Learning Invariants for Incomplete Heap Verification Engines

Abstract

Existing learning algorithms for synthesizing invariants for program verification work using concrete counterexamples returned by a verification engine. However, when the verification engine implements incomplete procedures for undecidable logics (such as a rich separation logic), it cannot give concrete counterexamples that a learner can use. We present a new learning framework to synthesize conjunctive invariants for sound but incomplete verification engines. The key idea in our learning framework is to encode *non-provability* information provided by verification engines as set constraints on predicates that make up the invariant. We then reduce learning from such non-provability information to an existing learning framework for invariants, called ICE. We hence build new invariant synthesis learning algorithms for incomplete verification oracles, both be using existing ICE learners as well as developing new ones. Finally, we apply our learning framework in the setting of a verification engine that uses a class of incomplete proofs for heap verification, called natural proofs. We implement our techniques, extensively evaluate them, and show that our technique can automatically synthesize inductive invariants for a large suite of heap-manipulating programs annotated using an undecidable separation logic.

1. Introduction

The paradigm of *deductive verification* [32, 43] combines manual annotations and semi-automated theorem proving to prove programs correct. Programmers annotate code they develop with contracts and inductive invariants, and use high-level directives to an underlying, mostly-automated logic engine to verify their programs correct. Several mature tools have emerged that support such verification, in particular tools based on the intermediate verification language BOOGIE [4] and the SMT solver Z3 [26], such as VCC [19] and DAFNY [54]. Several applications that use such tools to prove systems correct using manual annotations have been developed, including Microsoft Hypervisor verification [20], reliable systems code such as VERVE[73], ExpressOS [56], and Ironclad apps [41], as well as distributed systems in IronFleet [42]. Fully automated use of such engines for shallow specifications have also emerged, such as CORRAL [52] for verifying device drivers, CST [15] to certify transactions in online services, and GPUVERIFY [7] to ensure race-freedom of GPU kernels.

Viewed from the lens of deductive verification, the primary challenges to automating verification are two-fold. First, even when strong annotations in terms of contracts and inductive invariants are given, the validity problem for the resulting verification conditions is often undecidable (e.g., in reasoning about the heap). Second, the synthesis loop of invariants and strengthenings of contracts that prove a program correct needs to be automated so as to lift this burden currently borne by the programmer.

One approach to the first problem (intractability of validity checking of verification conditions) is to build automated, sound but incomplete verification engines for validating verification conditions that work for most programs, thus skirting the undecidability barrier. Several such techniques already exist; for instance, for reasoning with quantified formulas, tactics such as E-matching [25, 27], pattern-based quantifier instantiation [27], and model-based quantifier instantiation [37] are effective in practice, though they are not complete for most background theories. In the realm of heap verification, the so-called *natural proof* method explicitly aims to provide automated and sound but incomplete methods for checking validity of verification conditions with specifications in separation logic [18, 55, 61, 65].

Turning to the second problem of invariant generation, white-box techniques (such as interpolation [46, 57] and IC3 [11]) that generalize from information gathered in proving underapproximations of the program correct are quite effective [8], but their efficacy in dealing with programs where the underlying logics are undecidable is unclear. However, a class of *black-box* methods have emerged recently that propose to *learn* invariants from information provided through concrete program configurations. The key idea here is that the invariant synthesis engine (the *learner*) is largely agnostic of the program that is being verified, but is *data-driven*. This approach relies on a verification engine (the *teacher*) that examines candidate invariants the synthesis engine proposes and refutes them using concrete program configurations. In particular, the ICE learning model [34] is a robust learning model for invariant synthesis that asks the verification engine to provide positive counterexamples (if the invariant is too strict), negative counterexamples (if the invariant is too liberal leading to violating assertions), or implication counterexamples (if the invariant is non-inductive). Several effective ICE learning algorithms have been proposed, based on using constraint-solvers [34], stochastic search [69], and machine-learning algorithms for learning decision trees [36].

Combining learning and incomplete verification engines

Current learning-based synthesis techniques (including the ICE learning model) require to be paired with a verification engine that returns concrete program configurations. More precisely, the invariant synthesis works in rounds: in each round, the learner proposes an invariant and the verification engine attempts to verify the program using this invariant; if the verification fails, the verification engine returns concrete program configurations that help the learner to come up with a true invariant. The requirement to generate concrete program configurations, however, makes it impossible to employ verification engines for complex undecidable theories for which we have only incomplete decision procedures, which cannot provide such concrete counterexamples in case a validity check fails. When verification of the program fails with a purported invariant provided by the learner, the verification engine can only return information on what fragments of the proof failed.

The primary aim of this paper is to develop a new learning framework for invariant synthesis that allows pairing learning algorithms with incomplete verification engines, where the verification engines

provide only non-provability information, as opposed to concrete program configurations, as counterexamples.

The Learning Sets under Constraints (LSUC) Model for Synthesizing Conjunctive Invariants

The first and primary technical contribution of this paper is a learning framework that allows combining learners with incomplete verification engines to synthesize invariants. The key idea is that when the verification engine fails to prove the program using the learner-proposed invariant, it provides *non-provability information* to the learner.

In particular, we consider the problem of learning *conjunctive invariants* composed of predicates drawn from a large but fixed class of enumerated predicates. We show that when verification conditions fail to be proven, the non-provability information can be encoded using a particular class of constraints on the set of predicates that are allowed to appear conjunctively in the invariant. We introduce a learning model called *Learning Sets Under Constraints (LSUC)* where the verification engine encodes non-provability information as constraints on sets that it communicates to the learner. The LSUC model then allows us to pair any LSUC learner with a sound but incomplete verification engine that can return such non-provability information in order to synthesize invariants. Note that the goal here is to synthesize invariants that the incomplete prover can prove correct; correct invariants that are not provable by the verification engine are not very useful as they cannot be validated by the engine.

We then turn to solving the LSUC learning problem. We show that we can reduce, in general, the LSUC learning problem to the more well-studied problem of ICE learning of conjunctive formulas [34], where in the latter model the ICE learner learns from concrete Boolean vectors encoding values of predicates. This reduction allows us to use any ICE learner for conjunctive formulas available in the literature, hence giving us learners that synthesize invariants even when the verification engine in incomplete.

The HOUDINI algorithm [31] is in fact a classical ICE learner of conjunctions, and, using our reduction, can be readily applied to learn invariants from non-provability information. In addition, we propose a new ICE-learner for conjunctions, called SORCAR[1]. Both HOUDINI and SORCAR are polynomial-time learning algorithms (i.e., they are guaranteed to converge to an invariant, if one exists, in a polynomial number of interactions with the verification engine, and the time they spend in each round is polynomial in the number of counterexamples and predicates). However, unlike HOUDINI, SORCAR is a *property-driven* algorithm, and strives to identify the predicates relevant to prove the assertions in the program, leading to smaller invariants and often faster convergence.

We emphasize that our primary contribution is the LSUC model for learning invariants using non-provability information and the reduction of such learning to the ICE learning model; the new SORCAR algorithm is secondary.

Applying Invariant Learning for Programs with Heap Specifications

The second major contribution of the paper is an application of our LSUC learning framework for incomplete verification engines to prove correctness of programs that dynamically manipulate the heap. To this end, we employ the *natural proof verification engine* [61], which is an incomplete verification engine for programs that are annotated with an expressive variant of separation logic (called DRYAD) that combines separation logic, arithmetic, and properties of the multisets of keys stored in inductively-defined data-structures. The verification problem (even given adequate invariants) is undecidable for this logic, and natural proofs reduce validity tasks (soundly but incompletely) to validity of formulas in decidable logics that can be solved using off-the-shelf SMT solvers. In this setting, we show precisely how natural proofs can be augmented to provide the non-provability information when validity of verification conditions fail. Combined with our reduction to ICE learning and the two ICE learners LSUC_HOUDINI and LSUC_SORCAR, this results in a robust framework for learning invariants for heap verification using natural proofs.

Our final contribution is a thorough evaluation of our learning framework for natural proofs using LSUC learners. We implement our technique on top of the natural proof engine VCDRYAD [61] (which extends VCC and uses Boogie/Z3) and pair it with the LSUC_HOUDINI and LSUC_SORCAR learners. We evaluate our implementation on a large class of heap manipulating programs and show that our technique (using either learner) is extremely effective in automatically synthesizing loop invariants and strengthening contracts to prove these programs correct. We emphasize that this class of programs is very challenging even with manually given invariants as the verification conditions are undecidable in general; in fact, the incomplete verification engine based on natural proofs is the only approach currently known to handle them. In this paper, we completely automate verification of this class of programs by automatically synthesizing inductive invariants as well.

In summary, our contributions are:

- A learning framework called Learning Sets Under Constraints (LSUC) that allows learners to synthesize conjunctive invariants over a fixed set of predicates when paired with incomplete verification engines that provide non-provability information encoded as set constraints.
- A reduction of the LSUC problem to the ICE learning problem for conjunctive Boolean formulas that allows us to build LSUC learners using existing ICE learners such as HOUDINI. Also, we propose a new ICE learning algorithm called SORCAR, that is property-driven.
- An application of the LSUC framework for synthesizing separation logic invariants for programs that manipulate dynamic heaps, where the verification engine is an incomplete engine based on natural proofs.
- An evaluation that shows that our framework and the LSUC learners we develop are effective in proving a large class of heap-manipulating programs automatically correct. The SORCAR algorithm on this class of examples outperforms HOUDINI, learning smaller invariants as well as learning them faster.

Related Work

One can broadly categorize algorithms for invariant generation into white-box and black-box techniques. Most popular white-box techniques include abstract interpretation [23], interpolation [46, 57], IC3 [11], as well as an array of techniques relying on constraint solving [21, 39, 40]. On the other hand, Daikon [29] and HOUDINI [31] are two popular black-box techniques for invariant generation, which both use learning of conjunctive Boolean formulas.

In fact, learning has recently regained interest in the area of verification, specifically as an effective means to generate loop invariants [16, 33–35, 50, 51, 69–72]. In this context, Garg et al. [34] proposed a general learning framework for invariants, called ICE learning, which extends the classical learning setting (i.e., learning from classified data points) with implications. Algorithms that operate in this framework have been proposed for learning invariants over octagonal domains [33, 36], universally quantified invariants over linear data structures [33, 35], invariants over inductive data types [74], Regular Model Checking [60], and a general framework for generating invariants based on randomized search [69]. In

[1] Named after P.C. Sorcar, who was a magician, similar to Houdini.

addition, the HOUDINI algorithm [31] can also be seen as an ICE learning algorithm, as we show in Section 4.1.

