Figure 2: Refinement terms used to construct contract invariants in SolType [27].

2.3 Type Inference with Soft Constraints

SolType also provides capabilities for automatically inferring refinement type annotations, including those on storage variables. The type inference problem in this context can be formulated as an instance of "MaxCHC", the optimization variant of the Constrained Horn Clause (CHC) solving problem [11]. In particular, the basic idea is to introduce second-order variables V that represent unknown refinement type annotations and then add *hard* and *soft* constraints over these variables. Here, the hard constraints encode that the inferred type annotation must be a valid one — e.g., the annotation on a storage variable must correspond to a contract invariant that is preserved by all transactions. On the other hand, soft constraints correspond to properties that we would like to prove, such as the absence of arithmetic overflows. Then, the type inference problem is to find a mapping from each variable $v \in V$ to a term in Figure 2 such that all hard constraints and as many as possible soft constraints are satisfied.

Definition 2.1 (Type Inference with Soft Constraints). Let $C = C_h \cup C_s$ be the set of typing constraints generated by SolType's constraint generation procedure. The type inference problem is to infer a typing assignment such that (1) all hard constraints C_h are satisfied; and (2) the number of satisfied soft constraints in C_s is maximized.

Since performing type inference for local variables is often quite easy in Solidity, the key challenge of type inference is to find suitable type annotations on storage variables. Furthermore, because type annotations on storage variables correspond to contract invariants, we can also formulate the problem of inferring contract invariants as the MaxCHC problem outlined in Theorem 2.1. Thus, in the remainder of this paper, we use the terms contract invariant inference and type inference interchangeably.

2.4 Type Inference Using Reinforcement Learning

The SolType's authors implemented a tool called Solid to both verify and infer contract invariants. However, Solid's inference capability is limited by its reliance on existing CHC solvers to infer unknown refinements. Inspired by how human experts construct contract invariants, we propose Cider, the first framework for learning contract invariants using reinforcement learning. A high-level overview of Cider is illustrated in Figure 3. Specifically, Cider consists of a training phase and an inference phase.

- During the training phase, an RL agent is trained using a set of contracts, which are further encoded in our novel arithmetic dependency graph (ADG) representation. Intuitively, an ADG distills critical arithmetic-related information from a smart contract. Specifically, the agent repeatedly samples a contract from the training set and proposes candidate invariants to be checked by the Solid verifier. In the presence of an incorrect invariant, the verifier also provides the agent with fine-grained feedback on the quality of the proposed candidates. The agent learns from the feedback and strives to propose better candidates during the subsequent iterations. The goal of the training phase is to learn a *policy* that maximizes the contract invariants in the training set.
- During the inference phase, Cider uses the policy learnt from the training phase to generate invariants for unseen contracts. Specifically, based on the ADG representation of an input contract, the agent proposes a sequence of invariants of decreasing likelihood (representing the agent's confidence level for each proposed invariant). Then, Cider searches for an invariant that discharges the largest number of overflow checks, and returns a hardened contract in which the operations that can not be proven safe are guarded with run-time checks.

The next several chapters are organized as follows. In Section 3, we formulate the contract invariant generation as a reinforcement learning problem by giving a MDP formulation, and illustrating necessary terminology such as states, actions, transitions, and the reward function in our setting. In Section 4, we discuss the details of our neural architecture, including our ADG abstraction of smart contracts. The algorithm that underlies the inference phase of Cider is explained in Section 5. Finally, we systematically evaluate the effectiveness of Cider in Section 6.

3 RL FORMULATION FOR CONTRACT INVARIANT LEARNING

Because our goal is to learn a probability distribution over useful contract invariants from *unlabeled data*, we formulate this task as a reinforcement learning (RL) problem. As RL tasks are typically stated in terms of a Markov Decision Process (MDP), we first provide necessary background on this topic and then show how to formulate our problem as an MDP.

Definition 3.1 (Markov Decision Process). A Markov decision process (MDP) can be formalized as a tuple $\mathsf{M} = (\mathsf{S}, \mathsf{A}, r, tr)$, where:

- S is a set of states, with $S_0 \quad S$ as the initial states and $S_t \quad S$ as the final states;

