Active Learning of Points-To Specifications

Osbert Bastani
Stanford University
USA
obastani@cs.stanford.edu

Rahul Sharma
Microsoft Research
India
rahsha@microsoft.com

Alex Aiken
Stanford University
USA
aiken@cs.stanford.edu

Percy Liang
Stanford University
USA
pliang@cs.stanford.edu

Abstract

When analyzing programs, large libraries pose significant challenges to static points-to analysis. A popular solution is to have a human analyst provide *points-to specifications* that summarize relevant behaviors of library code, which can substantially improve precision and handle missing code such as native code. We propose Atlas, a tool that automatically infers points-to specifications. Atlas synthesizes unit tests that exercise the library code, and then infers points-to specifications based on observations from these executions. Atlas automatically infers specifications for the Java standard library, and produces better results for a client static information flow analysis on a benchmark of 46 Android apps compared to using existing handwritten specifications.

CCS Concepts • **Theory of computation** → *Program analysis*;

Keywords specification inference, static points-to analysis

ACM Reference Format:
Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-To Specifications. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192383

1 Introduction

When analyzing programs, dependencies on large libraries can significantly reduce the effectiveness of static analysis, since libraries frequently contain (i) native code that cannot be analyzed, (ii) use of challenging language features such as reflection, and (iii) deep call hierarchies that reduce precision. For example, the implementation of the `Vector` class in OpenJDK 1.7 uses multiple levels of call indirection and calls the native function `System.arrayCopy`.

A standard workaround is to use *specifications* that summarize the relevant behaviors of library functions [14, 48]. For a one-time cost of writing library specifications, the precision and soundness of the static analysis can improve dramatically when analyzing any client code. However, writing specifications for the entire library can be expensive [9, 48]—many libraries contain a large number of functions, manually written specifications are often error prone [18], and specifications must be updated whenever the library is updated.

To address these issues, approaches have been proposed for automatically *inferring* specifications for library code, both based on dynamic analysis [3, 8, 30, 35, 36] and on static analysis [4, 12, 24, 26, 32, 39]. In particular, tools have been designed to infer properties of missing code, including taint flow properties [13], function models [18, 19], and callback control flow [20]. While these approaches are incomplete, and may not infer sound specifications, current static analyses used in production already rely on user-provided specifications [14], and as we will show, tools that automatically infer specifications can outperform human analysts.

We propose an algorithm based on dynamic analysis that infers library specifications summarizing points-to effects relevant to a flow-insensitive points-to analysis. Our algorithm works by iteratively guessing candidate specifications and then checking whether each one is "correct". Intuitively, a specification is correct if it must be included in any sound set of specifications, i.e., it is precise. This property ensures that the specification does not cause any false positives. There are two constraints that make our problem substantially more challenging than previously studied settings:

- Points-to effects cannot be summarized for a library function in isolation, e.g., in Figure 1, `set`, `get`, and `clone` all refer to the same field `f`. Thus, a guessed candidate must simultaneously summarize the points-to effects of `set`, `get`, and `clone`. Furthermore, the inference algorithm may not know the library field `f` exists, and must invent a *ghost field* to represent it.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192383

- It is often difficult to instrument library code (e.g., native code); we assume only blackbox access.

To address these challenges, we introduce *path specifications* to describe points-to effects of library code. Each path specification summarizes a single points-to effect of a combination of functions. An example is:

> For two calls `box.set(x)` and `box.get()`, the return value of `get` may alias `x`.

Path specifications have two desirable properties: (i) we can check if a candidate path specification is correct using input-output examples, and (ii) correctness of path specifications is independent, i.e., a set of path specifications is correct if each specification in the set is correct. These two properties imply that we can infer path specifications incrementally. In particular, we propose an active learning algorithm that infers specifications by actively identifying promising candidate specifications, and then automatically checking the correctness of each candidate independently.

We implement our algorithm in a tool called Atlas,[1] which infers path specifications for functions in Java libraries. In particular, we evaluate Atlas by using it to infer specifications for the Java standard library, including the Collections API, which contains many functions that exhibit complex points-to effects. We evaluate Atlas using a standard static explicit information flow client [7, 8, 16, 17] on a benchmark of 46 Android apps. Our client uses handwritten points-to specifications for the Java standard library [9, 48]. Over the course of two years, we have handwritten a number of points-to specifications tailored to finding information flows in apps we analyzed, including those in our benchmark. However, these specifications remain incomplete.

We use Atlas to infer specifications for classes in six commonly used packages in the Java standard library. Atlas inferred more than 5× as many specifications as the existing, handwritten ones. We show that using the specifications inferred by Atlas instead of the handwritten ones improves the recall of our information flow client by 52%. Moreover, we manually wrote ground truth specifications for the most frequently used subset of classes in the Collections API, totalling 1,731 lines of code, and show that Atlas inferred the correct specifications (i.e., identical to handwritten ground truth specifications) for 97% of these functions. Finally, we show that on average, using specifications more than halves the number of false positive points-to edges compared to directly analyzing the library implementation—in particular, the library implementation contains deep call hierarchies, which our points-to analysis is insufficiently context-sensitive to handle. Our contributions are:

- We propose path specifications and prove they are sufficiently expressive to precisely model library functions for a standard flow-insensitive points-to analysis.
- We formulate the problem of inferring path specifications as a language inference problem and design a novel specification inference algorithm.
- We implement our approach in Atlas, and use it to infer a large number of specifications for the Java standard library. We use these inferred specifications to automatically replicate and even improve the results of an existing static information flow client.

[1] Atlas stands for AcTive Learning of Alias Specifications, and is available at https://github.com/obastani/atlas.

2 Overview

Consider the program `test` shown in Figure 1. We assume that the static analysis resolves aliasing using Andersen's analysis [5], a context- and flow-insensitive points-to analysis; our approach also applies to cloning-based context- and object-sensitive extensions [44]. To determine that variables `in` and `out` may be aliased, the points-to analysis has to reason about the heap effects of the `set` and `get` methods in the `Box` class. The analyst can write *specifications* for library functions that summarize their heap effects with respect to the semantics of the points-to analysis. For example, consider the `Stack` class in the Java library: its implementation extends `Vector`, which is challenging to analyze due to deep call hierarchies and native code. The following are the specifications for the `Stack` class, implemented as code fragments that overapproximate the heap effects of the methods:

```
class Stack { // specification
  Object f; // ghost field
  void push(Object ob) { f = ob; }
  Object pop() { return f; } }
```

In contrast to the implementation, the specifications for `Stack` are simple and easy to analyze. In fact, when used with our static points-to analysis, these specifications are not only sound but also precise, since our points-to analysis is flow-insensitive and collapses arrays into a single field.

A typical approach is to write specifications lazily [9, 48]—the analyst examines the program being analyzed, identifies the library functions most relevant to the static analysis, and writes specifications for them. The effort invested in writing specifications helps reduce the labor required to discharge false positives.[2] The analyst can trade off between manual effort and soundness by expending more effort as needed to increase the completeness of the library specifications.

Atlas helps bootstrap the specification writing process by automatically inferring points-to specifications. In accordance with the goal of minimizing false positives, Atlas employs a two-phase approach that prioritizes the precision of the specifications it infers (where precision is defined with respect to our points-to analysis). In the first phase, Atlas infers specifications guaranteed to be precise. In particular, each inferred specification s comes with a *witness*, which intuitively is a unit test that proves the precision of s by exhibiting the heap effects specified by s. In the second phase,

[2] In contrast to the recurring cost of debugging false positives, the cost of implementing specifications is amortized over many programs.

```
class Box { // library
  Object f;
  void set(Object ob) { f = ob; }
  Object get() { return f; }
  Box clone() {
    Box b = new Box(); // o_clone
    b.f = f;
    return b; } }
```

```
boolean test() { // program
  Object in = new Object(); // o_in
  Box box = new Box(); // o_box
  box.set(in);
  Object out = box.get();
  return in == out; }
```

Figure 1. Implementation of the library methods `set`, `get`, and `clone` in the Box class (left), and an example of a program using these functions (right).

Atlas inductively generalizes the specifications inferred in the first phase, using a large number of checks to minimize the possibility of imprecision.

Specification search space. The main challenge in inferring points-to specifications is to formulate the search space of candidate specifications. Naïvely searching over the space of code fragments is ineffective because specifications are highly interdependent, i.e., the specification of a library function is dependent on the specifications of other functions. For example, the specifications for each of the methods in the Box class all refer to the shared field f. Thus, to infer precise specifications, the naïve algorithm would have to guess a code fragment for every function in the library (or at least in a class) before it can check for precision; the likelihood that it could make a correct guess is tiny.