For the synthesis of invariants over dynamic data structures such as arrays and lists, several white-box techniques based-on Craig's interpolation [57] have emerged in recent years. Those techniques have the advantage of not requiring templates of predicates to be provided in advance. However, many of those techniques have a problem of potential divergence, requiring a special care for termination. Also, they are often tailored towards a specific class of properties, especially ones in theories admitting interpolation, such as range properties over arrays [3, 47, 68] or shape properties over linked-lists [58]. Thus, it is not clear how to extend them to richer properties over other types of heap data-structures. Recently, another technique based on spatial interpolation has been proposed [2], which can handle both shape and data properties written in a fragment of separation logic. Essentially, [2] infers each property separately– first it infers shape properties and then strengthens them with data properties. It is, however, not clear how to apply such a two-stage approach to more complicated heap structures (e.g., AVL trees) where shape, height, and data properties are not clearly separated.

There has been a lot of work on synthesizing invariants for separation logic using shape analysis [13, 14, 28, 53, 67]. In general, such techniques are more scalable than the ones based on logical reasoning such as interpolation or learning, and often guarantee termination. However, many of them are tailored for memory safety and shallow properties rather than richer properties that check full functional correctness. Also, they suffer from imprecision in the domain abstractions.

Abstract interpretation [23] has also been used for synthesizing invariants of array and heap manipulating programs [1, 24, 38]. Another example of such an approach is the framework by Bouajjani et al. [9, 10], in which the authors consider invariant synthesis for programs manipulating singly-linked lists. Recent work also includes synthesizing heap invariants by extending IC3 [48]. Finally, there also exist several learning-based black-box techniques, including methods based on predicate abstraction [50] and a reduction to a special type of register automata [33, 35].

Our work builds upon DRYAD[61, 65], a dialect of separation logic [66], and the natural proof technique line of work for heap verification developed by Qiu et al. [55, 61, 65], which build sound but incomplete techniques for verification for fairly expressive user-defined recursive predicates. In [61], the authors present VCDRYAD, which extends the VCC [19] framework to provide an automated deductive framework against separation logic specifications for C programs based on natural proofs. In contrast to the work we present here, VCDRYAD assumes that all loop invariants as well as strong pre- and postconditions are given. A technique, similar to DRYAD, for automated reasoning of dynamically manipulated data structures has recently been proposed by Chu et al. [18]. This method also relies on the unfold-and-match paradigm which employs systematic transformation steps of folding/unfolding the recursive rules. An earlier work [17] also uses similar ideas for folding and unfolding recursive rules.

There is a rich study of decidable fragments of separation logic in the context of deductive verification in the literature. Smallfoot [5, 6] implements a decision procedure for a fragment of separation logic for pure spatial properties of list segments, and there have been significant extensions to it [22], with some incorporating lists with arithmetic [59]. Encoding separation logic fragments in SMT to get decidability has been developed as well— by reducing separation logic over lists to a logic over graph reachability and stratified sets [62], and extending them to trees and data [63, 64]. The decision procedures for these logics, however, do not extend to arbitrary user-defined predicates (indeed, the problem is in general undecidable). More expressive fragments of separation logic have been shown to be decidable using monadic second-order logics on bounded tree-width graphs, but these are of high complexity [44]. Brotherston et al. [12] show a symbolic heap fragment of separation logic to be decidable. Some of the above work can be adapted to give concrete counterexamples using which we can build ICE learning algorithms. However, implementing black-box invariant learning using non-provability information for these approaches (even for complete procedures) using the framework proposed in this paper would be interesting future work.

2. Learning Inductive Invariants for Incomplete Verification Engines

In this section, we outline the problem of learning invariants from a sound-but-incomplete verification engine, the non-provability information that such verification engines can return, and how this information guides the invariants that are synthesized by the learner.

Let us fix a program P for the rest of this section that is annotated with assertions (and possibly with some partial annotations that describe pre-conditions/post-conditions). Let us also fix a sound-but-incomplete verification oracle SIVO.

The primary goal is to verify P against the assertions in it using the verification oracle. Let us assume that the program contains an annotation-hole A that needs to be filled in with some annotation (which must be an invariant at that point in the program) in order to verify the program. This annotation hole can be a loop-invariant, or a pre/post condition for a recursive function, for instance. We restrict the problem to synthesis of one annotation for simplicity; the framework we build smoothly extends to multiple annotations.

Our framework consists of pairing the verification oracle with a *learner* whose task is to synthesize an annotation for the hole A. In each round, the learner will come up with an annotation, and the verifier will attempt to verify the verification conditions that arise from plugging in this annotation. If it is unable to verify the program correct, then this could be either due to the fact that the annotation is incorrect (not an invariant), the annotation is inadequate to prove the assertions in the program (e.g., not inductive), or the annotations are logically adequate but the verification oracle is unable to prove them. We would like the verification oracle to return information that guides the learner towards learning an annotation that is adequate to prove the program with the verification oracle.

When the learner proposes an annotation for A, and the verification oracle fails to prove the verification condition for a Hoare triple $\{\alpha\}s\{\beta\}$, either α or β or both could involve the synthesized annotation. The verification oracle could simply just convey that the annotation for A is insufficient to prove the program correct—however, this gives little information on what aspects of the annotation were problematic. With this little information, the learner would gain no information, and will essentially only be able to *enumerate* annotations in some manner, and hence will be inefficient.

In the following, we specialize the above learning-based synthesis to *conjunctive annotations* where formulas for annotations are a subset of conjuncts drawn from a fixed class of formulas. As we will show, the verification oracle can return non-provability information when the space of annotations is restricted to conjunctions, and furthermore, we can build efficient learning algorithms for this space of annotations (see Sections 4 and 5). Throughout this paper, we use the notation $\bigwedge S$ for a set of predicates S as a shorthand for the conjunction $\bigwedge_{p\in S} p$.

We make a natural assumption for the verification oracle that it is *normal*, as defined below.

Definition 1. *A verification oracle is* normal *when for any finite set of predicates S, (a) the verification oracle can prove a Hoare triple* $\{\alpha\}s\{\bigwedge S\}$ *if and only if it can prove every Hoare triple* $\{\alpha\}s\{p\}$

where $p \in S$, and (b) if the verification oracle cannot prove a Hoare triple $\{\bigwedge S\}s\{\beta\}$ then it cannot prove $\{\bigwedge X\}s\{\beta\}$ for any set $X \subseteq S$.

The assumption that the verification oracle is essentially that it can do at least some minimal Boolean reasoning.

2.1 Synthesizing Conjunctive Invariants and Encoding Non-provability Using Set Constraints

Let us fix a finite set of predicates $\mathcal{P}$. Our goal will be to synthesize formulas in the logic $\mathcal{L}$ that consists of conjunctions of predicates drawn from $\mathcal{P}$ (i.e., $\mathcal{L} = \{\bigwedge_{p \in X} p \mid X \subseteq \mathcal{P}\}$). The formulas in $\mathcal{L}$ are those that are candidates for plugging into the annotation hole A.

We show in this section that non-provability information of verification conditions by the verification oracle can be encoded using a set of *constraints* on the set of predicates used for annotations. This results in a learning setting, called *learning sets under constraints (*LSUC*)*, in which the task is to learn a conjunct that satisfy such constraints. We formulate this precise learning model in Section 2.2. In Section 3, we reduce the LSUC problem to a more well-studied problem called *ICE learning* of conjunctions, and in Section 4 we present several ICE learning algorithms for conjunctions, which are hence solutions for LSUC.

What information can a sound-but-incomplete verification oracle provide? The overall verification goal is to find an annotation for the annotation-hole A that is a conjunct of a subset of predicates in $\mathcal{P}$ that leads the verification oracle to prove the program correct. For the following discussion, assume that the learner presents a candidate invariant $\gamma := \bigwedge S$ for some $S \subseteq \mathcal{P}$ to be used for A, but the sound-but-incomplete verifier SIVO is unable to verify the program using γ. Let us now consider each kind of verification condition and examine what useful non-provability information SIVO can provide the learner to help its future attempts to come up with an invariant. In the explanation below, α and β are annotations, s is a statement of P, and $\nvdash_{SIVO} \{\alpha\}s\{\beta\}$ denotes that the validity of the Hoare triple $\{\alpha\}s\{\beta\}$ is not provable by SIVO.

- If $\nvdash_{SIVO} \{\alpha\}s\{\gamma\}$ (i.e., we cannot prove a VC where the pre-condition is an existing annotation and the post-condition is the synthesized annotation γ), then it means that there is some predicate in γ that we cannot prove to hold after executing s from a state satisfying α.

  Since γ is the set of conjuncts of a set S, if the prover is unable to prove γ, then there exists some subset of $T_+ \subseteq S$ that the prover is not able to prove. Furthermore, clearly the prover cannot prove any set of conjuncts that includes some element of T_+ (as SIVO is normal).

  Hence, we can send the set T_+ to the learner and insist that in future rounds, the learner has to come up with a set X such that $X \cap T_+ = \emptyset$. We call T_+ an *avoid set* sample.

- If $\nvdash_{SIVO} \{\gamma\}s\{\beta\}$ (i.e., we cannot prove a VC where the pre-condition is the synthesized annotation γ and the postcondition is an existing annotation), then we know that the conjunction of predicates mentioned in γ is not strong enough for the verifier to prove that β holds after executing s. Now, let T_- be a *superset* of predicates in γ such that the verifier is unable to prove β holds after executing S with the conjunction of predicates in T_- as a precondition. (We will show later how the verifier can come up with T_- in some concrete verification scenarios.)

  We communicate T_- to the learner and insist that in future rounds, the learner come up with a set X such that $X \not\subseteq T_-$. This is correct since for any X with $X \subseteq T_-$, the verifier will not be able to prove the verification condition with $\bigwedge X$ as precondition since it is a weakening of $\bigwedge T_-$ (recall that SIVO is natural). We call T_- a *differ set* sample.

- If $\nvdash_{SIVO} \{\gamma\}s\{\gamma\}$ (i.e., we cannot prove the VC where both the pre- and post-condition is the synthesized annotation γ), then we cannot conclude that the annotation must be stronger or weaker than γ.

  We now assume that the verification oracle can come up with two sets of predicates U_1 and U_2 such that U_1 is a superset of predicates in γ and U_2 is a subset of predicates in γ such that the oracle cannot prove the verification condition $\{\bigwedge U_1\}s\{\bigwedge U_2\}$.

  Then we can communicate the pair (U_1, U_2) to the learner and demand that the learner come up with a set X such that *if* $X \subseteq U_1$ then $U_2 \cap X = \emptyset$. Intuitively, this condition is correct since if $X \subseteq U_1$ and $U_2 \cap X \neq \emptyset$, then the validity of the verification condition $\{\bigwedge X\}s\{\bigwedge X\}$ would imply the validity of $\{\gamma\}s\{\gamma\}$, which the verifier could not prove. We call the pair (U_1, U_2) a *double constraint* sample. (Again, we will show later how such a double constraint sample can be found in the context of certain verification oracles such as natural proofs for heap verification.)