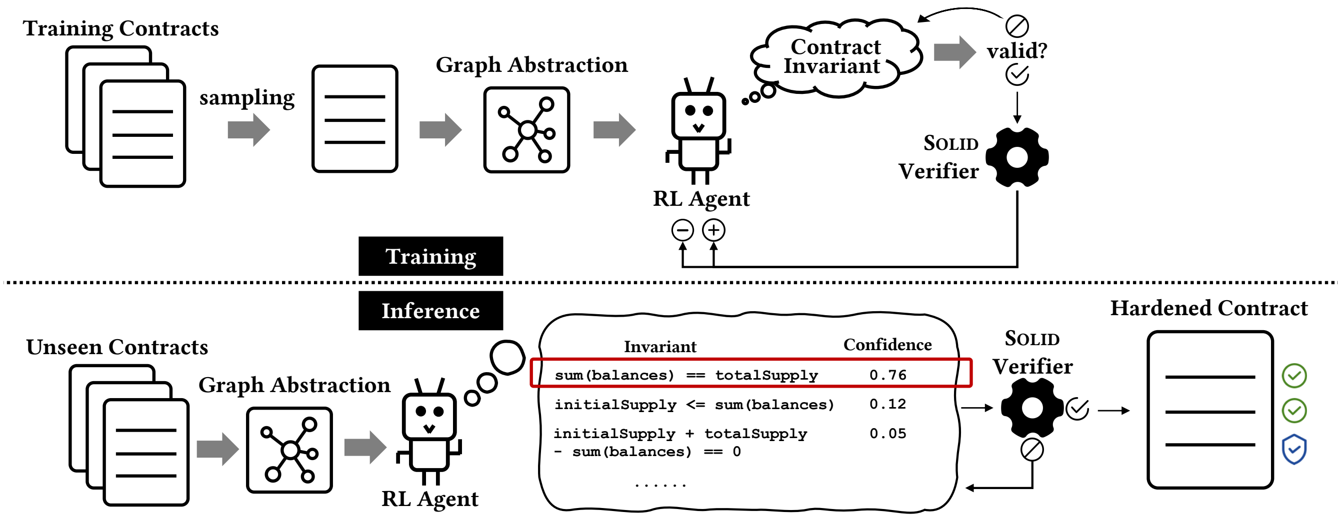


Figure 3: Overview of the CIDER framework.

- $\mathcal{A}$ is a set of actions;
- $r : \mathcal{S} \to \mathbb{R}$ is a function mapping each state to a reward;
- $tr : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a function that models the probability of transitioning from one state to another under a given action.

3.1 MDP Formulation for Contract Invariant Learning

In order to formulate our problem as an MDP, we next introduce the notion of a *partial contract invariant (PCI)*, which is a logical formula containing holes (i.e., missing expressions). More formally, a PCI is a sequence of grammar symbols (including both terminals and non-terminals) derived from the grammar of Figure 2. Words containing only terminals correspond to full contract invariants, and non-terminals in a word are referred to as holes. For instance, $\mathsf{Sum}(T)$ T is a PCI since the non-terminal T has not yet been expanded. Given a PCI Φ, we write $\Phi \rightsquigarrow \Phi'$ if Φ' can be obtained from Φ by expanding a single non-terminal using a grammar production from Figure 2.

Equipped with this definition of partial contract invariants, we can now formulate the contract invariant inference problem in terms of a Markov Decision Process with the following states, actions, transition, and reward function.

States. In our context, a state S is a pair (P, Φ) where P is a Solidity program and Φ is a partial contract invariant. For a given contract P, the initial state consists of P and the empty partial invariant (i.e., a single hole). Furthermore, (P, Φ) is a terminal (final) state if Φ is a complete contract invariant without any holes.

Actions and transitions. In our setting, actions correspond to expanding non-terminals in partial contract invariants using the grammar productions from Figure 2. In particular, applying an action to a state (P, Φ) yields another state (P, Φ') such that $\Phi \rightsquigarrow \Phi'$. Different actions can either expand different non-terminals or expand the same non-terminal using different grammar productions. Note that the transition function in our setting is deterministic, since there is a unique PCI that can be obtained by expanding a specific occurrence of a non-terminal using a specific grammar production.

Reward function. The design of the reward function promotes actions that lead to the construction of a useful contract invariant and penalizes those that will result in either incorrect "invariants" or valid but useless invariants. Thus, it takes into account both the hard and soft typing constraints from the formulation in Theorem 2.1. Specifically, the reward for a state $S = (P, \Phi)$ is defined as follows:

$$r(S) = \begin{cases} 0 & \text{if any hard constraint is violated} \\ \varsigma_s(S) & \text{otherwise,} \end{cases}$$

where $\varsigma_s(S)$ returns the percentage of soft typing constraints that are satisfied. Intuitively, we want a proposed invariant to satisfy all the hard constraints and as many soft constraints as possible. Thus, the reward function assigns a score of 0 to formulas that violate the hard constraint, and considers the *percentage* of satisfied soft constraints when all hard constraints are satisfied.

3.2 Learning Contract Invariants using RL

Given an MDP, the goal of reinforcement learning is to learn a *policy* π that will maximize expected reward. Specifically, a policy is a function $\pi(S, A) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that maps state, action pairs to a probability. A *rollout* of the policy is a sequence of moves ending in a terminal state. More formally, a rollout ξ is a sequence of triples of the form of:

$$\xi = ((S_0, A_0, r_0), \ldots, (S_{t-1}, A_{t-1}, r_{t-1}), (S_t, -, -)),$$

where S_0 is an initial state, S_t is a final state, A_i is sampled from the policy (denoted by $A_i \sim \pi(S_i)$), $r_i = r(S_{i+1})$ is the reward, and $S_i = tr(S_{i-1}, A_{i-1})$. Based on our MDP formulation, a rollout in our setting corresponds to the eventual construction of a full contract invariant. For example, the following sequence could be a rollout for the program P show in Figure 1:

$$\begin{aligned}\xi = \langle &((P, \square_0), \square_0 \to \square \leq \square, 0), ((P, \square_1 \leq \square_2), \square_1 \to totalSupply, 0),\\ &((P, totalSupply \leq \square_2), \square_2 \to initialSupply, 0.3),\\ &((P, totalSupply \leq initialSupply), \bot, \bot)\rangle,\end{aligned}$$

where $\square$ denotes a hole (i.e., non-terminal) and actions are represented as rewrite rules $\square \to t$ for some term t.