Our key insight is that while specifications cannot be broken at function boundaries, we can decompose them in a different way. In particular, we propose *path specifications*, which are independent units of points-to effects. Intuitively, a path specification is a dataflow path an object might take through library code. An example of a path specification is

$$s_{\mathrm{box}} = \mathtt{ob} \dashrightarrow \mathtt{this}_{\mathtt{set}} \rightarrow \mathtt{this}_{\mathtt{get}} \dashrightarrow r_{\mathtt{get}}. \tag{1}$$

This path specification roughly has the following semantics:

- Suppose that an object o enters the library code as the parameter ob of `set`; then, it is associated with the receiver $\mathtt{this}_{\mathtt{set}}$ of the `set` method (the edge $\mathtt{ob} \dashrightarrow \mathtt{this}_{\mathtt{set}}$).
- Suppose that in the client program, $\mathtt{this}_{\mathtt{set}}$ is aliased with $\mathtt{this}_{\mathtt{get}}$ (the edge $\mathtt{this}_{\mathtt{set}} \rightarrow \mathtt{this}_{\mathtt{get}}$); then, o enters the `get` method.
- Then, o exits the library as the return value $r_{\mathtt{get}}$ of `get`.

In particular, the dashed edges $\mathtt{ob} \dashrightarrow \mathtt{this}_{\mathtt{set}}$ and $\mathtt{this}_{\mathtt{get}} \dashrightarrow r_{\mathtt{get}}$ represent the effects of library code, and the solid edge $\mathtt{this}_{\mathtt{set}} \rightarrow \mathtt{this}_{\mathtt{get}}$ represents an assumption about the effects of client code. Then, the path specification says that, if the points-to analysis determines that $\mathtt{this}_{\mathtt{get}}$ and $\mathtt{this}_{\mathtt{set}}$ are aliased, then ob is *transfered* to $r_{\mathtt{get}}$, which intuitively means that ob is indirectly assigned to $r_{\mathtt{get}}$. More precisely, the semantics of s_{box} is the following logical formula:

$$(\mathtt{this}_{\mathtt{set}} \xrightarrow{\text{Alias}} \mathtt{this}_{\mathtt{get}}) \Rightarrow (\mathtt{ob} \xrightarrow{\text{Transfer}} r_{\mathtt{get}}). \tag{2}$$

Here, an edge $x \xrightarrow{A} y$ says that program variables x and y satisfy a binary relation A. We describe path specifications in detail in Section 4.

Testing specifications. Path specifications satisfy two key requirements. The first requirement is that we can check the precision of a single path specification. For example, for the specification s_{box}, consider the program `test` shown in Figure 1. This unit test satisfies three properties:

- It satisfies the antecedent of s_{box}, since $\mathtt{this}_{\mathtt{set}}$ and $\mathtt{this}_{\mathtt{get}}$ are aliased.
- It does not induce any other relations between the variables at the interface of the Box class (i.e., ob, $\mathtt{this}_{\mathtt{set}}$, i, $\mathtt{this}_{\mathtt{get}}$, and $r_{\mathtt{get}}$).
- It returns the consequent of s_{box}, i.e., whether ob and $r_{\mathtt{get}}$ point to the same object.

Upon executing `test`, it returns that the consequent of s_{box} is true; therefore, any sound set of specifications must include s_{box}, so s_{box} is precise (we formalize soundness and precision of path specifications in Section 4). We refer to such a unit test as a *witness* for the path specification. In summary, as long as we can find a witness for a candidate path specification s, then we can guarantee that s is precise. One caveat is that even if `test` returns false, s_{box} may still be precise.

Therefore, to check if a candidate path specification s is precise, our algorithm can synthesize a unit test P similar to `test` and execute it. If P returns true, then our algorithm concludes that s is precise; otherwise, it (conservatively) concludes that s is imprecise. Note that s may be precise even if P returns false; this possibility is unavoidable since executions are underapproximations, i.e., P may not exercise all paths of the relevant library functions. We use heuristics to ensure that our algorithm rarely concludes that precise specifications are imprecise. We describe our unit test synthesis algorithm in detail in Section 5.4.

The second requirement of path specifications is that they are independent, i.e., given a set of path specifications for which each specification has a witness, then the set as a whole is precise in the sense that it is a subset of any sound set of path specifications. In other words, we can use a potential witness to check the precision of a path specification in isolation. Thus, our specification inference algorithm can focus on inferring individual path specifications.

$$\text{(assign)}\ \frac{y \leftarrow x}{x \xrightarrow{\text{Assign}} y} \qquad \text{(allocation)}\ \frac{o = (x \leftarrow X())}{o \xrightarrow{\text{New}} o}$$

$$\text{(store)}\ \frac{y.f \leftarrow x}{x \xrightarrow{\text{Store}[f]} y} \qquad \text{(load)}\ \frac{y \leftarrow x.f}{x \xrightarrow{\text{Load}[f]} y} \qquad \text{(backwards)}\ \frac{x \xrightarrow{\sigma} y}{y \xrightarrow{\overline{\sigma}} x}$$

$$\text{(call parameter)}\ \frac{y \leftarrow m(x)}{x \xrightarrow{\text{Assign}} p_m} \qquad \text{(call return)}\ \frac{y \leftarrow m(x)}{r_m \xrightarrow{\text{Assign}} y}$$

Figure 2. Rules for constructing a graph G encoding the relevant semantics of program statements.

Specification inference. Our specification inference algorithm uses two inputs:

- **Library interface:** The type signature of each function in the library.
- **Blackbox access:** The ability to execute sequences of library functions on chosen inputs and obtain the corresponding outputs.

In the first phase (described in Section 5.2), Atlas randomly guesses a candidate path specification s, synthesizes a potential witness for s, and retains s if this unit test returns true. This process is repeated a large number of times to obtain a large but finite set of precise path specifications.

To soundly model the library, an infinite number of path specifications may be required, e.g., the path specifications required to soundly model `set`, `get`, and `clone` are

$$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} (\to \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}})^{*} \to \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}. \tag{3}$$

These specifications say that if we call `set`, then call `clone` n times in sequence, and finally call `get` (all with the specified aliasing between receivers and return values), then the parameter `ob` of `set` is transfered to the return value of `get`.

Thus, in the second phase (described in Section 5.3), we inductively generalize the set S of path specifications in the first phase to a possibly infinite set. We leverage the idea that a path specification can be represented as a sequence of variables $s \in \mathcal{V}_{\text{path}}^{*}$, where $\mathcal{V}_{\text{path}}$ are the variables in the library interface—for example, s_{box} corresponds to

$$\texttt{ob}\ \texttt{this}_{\texttt{set}}\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}} \in \mathcal{V}_{\text{path}}^{*}.$$

Thus, a set of path specifications is a formal language over the alphabet $\mathcal{V}_{\text{path}}$. As a consequence, we can frame the inductive generalization problem as a language inference problem: given (i) the finite set of positive examples from phase one, and (ii) an oracle we can query to determine whether a given path specification s is precise (though this oracle is *noisy*, i.e., it may return false even if s is precise), the goal is to infer a (possibly infinite) language $S \subseteq \mathcal{V}_{\text{path}}^{*}$.

We devise an active language learning algorithm to solve this problem. Our algorithm proposes candidate inductive generalizations of S, and then checks the precision of these

$$\begin{aligned}
\text{Transfer} &\to \varepsilon \mid \text{Transfer Assign} \mid \text{Transfer Store}[f]\ \text{Alias Load}[f] \\
\overline{\text{Transfer}} &\to \varepsilon \mid \overline{\text{Assign}}\ \overline{\text{Transfer}} \mid \overline{\text{Load}[f]}\ \text{Alias}\ \overline{\text{Store}[f]}\ \overline{\text{Transfer}} \\
\text{Alias} &\to \overline{\text{Transfer}}\ \overline{\text{New}}\ \text{New Transfer} \\
\text{FlowsTo} &\to \text{New Transfer}
\end{aligned}$$

Figure 3. Productions for the context-free grammar C_{pt}. The start symbol of C_{pt} is FlowsTo.

candidates using a large number of synthesized unit tests. Unlike phase one, we may introduce imprecision even if all the unit tests pass; we show empirically that precision is maintained. Finally, our algorithm infers a regular set of path specifications. In theory, no regular set may suffice to model the library code and more expressive language inference techniques might be required [11]. In practice, we find that regular sets are sufficient.

In our example, suppose that phase one infers

$$\begin{aligned}\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} &\to \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}} \\ &\to \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}} \to \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}.\end{aligned}$$

Then, phase two would inductively generalize it to (3). Finally, our tool automatically converts these path specifications to the equivalent code fragment specifications shown in Figure 1. These code fragment specifications can be used in place of the (possibly unavailable) library implementation when analyzing client code.

3 Background on Points-To Analysis

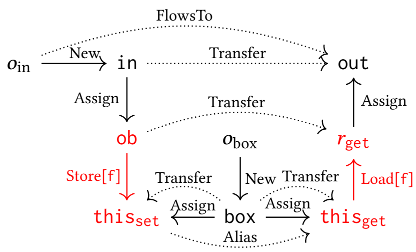
We consider programs with assignments $y \leftarrow x$ (where $x, y \in \mathcal{V}$ are variables), allocations $x \leftarrow X()$ (where $X \in C$ is a type), stores $y.f \leftarrow x$ and loads $y \leftarrow x.f$ (where $f \in \mathcal{F}$ is a field), and calls to library functions $y \leftarrow m(x)$ (where $m \in \mathcal{M}$ is a library function). For simplicity, we assume that each library function m has a single parameter p_m and a return value r_m.