2.2 The Models of Learning Sets Under Constraints (LSUC)

The above analysis clearly shows that when the sound-but-incomplete verifier cannot prove the verification conditions resulting from plugging in a candidate annotation γ, it can return an avoid set T_+, a differ set T_-, or a double constraint (U_1, U_2).[2] We can accumulate such constraints over successive rounds in a data structure called a *sample set*. A learning algorithm (annotation synthesizer) can then use this information to propose a new candidate annotation. The process of proposing candidate annotations and checking them repeats until the learning algorithm proposes a valid annotation or the learning algorithm detects that no valid annotation exists (recall that the set of predicates is assumed to be finite). This idea leads to two learning models: (a) a *passive* learning model that describes the learning in one iteration of the process described above, and (b) an *iterative* learning model that describes the learning as a whole.

For the sake of simplicity, let us assume that the learning algorithm accumulates the constraints returned by the verification oracle in a data structure that we call sample. Formally, a *sample* is a triple $\mathfrak{S} = (AS, DS, DC)$ where $AS \subseteq 2^{\mathcal{P}}$ is a set of avoid sets, $DS \subseteq 2^{\mathcal{P}}$ is a set of differ sets, and $DC \subseteq 2^{\mathcal{P}} \times 2^{\mathcal{P}}$ is a set of double constraints. We say that X is consistent with a sample $\mathfrak{S} = (AS, DS, DC)$ if

- $X \cap T_+ = \emptyset$ for every $T_+ \in AS$;
- $X \not\subseteq T_-$ for every $T_- \in DS$; and
- if $X \subseteq U_1$, then $U_2 \cap X = \emptyset$ for every $(U_1, U_2) \in DC$.

Passively learning sets under constraints: The *passive learning of sets under constraints (passive* LSUC*)* is a learning model where the learner needs to find, given a sample $\mathfrak{S} = (AS, DS, DC)$, some set X consistent with it.

Iteratively learning sets under constraints: In the *iterative learning of sets under constraints (iterative* LSUC*)*, the learner interacts with an oracle in order to learn *some* set in a target set $\mathcal{T} \subseteq 2^{\mathcal{P}}$. In each round, the learner has a sample $\mathfrak{S} = (AS, DS, DC)$, and learns a set X consistent with the sample. If X is in the target set $\mathcal{T}$, then the learner has learned the concept, and we stop. Otherwise, the oracle *adds* a new avoid set, a new differ set, or a new double constraint to the sample (consistent with targets in $\mathcal{T}$), and the process iterates.

Note that the iterative learning model is the one of primary interest. However, *any* passive learning model can be used in an iterative setting. Our goal is to build learners that (a) in each round, learn the set X efficiently, and (b) minimize the number of rounds to

[2] Existence of T_+, T_-, and (U_1, U_2) is obvious. One can simply return the precise γ that the learner gave. We will show later how the verifier can come up with non-trivial constraints.

learn a target concept. Any such iterative learner can be plugged in into our framework to obtain an *automatic* verification engine that synthesizes invariants that prove the program using the verification oracle.

We can in fact show a *relative completeness* result:

Theorem 1. *Let P be a program with assertions and an annotation-hole and* SIVO *be a* normal *sound-but-incomplete verification oracle that can compute set constraints (as in the LSUC model) whenever an annotation fails to prove the program correct. Let $\mathcal{P}$ be a finite set of predicates, and assume that there exists some $S \subseteq \mathcal{P}$ such that the program verifies with the annotation $\bigwedge S$ under* SIVO. *Then, when the verification oracle is paired with any consistent* LSUC *learner (learner of sets under constraints), the learner will eventually learn an annotation that proves P correct using* SIVO. □

The proof follows from the observation that the set S will always be a set that is consistent with the sample in every round of interaction and the fact that a consistent learner cannot produce the same set twice (since the sample given by the oracle for any hypothesized set R is guaranteed to refute R), and the fact that there are only finitely many subsets of $\mathcal{P}$.

3. Reducing Learning of Sets under Constraints to ICE Learning

In this section, we show how the LSUC problem can be reduced to learning conjunctive formulas in the so-called *ICE learning framework* [34]. In fact, it is enough to reduce the passive LSUC problem to passive ICE-learning as this immediately gives us the reduction of learning sets in the iterative setting to iterative ICE-learning, including the properties of time-complexity and round-complexity (this will be discussed in Section 4).

We first briefly review the ICE learning framework in Section 3.1. In Section 3.2, we describe the reduction of the passive learning of sets under constraints to learning conjunctive formulas in the ICE learning framework.

Before we begin, however, let us fix some notation. First, let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a finite set of predicates; we assume that $\mathcal{P}$ is equipped with an ordering such that the i-th element is denoted by p_i. Next, let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a set of Boolean variables, again equipped with an ordering such that b_i corresponds to p_i, and $\mathcal{BV}_n$ the set of all Boolean vectors of length n. Given a conjunctive Boolean formula $\varphi = b_{i_1} \wedge \ldots \wedge b_{i_\ell}$, $i_j \in \{1, \ldots, n\}$ for $j \in \{1, \ldots, \ell\}$, we say that φ *satisfies* a Boolean vector $\vec{v} = (v_1, \ldots, v_n) \in \mathcal{BV}_n$, denoted by $\vec{v} \models \varphi$, if and only if $v_{i_j} = 1$ for each $j \in \{1, \ldots, \ell\}$. Similarly, given a set $X \subseteq \mathcal{B}$, we say that X *satisfies* $\vec{v} \in \mathcal{BV}_n$, denoted by $\vec{v} \models X$, if $\vec{v} \models \varphi_X$ where $\varphi_X = \bigwedge_{b_i \in X} b_i$ (i.e., we can use conjunctions and sets of variables interchangeably).

3.1 The Passive ICE Learning Framework

The ICE learning framework [34] is a general framework for learning inductive invariants (as in this paper, the ICE learning framework describes a passive and an iterative setting). However, we are here interested in learning conjunctive formulas over Boolean variables and, hence, simplify the description of the framework slightly to better fit our setting.

Roughly speaking, the objective of a learning algorithm in the passive ICE learning framework is to learn a conjunctive formula over $\mathcal{B}$ from positive examples, negative examples, and implications. These examples are given as a so-called ICE sample: an *ICE sample* is a triple $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$ where (a) $S_+ \subseteq \mathcal{BV}_n$ is a set of *positive examples*, (b) $S_- \subseteq \mathcal{BV}_n$ is a set of *negative examples*, and (c) $S_\Rightarrow \subseteq \mathcal{BV}_n \times \mathcal{BV}_n$ is a set of *implications*. We call a conjunctive formula φ over $\mathcal{B}$ (analogously a set $X \subseteq \mathcal{B}$ of Boolean variables) *consistent with* $\mathcal{S}$ if it satisfies the following three conditions:

1. $\vec{v} \models \varphi$ for all $\vec{v} \in S_+$;
2. $\vec{v} \not\models \varphi$ for all $\vec{v} \in S_-$; and
3. $\vec{v}_1 \models \varphi$ implies $\vec{v}_2 \models \varphi$ for all $(\vec{v}_1, \vec{v}_2) \in S_\Rightarrow$.

Having defined this, we can now define the *passive* ICE learning problem formally:

> *Given an ICE sample $\mathcal{S}$, construct a conjunctive Boolean formula φ that is consistent with $\mathcal{S}$.*

The remainder of this section shows how passive ICE learning algorithms can be used to learn sets under constraints, which, in turn, model counterexamples of non-provability.

3.2 Reducing passive LSUC to passive ICE Learning of Conjuncts

Recall the passive LSUC problem from Section 2.2, and the notion of when a set $X \subseteq \mathcal{P}$ is consistent with a sample $\mathfrak{S} = (AS, DS, DC)$.

The key idea of our reduction is to translate the sample $\mathfrak{S}$ into an ICE sample $\mathcal{S}$ such that a Boolean formula that is consistent with $\mathcal{S}$ can be translated into a set $X \subseteq \mathcal{P}$ that is consistent with $\mathfrak{S}$. This translation relies on two bijective mappings τ and κ. Intuitively, τ translates between subsets of $\mathcal{P} = \{p_1, \ldots, p_n\}$ and Boolean vectors of length n, whereas κ translates between subsets of $\mathcal{P}$ and conjunctive formulas over the Boolean variables $\mathcal{B} = \{b_1, \ldots, b_n\}$. Formally, we define τ and κ as follows:

- The mapping τ maps a set $Y \subseteq \mathcal{P}$ of predicates to the Boolean vector $\vec{v} = (v_1, \ldots, v_n) \in \mathcal{BV}_n$ with

$$v_i = 1 \text{ if and only if } p_i \notin Y.$$

- The mapping κ maps a set $Y \subseteq \mathcal{P}$ of predicates to the conjunctive formula $\varphi = \bigwedge_{p_i \in Y} b_i$.

Note that both τ and κ are in fact bijective mappings and, hence, unique inverse mappings τ^{-1} and κ^{-1} exist.

Our reduction now proceeds in three steps:

1. Given the sample $\mathfrak{S} = (AS, DS, DC)$, we translate $\mathfrak{S}$ into the ICE sample $\tau(\mathfrak{S}) = (S_+, S_-, S_\Rightarrow)$ where
   - $S_+ = \{\tau(T_+) \mid T_+ \in AS\}$;
   - $S_- = \{\tau(\mathcal{P} \setminus T_-) \mid T_- \in DS\}$; and
   - $S_\Rightarrow = \{(\tau(\mathcal{P} \setminus U_1), \tau(U_2)) \mid (U_1, U_2) \in DC\}$.
2. We learn a conjunctive Boolean formula φ that is consistent with $\tau(\mathfrak{S})$.
3. We return the set $\kappa^{-1}(\varphi)$.

It is left to show that this reduction is indeed correct. In other words, if φ is a solution of the ICE learning problem (i.e., φ is a conjunctive Boolean formula that is consistent with the ICE sample $\tau(\mathfrak{S})$), then $\kappa^{-1}(\varphi)$ is a set consistent with $\mathfrak{S}$.

Theorem 2. *Let $\mathfrak{S} = (AS, DS, DC)$ be a sample and $X \subseteq \mathcal{P}$ a set of predicates. Then, X is consistent with the sample $\mathfrak{S}$ if and only if $\kappa(X)$ is consistent with the ICE sample $\tau(\mathfrak{S})$.*

The remaining of this section is devoted to prove the above theorem. In preparation of the proof of Theorem 2, let us first state a simple property of τ and κ, which we exploit later.