In this work, we solve the RL problem of learning an optimal policy using the standard *policy gradient technique* [26] which finds an optimal policy π^* that maximizes the following cumulative expected reward:

$$J(\pi) = \mathbb{E}_{\xi \sim \pi}\left[\sum_{i=0}^{|\xi|-1} R_i\right], \; where \; R_i = \sum_{j=i}^{|\xi|-1} r_j.$$

The policy gradient technique optimizes this function $J(\pi)$ by computing the gradient of $J(\pi)$ and then repeatedly taking a step in the direction of the gradient until it converges to a local maximum. In particular, given a policy π_θ parameterized over θ, the well-known *policy gradient theorem* [26] allows computing the gradient of $J(\pi_\theta)$ as follows:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\xi \sim \pi_\theta} \ell(\xi) \approx \frac{1}{n}\sum_{k=1}^{n} \ell(\xi_k), \tag{3}$$

where:

$$\ell(\xi) = \sum_{i=0}^{|\xi|-1} \nabla_\theta R_i = \sum_{i=0}^{|\xi|-1} R_i \nabla_\theta \log \pi_\theta(S_i, A_i).$$

In the next section, we show how to represent such a parameterized policy in our setting as a deep neural network with an encoder-decoder style architecture. Since the parameters of the policy correspond to the weights of this neural network, we can optimize this neural network using Equation 3.

4 NEURAL ARCHITECTURE FOR CONTRACT INVARIANT LEARNING

Recall that states in our MDP formulation from Section 3 are pairs of Solidity programs and partial contract invariants, and the actions are grammar productions (parameterized over the storage variables in the input contract). Since a policy takes as input a state and outputs an action, we need suitable representations of such complex states and actions. Our approach utilizes (1) a compact graph abstraction that captures relevant features of the smart contract, and (2) an encoder-decoder style neural network that featurizes states and actions.

Figure 4 illustrates our high-level approach for mapping states to actions. Specifically, given a Solidity program, we first statically analyze its source code to construct a so-called *arithmetic dependency graph* (ADG) that captures its key features relevant for discharging arithmetic overflows. We then compute a vector encoding of the ADG by using a graph neural network (GNN) and vectorize the PCI (which is also part of the state) using a gated recurrent unit (GRU) network. In particular, because GNNs are designed to generate vector embeddings for graphs, they are a natural choice for encoding our ADG abstraction. Furthermore, since we construct PCIs gradually by expanding each hole with a grammar production, the GRU architecture (commonly used for encoding sequences) is a natural choice for encoding PCIs. We refer to the combination of GNN and GRU as the encoder part of the network. Once we have a vector encoding of the contract and the PCI, we concatenate their vector representations and use a decoder network to generate a probability distribution over actions. In particular, our decoder network is a feed-forward neural network (FFNN), augmented with a pointer mechanism [28] for handling the program-specific aspect of the action space. In the remainder of this section, we discuss the ADG abstraction and various aspects of the neural architecture in more detail.

4.1 Arithmetic Dependency Graph Abstraction

While smart contracts typically consist of hundreds of lines of code, most of this code is actually irrelevant for our goal of proving arithmetic safety. Hence, instead of representing Solidity programs using a sequence of tokens or as a control-flow-graph, we instead propose the arithmetic dependency graph abstraction (ADG) to capture features that are more relevant to proving arithmetic safety.

Our ADG representation is motivated by the following observation: invariants that are useful for discharging overflow checks often relate storage variables that (1) have arithmetic operations performed on them, and (2) have a data dependence on each other either directly or indirectly. Based on this observation, our ADG abstraction encodes both arithmetic- and non-arithmetic-related data flow between storage variables. More formally, given a program P, the ADG abstraction is a graph (V, E) where vertices V correspond to variables used in the program, including both local and storage variables, and edges E encode dependencies between these variables. In particular, we distinguish between the following types of edges:

Decrement/increment edges: A so-called decrement edge (v, u) indicates that v is decremented by u. Dually, an increment edge (v, u) indicates that v is incremented by u.

Data-flow edges: To capture data dependencies beyond increment and decrement operations, the ADG abstraction also includes more generic data-flow edges (v, u) indicating the presence of an assignment where v appears on the left-hand-side and u appears on the right-hand-side.

For instance, Figure 5 shows the ADG abstraction for the code from Figure 1. In this example, there is a data flow edge from *totalSupply* to *balances* due to the assignment in the constructor. Similarly, there is an increment edge from the *_value* parameter of *mint* to both *totalSupply* and *balances* since they are both incremented by itvalue in that function.

4.2 Encoder Network

Recall that the goal of the encoder network is to generate a vector representation of each MDP state. In this subsection, we describe the GNN used for encoding the ADG abstraction as well as the GRU network for encoding partial contract invariants.

GNN for Encoding ADGs. Since the standard way to generate a vector encoding of graphs is via graph neural networks, GNNs are a natural choice for encoding our ADG abstraction. Given an

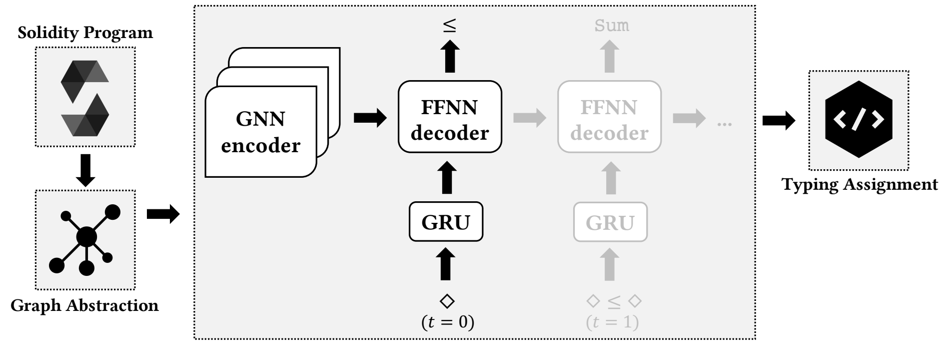


Figure 4: Overview of neural architecture.