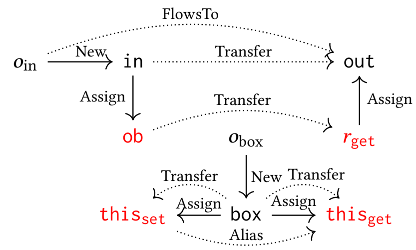
An *abstract object* $o \in O$ is an allocation statement $o = (x \leftarrow X())$. A *points-to edge* is a pair $x \hookrightarrow o \in \mathcal{V} \times O$. A static points-to analysis computes points-to edges $\Pi \subseteq \mathcal{V} \times O$. Our results are for Andersen's analysis, a flow-insensitive points-to analysis [5], but generalize to object- and context-sensitive extensions based on cloning [44]. We describe the formulation of Andersen's analysis as a context-free language reachability problem [22, 23, 33, 41, 42].

Graph representation. First, our static analysis constructs a labeled graph G representing the program semantics. The vertices of G are $\mathcal{V} \cup O$. The edge labels

$$\Sigma_{\text{pt}} = \{\text{Assign}, \text{New}, \text{Store}, \text{Load}, \overline{\text{Assign}}, \overline{\text{New}}, \overline{\text{Load}}, \overline{\text{Store}}\}$$

encode the semantics of program statements. The rules for constructing G are in Figure 2. For example, the edges extracted for the program `test` in Figure 1 are the solid edges in Figure 4.





$$(\texttt{this}_{\text{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\text{get}}) \Rightarrow (\texttt{ob} \xrightarrow{\text{Transfer}} r_{\text{get}})$$

Figure 4. The left-hand side shows the graph $\widetilde{G}$ computed by the static analysis with the library code, and the right-hand side shows $\widetilde{G}$ computed with path specifications (the relevant path specification is shown below the graph). The solid edges are the graph G extracted from the program `test` shown in Figure 1. In addition, the dashed edges are a few of the edges in $\widetilde{G}$ when computing the transitive closure. We omit backward edges (i.e., with labels $\overline{A}$) for clarity. Vertices and edges corresponding to library code are highlighted in red.

Transitive closure. Second, our static analysis computes the *transitive closure* $\widetilde{G}$ of G according to the context-free grammar C_{pt} in Figure 3. A *path* $y \overset{\alpha}{\dashrightarrow} x$ in G is a sequence

$$x \xrightarrow{\sigma_1} v_1 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_k} y$$

of edges in G such that $\alpha = \sigma_1 \dots \sigma_k \in \Sigma_{\text{pt}}^*$. Then, $\widetilde{G}$ contains (i) each edge $x \xrightarrow{\sigma} y$ in the original graph G, and (ii) if there is a path $x \overset{\alpha}{\dashrightarrow} y$ in G such that $A \overset{*}{\Rightarrow} \alpha$ (where A is a nonterminal of C_{pt}), the edge $x \xrightarrow{A} y$. Our static analysis computes $\widetilde{G}$ using dynamic programming; e.g., see [27].

The first production in Figure 3 constructs the *transfer* relation $x \xrightarrow{\text{Transfer}} y$, which says that x may be "indirectly assigned" to y. The second production constructs the "backwards" transfer relation. The third production constructs the *alias* relation $x \xrightarrow{\text{Alias}} y$, which says that x may alias y. The fourth production computes the points-to relation, i.e., $x \hookrightarrow o$ whenever $o \xrightarrow{\text{FlowsTo}} x \in \widetilde{G}$.

4 Path Specifications

At a high level, a path specification encodes when edges in G would have been connected by (missing) edges from the library implementation. In particular, suppose that our static analysis could analyze the library implementation, and that while computing the transitive closure $\widetilde{G}$, it contains a path

$$z_1 \overset{\beta_1}{\dashrightarrow} w_1 \xrightarrow{A_1} z_2 \overset{\beta_2}{\dashrightarrow} \dots \xrightarrow{A_{k-1}} z_k \overset{\beta_k}{\dashrightarrow} w_k. \tag{4}$$

Here, $z_1, w_1, \dots, z_k, w_k \in \mathcal{V}_{\text{path}}$ where $\mathcal{V}_{\text{path}}$ is the set of variables in the library interface (i.e., parameters and return values of library functions), $\beta_1, \dots, \beta_k \in \Sigma_{\text{pt}}^*$ are labels for paths corresponding to library code, and $A_1, \dots, A_{k-1}$ are nonterminals in C_{pt} labeling relations computed so far. If

$$A \overset{*}{\Rightarrow} \beta_1 A_1 \dots \beta_{k-1} A_{k-1} \beta_k,$$

then our analysis adds $z_1 \xrightarrow{A} w_k$ to $\widetilde{G}$. Path specifications ensure that our analysis adds such edges to $\widetilde{G}$ when the library code is unavailable (so the paths $z_i \overset{\beta_i}{\dashrightarrow} w_i$ are missing from $\widetilde{G}$); e.g., a path specification for (4) says that if

$$w_1 \xrightarrow{A_1} z_2, \ \dots, \ w_{k-1} \xrightarrow{A_{k-1}} z_k \in \widetilde{G},$$

then the static analysis should add $z_1 \xrightarrow{A} w_k$ to $\widetilde{G}$.

For example, while analyzing `test` in Figure 1 with the library code on the right available, the analysis sees the path

$$\texttt{ob} \xrightarrow{\texttt{Store[f]}} \texttt{this}_{\text{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\text{get}} \xrightarrow{\texttt{Load[f]}} r_{\text{get}}.$$

Since Transfer $\overset{*}{\Rightarrow}$ `Store[f]` Transfer `Load[f]`, the analysis adds edge $\texttt{ob} \xrightarrow{\text{Transfer}} r_{\text{get}}$. As we describe below, the path specification s_{box} shown in (1) ensures that this edge is added to $\widetilde{G}$ when the library code is unavailable.

Syntax. Let $\mathcal{V}_{\text{prog}}$ be the variables in the program (i.e., excluding variables in the library), let $\mathcal{V}_m = \{p_m, r_m\}$ be the parameter and return value of library function $m \in \mathcal{M}$, and let $\mathcal{V}_{\text{path}} = \bigcup_{m \in \mathcal{M}} \mathcal{V}_m$ be the *visible variables* (i.e., variables at the library interface). A *path specification* is a sequence

$$z_1 w_1 z_2 w_2 \dots z_k w_k \in \mathcal{V}_{\text{path}}^*,$$

where $z_i, w_i \in \mathcal{V}_{m_i}$ for library function $m_i \in \mathcal{M}$. We require that w_i and z_{i+1} are not both return values, and that w_k is a return value. For clarity, we typically use the syntax

$$z_1 \dashrightarrow w_1 \rightarrow z_2 \dashrightarrow \dots \dashrightarrow w_{k-1} \rightarrow z_k \dashrightarrow w_k. \tag{5}$$

Semantics. Given path specification (5), for each $i \in [k]$, define the nonterminal A_i in the grammar C_{pt} to be

$$A_i = \begin{cases} \text{Transfer} & \text{if } w_i = r_{m_i} \text{ and } z_{i+1} = p_{m_{i+1}} \\ \text{Alias} & \text{if } w_i = p_{m_i} \text{ and } z_{i+1} = p_{m_{i+1}} \\ \overline{\text{Transfer}} & \text{if } w_i = p_{m_i} \text{ and } z_{i+1} = r_{m_{i+1}}. \end{cases}$$

Candidate Code Fragment Specification	Candidate Path Specification(s)	Synthesized Unit Test	

`class Box { // candidate specification`
`  Object f; // ghost field`
`  void set(Object ob) { f = ob; }`
`  Object get() { return f; } }`
	$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} \to \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}$	`boolean test() {`
		`  Object in = new Object(); // o_in`
		`  Box box = new Box(); // o_box`
		`  box.set(in);`
		`  Object out = box.get();`
		`  return in == out; }`
			✓

`class Box { // candidate specification`
`  Object f; // ghost field`
`  void set(Object ob) { f = ob; }`
`  Object clone() { return f; } }`
	$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} \to \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}}$	`boolean test() {`
		`  Object in = new Object(); // o_in`
		`  Box box = new Box(); // o_box`
		`  box.set(in);`
		`  Object out = box.clone();`
		`  return in == out; }`
			✗

`class Box { // candidate specification`
`  Object f; // ghost field`
`  void set(Object ob) { f = ob; }`
`  Object get() { return f; }`
`  Box clone() {`
`    Box b = new Box(); // ~o_clone`
`    b.f = f;`
`    return b; } }`
	$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} \left(\to \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}}\right)^* \to \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}$	`boolean test0() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Object out = box0.get();`
		`  return in == out; }`
		`boolean test1() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Object out = box1.get();`
		`  return in == out; }`
		`boolean test2() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Box box2 = box1.clone();`
		`  Object out = box2.get();`
		`  return in == out; }`
		`...`
			✓

Figure 5. Examples of hypothesized library implementations (left column), an equivalent set of path specifications (middle column), and the synthesized unit tests to check the precision of these specifications (right column), with a check mark ✓ (indicating that the unit tests pass) or a cross mark ✗ (indicating that the unit tests fail).