Lemma 1. *Let $X, Y \subseteq \mathcal{P}$ be two sets of predicates. Then, $X \cap Y = \emptyset$ if and only if $\tau(X) \models \kappa(Y)$.*

Proof. To proof Lemma 1, we require the notation of what it means that a Boolean variable occurs in a conjunction. To this end, let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a set of Boolean variables and $\varphi = b_{i_1} \wedge \ldots \wedge b_{i_\ell}$

with $1 \leq \ell \leq n$ and $i_j \in \{1,\ldots,n\}$ for $j \in \{1,\ldots,\ell\}$ a conjunction over $\mathcal{B}$. We then write

$$b \in \varphi \text{ if and only if } b \in \{b_{i_1},\ldots,b_{i_\ell}\}$$

(read "b_i occurs in φ").

We can now prove Lemma 1. Let $\mathcal{P} = \{p_1,\ldots,p_n\}$ be a finite set of predicates and $X,Y \subseteq \mathcal{P}$ two sets. Moreover, let $\kappa(X) = (v_1,\ldots,v_n)$ and $\tau(Y) = \varphi$.

From left to right: Let $X \cap Y = \emptyset$. By definition of τ, $v_i = 1$ if and only if $p_i \notin X$ and, hence, $v_i = 1$ for all $p_i \in \mathcal{P} \setminus X$. By definition of κ, on the other hand, $b_i \in \varphi$ if and only if $p_i \in Y$. Moreover, since $X \cap Y = \emptyset$, we have $Y \subseteq \mathcal{P} \setminus X$. Hence, $v_i = 1$ if $b_i \in \varphi$ and, thus, $(v_1,\ldots,v_n) \models \varphi$ (i.e., $\kappa(X) \models \tau(Y)$).

From right to left: Let $\tau(X) \models \kappa(Y)$ (i.e., $(v_1,\ldots,v_n) \models \varphi$). By definition of $\models$, this means that $b_i \in \varphi$ implies $v_i = 1$. Moreover, $b_i \in \varphi$ implies $p_i \notin X$ by definition of κ and, hence, $p_i \in Y$ implies $p_i \notin X$ by definition of τ. Hence, $X \cap Y = \emptyset$. □

To prove Theorem 2, let us briefly recall the definition of consistency of a set with a sample: a set $X \subseteq \mathcal{P}$, where $\mathcal{P}$ is a finite set of predicates, is said to be consistent with a sample $\mathfrak{S} = (AS, DS, DC)$ if

a) $X \cap T_+ = \emptyset$ for every $T_+ \in AS$;

b) $X \not\subseteq T_-$ for every $T_- \in DS$; and

c) if $X \subseteq U_1$, then $X \cap U_2 = \emptyset$ for every $(U_1,U_2) \in DC$.

Let us now bring this definition into an equivalent, yet more suitable form: a set $X \subseteq \mathcal{P}$ is consistent with a sample $\mathfrak{S} = (AS, DS, DC)$ if

a′) $X \cap T_+ = \emptyset$ for every $T_+ \in AS$;

b′) $X \cap (\mathcal{P} \setminus T_-) \neq \emptyset$ for every $T_- \in DS$; and

c′) if $X \cap (\mathcal{P} \setminus U_1) = \emptyset$, then $X \cap U_2 = \emptyset$ for every $(U_1,U_2) \in DC$.

We can now prove Theorem 2.

Proof. We show the direction from left to right. The reverse direction is analogous and, therefore, skipped.

Let $X \subseteq \mathcal{P}$ and assume that X is consistent with $\mathfrak{S} = (AS, DS, DC)$. We now show that $\kappa(X)$ is consistent with $\tau(\mathfrak{S}) = (S_+, S_-, S_\Rightarrow)$ by showing

- $\vec{v} \models \kappa(X)$ for all $\vec{v} \in S_+$;
- $\vec{v} \not\models \kappa(X)$ for all $\vec{v} \in S_-$; and
- if $\vec{v}_1 \models \kappa(X)$, then $\vec{v}_2 \models \kappa(X)$ for all $(\vec{v}_1,\vec{v}_2) \in S_\Rightarrow$.

Let us investigate positive examples, negative examples, and implications individually.

Positive examples: Pick some element $\vec{v} \in S_+$, say $\vec{v} = \tau(T_+)$ where $T_+ \in AS$ (recall the definition of $\tau(\mathfrak{S})$). Since X is consistent with $\mathfrak{S}$, we know that $X \cap T_+ = \emptyset$. By Lemma 1, we immediately obtain $\vec{v} = \tau(T_+) \models \kappa(X)$.

Negative examples: Pick some element $\vec{v} \in S_-$, say $\vec{v} = \tau(\mathcal{P} \setminus T_-)$ where $T_- \in DS$. Since X is consistent with $\mathfrak{S}$, we know that $X \cap (\mathcal{P} \setminus T_-) \neq \emptyset$. By Lemma 1, we immediately obtain $\vec{v} = \tau(\mathcal{P} \setminus T_-) \not\models \kappa(X)$.

Implications: Pick some implication $(\vec{v}_1,\vec{v}_2) \in S_\Rightarrow$, say $\vec{v}_1 = \tau(\mathcal{P} \setminus U_1)$ and $\vec{v}_2 = \tau(U_2)$ where $(U_1,U_2) \in DC$. Since X is consistent with $\mathfrak{S}$, we know that $X \cap (\mathcal{P} \setminus U_1) = \emptyset$ implies $X \cap U_2 = \emptyset$. By Lemma 1, we immediately obtain that if $\vec{v}_1 = \tau(\mathcal{P} \setminus U_1) \models \kappa(X)$, then $\vec{v}_2 = \tau(U_2) \models \kappa(X)$.

Since $\kappa(X)$ satisfies all consistency constraints, $\kappa(X)$ is consistent with $\tau(\mathfrak{S})$. □

4. ICE Learners for Conjunctive Boolean Formulas

In this section, we first recap an existing ICE learner for conjunctive Boolean formulas called HOUDINI [31], then develop *a novel property-driven ICE learning algorithm* called SORCAR, and compare and contrast them. Both algorithms, in combination with our reduction, become LSUC learners that can synthesize conjunctive invariants for incomplete verification engines.

HOUDINI is typically seen as a particular way to synthesize invariants, but is best understood as an ICE learner for conjuncts, as described in the work by Garg et al. [34]. In fact, Houdini is similar to the classical PAC learning algorithm for conjunctions [49], but extended to implication counterexamples as well. In every round, the HOUDINI algorithm learns the *largest* set of conjuncts that satisfies the current positive and implication samples (thus, it can never misclassify a negative sample if a consistent conjunction exists), and hence learns, in the iterative setting, the invariant that can be expressed using the largest set of conjuncts. The time HOUDINI spends in each round is *polynomial* and, furthermore, when used in an iterative setting, is guaranteed to converge in at most n rounds (where $n = |\mathcal{P}|$) or report that no conjunctive invariant over $\mathcal{P}$ exists.

The property of the HOUDINI algorithm that it converges in at most $n = \mathcal{P}$ rounds in the iterative setting is very important in practice. We can, for instance, in every round learn the *smallest* set of conjuncts satisfying the sample, say using a SAT solver. Doing so would not significantly increase the time taken for learning in each round, but the worst-case number of iterations to converge to an invariant becomes exponential. An exponential number of rounds, however, hurts in practice (we implemented such a constraint-solver based learner, but it performed poorly on our set of benchmarks). Hence, it is important to keep the number of iterations small when learning invariants.

One disadvantage of HOUDINI is that it learns in each round the largest set of conjuncts, *independent* of negative samples, and hence independent of the assertions/specifications in a program— it learns the tightest inductive invariant expressible as a set of conjuncts over $\mathcal{P}$.

This motivates the development of the new SORCAR ICE learning algorithm for conjuncts, which has a bias towards learning smaller invariants. SORCAR always learns invariants involving what we call *relevant* predicates, which are predicates that have shown some evidence to effect the assertions in the program. However, the SORCAR algorithm is more complex and subtle than HOUDINI, and its correctness proof is more involved. Intuitively, a negative sample stems from a failed verification condition involving an assertion in a program. We say that a predicate is *relevant* if it is marked *false* in some negative example. When a predicate occurs *false* in a negative sample, it signifies that *not* assuming the predicate leads to an assertion violation, and hence deemed important as a candidate predicate in the synthesized formula. The algorithm is more involved, however, as implication samples also can contribute to relevancy of predicates, and further, naively growing the relevant predicates would lead to an exponential number of rounds in general. The SORCAR algorithm, on the other hand, requires at most $2n = 2\mathcal{P}$ iterations, despite simultaneously learning formulas only over relevant variables.

Table 1 summarizes the key properties of HOUDINI and SORCAR. Despite requiring twice the number of iteration as HOUDINI, the SORCAR algorithm performs very well in practice and significantly outperforms HOUDINI on our benchmarks in overall time to learn invariants; it also produces *smaller* invariants (but nowhere close to the minimum), which is a pleasant side-effect of concentrating on relevant predicates.

Table 1: Comparison of HOUDINI and SORCAR

Learning Algorithm	Property driven?	Complexity per round	Maximum # Rounds	Final # of conjuncts
HOUDINI	No	Polynomial	n	Largest set
SORCAR	Yes	Polynomial	$2n$	Bias towards smaller sets involving only relevant predicates

We now present briefly the HOUDINI algorithm (as it will be used by the SORCAR algorithm as well), and then present our new SORCAR ICE-learning algorithm.

4.1 HOUDINI as an ICE learner

HOUDINI [31] is a learning algorithm for conjunctive formulas over an a priori given set $\mathcal{B} = \{b_1, \ldots, b_n\}$ of Boolean variables. Given an ICE sample $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$, HOUDINI computes the largest conjunctive formula φ in terms of the number of Boolean variables occurring in φ (i.e., the semantically strongest conjunctive formula) that is consistent with $\mathcal{S}$ in the following way. First, it computes the largest conjunction φ that is consistent with the positive examples (i.e., $\vec{v} \models \varphi$ for all $\vec{v} \in S_+$); note that this conjunction is unique. Next, HOUDINI checks whether the implications are satisfied. If this is not the case, then we know for each non-satisfied implication $(\vec{v}_1, \vec{v}_2) \in S_\Rightarrow$ that $\vec{v}_2$ has to be classified positively because $\vec{v}_1$ belongs to every set that includes S_+. Hence, HOUDINI adds all such $\vec{v}_2$ to S_+, resulting in a new set S'_+. Subsequently, it constructs the largest conjunction φ' that is consistent with the positive examples in S'_+ (i.e., $\vec{v} \models \varphi'$ for all $\vec{v} \in S'_+$). HOUDINI repeats this procedure until it arrives at the largest conjunctive formula φ^* that is consistent with S_+ and $S_\Rightarrow$ (again, note that this set is unique). Finally, HOUDINI checks whether each negative example violates φ^* (i.e., $\vec{v} \not\models \varphi^*$ for all $\vec{v} \in S_-$). If this is the case, φ^* is the largest conjunctive formula over $\mathcal{B}$ that is consistent with $\mathcal{S}$; otherwise, no consistent conjunctive formula exists.