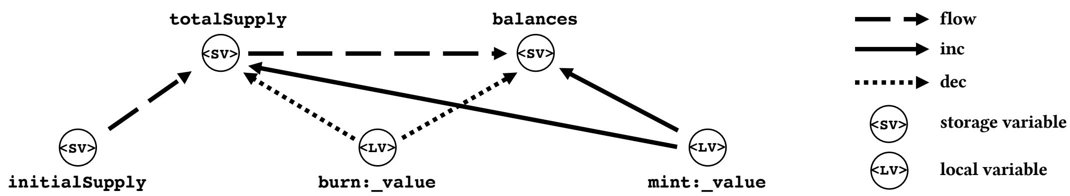


Figure 5: Graph abstraction of the example contract in Figure 1.

ADG $G = (V, E)$ as input, we train the GNN via a message passing mechanism that exchanges information between neighboring nodes to update the vector encoding h_v of each node in $v \in V$. Let h_v^l denote the vector encoding of node v during the l'th message passing iteration as h_v^l. Then, we update the encoding as follows:

$$h_v^{l+1} = \sigma\Big(\sum_{u \in \mathsf{N}_v} f(h_v^l, h_u^l, e(v, u))\Big) \qquad h_v^0 = \psi(v), \tag{4}$$

Here, N_v is the set of neighboring nodes of v, $e(v, u)$ is the edge type (increment, decrement, or data flow), and σ is an activation function. Because an ADG has a fixed set of node types (local vs storage), the initial representation h_v^0 for a given node v is given by a learnable embedding of its node type $\psi(v)$. Function f used in Equation 4 performs aggregation over the neighboring nodes and is defined as follows in our setting:

$$f(h_v, h_u, e(v, u)) = W_{e(v,u)} h_u,$$

where $W_{e(v,u)}$ is a learnable neural network parameter. Finally, given the vector encoding h_v of each node, we compute the encoding of the entire ADG as follows:

$$enc(G) = h_P = \frac{1}{|V|} \sum_{v \in V} h_v,$$

GRU for Encoding PCIs. Given a PCI Φ that is represented by a sequence of actions, we generate its vector encoding $enc(\Phi)$ using a standard GRU network:

$$enc(\Phi) = h_\Phi = GRU(\psi(\Phi)), \ where \ \psi(\Phi) = (\psi(A_0), \psi(A_1), \ldots).$$

Here $\psi(A)$ is a learnable embedding for an action A.

4.3 Decoder Network

In this subsection, we turn our attention to the decoder network, which maps the output of the encoder network (concatenation of h_P and h_Φ from the previous section) to a probability distribution over actions. The decoder network is a feed-forward neural network augmented with a pointer-like mechanism [28] (to handle our program-dependent action space).

Given the output h_s of the encoder, the goal of the decoder is to assign a probability to the set of all available actions A. To achieve this goal, we first project h_s to the feature space of the actions using an FFNN:

$$h = FFNN(h_s).$$

where h is the projected state vector encoding. Then, because the set of actions varies across different programs, we employ a pointer mechanism to compute a preference score for every action as follows:

$$\varphi(h, \mathsf{A}) = \mathrm{softmax}(\Psi_{\mathsf{A}} h + W_b \Psi_{\mathsf{A}}) \qquad where \ \ \Psi_{\mathsf{A}} = [\psi(A_0); \psi(A_1); \ldots], A \in \mathsf{A}. \tag{5}$$

Here, W_b is a neural network parameter, $\psi(A)$ is the encoding for action A (see below), and the resulting $\varphi(h, \mathsf{A})$ is a vector of assigned probabilities for all actions. The encoding $\psi(A)$ of an action A depends on whether the action (i.e., grammar production) involves a program-dependent term, such as the name of a storage variable in the input contract. Specifically, if A is a production of the form $\rightarrow v$ where v is a program variable, then $\psi(A)$ is the vector representation of v in the GNN from the previous subsection. Otherwise $\psi(A)$ is learnable embedding (denoted by ψ_A) for the program-agnostic grammar production:

$$\psi(A) = \begin{cases} h_v & \text{if } A \text{ is program-dependent and node } v \\ & \quad \text{represents } A \text{ in the GNN} \\ \psi_A & \text{otherwise .} \end{cases}$$

5 USING LEARNED POLICY FOR VERIFYING ARITHMETIC SAFETY

In this section, we describe how to use the learned neural policy from the previous section to perform verification, as summarized in Algorithm 1. This algorithm takes as input (1) a Solidity program P, (2) a trained encoder network N_e (described in Section 4.2), (3) a trained decoder network N_d (from Section 4.3), and (4) a probability threshold ϵ that controls how many candidate invariants are enumerated. If verification is successful, the program returns "Verified"; otherwise, it returns a new "hardened" program P' with appropriate runtime checks inserted to ensure the absence of arithmetic overflows.

The verification algorithm starts by using the type system from Section 2 to generate a set of hard and soft type constraints as discussed in Theorem 2.1. We assume that the only unknown in the generated type constraints $C_h \cup C_s$ is the contract invariant because (a) the types of storage variables are easily derived from the contract invariant and (b) the types of local variables can be easily inferred via local analysis from the types of storage variables.[2] Hence, given a contract invariant $\mathcal{I}$, it is easy to check which of the hard and soft constraints are satisfied.