Also, define the nonterminal A by

$$A = \begin{cases} \text{Transfer} & \text{if } z_1 = p_{m_1} \\ \text{Alias} & \text{if } z_1 = r_{m_1}. \end{cases}$$

Then, the path specification corresponds to adding a rule

$$\left(\bigwedge_{i=1}^{k-1} w_i \xrightarrow{A_i} z_{i+1} \in \widetilde{G} \right) \Rightarrow (z_1 \xrightarrow{A} w_k \in \widetilde{G})$$

to the static points-to analysis. The rule also adds the backwards edge $w_k \xrightarrow{\overline{A}} z_1$ to $\widetilde{G}$, but we omit it for clarity. We refer to the antecedent of this rule as the *premise* of the path specification, and the consequent of this rule as the *conclusion* of the path specification. Continuing our example, the path specification s_{box} shown in (1) has semantics

$$(\texttt{this}_{\texttt{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}}) \Rightarrow (\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}}).$$

This rule says that if the static analysis computes the edge $\texttt{this}_{\texttt{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}} \in \widetilde{G}$, then it must add $\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}}$ to $\widetilde{G}$. For example, this rule is applied in Figure 4 (right) to compute $\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}}$.

The middle column of Figure 5 shows examples of path specifications, and the first column shows equivalent code fragment specifications (the last column is described below). The specifications on the first and last rows are precise, whereas the specification on the second row is imprecise.

Soundness and precision. Let $\widetilde{G}_*(P)$ denote the true set of relations for a program P (i.e., relations that hold dynamically for some execution of P); note that because the library code is omitted from analysis, we only include relations between program variables in $\widetilde{G}_*(P)$. Then, given path specifications S, let $\widetilde{G}(P, S)$ denote the points-to edges computed using S for P, let $\widetilde{G}_+(P, S) = \widetilde{G}(P, S) \setminus \widetilde{G}_*(P)$ be the false positives, and let $\widetilde{G}_-(P, S) = \widetilde{G}_*(P) \setminus \widetilde{G}(P, S)$ be the false negatives.

Specification set S is *sound* if $\widetilde{G}_-(P, S) = \varnothing$. We say S and S' are *equivalent* (written $S \equiv S'$) if for every program P, $\widetilde{G}(P, S) = \widetilde{G}(P, S')$. Finally, S is *precise* if for every sound

S', $S' \cup S \equiv S'$. In other words, using S computes no false positive points-to edges compared to any sound set S'. We say a single path specification s is precise if $\{s\}$ is precise. We have the following result, which follows by induction:

Theorem 4.1. For any set S of path specifications, if each $s \in S$ is precise, then S is precise.

Witnesses. Our algorithm needs to synthesize unit tests that check whether a candidate path specification s is precise. In particular, a program P is a *potential witness* for a candidate path specification s if P returns true only if s is precise. If P is a potential witness for s, and upon execution, P in fact returns true, then we say P is a *witness* for s. The last column of Figure 5 shows potential witnesses for the candidate path specification in the middle column; a green check indicates the potential witness returns true (we say P *passes*), and a red check indicates it returns false or raises an exception (we say P *fails*). In particular, the synthesized unit test correctly rejects the imprecise specification on the second row.

Note that P may return false even if s is precise. This property is inevitable since executions are underapproximations; we show empirically that if s is precise, then typically the potential witness synthesized by our algorithm passes.

Soundly and precisely modeling library code. It is not obvious that path specifications are sufficiently expressive to precisely model library code. In this section, we show that path specifications are in fact sufficiently expressive to do so in the case of Andersen's analysis (and its cloning-based context- and object-sensitive extensions). More precisely, for any implementation of the library, there exists a (possibly infinite) set of path specifications such that the points-to sets computed using path specifications are both sound and at least as precise as analyzing the library implementation:

Theorem 4.2. Let $\widetilde{G}(P)$ be the set of points-to edges computed for program P assuming the library code is available. Then, there exists a set S of path specifications such that for every program P, $\widetilde{G}(P, S)$ is sound and $\widetilde{G}(P, S) \subseteq \widetilde{G}(P)$.

We give a proof in Appendix C. Note that the set S of path specifications may be infinite. This infinite blowup is unavoidable since we want the ability to test the precision of an individual path specification. In particular, the library implementation may exhibit effects that require infinitely many unit tests to check precision (e.g., the path specifications shown on the third row of Figure 5).

Regular sets of path specifications. Since the library implementation may correspond to an infinite set of path specifications, we need a mechanism for describing such sets. In particular, since a path specification is a sequence $s \in \mathcal{V}_{\text{path}}^*$, we can think of a set S of path specifications as a formal language $S \subseteq \mathcal{V}_{\text{path}}^*$ over the alphabet $\mathcal{V}_{\text{path}}$. Then, we can express an infinite set of path specifications using standard representations such as regular expressions or context-free grammars.

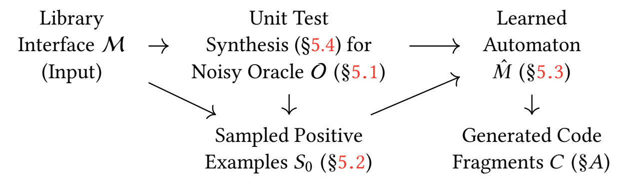


Figure 6. An overview of our specification inference system. The section describing each component is in parentheses.

We make the empirical observation that the library implementation is equivalent to a regular set S_* of path specifications (i.e., S_* is sound and precise). There is no particular reason that this fact should be true, but it holds for all the Java library functions we have examined so far. For example, consider the code fragment shown in the first column of the third row of Figure 5. This specification corresonds to the set of path specifications shown as a regular expression in the middle column of the same line (tokens in the regular expression are highlighted in blue for clarity).

Static points-to analysis with path specifications. To perform static points-to analysis with regular sets of path specifications, we convert the path specifications into equivalent code fragments and then analyze the client along with the code fragments; see Appendix A for details.

5 Specification Inference Algorithm

We describe our algorithm for inferring path specifications. Our system is summarized in Figure 6, which also shows the section where each component is described in detail.

5.1 Overview

Let the *target language* $S_* \subseteq \mathcal{V}_{\text{path}}^*$ be the set of all path specifications that are precise. By Theorem 4.1, S_* is precise. The goal of our algorithm is to infer a set of path specifications that approximates S_* as closely as possible.

Inputs. Recall that our algorithm is given two inputs: (i) the library interface, and (ii) blackbox access to the library functions. We use these two inputs to construct the noisy oracle and positive examples as describe below.

Noisy oracle. Given a path specification s, the *noisy oracle* $O : \mathcal{V}_{\text{path}}^* \to \{0, 1\}$ (i) always returns 0 if s is imprecise, and (ii) ideally returns 1 if s is precise (but may return 0). [3] This oracle is implemented by synthesizing a potential witness P for s and returning the result of executing P. We describe how we synthesize a witness for s in Section 5.4.

[3] While our implementation of the oracle is deterministic, our specification inference algorithm can also make use of a stochastic oracle (as long as it satisfies these two properties).

Positive examples. Phase one of our algorithm constructs a set of positive examples: our algorithm randomly samples candidate path specifications $s \sim \mathcal{V}_{\text{path}}^*$, and then uses $\mathcal{O}$ to determine whether each s is precise. More precisely, given a set $S = \{s \sim \mathcal{V}_{\text{path}}^*\}$ of random samples, it constructs positive examples $S_0 = \{s \in S \mid \mathcal{O}(s) = 1\}$. We describe how we sample $s \sim \mathcal{V}_{\text{path}}^*$ in Section 5.2.

Language inference problem. Phase two of our algorithm inductively generalizes S_0 to a regular set of path specifications. We formulate this problem as follows:

Definition 5.1. The *language inference problem* is to, given the noisy oracle $\mathcal{O}$ and the positive examples $S_0 \subseteq S_*$, infer a language $\hat{S}$ that approximates S_* as closely as possible.

The approximation quality of $\hat{S}$ compared to S_* must take into account both the false positive rate and the false negative rate. Intuitively, we prioritize minimizing false positives over minimizing false negatives—i.e., we aim to maximize the size of $\hat{S}$ subject to $\hat{S} \subseteq S_*$; however, $\hat{S}$ and S_* may be infinitely large. In our evaluation, we use a heuristic to measure approximation quality; see Section 6 for details.