4.2 The SORCAR ICE-learning algorithm

In this section we present SORCAR, a new ICE algorithm for learning conjunctive Boolean formulas. The salient aspect of SORCAR is that it uses the negative examples in the ICE sample to learn relevant variables, and biases learning towards conjuncts involving relevant variables only. At the same time, it learns concepts in the iterative setting using only a linear number of rounds, similar to HOUDINI (though it requires at most $2n$ rounds in the worst-case).

The SORCAR ICE learner is presented as Algorithm 1. In contrast to HOUDINI, it is not a purely passive learning algorithm but maintains a state in the course of the iterative learning in form of a set $R \subseteq \mathcal{B}$. This set is used to accumulate *relevant variables*, which, intuitively, are Boolean variables in $\mathcal{B}$ that are consistent with all positive examples in the samples S_+ and have shown some relevancy in the sense that they can be used to establish consistency with negative examples and implications. In the beginning of the iterative learning, when the ICE sample $\mathcal{S}$ is empty, R is also empty.

Given a sample $\mathcal{S}$ and the set R, the passive SORCAR algorithm first constructs the largest set $X \subseteq \mathcal{B}$ of variables that is consistent with $\mathcal{S}$ (Line 4). This construction follows the HOUDINI algorithm described in Section 4.1 and ensures that X is consistent with all examples S. Since X is the largest set of predicates consistent with S, it follows that $X \cap R$ (in fact, any subset of X) is consistent with S_+. However, $X \cap R$ might not be consistent with S_- and $S_\Rightarrow$. If $X \cap R$ is not consistent with a negative sample $\vec{v} \in S_-$, then SORCAR adds all variables b_i with $\vec{v} \not\models b_i$ to R (see Line 7 and Line 2), effectively resulting in $X \cap R \not\models \vec{v}$. If $X \cap R$ is not consistent with an implication $(\vec{v}_1, \vec{v}_2) \in S_\Rightarrow$, then $X \cap R \models \vec{v}_1$ and $X \cap R \not\models \vec{v}_2$, which in turn means

```
1  Function relevant(v⃗):
2      return {b_i | v⃗ ⊭ b_i, b_i ∈ B} ;
3  Function SORCAR-PASSIVE(S = (S_+, S_-, S_⇒), R):
4      X ← {b_{i_1}, ..., b_{i_m}} s.t. ⋀_{i=1}^{m} b_{i_j} is the largest conjunctive formula
       consistent with S (abort if such an X does not exist);
5      while X ∩ R is not consistent with S do
6          if X ∩ R is not consistent with v⃗ ∈ S_- then
7              R ← R ∪ relevant(v⃗) ;
8          end
9          if X ∩ R is not consistent with (v⃗_1, v⃗_2) ∈ S_⇒ then
10             R ← R ∪ relevant(v⃗_1) ;
11         end
12     end
13     return (X ∩ R, R) ;
14 Function SORCAR-ITERATIVE(S):
15     static var R ⊆ B, initialized to ∅. ;
16     (H, R) ← SORCAR-PASSIVE(S, R) ;
17     return H ;
```

Algorithm 1: The SORCAR ICE learning algorithm

that $X \not\models \vec{v}_2$ as well (since $X \cap R \subseteq X$). Consequently, we would like to *expand* R so that then $X \cap R \not\models \vec{v}_1$. Thus, SORCAR-PASSIVE adds all variables b_i with $\vec{v}_1 \not\models b_i$ to R, hence, establishing $X \cap R \not\models \vec{v}_1$. The process continues (R grows monotonically larger) till a consistent conjunct is found, and SORCAR-PASSIVE returns both the satisfying conjunct $X \cap R$ as well as the new set of relevant variables R.

The condition of the loop in Line 5 immediately shows that the set $X \cap R$ is consistent with the input sample $\mathcal{S}$ once the passive SORCAR algorithm terminates. However, the termination argument is not obvious. To argue termination, we first observe that X is consistent with each positive sample in S_+ and, hence, $X \cap R$ remains consistent with all positive examples during the run of the passive SORCAR algorithm. Next, we observe that $X \cap R$ can be inconsistent with each negative example and implication at most once since adding a variable to R makes $X \cap R$ consistent with the current negative example or implication. Hence, the passive SORCAR algorithm terminates with a consistent set $X \cap R$ after at most $|\mathcal{S}|$ iterations (in the worst-case with $R = X$, which is consistent with $\mathcal{S}$). It also takes worst-case $|\mathcal{B}|$ iterations since $R \subseteq \mathcal{B}$.

Theorem 3. *Given an ICE sample $\mathcal{S}$ and a set of relevant variables $R \subseteq \mathcal{B}$, the passive* SORCAR *algorithm learns a consistent set of variables (i.e., a consistent conjunction over $\mathcal{B}$) in time polynomial in $|\mathcal{B}|$ and $|\mathcal{S}|$.*

Turning to the iterative SORCAR algorithm, let us assume that the samples in each round are generated so that there is at least some conjunct consistent with all the samples. Then we can show that the iterative SORCAR algorithm learns some conjunct consistent with the samples in at most $2|\mathcal{B}|$ rounds, as stated in the following theorem.

Theorem 4. *Given a set of Boolean variables $\mathcal{B}$, the* SORCAR *algorithm, paired with any ICE teacher that feeds it samples consistent with all conjuncts in $\mathcal{T}$ (where $\mathcal{T}$ is any nonempty set of conjunctive formulas), terminates and learns a conjunctive formula in $\mathcal{T}$ in at most $2|\mathcal{B}|$ iterations.*

Proof of Theorem 4. First, note that the computation of X in Line 4 can never fail since there is at least one conjunctive formula consistent with all samples, since $\mathcal{T}$ is nonempty. To prove Theorem 4, we show that either the size of X (computed in Line 4 in successive calls to the passive learner) decreases by at least one or the size of R increases by at least one, in each iteration. Since both X and R are

subsets of $\mathcal{B}$, this proves that the iterative algorithm terminates in at most $2|\mathcal{B}|$ rounds.

We prove the claim above by carefully examining the updates to X and R as counterexamples are added to the ICE sample $\mathcal{S}$:

- If a positive counterexample $\vec{v}$ is added to $\mathcal{S}$, then because $\vec{v} \not\models X \cap R$, therefore, $\vec{v} \not\models X$. This implies that there exists a variable $b_i \in X$ with $\vec{v} \not\models b_i$. In the subsequent round of the passive SORCAR algorithm, b_i will no longer be present in X and as a result $|X|$ decreases by at least one.
- If a negative counterexample $\vec{v}$ is added to $\mathcal{S}$, the set X remains unchanged in the next iteration, but R will get updated to account for this new negative sample (Line 7) and increases by at least one.
- If an implication counterexample $(\vec{v}_1, \vec{v}_2)$ is added to $\mathcal{S}$, then $\vec{v}_1 \models X \cap R$ but $\vec{v}_2 \not\models X \cap R$. Consequently, $\vec{v}_2 \not\models X$. Hence, SORCAR adds new variables b_i with $\vec{v}_1 \not\models b_i$ to R, hence increasing $|R|$ by at least one.

Consequently, after $2|\mathcal{B}|$ rounds, the iterative learner must return some conjunctive formula in $\mathcal{T}$, hence stopping the teacher from adding new samples. □

5. Learning Invariants that Aid Natural Proofs for Heap Reasoning

We now develop an instantiation of our learning framework for synthesizing invariants for verification engines based on natural proofs for heap reasoning, the latter as developed in the work in [61, 65].

We first provide some background on the separation logic DRYAD and natural proofs for DRYAD, which is a sound but incomplete verification procedure. Then, we present our verification framework, which pairs a verification oracle based on natural proofs with black-box learners that learn from non-provability information encoded using constraints on sets. The main technical contributions here are (a) to show how the natural proof method, which is based on abstractions to decidable theories, can be augmented to yield non-provability information in terms of set constraints (Section 5.2), and (b) the instantiation of the framework (Section 5.3), including the choice of an appropriately large class of predicates encoding separation, shape, and data information appropriate for heap verification.

5.1 Background: Natural Proofs and DRYAD

DRYAD [61, 65] is a dialect of separation logic that comes with a heaplet semantics and allows expressing second order properties such as pointing to a list segment or a list using recursive functions and predicates. The syntax of DRYAD is a standard separation logic syntax with a few restrictions, such as disallowing negations inside recursive definitions and in sub-formulas connected by spatial conjunction (please refer to [61] for more details about the DRYAD syntax). DRYAD is expressive enough to state a variety of data-structures (singly and doubly linked lists, sorted lists, binary search trees, AVL trees, maxheaps, treaps, etc.), recursive definitions over them that map to numbers (length, height, etc.), as well as data stored within the heaps (the multiset of keys stored in lists, trees, and so on). The main difference between DRYAD and classical separation logic lies in its semantics. Unlike classical separation logic, the heap domain on which any DRYAD formula holds true is syntactically determined. With this semantics, one can construct the heap domain for any DRYAD formula through one bottom-up traversal of the syntactic parse tree of the formula (see [65] for details).

Natural proofs [61, 65] is a sound but incomplete strategy for deciding satisfiability of DRYAD formulas. The first step the natural proof verifier performs is to convert all predicates and functions in a DRYAD-annotated program to *classical logic*. This translation introduces *heaplets* (modeled as sets of locations) explicitly in the logic, and furthermore, introduces assertions that demand that the accesses of each method are contained in the heaplet implicitly defined by its precondition (taking into account newly allocated or freed nodes), and that at the end of the program, the modified heaplet precisely matches the implicit heaplet defined by the post-condition.

The second step the natural proof verifier does is to perform *transformations* on the program and translate it to BOOGIE [31], an intermediate verification language that handles proof obligations using automatic theorem provers (typically SMT solvers). VCDRYAD extends VCC [19] to perform several natural proof transformations that essentially perform three tasks: (a) abstract all recursive definitions on the heap using uninterpreted functions but introduce finite-depth unfoldings of recursive definitions at every place in the code where locations are dereferenced, (b) model heaplets and other sets using a decidable theory of maps, (c) insert *frame reasoning* explicitly in the code that allows the verifier to derive that certain properties continue to hold across a heap update (or function call) using the heaplet that is modified.

The resulting program is a BOOGIE program with no recursive definitions and where all verification conditions are in decidable logics (caveat: there is some quantification introduced byVCC itself to define the memory model and semantics of C, but this does not typically derail decidable reasoning). Consequently, the program can be verified if supplied with correct inductive loop-invariants and adequate pre/post conditions.

5.2 Extracting Non-provability Information from Natural Proof Verifiers

While natural proofs is a sound (but incomplete) approach to verifying programs against heap properties, it can also provide non-provability information. Natural proofs transform verification conditions involving undecidable logics that include recursion (amongst others) to decidable logics such that the latter's validity implies the validity of the former. This aspect of natural proofs is the crucial property that we use to generate counterexamples. While failure to find a natural proof cannot be used to produce true counterexamples, we show how a model showing the failure to find a natural proof can be used to provide *constraints on the set of conjuncts in the invariant* as described in Section 2.