Once the type constraints over the unknown invariant are generated, the Verify procedure enters a loop (lines 4–12) that terminates either when the contract is verified or an invariant that satisfies the largest number of soft constraints is found. In each iteration, the GetInv procedure (discussed later) is called to find a candidate contract invariant (line 5). If the returned candidate $\mathcal{I}$ does not satisfy the hard constraints (line 7), this means that $\mathcal{I}$ is not a valid contract invariant; so the algorithm moves on to the next candidate. Otherwise, it checks how many soft constraints $\mathcal{I}$ satisfies. If all of them are satisfied, then contract P is reported as being safe (line 10). If this is not the case but $\mathcal{I}$ satisfies more soft constraints than previously enumerated contract invariants, the best invariant found so far is updated at line 12. Once the main loop (lines 4–12) terminates, lines 13–15 identify potentially unsafe arithmetic operations, which correspond to soft constraints that cannot be verified using *inv*. To ensure arithmetic safety, Algorithm 1 generates an instrumented contract by adding a run-time overflow check for each unsafe operation (lines 15–16).

Next, we discuss the GetInv procedure (shown on the right side of Algorithm 1) for generating candidate contract invariants. This procedure uses the encoder and decoder neural networks from Section 4 (trained via policy gradient) to construct candidate invariants in decreasing order of probability. To this end, it maintains a worklist W of partial contract invariants (along with their probability according to the neural policy). The algorithm starts by constructing the Arithmetic Dependency Graph (ADG) abstraction discussed in Section 4.1. Then, it enters a loop (lines 21–31) where, in each iteration, the highest probability PCI is dequeued from the worklist and expanded using a grammar production. Specifically, the procedure first generates a vector encoding of the program and the current PCI using encoder network N_e (line 23) and maps this state to a probability distribution over actions using the decoder network N_d (line 24). Here, each action is a triple of the form (N, R, p_i) where N is a hole (i.e., specific occurrence in of a non-terminal) in the PCI, R is a grammar production from Figure 2, and p_i is the probability of this action according to the learned policy. The GenInv procedure expands the current PCI using the most likely action (line 26) and returns it as the candidate invariant if there are no holes left. Otherwise, the new PCI is added to the worklist if its probability exceeds the specified threshold ϵ.

6 IMPLEMENTATION AND EVALUATION

We have implemented the proposed algorithm in a new tool called Cider[3] written in Python. Cider's GNN is instantiated by TransformerConv [20] built on top of PyTorch Geometric [6] and RLlib [14]. As discussed in the previous section, Cider is incorporated into Solid [27] as its type inference engine and leverages Solid's type checking capabilities.

In what follows, we describe the results of our empirical evaluation which is designed to answer the following research questions:

> **RQ1:** Can Cider infer better contract invariants than existing tools in the context of arithmetic overflow checking?
> **RQ2:** Does Cider improve inference time compared to other baselines?
> **RQ3:** Is the ADG abstraction more useful compared to a more standard program representation?

Benchmarks. We evaluate Cider on a set of 120 Solidity programs taken from prior work. 60 of our evaluation benchmarks come from [27], and the remaining 60 contracts are taken from [15].

Training. We trained Cider on 100 contracts that are disjoint from the test set and that are sampled from Etherscan. To avoid duplication between the testing and the training sets (and among themselves), we cluster the contracts based on a number of syntactic (e.g., LOC, similarity scores) and semantic features (e.g., total number of soft and hard constraints, percentage of soft constraints that can be proved using the "true" invariant, the similarity scores of the ADGs, etc.). We then sample one candidate from each cluster and perform manual inspection to avoid duplication. We note that because the verification procedure dominates the overall running time, to reduce the actual training time, we used smaller contracts for training than for testing, which can also be used to validate whether the agent can generalize to complex, unseen instances. The training set only contains contracts that take fewer than 10 seconds per verifier call, and on average contain 250 lines of code (LOC). The testing set consists of contracts with an average LOC of 400 and maximum of over 1500. The training was performed on a MacBook Pro with 2GHz Quad-Core Intel Core i5 CPU and 16GB of RAM.

[2] Most contracts do not contain loops inside methods, so loop invariants are typically not needed in this context.

[3] stands for Contract Invariants via DEep Reinforcement learning

Algorithm 1 for verifying/hardening programs against arithmetic overflow

Input: Program P, Encoder N_e, Decoder N_d, Threshold ϵ
Returns: Verified(P) or Hardened(P')

```
procedure Verify(P, N_e, N_d, ε)
  best ← 0; inv ← true
  (C_h, C_s) ← GenTypeConstraints(P)
  while true do
    φ ← GetInv(P, N_e, N_d, ε)
    if φ = ⊥ then break
    if φ ⊭ C_h then continue
    v ← NumSatisfied(C_s, φ)
    if v = |C_s| then
      return Verified(P)
    if v > best then
      best ← v; inv ← φ
  P' ← P
  Υ ← GetUnsafeOps(P, C_s, inv)
  for each op ∈ Υ do
    P' ← AddOverflowCheck(P', op)
  return Hardened(P')

procedure GetInv(P, N_e, N_d, ε)
  W ← {(⊥, 1.0)};
  G ← ConstructADG(P)
  while W ≠ ∅ do
    (pci, p) ← PopBest(W);
    h_s ← N_e(G, pci)
    A ← N_d(h_s)
    for each (N, R, p_i) ∈ A do
      pci' ← ExpandNonTerminal(pci, N, R)
      p' ← p · p_i
      if pci' is complete then
        yield pci'
      else if p' > ε then
        W ← W ∪ {(pci', p')}
  return ⊥
```

Baselines. To evaluate the quality of the learned invariants as well as inference time, we compare Cider against the following two baselines:

- SolidChc: As mentioned earlier, Solid [27] incorporates a type inference engine that leverages a state-of-the-art CHC solver, namely Spacer [13]. We refer to this version of Solid as SolidChc and use it as one of our baselines.
- Houdini: We also implement a Houdini-style inference algorithm that generates conjunctive invariants. This baseline generates all possible atomic predicates by unwinding the grammar from Figure 2 up to a fixed bound and then generates the strongest conjunctive invariant over this universe in the standard way [7].