In Section 5.3, we describe our algorithm for solving the language inference problem. It outputs a regular language $\hat{S} = \mathcal{L}(\hat{M})$, where $\hat{M}$ is a finite state automaton—e.g., given

$$S_0 = \{\texttt{ob}\ \texttt{this}_{\texttt{set}}\ \texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}}\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}\},$$

our language inference algorithm returns an automaton encoding the regular language

$$\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^*\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}.$$

5.2 Sampling Positive Examples

We sample a path specification $s \in \mathcal{V}_{\text{path}}^*$ by building it one variable at a time, starting from $s = \epsilon$ (where ϵ denotes the empty string). At each step, we ensure that s satisfies the path specification constraints, i.e., (i) z_i and w_i are parameters or return values of the same library function, (ii) w_i and z_{i+1} are not both return values, and (iii) the last variable w_k is a return value. In particular, given current sequence s, the set $\mathcal{T}(s) \subseteq \mathcal{V}_{\text{path}} \cup \{\phi\}$ of choices for the next variable (where ϕ indicates to terminate and return s) is:

- If $s = z_1 w_1 z_2 ... z_i$, then the choices for w_i are $\mathcal{T}(s) = \{p_m, r_m\}$, where $z_i \in \{p_m, r_m\}$.
- If $s = z_1 w_1 z_2 ... z_i w_i$, and w_i is a parameter, then the choices for z_{i+1} are $\mathcal{T}(s) = \mathcal{V}_{\text{path}}$.
- If $s = z_1 w_1 z_2 ... z_i w_i$, and w_i is a return value, then the choices for z_{i+1} are

$$\mathcal{T}(s) = \{z \in \mathcal{V}_{\text{path}} \mid z \text{ is a parameter}\} \cup \{\phi\}.$$

At each step, our algorithm samples $x \sim \mathcal{T}(s)$, and either constructs $s' = sx$ and continues if $x \neq \phi$ or returns s if $x = \phi$. We consider two sampling strategies.

Random sampling. We choose $x \sim \mathcal{T}(s)$ uniformly at random at every step.

Monte Carlo tree search. We can exploit the fact that certain choices $x \in \mathcal{T}(s)$ are much more likely to yield a precise path specification than others. To do so, note that our search space is structured as a tree. Each edge in this tree is labeled with a symbol $x \in \mathcal{V}_{\text{path}} \cup \{\phi\}$; then, we can associate each node N in the tree with the sequence $s_N \in (\mathcal{V}_{\text{path}} \cup \{\phi\})^*$ obtained by traversing the tree from the root to N and collecting the labels along the edges (if N is the root node, then $s_N = \epsilon$). Given an internal node N with corresponding sequence s_N, its children are determined by $\mathcal{T}$ as follows:

$$\{N \xrightarrow{x} N' \mid x \in \mathcal{T}(s_N)\}.$$

Therefore, a leaf node L corresponds to a sequence of the form $s_L = x_1 ... x_k \phi$, which in turn corresponds to a candidate path specification $s = x_1 ... x_n$. Thus, we can sample $x \sim \mathcal{T}(s)$ using Monte Carlo tree search (MCTS) [21], a search algorithm that learns over time which choices are more likely to succeed. In particular, MCTS keeps track of a score $Q(N, x)$ for every visited node N and every $x \in \mathcal{T}(s_N)$. Then, the choices are sampled according to the distribution

$$\Pr[x \mid N] = \frac{1}{Z} e^{Q(N,x)} \quad \text{where} \quad Z = \sum_{x' \in \mathcal{T}(s_N)} e^{Q(N,x')}.$$

Whenever a candidate $s = x_1 ... x_k$ is found, we increase the score $Q(x_1 ... x_i, x_{i+1})$ (for each $0 \leq i < k$) if s is a positive example ($\mathcal{O}(s) = 1$) and decrease it otherwise ($\mathcal{O}(s) = 0$):

$$Q(x_1 ... x_i, x_{i+1}) \leftarrow (1 - \alpha) Q(x_1 ... x_i, x_{i+1}) + \alpha \mathcal{O}(s).$$

We choose the *learning rate* α to be $\alpha = 1/2$.

5.3 Language Inference Algorithm

Our language inference algorithm is based on RPNI [31]. [4] In particular, we modify RPNI to leverage access to the noisy oracle—whereas RPNI takes as input a set of negative examples, we use the oracle to generate them on-the-fly. Our algorithm learns a regular language $\hat{S} = \mathcal{L}(\hat{M})$ represented by the (nondeterministic) finite state automaton (FSA) $\hat{M} = (Q, \mathcal{V}_{\text{path}}, \delta, q_{\text{init}}, Q_{\text{fin}})$, where Q is the set of states, $\delta : Q \times \mathcal{V}_{\text{path}} \to 2^Q$ is the transition function, $q_{\text{init}} \in Q$ is the start state, and $Q_{\text{fin}} \subseteq Q$ are the accept states. If there is a single accept state, we denote it by q_{fin}. We denote transitions $q \in \delta(p, \sigma)$ by $p \xrightarrow{\sigma} q$.

Our algorithm initializes $\hat{M}$ to be the FSA representing the finite language S_0. In particular, it initializes $\hat{M}$ to be the prefix tree acceptor [31], which is the FSA where the underlying transition graph is the prefix tree of S_0, the start

[4] We also considered the L^* algorithm [6]; however, the L^* algorithm depends on an *equivalence oracle* that reports whether a candidate language is correct, which is unavailable in our setting. It is possible to approximate the equivalence oracle using sampling, but in our experience, this approximation is very poor and can introduce substantial imprecision.

state is the root of this prefix tree, and the accept states are the leaves of this prefix tree.

Then, our algorithm iteratively considers *merging* pairs of states of $\hat{M}$. More precisely, given two states $p, q \in Q$ (without loss of generality, assume $q \neq q_{\text{init}}$), $\text{Merge}(\hat{M}, q, p)$ is the FSA obtained by (i) replacing transitions

$$(r \xrightarrow{\sigma} q) \text{ becomes } (r \xrightarrow{\sigma} p), \qquad (q \xrightarrow{\sigma} r) \text{ becomes } (p \xrightarrow{\sigma} r),$$

(ii) adding p to Q_{fin} if $q \in Q_{\text{fin}}$, and (iii) removing q from Q.

Our algorithm iterates once over all the states Q; we describe how a single iteration proceeds. Let q be the state being processed in the current step, let Q_0 be the states that have been processed so far but not removed from Q, and let $\hat{M}$ be the current FSA. For each $p \in Q_0$, our algorithm checks whether merging q and p adds imprecise path specifications to the language $\mathcal{L}(\hat{M})$; if not, it greedily performs the merge. More precisely, for each $p \in Q_0$, our algorithm constructs

$$M_{\text{diff}} = \text{Merge}(\hat{M}, q, p) \setminus \hat{M},$$

which represents the set of strings that are added to $\mathcal{L}(\hat{M})$ if q and p are merged. Then, for each $s \in M_{\text{diff}}$ up to some maximum length N (we take $N = 8$), our algorithm queries $\mathcal{O}(s)$. If all queries pass (i.e., $\mathcal{O}(s) = 1$), then our algorithm greedily accepts the merge, i.e., $\hat{M} \leftarrow \text{Merge}(\hat{M}, q, p)$ and continues to the next $q \in Q$. Otherwise, it considers merging q with the next $p \in Q_0$. Finally, if q is not merged with any state $p \in Q_0$, then our algorithm does not modify $\hat{M}$ and adds q to Q_0. Once it has completed a pass over all states in Q, our algorithm returns $\hat{M}$. For example, suppose our language learning algorithm is given a single positive example

$$\texttt{ob}\ \texttt{this}_{\texttt{set}}\ \texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}}\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}.$$

Then, our algorithm constructs the finite state automaton

$$q_{\text{init}} \xrightarrow{\texttt{ob}} q_1 \xrightarrow{\texttt{this}_{\texttt{set}}} q_2 \xrightarrow{\texttt{this}_{\texttt{clone}}} q_3 \xrightarrow{r_{\texttt{clone}}} q_4 \xrightarrow{\texttt{this}_{\texttt{get}}} q_5 \xrightarrow{r_{\texttt{get}}} q_{\text{fin}}.$$

Our algorithm fails to merge q_{init}, q_1, q_2, or q_3 with any previous states. It then tries to merge q_4 with each state $\{q_{\text{init}}, q_1, q_2, q_3\}$; the first two merges fail, but merging q_4 with q_2 produces

$$\begin{tikzcd} q_{\text{init}} \arrow[r, "\texttt{ob}"] & q_1 \arrow[r, "\texttt{this}_{\texttt{set}}"] & q_2 \arrow[r, "\texttt{this}_{\texttt{get}}"] \arrow[d, bend right, "\texttt{this}_{\texttt{clone}}"'] & q_4 \arrow[r, "r_{\texttt{get}}"] & q_{\text{fin}}. \\ & & q_3 \arrow[u, bend right, "r_{\texttt{clone}}"'] & & \end{tikzcd}$$

Then, the specifications of length at most N in M_{diff} are

$$\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^0\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}$$

$$\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^1\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}$$

$$\ldots$$

$$\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^N\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}},$$

all of which are accepted by $\mathcal{O}$. Therefore, our algorithm greedily accepts this merge and continues. The remaining merges fail and our algorithm returns this automaton.

input	$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} \to \texttt{this}_{\texttt{clone}} \to r_{\texttt{clone}}$ $\to \texttt{this}_{\texttt{get}} \to r_{\texttt{get}}$

skeleton	`??.set(??);`
	 `?? = ??.clone();`
	 `?? = ??.get();`

fill holes	`box.set(in);`
	 `Box boxClone = box.clone();`
	 `Object out = boxClone.get();`

initialization & scheduling	`Object in = new Object();`
	 `Box box = new Box()`
	 `box.set(in);`
	 `Box boxClone = box.clone();`
	 `Object out = boxClone.get();`
	 `return in == out;`

Figure 7. Steps in the witness synthesis algorithm for a candidate path specification for Box. Code added at each step is highlighted in blue. Scheduling is shown in the same line as initialization.