First, let us fix a set of predicates $\mathcal{P}$ in classical logic such that the space of invariants we want to learn from are conjuncts of formulas from $\mathcal{P}$. The set of predicates typically depends on the data-structures and recursive definitions used in the program and its annotations, and we will describe how we choose this later.

Given a Hoare triple $\{\alpha\}s\{\beta\}$, the natural proof technique fails to prove the validity of the associated verification condition when the associated *abstraction* of the verification condition expressed in a decidable logic fails to be proved valid. In this case, the SMT solver would find that the negation of the abstracted verification condition is satisfiable.

To extract non-provability information, we first augment the program snippets s so that we record, using a set of ghost variables $b_1 \ldots b_n$, whether the predicates $p_1, \ldots, p_n$ are true, before and after the execution of s. Consequently, when the SMT solver finds the negation of the verification condition to be satisfiable, it would also find a valuation of the variables $b_1 \ldots b_n$ before and after the snippet s. We can now use this information to find the constraints on sets.

More precisely, we find the LSUC sample from such a counterexample model as follows. Let $v_1, \ldots, v_n$ denote the valuation of $b_1, \ldots, b_n$ before executing s, and let $v'_1, \ldots, v'_n$ denote the valuation of $b_1, \ldots, b_n$ after executing s.

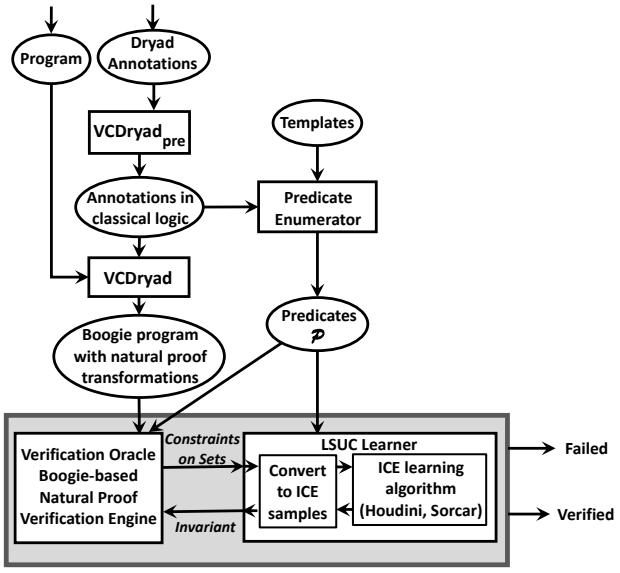

Figure 1: Framework for proving programs using natural proofs and invariant synthesis using learning

- If the Hoare triple is $\{\alpha\}s\{\gamma\}$, where γ is the synthesized invariant, then we take the avoid set T_+ to be the set of predicates p_i such that $v_i' = \mathit{false}$.
- If the Hoare triple is $\{\gamma\}s\{\alpha\}$, where γ is the synthesized invariant, then we take differ set T_- to be the set of predicates p_i such that $v_i = \mathit{true}$.
- If the Hoare triple is $\{\gamma\}s\{\gamma\}$, where γ is the synthesized invariant, then we form the double constraint (U_1, U_2), where U_1 is the set of predicates p_i such that $v_i = \mathit{true}$ and U_2 is the set of predicates p_i such that $v_i' = \mathit{false}$.

It is straightforward to see that this accurately gives the non-provability entailed by the counterexample model found by the SMT engine, and the sample meets the criteria of avoid sets, differ sets, and double constraints described in Section 2.1. For instance, when a proving the abstracted verification condition of the form $\{\alpha\}s\{\gamma\}$ fails, and the SMT solver finds a model for the negation of the verification condition, this model will assign a nonempty subset of predicates in γ to false, signifying that *at least* this subset of predicates are not provable, hence justifying that subset as an avoid set (note that the predicates in γ marked true are not necessarily provable). Similarly, when an abstracted verification condition for a Hoare triple $\{\gamma\}s\{\alpha\}$ fails, the satisfying model for the negated formula will assign some predicates in γ to true, signifying that even with that subset is set to true, proving the postcondition would be impossible; this justifies returning this subset of predicates as a differ set. The intuition for implication examples is similar.

The above gives an adequate mechanism to extract and report non-provability information when natural proofs fail, and hence allows us to pair the natural verification engine with any LSUC learner to obtain an invariant synthesis tool.

5.3 A Framework Combining Natural Proofs and Invariant Learning

We now describe our proposed framework that combines natural proofs and black-box invariant learning of conjunctions, as depicted in Figure 1. As input we take a program P with DRYAD definitions and annotations. We then apply the translation of DRYAD constraints to classical logic from [61, 65] to obtain the program P with purely classical logic annotations, which involve recursive definitions over sets of locations and integers. This program can then be transformed using natural proof transformations that unfold definitions depending on the dereferences the program makes, and abstract recursive definitions to uninterpreted functions, again from [61, 65].

Templates for enumerating predicates that occur in invariants

Given the DRYAD definitions of data structures, we generate a set of *predicates*, which serve as the basic building blocks of our invariants. Figure 2 shows the class of templates from which the actual predicates are generated; the templates are instantiated using all combinations of program variables that occur in the program being verified.

The templates define a fairly exhaustive set of predicates that are allowed to occur in an invariant. These predicates include properties of the store (equality of pointer variables, equality and inequalities between integer variables, etc.), shape properties (singly and doubly linked lists and list segments, sorted lists, trees, BST, AVL, treaps, etc.), and recursive definitions that map data structures to numbers (keys/data stored in a structure, lengths of lists and list segments, height of trees) involving arithmetic relationships and set relationships. In addition, there are also predicates describing heaplets of various structures (with suffix *_heaplet*), involving set operations, disjointness, and equalities. The structures and predicates are extensible, of course, to any recursive definition expressed in DRYAD .

The predicates are grouped into three categories, roughly in increasing complexity. Category 1 predicates involve shape-related properties, Category 2 involves properties related to the keys stored in the data-structure, and Category 3 predicates involve size-predicates on data structures (lengths of lists and heights of trees). Given a program to verify and its annotations, depending on what kinds of properties the specification refers to (shape only, shapes and keys, or shapes, keys, and sizes), we pick the category of predicates automatically (choosing a category automatically includes the predicates of lower category as well). Predicates are enumerated according to these templates uniformly, and this results in a large class of predicates that grow with the number of variables of each type in the program.

Learning Heap Invariants

The program verifier is then paired with an LSUC learner that learns invariants expressed as a conjunction of the enumerated predicates, in an iterative learning loop (see the bottom part of Figure 1). Using the reduction of learning sets under constraints to ICE learning of conjuncts (Section 3), we can use either of the ICE learners for conjuncts we developed in Section 4. This phase either succeeds in finding an inductive invariant *that is provable using natural proofs* or terminates reporting that no such inductive invariant exists.

6. Evaluation

We implemented and evaluated our verification framework shown in Figure 1. Our main goal in the evaluation was to determine whether the LSUC learning framework, in particular the learners LSUC_HOUDINI and LSUC_SORCAR, are able to, in practice, effectively learn inductive invariants for natural proofs on heap-manipulating programs using non-provability information. Secondly, we wanted to compare the relative efficacy of LSUC_HOUDINI and LSUC_SORCAR in the domain of natural proofs for heap verification.

The programs in our suite of benchmarks are annotated in a special expressive fragment of separation logic called DRYAD , and our focus here is to evaluate the invariant synthesis learning framework for natural proof verification engines for this logic (there

$x, y, x_1, x_2 \in \mathit{PointerVars} \quad \vec{x}, \vec{y}, \vec{z} \in \mathit{PointerVars}^* \quad i, j \in \mathit{IntegerVars} \cup \{0, \mathsf{IntMax}, \mathsf{IntMin}\} \quad \mathit{pf} \in \mathit{PointerFields} \quad \mathit{key}, \mathit{df} \in \mathit{DataFields}$

$$
\begin{array}{rcl}
\mathit{listshape}(\vec{x}) & := & \mathsf{LinkedList}(x_1) \mid \mathsf{DoublyLinkedList}(x_1) \mid \mathsf{SortedLinkedList}(x_1) \\
& \mid & \mathsf{LinkedListSeg}(x_1, x_2) \mid \mathsf{DoublyLinkedListSeg}(x_1, x_2) \mid \mathsf{SortedLinkedListSeg}(x_1, x_2) \\
\mathit{treeshape}(x) & := & \mathsf{BST}(x) \mid \mathsf{AVLtree}(x) \mid \mathsf{Treap}(x) \\
\mathit{shape}(\vec{x}) & := & \mathit{listshape}(\vec{x}) \mid \mathit{treeshape}(\vec{x}) \\
\mathit{size}(\vec{x}) & := & \mathit{listshape}\_\mathsf{length}(\vec{x}) \mid \mathit{treeshape}\_\mathsf{height}(\vec{x})
\end{array}
$$

Category 1	$\mathit{shape}(\vec{x})$	$x = y$	$x = \mathsf{nil}$
	$x \in \mathit{shape}\_\mathsf{heaplet}(\vec{y})$	$x \neq y$	$x \neq \mathsf{nil}$
	$x \notin \mathit{shape}\_\mathsf{heaplet}(\vec{y})$	$x.\mathit{pf} = y$	$x.\mathit{pf} = \mathsf{nil}$
	$\mathit{shape}\_\mathsf{heaplet}(\vec{x}) \cap \mathit{shape}\_\mathsf{heaplet}(\vec{y}) = \emptyset$	$x.\mathit{pf} \neq y$	$x.\mathit{pf} \neq \mathsf{nil}$
Category 2	$i \in \mathit{shape\_key}\_\mathsf{set}(\vec{x})$	$x.\mathit{df} \leq y.\mathit{df}$	$x.\mathit{df} \leq i$
	$i \notin \mathit{shape\_key}\_\mathsf{set}(\vec{x})$	$x.\mathit{df} \geq y.\mathit{df}$	$x.\mathit{df} \geq i$
	$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) = \mathit{shape\_key}\_\mathsf{set}(\vec{y})$	$x.\mathit{df} = y.\mathit{df}$	$x.\mathit{df} = i$
	$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) = \mathit{shape\_key}\_\mathsf{set}(\vec{y}) \cup \mathit{shape\_key}\_\mathsf{set}(\vec{z})$	$x.\mathit{df} \neq y.\mathit{df}$	$x.\mathit{df} \neq i$
	$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \mathit{shape\_key}\_\mathsf{set}(\vec{y})$	$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \{y.\mathit{df}\}$	$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \{i\}$
	$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \mathit{shape\_key}\_\mathsf{set}(\vec{y})$	$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \{y.\mathit{df}\}$	$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \{i\}$

Category 3	$\mathit{size}(\vec{x}) = i$	$\mathit{size}(\vec{x}) - \mathit{size}(\vec{y}) = i$	$\mathit{size}(\vec{x}) \leq i$
	$\mathit{size}(\vec{x}) = i - j$	$\mathit{size}(\vec{x}) - \mathit{size}(\vec{y}) = i - j$	$\mathit{size}(\vec{x}) \geq i$

Figure 2: **Templates for predicates.** The operator $\leq_{\mathsf{set}}$ denotes comparison between integer sets, where $A \leq_{\mathsf{set}} B$ if and only if $\forall x \in A. \forall y \in B.\ x \leq y$. The operator $\geq_{\mathsf{set}}$ is similarly defined. Shape properties such as LinkedList, AVLtree, etc., are recursively defined in DRYAD , separately, and is extensible to any class of DRYAD defined shapes. Similarly, the definitions related to keys stored in a data structure and the sizes of data structures also stem from recursive definitions of them in DRYAD .

are no invariant synthesis engines for DRYAD that we can compare to). There are, of course, several other interesting fragments of separation logic (some decidable, some not) studied in the literature—we refer the reader to the related work in Section 1 for a survey of such work. Implementing the invariant synthesis using our LSUC learners for these systems would require extracting non-provability information from such engines, and comparing their efficacy to existing invariant synthesis engines for them is interesting future work.