Experimental setup. Recall from Algorithm 1 that our algorithm uses a parameter ϵ that controls how we sample rollouts from the policy. In our evaluation, we use $\epsilon = 0.2$ and sample at most 5 rollouts. We also set a time limit of 10 minutes and, for runs that exceed this limit, we consider the best invariant found within that limit. To perform the comparison against SolidChc, we use a time limit of 10 seconds per Z3 query and a ten-minute total timeout. We also use a 10 minute time limit when evaluating Houdini.

6.1 Evaluating the Quality of Contract Invariants

In this subsection, we evaluate the quality of the contract invariants inferred by Cider in terms of the percentage of safe arithmetic overflows that can be discharged using contract invariants inferred by Cider and our two baselines.

The results of this evaluation are shown in Figure 7. Here, the column labeled "Ops" shows the total number of arithmetic operations across all contracts, and the column labeled "Safe" shows the total number of *safe* operations. The next two columns show the number and percentage of safe operations that can be discharged for each of the three tools. As is evident from these numbers, the invariants inferred by Cider allow discharging more arithmetic operations.

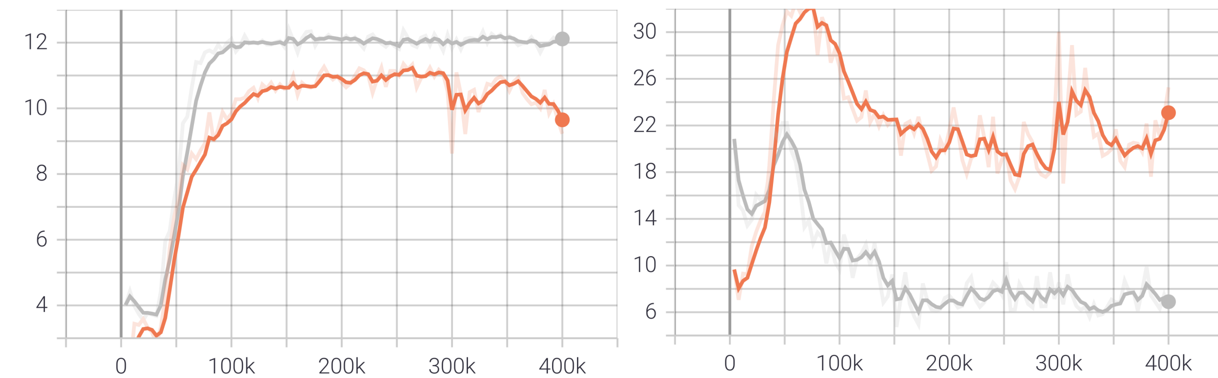
To provide further intuition about these results, Figure 8 shows the number and percentage of non-trivial contract invariants that can be inferred by SolidChc and Cider. Here, by "non-trivial", we mean a contract invariant that is not logically equivalent to true. Across all contracts, Cider is able to infer a non-trivial invariant for 85.8% of the contracts, whereas SolidChc infers non-trivial invariants for 50.8% of the contracts.

Result for RQ1: Invariants inferred by Cider discharge 97.1% of the arithmetic checks, while the SolidChc and Houdini baselines discharge 87.2% and 79.5%, respectively.

6.2 Evaluating Efficiency of Inference

In addition to the quality of the contract invariants, another important evaluation metric is the time it takes to infer these invariants. This is particularly important in a setting like the one we target, where the invariant generation engine is used for type inference and compile-time arithmetic overflow checking. Thus, we also compare the invariant inference time of Cider against our two baselines. The inference time includes query timeouts, which typically happen when the solver cannot prove a soft constraint using the supplied invariant. We note, however, that timeouts affect Cider and SolidChc equally, since Cider also relies on the verifier component of SolidChc to check proposed invariants.

The results of this evaluation are shown in Figure 9. In terms of average (resp. median) verification time, Cider is 1.8× (resp. 6.5×) faster than SolidChc and 15.5× (resp. 53.4×) faster than Houdini.



(a) Average reward per episode

(b) Loss function

Figure 6: Comparison of training trajectories for Cider (grey) and Cider-NoADG (orange).

Ops	Safe	Houdini		SolidChc		Cider	
		Safe	%	Safe	%	Safe	%
1828	1190	946	79.5%	1038	87.2%	1155	97.1%

Figure 7: Evaluation of the quality of inferred invariants by each tool. "Ops" is the total number of arithmetic operations across all contracts, and "Safe" shows the total number of safe operations. For each tool, the "Safe" and "%" columns record the number and percentage of arithmetic operations that can be proven safe using invariants proposed by the tool.

Total	SolidChc		Cider	
	Non-trivial	%	Non-trivial	%
120	61	50.8%	103	85.8%

Figure 8: The number and percentage of contracts for which Houdini, SolidChc, and Cider can infer non-trivial invariants.

	Houdini	SolidChc	Cider
Avg (s)	530.2	72.9	38.6
Median (s)	600.0	44.9	6.9

Figure 9: Inference time statistics for Houdini, SolidChc, and Cider

Thus, overall, Cider increases the precision of the verifier while also reducing its running time significantly.

Result for RQ2: Cider performs verification 1.8 faster compared to SolidChc and 15.5 compared to Houdini on average.

Ops	Safe	Cider-NoADG		Cider	
		Safe	%	Safe	%
1828	1190	1040	87.4%	1155	97.1%

Figure 10: Comparison of the number and percentage of arithmetic operations can be proven safe using invariants inferred by Cider and Cider-NoADG.