5.4 Unit Test Synthesis

We describe how we synthesize a unit test that is a potential witness for a given specification

$$s = (z_1 \dashrightarrow w_1 \to \ldots \to z_k \dashrightarrow w_k),$$

relegating details to Appendix B. Figure 7 shows how the unit test synthesis algorithm synthesizes a unit test for the candidate specification s_{box}.

Recall that the semantics of s are $\left(\bigwedge_{i=1}^{k-1} w_i \xrightarrow{A_i} z_{i+1} \in \widetilde{G}\right) \Rightarrow (z_1 \xrightarrow{A} w_k \in \widetilde{G})$. Then, consider a program P that satisfies the following properties:

- The conclusion of s does not hold statically for P with empty specifications, i.e., $z_1 \xrightarrow{A} w_k \notin \widetilde{G}(P, \varnothing)$.
- The premise of s holds for P, i.e., $w_i \xrightarrow{A_i} z_{i+1} \in \widetilde{G}(P, \{s\})$ for each $i \in [k-1]$.
- For every set S of path specifications, if $z_1 \xrightarrow{A} w_k \in \widetilde{G}(P, S)$, then $S \cup \{s\}$ is equivalent to S.

Intuitively, if program P is a potential witness for path specification s with premise ψ and conclusion $\phi = (e \in \widetilde{G})$, then s is the only path specification that can be used by the static analysis to compute relation e for P. Thus, if P witnesses s, then s is guaranteed to be precise. In particular, we have the following important guarantee for the unit test synthesis algorithm (see Appendix E for a proof):

Theorem 5.2. The unit test P synthesized for path specification s is a potential witness for s.

Skeleton construction. Our algorithm first constructs the *skeleton* of the unit test. In particular, a witness for s must include a call to each function $m_1, \ldots, m_k$, where the variables

$z_i, w_i \in \mathcal{V}_{m_i}$ are parameters or return values of m_i, since the graph G extracted from the unit test must by definition contain edges connecting the w_i to z_{i+1}. For each function call $y \leftarrow m(x)$, the argument x and the left-hand side variable y are left as holes ?? to be filled in subsequent steps.

Fill holes. Second, our algorithm fills the holes in the skeleton corresponding to reference variables. In particular, for each pair of function calls $??_{y,i} \leftarrow m_i(??_{x,i})$ and $??_{y,i+1} \leftarrow m_{i+1}(??_{x,i+1})$, it fills the holes $??_{y,i}$ and $??_{x,i+1}$ depending on the edge $w_i \xrightarrow{A_i} z_{i+1}$:

- **Case** A_i = **Transfer:** In this case, w_i is a return value and z_{i+1} is a parameter. Thus, the algorithm fills $??_{y,i}$ and $??_{x,i+1}$ with the same fresh variable x.
- **Case** A_i = $\overline{\textbf{Transfer}}$**:** This case is analogous to the case A_i = Transfer.
- **Case** A_i = **Alias:** In this case, w_i and z_{i+1} are both parameters. Thus, the algorithm fills $??_{y,i}$ and $??_{x,i+1}$ with the same fresh variable x, and additionally adds to the test an allocation statement $x \leftarrow X()$.

As shown in the proof of Theorem 5.2, the added statements ensure that P is a potential witness for s.

Initialization. Third, our algorithm initializes the remaining reference and primitive variables in the unit test. In particular, function calls $y \leftarrow m_i(x)$ may have additional parameters that need to be filled. For Theorem 5.2 to hold, the remaining reference variables must be initialized to `null`.

However, this approach is likely to synthesize unit tests that fail (by raising exceptions) even when s is precise. Therefore, we alternatively use a heuristic where we allocate a fresh variable for each reference variable. For allocating reference variables that are passed as arguments to constructors, we have to be careful to avoid infinite recursion; for example, a constructor `Integer(Integer i)` should be avoided. Our algorithm uses a shortest-path algorithm to generate the smallest possible initialization statements; see Appendix B.3 for details. With this approach, we can no longer guarantee that P is a witness and our oracle may be susceptible to false positives. In our evaluation, we show that this heuristic substantially improves recall with no reduction in precision.

Primitive variables can be initialized arbitrarily for Theorem 5.2 to hold, but this choice affects whether P is a witness when s is precise. We initialize primitive variables using default values (`0` for numeric variables and `true` for boolean variables) that work well in practice.

Scheduling. Fourth, our algorithm determines the ordering of the statements in the unit test. There are many possible choices of statement ordering, which affect whether P is a witness when s is precise. There are *hard constraints* on the ordering (in particular, a variable must be defined before it is used) and *soft constraints* (in particular, statements corresponding to edges $w_i \to z_{i+1}$ for smaller i should occur earlier in P). Our scheduling algorithm produces an ordering that satisfies the hard constraints while trying to satisfy as many soft constraints as possible. It uses a greedy strategy, i.e., it orders the statements sequentially from first to last, choosing at each step the statement that satisfies all the hard constraints and the most soft constraints; see Appendix B.4.

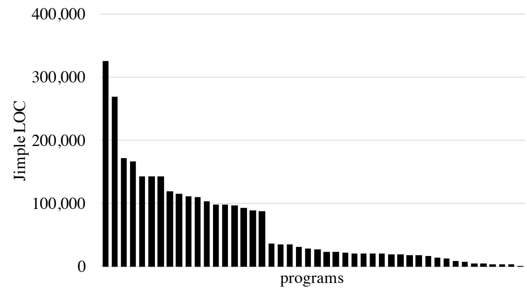


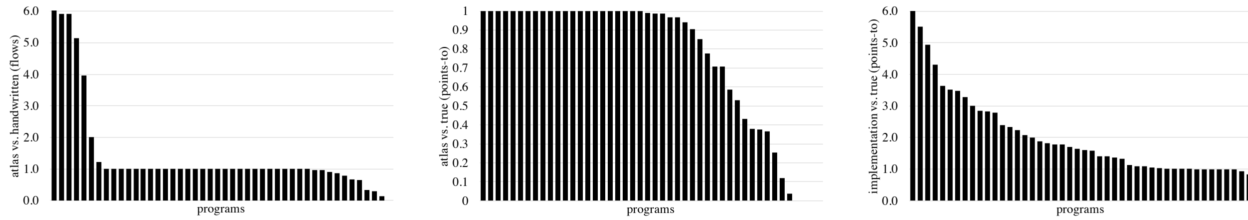
Figure 8. Jimple lines of code of the apps in our benchmark.

6 Evaluation

We implemented our specification inference algorithm in a tool called Atlas and evaluated its ability to infer points-to specifications for the Java standard library. First, we demonstrate the usefulness of the inferred specifications for a static information flow analysis, comparing to the existing, handwritten specifications, even though many of these specifications were written specifically for apps in our benchmark. Second, we evaluate the precision and recall of the inferred specifications by comparing to ground truth specifications; furthermore, we demonstrate the effectiveness of using specifications by showing that using ground truth specifications significantly decreases false positives compared to analyzing the actual library implementation. Finally, we analyze some of the choices we made when designing our algorithm.

Benchmark. We base our evaluation on a benchmark of 46 Android apps, including a benchmark of 26 malicious and benign apps given to us by a major security company. The remaining apps were obtained as part of a DARPA program on detecting Android malware. Overall, our benchmark contains a mix of utility apps (e.g., flashlights, note taking apps, battery monitors, wallpaper apps, etc.) and Android games. We show the sizes of these apps in Jimple lines of code (Jimple is the intermediate representation used by Soot) in Figure 8. The malware in this benchmark consist primarily of apps that leak sensitive user information, including location, contacts, phone number, and SMS messages.