Later in this section (Section 6.3), we also report on an optional optimization using dynamic runs on test-inputs and report an evaluation of it.

6.1 Implementation

We implemented the verification framework (described in Section 5). We use VCDryad [61], a Boogie/Z3-based natural proof engine that builds on VCC for verifying C programs, as the verification engine. We implemented the HOUDINI [4, 31] and SORCAR ICE-learning algorithms, and used that to build the LSUC learners LSUC_HOUDINI and LSUC_SORCAR, using our reduction from LSUC to ICE learning.

6.2 Benchmarks

Our benchmarks are primarily drawn from the benchmarks used in evaluating VCDRYAD [61], since this was our verification oracle. This benchmark suite consists of various list and tree manipulating programs, including singly-linked lists, doubly-linked lists, sorted lists, binary search trees, AVL trees, and treaps. The programs were taken from GNU C library, OpenBSD operating system, ExpressOS [56], a secure operating system for mobile devices, benchmark suites of other verification engines, namely GRASShopper [62] and AFWP [45]. The specifications for these programs are generally for full functional correctness properties, such as preserving or altering shapes of data structures, inserting or deleting keys, filtering or finding elements, and sorted-ness of elements in structures. The specifications hence involve separation logic with arithmetic as well as recursive definitions that compute numbers (like lengths and heights) and data-aggregating recursive functions (like multisets of keys stored in data-structures), and complex data-structures that combine these properties (like binary search trees, AVL trees and treaps).

From this benchmark suite, we first picked all programs that contained iterative loops, and *erased* the user-provided loop invariants, and used our framework to find an adequate inductive invariant. We also selected some of the programs that were purely recursive, where the contract for the function had been strengthened to make the verification succeed. We *weakened* these contracts to only state the specification (typically by removing formulas in the post-conditions of recursively-called functions), and introduced annotation holes instead, and use our framework to synthesize strengthening. Our tool hence was used in both learning invariants for iterative loops as well as strengthening contracts for recursive programs.

We also chose five more non-recursive programs, and deleted their post-conditions, and evaluated whether we can learn the post-conditions for them. Note that in this case, the formula *true* is always a valid annotation for the post-condition, and hence it makes sense only to learn the *strongest* post-condition expressible as a conjunct. Consequently, we can only use LSUC_HOUDINI for this problem as it is the one that guarantees learning the strongest post-condition expressible as a conjunction of base predicates (LSUC_SORCAR will typically learn *true* for these programs).

6.3 Experimental Results

We evaluate our framework against 79 routines, and the results of the experiments are tabulated in Table 2. For each routine, we report the number of candidate predicates enumerated using our templates and the category of predicates used for it (see Section 5.3 for details on categories). Note that there are five programs that are subject to post-condition learning, and hence were evaluated only using LSUC_HOUDINI.

All experiments were performed on a guest virtual machine running Windows 7 on a single processor of Intel Core i7 2.2 GHz and 2GB memory.

Observation#1: The LSUC learners are effective in finding inductive invariants: Both LSUC learners (LSUC_HOUDINI and LSUC_SORCAR) are extremely effective in finding inductive invariants that result in proving the programs correct using natural proofs (with an average of less than 2 minutes per routine). Despite having

Program	#Cand-idate Pred	Cate-gory	LSUC_HOUDINI			LSUC_SORCAR		
			#Rou-nds	|Inv|	Time (s)	#Rou-nds	|Inv|	Time (s)
GNU C Library(glibc) Singly and Sorted Linked-List								
g_slist_copy	428	2	9	137	17.2	3	85	13.5
g_slist_find	50	2	7	10	3.4	8	5	3.5
g_slist_free	22	1	5	2	2.8	5	1	4.0
g_slist_index	261	3	10	68	9.7	10	50	10.1
g_slist_insert	548	2	22	76	954.3	9	36	**173.9**
g_slist_insert_before	935	2	15	160	581.0	10	43	**193.4**
g_slist_insert_sorted	556	2	10	139	217.7	9	36	206.2
g_slist_last	32	2	4	9	1.8	5	6	2.4
g_slist_length	56	3	6	13	5.0	6	9	4.0
g_slist_nth	92	3	10	18	6.1	9	10	5.8
g_slist_nth_data	372	3	9	75	9.6	11	40	12.2
g_slist_position	180	3	7	20	9.2	8	12	9.3
g_slist_remove	188	1	5	53	3.9	5	24	3.9
g_slist_remove_all	428	2	18	26	204.8	20	13	190.7
g_slist_remove_link	385	1	7	66	**10.8**	8	27	118.1
g_slist_reverse	129	2	8	13	13.4	11	8	23.7
GNU C Library(glibc) Doubly Linked-List								
g_list_find	50	2	7	10	4.2	8	6	4.0
g_list_free	22	1	5	2	3.8	5	1	4.7
g_list_index	261	3	9	68	11.1	11	50	10.5
g_list_last	22	1	3	7	1.3	5	4	2.8
g_list_length	92	3	10	17	9.9	5	11	3.1
g_list_nth	92	3	10	18	6.4	9	10	6.1
g_list_nth_data	372	3	10	75	10.7	9	52	11.3
g_list_position	180	3	8	20	20.7	8	12	8.8
g_list_reverse	368	2	13	14	146.5	10	11	**89.1**
AWFP Singly and Sorted Linked-List								
SLL-create	5	1	4	2	1.2	4	1	1.4
SLL-delete-all	22	1	4	2	3.2	4	1	3.7
SLL-delete	385	1	7	91	6.0	6	32	6.6
SLL-filter	75	1	5	14	2.7	7	10	5.7
SLL-find	188	1	8	58	3.3	6	27	2.4
SLL-insert	219	2	11	31	115.5	7	15	**41.8**
SLL-last	75	1	5	14	1.7	7	10	5.0
SLL-merge	464	2	12	70	736.0	13	57	**633.6**
SLL-reverse	75	1	6	11	2.5	9	10	5.9
ExpressOS MemoryRegion								
find	24	1	6	3	1.5	5	1	1.6
insert	51	1	8	4	1.6	7	1	2.0
init*	7	1	3	5	1.0			N/A
split*	24	1	7	7	4.3			N/A
OpenBSD SysQueue								
insert_head*	5	1	3	3	17.0			N/A
insert_tail*	5	1	2	4	9.4			N/A
remove_head*	12	1	3	5	2.2			N/A

Program	#Cand-idate Pred	Cate-gory	LSUC_HOUDINI			LSUC_SORCAR		
			#Rou-nds	|Inv|	Time (s)	#Rou-nds	|Inv|	Time (s)
GRASShopper Singly Linked-List								
sl_concat	75	1	6	23	2.4	6	12	2.8
sl_copy	75	1	6	21	4.8	10	20	12.3
sl_dispose	22	1	5	2	2.4	5	1	2.3
sl_filter	188	1	9	20	10.4	9	16	14.0
sl_insert	75	1	3	27	1.4	6	12	3.1
sl_remove	22	1	3	7	1.6	4	4	2.0
sl_reverse	75	1	6	11	4.7	9	10	6.4
GRASShopper Sorted Linked-List								
sls_concat	165	2	8	35	17.1	7	20	19.6
sls_copy	556	2	12	129	1589.1	11	83	**1213.0**
sls_dispose	40	2	8	4	6.5	10	3	8.3
sls_double_all	556	2	16	135	48.8	9	73	49.9
sls_filter	556	2	19	41	**439.6**	17	19	526.1
sls_insert	165	2	5	37	10.2	7	20	16.1
sls_merge	464	2	13	48	**350.9**	18	39	576.9
sls_remove	556	2	10	146	82.2	10	60	**43.7**
sls_reverse	102	2	6	14	23.9	9	12	38.2
sls_split	165	2	8	34	**250.2**	10	25	359.7
GRASShopper Doubly Linked-List								
dl_concat	75	1	7	23	2.2	5	12	3.6
dl_copy	75	1	6	21	7.2	5	14	6.2
dl_dispose	188	1	9	14	11.2	12	10	16.2
dl_filter	188	1	9	20	8.6	8	16	10.9
dl_insert	75	1	4	27	1.4	6	12	3.8
dl_remove	22	1	4	7	1.4	5	4	2.0
dl_reverse	75	1	6	11	3.9	9	10	5.8
VCDryad Sorted Linked-List								
insert_sorted	219	2	10	54	120.4	8	28	**83.4**
find_last_sorted	40	2	4	12	2.5	6	9	3.5
reverse_sorted	102	2	6	14	12.9	7	8	15.5
VCDryad Trees								
avl-delete-rec†	72	3	7	6	306.7	5	2	313.0
avl-find-smallest†	19	3	4	12	1.4	3	7	1.7
avl-insert-rec†	72	3	9	15	252.3	5	2	**137.0**
bst-delete-rec†	68	2	9	12	143.3	3	2	**55.5**
bst-find-rec†	23	2	4	10	2.3	3	4	1.9
bst-insert-rec†	68	2	8	17	42.1	7	4	60.0
traverse-inorder†	9	3	4	4	1.3	4	2	2.7
traverse-posttorder†	9	3	4	4	1.3	4	2	1.8
traverse-preorder†	9	3	4	4	1.3	4	2	6.2
treap-delete-rec†	80	3	10	14	427.2	3	2	**200.3**
treap-find-rec†	25	3	4	12	2.3	3	6	2.3

Table 2: Experimental results comparing the performance of HOUDINI and SORCAR. Total time, HOUDINI:7374s SORCAR:5652s. The times in bold indicates an improvement of over 20s. † Contract strengthening programs. * Post-condition learning programs (only Houdini).