6.3 Ablation Study

In this section, we describe the results of an ablation study that we use to evaluate the effectiveness of the ADG abstraction discussed in Section 4.1. In particular, we consider Cider-**NoADG**, which does not use our proposed ADG abstraction. Instead, it feeds the AST representation of the program directly to the graph neural network. To study the effect of our ADG graph abstraction, we compare Cider with Cider-NoADG in terms of both their training trajectories and their effectiveness during testing.

Training. Figure 6 shows the average reward and the loss function per episode for each variant. We observe that Cider-NoADG, which utilizes larger, more noisy graphs, converges more slowly than Cider. In the long run, Cider-NoADG also fails to learn a good policy even on the training data, and consistently scores a lower reward and a higher loss than Cider.

Impact on quality of invariants. Figure 10 compares the number and percentage of arithmetic operations that can be proven safe by Cider and Cider-NoADG. The policy learned by Cider-NoADG fails to generalize to unseen contracts, resulting in lower-quality contract invariants that can only discharge 87.3% of overflow checks.

Result for RQ3: Our ADG representation is useful for effective training, and it improves the quality of invariants needed for proving arithmetic safety.

6.4 Threats to Validity

Quality of the corpus. Even though the neural architecture in Figure 4 is more resilient to the limitation of the existing data set, the performance of CIDER may still be sensitive to the quality of the training data. To mitigate this concern, we avoid duplication and obtain representative training data by clustering contracts based on their syntactic (e.g., LOC, similarity scores) and semantic features (e.g., number of constraints, similarity scores of ADGs, etc.). We then sample candidates from each cluster and perform manual inspection to avoid duplication. In the future, we also plan to leverage transfer learning to incorporate smart contracts written in other languages (e.g., Vyper and Rust, etc.).

Benchmark selection. Because SOLID—which CIDER uses as the verifier—only supports a core subset of Solidity features, the benchmarks in our testing set may not represent the actual distribution of the contracts on ETHERSCAN. Supporting every nascent or deprecated feature used by contracts "in the wild" may require significant engineering effort. However, since we use small contracts for training and leave the medium and larger contracts for testing, it suggests that the neural agent is able to generalize to more complex instances. Therefore, we believe our comparison is sufficient to show the strength of our technique. Furthermore, since both our neural architecture and the inference algorithm are designed in domain-agnostic way, we also believe our technique can generalize to other unseen contracts.

7 RELATED WORK

Smart contract security has become a very active research topic in the past few years. In what follows, we discuss prior work in this space that is most closely related to our proposed approach.

Smart contract verification. Due to their immutable nature once deployed, verification of smart contracts has received significant attention from both the formal methods and security communities. Among recent efforts, many address the problem of verifying arithmetic overflow safety, as overflows in this setting are known to be a serious security concern. One of the efforts in this space [12] encodes the semantics of arithmetic operations into verification conditions and tries to discharge them using off-the-shelf SMT solvers. Another recent effort approaches this problem from the synthesis angle and employs a CEGIS-style algorithm to infer arithmetic invariants to prove overflow safety [24]. As mentioned earlier, our work builds on SOLTYPE [27], a refinement type system designed for checking arithmetic safety. However, as demonstrated in our evaluation, it provides a more effective type inference mechanism that can be used to further increase automation while improving inference time. We do not perform an empirical comparison against arithmetic safety checkers other than SOLID, as prior work [27] has shown that they are outperformed by SOLID.

Going beyond arithmetic safety, several verification techniques target a more general class of functional correctness properties. For example, VERX [18] uses symbolic execution to verify functional correctness properties written in Past LTL. Other tools for verifying a more general class of properties include SMARTPULSE [25], VERISOL [29], SAILFISH[2], and K-framework [17]. We note that all of these techniques utilize contract invariants and would benefit from advances in automated inference of such invariants. While our proposed approach primarily targets invariants that are useful for arithmetic overflow checking, we believe that adaptations of our proposed approach (e.g., using a different program abstraction) could help infer higher-quality contract invariants for other verification tasks as well.

Invariant generation. There has been extensive work on invariant generation—particularly on generating numeric loop invariants—using techniques such as abstract interpretation [4, 9, 16, 23], predicate abstraction [7], logical abduction [5], randomized search [19], decision trees [8, 30], and deep learning [21]. However, contract invariants differ from loop invariants in other settings because they typically require reasoning about aggregate properties of complex data structures like nested mappings. Hence, verifying important security properties of smart contracts often requires inference of more complex invariants than numeric relationships between scalar variables.

Machine learning for verification. Recently, there has been significant interest in applying machine learning techniques to program verification. For example, [21] uses stochastic search to generate candidate invariants; [3] applies reinforcement learning for relational verification, and [30] combines linear classification with decision tree learning for solving CHC constraints. The work that most closely resembles ours is [21], which leverages reinforcement learning to infer loop invariants of C programs. Their technique utilizes a neural architecture that is designed to mimic the way a human reasons about program correctness. To that end, their proposed solution consists of three key pieces, including (1) a structured memory representation of the program, (2) a multi-step auto-regressive model to incrementally construct the loop invariant, and (3) an attention component that allows focusing on different parts of the program. There are several differences between their approach and ours, including the application domain (loop invariants in C vs. contract invariants in Solidity), the abstraction used for generating a program embedding (ADG vs. AST/CFG), the reward function, the neural architecture, and the underlying RL algorithm. In addition to these differences, our method is designed to be used as the type inference backend for a refinement type system.

8 CONCLUSION

We presented a new technique, based on reinforcement learning, for inferring smart contract invariants that are useful for proving arithmetic safety properties. Our approach learns a neural policy that produces a distribution over candidate invariants and uses the learned policy to perform type inference in a refinement type system.