Information flow client. We use Atlas to infer specifications for a client static information flow analysis for Android apps [7, 8, 16, 17]; in particular, it is based closely on [9]. This information flow client is specifically designed to find Android malware exhibiting malicious behaviors such as the



(a) (b) (c)

Figure 9. In (a), we show the ratio of nontrivial information flows discovered using Atlas versus existing specifications. We show the ratio of nontrivial points-to edges discovered using (b) Atlas versus ground truth and (c) ground truth versus implementation. The ratios are sorted from highest to lowest for the 46 benchmark programs with nontrivial points-to edges. Note that some values exceeded the graph scale; in (a), the largest value is 9.3, and in (c), the largest value is 11.7.

ones in our benchmark. The client is built on Chord [29] modified to use Soot [43] as a backend. It computes an Andersen-style, 1-object-sensitive, flow- and path-insensitive points-to analysis. It then computes an explicit information flow analysis [34] between manually annotated sources and sinks, using the computed points-to sets to resolve flows through the heap. Sources include location, contacts, and device information (e.g., the return value of `getLocation`), and sinks include the Internet and SMS messages (e.g., the `text` parameter of `sendTextMessage`).

Existing specifications. Our tool omits analyzing the Java standard library (version 7), and instead analyzes client code and user-provided code fragment specifications. Over the course of two years, we have handwritten several hundred code fragment specifications for our static information flow client for 90 classes in the Java standard library, including many written specifically for our benchmark of Android apps. In particular, these specifications were written as needed for the apps that we analyzed. The most time consuming aspect of developing the existing specifications was not writing them (they total a few thousand lines of code), but identifying which functions required specifications. Thus, they cover many fewer functions than the handwritten ground truth specifications described in Section 6.2, but are tailored to finding information flows for apps in our benchmark.

Evaluating inferred specifications. To measure precision and recall of our inferred specifications, we begin by converting inferred specifications $\hat{S}$ to code fragment specifications using the algorithm described in Appendix A. Next, recall our observation in Section 4 that the library implementation can be soundly and precisely represented by a regular set of path specifications S_*; we can similarly convert S_* to a set of code fragment specifications. Then, we count a code fragment specification in $\hat{S}$ as a false positive if it does not appear in S_*, and similarly count a code fragment specification in S_* as a false negative if it does not appear in $\hat{S}$. For code fragment specifications with multiple statements, we count each statement fractionally. For example, consider the sound and precise specification

```
class Box { // ground truth specification
  Object f; // ghost field
  Object set(Object ob) {
    f = ob;
    return ob;
    return f; }
```

for the `set` method in the `Box` class (note that the statement `return ob` is redundant, but it is generated by the algorithm described in Appendix A). Then, the specification

```
Object set(Object ob) { // inferred specification
  return ob; }
```

is missing two statements, so we count it as 2/3 of a false negative. Each edge $z_i \dashrightarrow w_i$ in a path specification roughly corresponds to a single statement in the generated code fragments, so this heuristic intuitively counts false negative and false positive path specifications weighted by their length.

Evaluating computed relations. To compare how using two different sets of specifications impacts the static analysis, we examine the ratio of the sizes of the computed relations. We preprocess these sizes in two ways: (i) we only consider relations between program variables, and (ii) we ignore *trivial* relations that can be computed even when using empty specifications (i.e., all library functions are treated as no-ops with respect to heap effects). For example, for points-to edges, we use the metric $R_{\text{pt}}(S, S') = \frac{|\Pi(S) \setminus \Pi(\varnothing)|}{|\Pi(S') \setminus \Pi(\varnothing)|}$, where $\Pi(S) \subseteq \mathcal{V} \times O$ are the points-to edges computed using specifications S, and $\Pi(\varnothing) \subseteq \mathcal{V} \times O$ are the trivial edges.

6.1 Comparison to Our Existing Specifications

We compare the quality of the inferred specifications to our existing, handwritten specifications, in particular, aiming to improve our information flow client. We focus on inferring specifications for the commonly used packages `java.lang`, `java.util`, `java.io`, `java.nio`, `java.net`, and `java.math`; all the specifications we have manually written for the Java standard library are for classes in these packages. We note that the handwritten specifications for these packages are high-quality—we manually examined all of these handwritten specifications, and found that they were precise (i.e., there were no false positives compared to S_*).

Inferred specifications. We used a total of 12 million random samples for phase one, which ran in 44.9 minutes. Phase two of our algorithm ran in 31.0 minutes; the initial FSA had 10,969 states, and the final FSA had 6,855 states. We compare our inferred specifications to the handwritten ones, i.e., measuring precision and recall compared to the handwritten specifications rather than to S_*. Most strikingly, Atlas infers 5× as many specifications as the existing, handwritten ones—Atlas infers specifications for 878 library functions, whereas handwritten specifications cover only 159 library functions. Furthermore, Atlas infers 89% of the handwritten specifications. We manually examined the 5 false negatives (i.e., handwritten specifications that Atlas fails to infer) for the Collections API. Each one is due to a false negative in the unit test synthesis. For example, the function `subList(int,int)` in the `List` class requires a call of the form `subList(0, 1)` to retrieve the first object in the list. Similarly, the function `set(int,Object)` in the `List` class requires an object to already be in the list or it raises an index out-of-bounds exception. The potential witnesses synthesized by Atlas fail to exercise the relevant behaviors in these instances. Finally, we manually examined more than 200 of the inferred specifications that were new; all of them were precise, which is strong evidence that our tool has very few if any false positives despite the heuristics we employ.

Information flows. To show how Atlas can improve upon our existing handwritten specifications, we study the ratio $R_{\text{flow}}(S_{\text{atlas}}, S_{\text{hand}})$ of information flows computed using Atlas versus using the existing, handwritten specifications. A higher ratio ($R_{\text{flow}} > 1$) says that Atlas has higher recall, and a lower ratio ($R_{\text{flow}} < 1$) says that handwritten specifications have higher recall. Figure 9 (a) shows $R(S_{\text{atlas}}, S_{\text{hand}})$. Overall, Atlas finds 52% more information flows compared to the handwritten specifications. The size of this gap is noteworthy because we have already analyzed these apps over the past few years—many of the existing specifications were written specifically for this benchmark.

Finally, for the subset of apps in our benchmark given to us by a major security company (27 of the 46) and for a subset of information sources and sinks, the security company provided us with ground truth information flows that they considered to be malicious. In particular, they consider 86.5% of the information flows newly identified using the inferred specifications to be actual malicious behaviors.

6.2 Comparison to Ground Truth

Since the existing handwritten specifications are incomplete, we additionally compare to the ground truth specifications S_* to evaluate the precision and recall of our inferred specifications. Because of the manual effort required to write ground truth specifications, we do so only for the 12 classes in the Java Collections API that are most frequently used by our benchmark (98.5% of calls to the Collections API target these 12 classes). We focus on the Java Collections API (i.e., classes that implement the `Collection` or `Map` interfaces), because it requires by far the most complex points-to specifications.

Inferred specifications. We examine the top 50 most frequently called functions in our benchmark (in total, accounting for 95% of the function calls). The recall of our algorithm is 97% (i.e., we inferred the ground truth specification for 97% of the 50 functions) and the precision is 100% (i.e., each specification is as precise as the ground truth specification). The false negatives occured for the same reasons as the false negatives discussed in the previous section.

Points-to sets. To show the quality of the specifications inferred by Atlas, we study the ratio $R_{\text{pt}}(S_{\text{atlas}}, S_*)$ of using specifications inferred by Atlas to using ground truth specifications. We found that using Atlas does not compute a single false positive points-to edge compared to using ground truth specifications, i.e., the precision of Atlas is 100%. Thus, $1 - R_{\text{pt}}(S_{\text{atlas}}, S_*)$ is the rate of false negative points-to edges when using Atlas. Figure 9 (b) shows $R_{\text{pt}}(S_{\text{atlas}}, S_*)$ for each app in our benchmark, sorted by magnitude. This ratio is 1.0 for almost half of the programs, i.e., for almost half the programs, there are no false negatives. The median recall is 99.0%, and the average recall is 75.8%.

Benefits of using specifications. We show that using specifications can greatly improve the precision and soundness of a static analysis. In particular, we compare the ground truth specifications to the library implementation, i.e., the class files comprising the actual implementation of the Collections API (developed by Oracle). We compute the ratio $R_{\text{pt}}(S_{\text{impl}}, S_*)$ of analyzing the library implementation S_{impl} to analyzing the ground truth specifications S_*. This ratio measures the number of false positives due to analyzing the library implementation instead of using ground truth specifications, since every points-to edge computed using the implementation but not the ground truth specifications is a false positive. Figure 9 (c) shows this ratio $R_{\text{pt}}(S_{\text{impl}}, S_*)$. For a third of programs, the false positive rate is more than 100% (i.e., when $R_{\text{pt}} \geq 2$), and for four programs, the false positive rate is more than 300% (i.e., $R_{\text{pt}} \geq 4$). The average false positive rate is 115.2%, and the median is 62.1%. Furthermore, for two of the programs, there are false negatives (i.e., $R_{\text{pt}} < 1$) due to unanalyzable calls to native code.