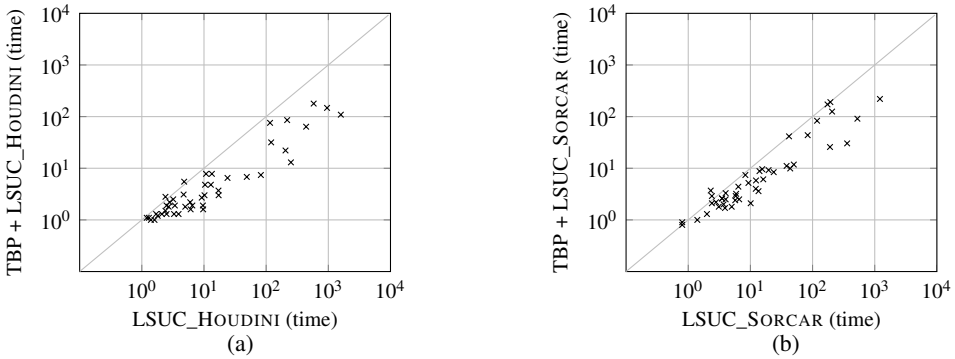

Figure 3: Scatter plot comparison. Time in s. TBP=Testing-based Pruning.

hundreds of candidate predicates in many examples, which in turn suggests a worst-case complexity of hundreds of rounds, the learners were able to learn with much fewer rounds. The non-provability information provided by the natural proof engine provides much more information than the worst-case suggests. Also, on the five programs, LSUC_HOUDINI was able to learn the strongest post-condition (expressible as a conjunction of the given predicates) in reasonable time.

The above is the most important observation as it validates the framework of learning from non-provability information proposed in this paper.

Observation#2: LSUC_SORCAR mildly outperforms LSUC_HOUDINI: There is no clear winner between the two learners— on some programs, LSUC_HOUDINI performs better while on some others, LSUC_SORCAR performs better. However, overall, LSUC_SORCAR performs slightly faster, taking about 20% less time than LSUC_HOUDINI in completing the entire set of benchmarks. Furthermore, the size of the invariants learned by LSUC_SORCAR were significantly smaller than those learned by LSUC_HOUDINI. These results suggest that property-directed invariant generation can help both reduce time and succinctness of invariants synthesized.

6.4 An optimization: Testing-based Pruning

When enumerating predicates according to templates, we create hundreds of candidate predicates, most of which are not invariants in the program. Our learning framework requires the learning algorithm to eliminate such predicates with the help of the verification oracle, which is expensive.

However, if we have a few test inputs to the program being verified (that satisfy the program's preconditions), then we can execute the program on these inputs, and eliminate all predicates that are falsified at least once when the program reaches the annotation we are trying to synthesize. This is sound because the predicates in a conjunctive invariant obviously cannot contain a predicate that is violated in a test. This technique is in fact precisely the same as Daikon [30], which learns *likely invariants* using dynamic executions on concrete test inputs. Our idea here is to use these likely invariants to prune the set of predicates initially, before subjecting the program to invariant synthesis using our LSUC learning framework. This idea is not new— the original paper on Houdini [31] already suggested such an idea.

We implement the testing-based pruning and evaluate its effectiveness. We consider all the singly linked-list and sorted linked-list benchmarks (45 programs in total). For each program, we created in fact precisely *one* test input that satisfies the precondition (this test input was not chosen with care, but chosen as a typical input to the program). By executing each program on its single test input, we instrumented the program to automatically check and prune predicates that are falsified during the execution. Our LSUC learning framework was then used to find inductive invariants using the pruned predicates.

Figure 3(a) and Figure 3(b) show the comparison of the times taken by LSUC_HOUDINI and LSUC_SORCAR, with and without testing based reduction. On an average, we found testing based reduction reduces the initial set of candidate predicates by 64%, decreasing the average time taken for each program by 83% for HOUDINI and 66% for SORCAR.

We observe that testing-based pruning helps LSUC_HOUDINI much more than it helps LSUC_SORCAR. This suggests that the relevancy mining of predicates in LSUC_SORCAR is already able to filter out irrelevant predicates, leading to less advantage of using testing-based pruning.

We conclude that when test inputs for programs exist, testing-based pruning can be a very effective strategy to better scale the techniques presented in this paper.

7. Conclusions and Future Work

The Learning Sets Under Constraints model introduced in this paper is a learning framework that allows incomplete verification engines to communicate non-provability information that allows the learner to synthesize conjunctive invariants. We have developed two learners under this framework and shown the efficacy of the model and learner to synthesize separation logic invariants on a suite of heap-manipulating programs.

Several future directions are interesting. First, it would be interesting to implement our learning framework and learners in the context of other verification problems where typically only incomplete verification engines exist, such as programs that require non-linear arithmetic reasoning or for verification engines for other variants of separation logics [2, 17, 18]. Second, it would be interesting to extend our learning model beyond conjuncts to more general logics that express the invariant. The key difficulty here would be to define appropriate non-provability information that verification engines can furnish and that effective algorithmic learning techniques can use to construct invariants. Finally, we believe that the SORCAR learning algorithm developed here is a promising ICE-learning algorithm that can be readily useful even in invariant synthesis contexts where the verification oracle provides concrete counterexamples. It would be interesting to see its efficacy by replacing HOUDINI with SORCAR in practical fully automated tools that perform invariant synthesis, in applications including device-driver verification [52], automatically proving absence of dataraces in GPU programs [7], and certified symbolic transactions [15].

References

[1] P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 224–239, 2013.

[2] A. Albarghouthi, J. Berdine, B. Cook, and Z. Kincaid. Spatial interpolants. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 634–660, 2015.

[3] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: smt-based abstraction for arrays with interpolants. In *CAV Berkeley*, volume 7358 of *LNCS*, pages 679–685. Springer, 2012.

[4] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, pages 52–68, 2005.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 115–137, 2005.

[7] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: a verifier for GPU kernels. In *OOPSLA 2012*, pages 113–132. ACM, 2012.

[8] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 184–190, 2011.

[9] A. Bouajjani, C. Dragoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *CAV 2010*, volume 6174 of *LNCS*, pages 72–88. Springer, 2010.

[10] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 578–589, 2011.

[11] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI 2011*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[12] J. Brotherston, C. Fuhs, J. A. N. Pérez, and N. Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 25:1–25:10, 2014.

[13] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.

[14] B. E. Chang and X. Rival. Relational inductive shape analysis. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 247–260, 2008.

[15] E. Y. Chen, S. Chen, S. Qadeer, and R. Wang. Securing multiparty online services via certification of symbolic transactions. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 833–849. IEEE Computer Society, 2015.

[16] Y.-F. Chen and B.-Y. Wang. Learning boolean functions incrementally. In *CAV 2012*, volume 7358 of *LNCS*, pages 55–70. Springer, 2012.

[17] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, Aug. 2012. ISSN 0167-6423.

[18] D. Chu, J. Jaffar, and M. Trinh. Automatic induction proofs of data-structures in imperative programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 457–466. ACM, 2015.

[19] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[20] E. Cohen, W. J. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In *SOFSEM 2013: Theory and Practice of Computer Science, 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 26-31, 2013. Proceedings*, volume 7741 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2013.

[21] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV 2003*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.

[22] B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *Proceedings of the 22Nd International Conference on Concurrency Theory*, CONCUR'11, pages 235–249, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23216-9.

[23] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, pages 238–252. ACM Press, 1977.

[24] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 105–118, 2011.

[25] L. M. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 183–198, 2007.

[26] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[27] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[28] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33056-9, 978-3-540-33056-1.

[29] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE 2000*, pages 449–458. ACM Press, 2000.

[30] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[31] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[32] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[33] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV 2013*, volume 8044 of *LNCS*, pages 813–829. Springer, 2013.

[34] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV 2014*, volume 8559 of *LNCS*, pages 69–87. Springer, 2014.

[35] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Quantified data automata for linear data structures: a register automaton model with

applications to learning invariants of programs manipulating arrays and lists. *Formal Methods in System Design*, 47(1):120–157, 2015.

[36] P. Garg, P. Madhusudan, D. Neider, and D. Roth. Learning Invariants using Decision Trees and Implication Counterexamples. In *Proceedings of the 43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, to appear.

[37] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 306–320, 2009.

[38] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 235–246, 2008.

[39] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI 2008*, pages 281–292. ACM, 2008.

[40] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *CAV 2009*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.

[41] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 165–181. USENIX Association, 2014.

[42] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17, 2015.

[43] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[44] R. Iosif, A. Rogalewicz, and J. Simacek. The tree width of separation logic with recursive definitions. In *Proceedings of the 24th International Conference on Automated Deduction*, CADE'13, pages 21–38, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-38573-5.

[45] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 756–772. Springer, 2013.

[46] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS 2006*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[47] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV 2007*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.

[48] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 583–602, 2015.

[49] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

[50] S. Kong, Y. Jung, C. David, B. Wang, and K. Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *APLAS 2010*, volume 6461 of *LNCS*, pages 328–343. Springer, 2010.

[51] S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[52] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV 2012*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.

[53] Q. L. Le, C. Gherghina, S. Qin, and W. Chin. Shape analysis via second-order bi-abduction. 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*, pages 52–68, 2014.

[54] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR 2010*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[55] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 123–136, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3.

[56] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in expressos. In *ASPLOS 2013*, pages 293–304. ACM, 2013.

[57] K. L. McMillan. Interpolation and SAT-Based model checking. In *CAV 2003*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[58] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS 2008*, volume 4963 of *LNCS*, pages 413–427. Springer, 2008.

[59] J. A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 556–566, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8.

[60] D. Neider. *Applications of Automata Learning in Versification and Synthesis*. PhD thesis, RWTH Aachen University, April 2014.

[61] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI 2014*, page 46. ACM, 2014.

[62] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 773–789. Springer, 2013.

[63] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. 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*, pages 711–728, 2014.

[64] R. Piskac, T. Wies, and D. Zufferey. Grasshopper - complete heap verification with mixed specifications. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 124–139, 2014.

[65] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI 2013*, pages 231–242. ACM, 2013.

[66] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[67] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3.

[68] M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS 2009*, volume 5673 of *LNCS*, pages 3–18. Springer, 2009.

[69] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV 2014*, volume 8559 of *LNCS*, pages 88–105. Springer, 2014.

[70] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV 2012*, volume 7358 of *LNCS*, pages 71–87. Springer, 2012.

[71] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP 2013*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.

[72] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS 2013*, volume 7935 of *LNCS*, pages 388–411. Springer, 2013.

[73] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 99–110, 2010.

[74] H. Zhu, G. Petri, and S. Jagannathan. Automatically learning shape specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 491–507, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908125.