We have implemented this approach in a tool called CIDER and incorporated it into an existing verifier (based on refinement type checking) for proving arithmetic safety. Our evaluation shows that CIDER can discharge more arithmetic operations while significantly reducing verification time compared to two different baselines, leading to faster verification and hardened contracts with fewer run-time assertions.

REFERENCES

[1] Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn Clause Solvers for Program Verification, pp. 24–51. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2, https://doi.org/10.1007/978-3-319-23534-9_2

[2] Bose, P., Das, D., Chen, Y., Feng, Y., Kruegel, C., Vigna, G.: SAILFISH: vetting smart contract state-inconsistency bugs in seconds. In: 43rd IEEE Symposium on Security and Privacy, S&P 2022, San Francisco, US, May 22-26, 2022. IEEE (2022), https://arxiv.org/abs/2104.08638

[3] Chen, J., Wei, J., Feng, Y., Bastani, O., Dillig, I.: Relational verification using reinforcement learning. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019). https://doi.org/10.1145/3360567, https://doi.org/10.1145/3360567

[4] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977)

[5] Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013. pp. 443–456. ACM (2013). https://doi.org/10.1145/2509136.2509511, https://doi.org/10.1145/2509136.2509511

[6] Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)

[7] Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/java. In: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity. p. 500–517. FME ’01, Springer-Verlag, Berlin, Heidelberg (2001)

[8] Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) 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, vol. 8559, pp. 69–87. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_5, https://doi.org/10.1007/978-3-319-08867-9_5

[9] Ghorbal, K., Goubault, E., Putot, S.: The zonotope abstract domain taylor1+. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 627–633. Springer (2009)

[10] Graf, S., Saidi, H.: Construction of abstract state graphs with pvs. In: Grumberg, O. (ed.) Computer Aided Verification. pp. 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)

[11] Gurfinkel, A., Bjørner, N.: The science, art, and magic of constrained horn clauses. In: 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2019, Timisoara, Romania, September 4-7, 2019. pp. 6–10. IEEE (2019)

[12] Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12031, pp. 161–179. Springer (2019)

[13] Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: International Conference on Computer Aided Verification. pp. 846–862. Springer (2013)

[14] Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., Stoica, I.: RLlib: Abstractions for distributed reinforcement learning. In: Dy, J., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 3053–3062. PMLR (10–15 Jul 2018), https://proceedings.mlr.press/v80/liang18b.html

[15] Mariano, B., Chen, Y., Feng, Y., Lahiri, S.K., Dillig, I.: Demystifying loops in smart contracts. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. pp. 262–274. IEEE (2020). https://doi.org/10.1145/3324884.3416626, https://doi.org/10.1145/3324884.3416626

[16] Miné, A.: The octagon abstract domain. In: Burd, E., Aiken, P., Koschke, R. (eds.) Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE’01, Stuttgart, Germany, October 2-5, 2001. p. 310. IEEE Computer Society (2001). https://doi.org/10.1109/WCRE.2001.957836, https://doi.org/10.1109/WCRE.2001.957836

[17] Park, D., Zhang, Y., Rosu, G.: End-to-end formal verification of ethereum 2.0 deposit smart contract. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 151–164. Springer (2020)

[18] Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP) (2020)

[19] Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. Formal Methods Syst. Des. **48**(3), 235–256 (2016). https://doi.org/10.1007/s10703-016-0248-5, https://doi.org/10.1007/s10703-016-0248-5

[20] Shi, Y., Huang, Z., Feng, S., Zhong, H., Wang, W., Sun, Y.: Masked label prediction: Unified message passing model for semi-supervised classification. In: Zhou, Z.H. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21. pp. 1548–1554. International Joint Conferences on Artificial Intelligence Organization (8 2021). https://doi.org/10.24963/ijcai.2021/214, https://doi.org/10.24963/ijcai.2021/214, main Track

[21] Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada. pp. 7762–7773 (2018), https://proceedings.neurips.cc/paper/2018/hash/65b1e92c585fd4c2159d5f33b5030ff2-Abstract.html

[22] Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2inv: A deep learning framework for program verification. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 151–164. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_9, https://doi.org/10.1007/978-3-030-53291-8_9

[23] Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 46–59. ACM (2017)

[24] So, S., Lee, M., Park, J., Lee, H., Oh, H.: Verismart: A highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1678–1694. IEEE Computer Society, Los Alamitos, CA, USA (may 2020). https://doi.org/10.1109/SP40000.2020.00032, https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00032

[25] Stephens, J., Ferles, K., Mariano, B., Lahiri, S.K., Dillig, I.: Smartpulse: Automated checking of temporal properties in smart contracts. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 555–571. IEEE (2021)

[26] Sutton, R.S., McAllester, D., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Solla, S., Leen, T., Müller, K. (eds.) Advances in Neural Information Processing Systems. vol. 12. MIT Press (2000), https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf

[27] Tan, B., Mariano, B., Lahiri, S., Dillig, I., Feng, Y.: Soltype: Refinement types for solidity. arXiv preprint arXiv:2110.00677 (2021)

[28] Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 28. Curran Associates, Inc. (2015), https://proceedings.neurips.cc/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf

[29] Wang, Y., Lahiri, S.K., Chen, S., Pan, R., Dillig, I., Born, C., Naseer, I., Ferles, K.: Formal verification of workflow policies for smart contracts in azure blockchain. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 87–106. Springer (2019)

[30] Zhu, H., Magill, S., Jagannathan, S.: A data-driven chc solver. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 707–721. PLDI 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192416, https://doi.org/10.1145/3192366.3192416