6.3 Design Choices

Finally, we compare the performance of different design choices for our specification inference algorithm; in particular, we infer specifications for 733 library functions in the Java Collections API using different design choices.

```
class StrangeBox { // library
  Object f;
  void set(Object ob) {
    f = ob;
    f = null; }
  Object get() { return f; } }
```

```
boolean test() { // program
  Object in = new Object(); // o_in
  Box box = new Box(); // o_box
  Object out;
  new Thread(() -> box.set(in)).execute();
  new Thread(() -> out = box.get()).execute();
  return in == out; }
```

Figure 10. Implementation of the library methods `set`, `get`, and `clone` in the `StrangeBox` class (left), and an example of a concurrent program using these functions (right).

Positive examples: random sampling vs. MCTS. We sampled 2 million candidate path specifications using each algorithm. Random sampling found 3,124 positive examples, whereas MCTS found 10,153.

Object initialization: null vs. instantiation. Each of the 11,613 positive examples passed the unit test constructed using instantiation, but only 7,721 passed when using null initialization, i.e., instantiation finds 50% more specifications. As discussed above, even with this heuristic, we estimate that the false positive rate of our tool is zero.

6.4 Discussion

As we have shown, using ground truth substantially improves precision and soundness compared to analyzing the library implementation. For the one-time cost of writing specifications, we can eliminate imprecision due to deep call hierarchies and unsoundness due to native code, reflection, etc. Such an approach is already used in production static analysis systems to handle hard-to-analyze code [14]. However, manually writing specifications that are complete or close to complete is impractical—ground truth specifications for just 12 classes took more than a week to write and contain 1,731 lines of code, but there are more than 4,000 classes in the Java standard library. Typically, manual effort is focused on writing specifications for the most commonly used functions, but this approach leaves a long tail of missing specifications [9]. Moreover, handwritten specifications can be error prone [18], and libraries grow over time—the Java 9 standard library contains more than 2,000 new classes—so specifications must be maintained over time.

We have shown that Atlas automatically covers an order of magnitude more of the Java Collections API compared to the existing, handwritten specifications, and is furthermore very close to ground truth. In addition, we have shown that Atlas can substantially improve recall on an information flow client compared to the handwritten specifications, despite the fact that we have already written specifications specifically for apps in this benchmark. Thus, Atlas substantially improves the practicality of using specifications to model hard-to-analyze code in static analysis. We believe that the remaining gap can be bridged by the human analyst, e.g., by manually inspecting important specifications or by using interactive specification inference [9, 48].

7 Discussion

Sources of imprecision. We briefly summarize the two potential sources of imprecision in our specification inference algorithm. First, in the second phase of our algorithm, imprecision can be introduced when we merge FSA states. In particular, such a merge can add infinitely many new path specifications to our set of inferred specifications; we only check precision for specifications up to a bounded length. Thus, newly added specifications that are longer than this bound may be imprecise. In our evaluation, we observe that all the added specifications are precise. Second, in both phases of our analysis, the unit test synthesis algorithm uses a heuristic where it initializes all parameters to non-null values. Thus, the unit test synthesized for a given specification may pass even if the specification is imprecise. In our evaluation, we show that this approximation helps us find many more specifications without introducing any imprecision.

Sources of unsoundness. We briefly summarize the potential sources of unsoundness in our specification inference algorithm, i.e., why it may miss correct specifications. At a high level, there are three sources: (i) the restriction to regular languages, (ii) incompleteness in the search (both in the random samples in phase one, and in the language inference algorithm), and (iii) shortcomings in the unit test synthesis algorithm, i.e., the unit test synthesized for a given specification fails even though the specification is precise.

In our evaluation, we found that unsoundness was entirely due to reason (iii). We believe there are three reasons why a synthesized unit test may fail to pass: (a) heuristics used for scheduling, (b) heuristics used to initialize variables, and (c) concurrency. First, in the scheduling step, there are multiple possible schedules of the statements in the unit test, and our synthesis algorithm uses heuristics to choose a good one (i.e., one that is likely to be a witness). Second, in the initialization step, there are multiple possible ways to initialize variables in the unit test; again, our algorithm uses heuristics to choose a good one. As described in Section 6, in our evaluation, all incorrectly rejected specifications were due to this second reason (e.g., this reason explains our algorithm failed to infer the specifications for `subList` and `set`).

The third possible reason is due to concurrency. So far, we have implicitly assumed that code is executed sequentially. In principle, because we are using a flow-insensitive points-to analysis, path specifications are sound with respect to concurrency. However, it may be possible that a unit test must include concurrent code to avoid incorrectly rejecting a specification. For example, consider the specifications for the class `StrangeBox` and the test program using this class shown in Figure 10. Because our static analysis is flow insensitive, it soundly determines that the argument of `set` may be aliased with the return value of `get`. However, for any sequential client code, the statement `f = ob` in the `set` method has no effects; thus, such code can never observe that `ob` and $r_{\texttt{get}}$ may be aliased. In particular, the unit test synthesized by our algorithm for the candidate path specification

$$s_{\text{strange}} = \texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} \rightarrow \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}.$$

would fail (thereby rejecting s_{strange}), even though s_{strange} is precise. Still, a witness exists, i.e., the unit test shown in Figure 10, which executes `set` and `get` concurrently.

Potential clients. We believe that the points-to summaries inferred by our algorithm may be useful for a number of clients beyond information flow analysis. In particular, our approach is applicable to any static analysis where soundness is highly desirable, but where state-of-the-art tools nevertheless sacrifice soundness in favor of reducing false positives. This criterion is true for nearly all static analyses designed to find bugs and security vulnerabilities, which frequently depend heavily on points-to analysis, information flow analysis, and taint analysis. Even for the security-critical task of finding malware, existing static analyses are typically unsound with respect to features such as reflection, dynamically loaded code, and native code [7, 16]. Using specifications not only improves precision, but also reduces unsoundness due to the use of these features in large libraries. If necessary, the human security analyst can always add specifications to improve the static analysis—for this use case, ATLAS substantially reduces human workload, e.g., in our evaluation, ATLAS inferred 92% of handwritten specifications for the Java Collections API. On the other hand, our approach is not suitable for static analyses where soundness is crucial, such as compiler optimizations.

8 Related Work

Inferring specifications for library code. Techniques have been proposed for mining specifications for library code from executions, e.g., taint specifications (i.e., whether taint flows from the argument to the return value) [13], functional specifications of library functions [19], specifications for x86 instructions [18], and specifications for callback control flow [20]. In contrast, points-to specifications that span multiple functions are more complex properties. One approach is to infer points-to specifications using data gathered from deployed instrumented apps [10]. In contrast, our algorithm actively synthesizes unit tests that exercise the library code and requires no instrumentation of deployed apps. Another approach is to interact with a human analyst to infer specifications [1, 9, 48]. These approach guarantee soundness, but still require substantial human effort, e.g., in [9], the analyst may need to write more than a dozen points-to specifications to analyze a single app. Finally, [2] uses a static approach to infer callgraph specifications for library code.

Inferring program properties. There has been work inferring program invariants from executions [30], including approaches using machine learning [35–37]. The most closely related work is [11], which uses an active learning strategy to infer program input grammars for blackbox code. In contrast, our goal is to infer points-to specifications for library code. There has also been work on specifications encoding desired properties of client programs (rather than encoding behaviors of the library code), both using dynamic analysis [8, 12, 24, 26, 32, 39] and using static analysis [4, 46].

Static points-to analysis. There is a large literature on static points-to analysis [5, 15, 28, 38, 45], including formulations based on set-constraints and context-free language reachability [22, 23, 33, 42]. Recent work has focused on improving context-sensitivity [25, 40, 41, 44, 47]. Using specifications in conjunction with these analyses can improve precision, scalability, and even soundness. One alternative is to use demand driven static analyses to avoid analyzing the entire library code [42]; however, these approaches are not designed to work with missing code, and furthermore do not provide much benefit for demanding clients that require analyzing a substantial fraction of the library code.

9 Conclusion

Specifications summarizing the points-to effects of library code can be used to increase precision, recall, and scalability of running a static points-to analysis on any client code. By automatically inferring such specifications, Atlas fully automatically achieves all of these benefits without the typical time-consuming and error-prone process of writing specifications by hand. We believe that Atlas is an important step towards improving the usability of static analysis.

Acknowledgments

This material is based on research sponsored by DARPA under agreement number FA84750-14-2-0006. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied of DARPA or the U.S. Government. This work was also supported by NSF grant CCF-1160904 and a Google Fellowship.

References

[1] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *POPL*.

[2] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *ECOOP*.

[3] Rajeev Alur, Pavol Černỳ, Parthasarathy Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. In *POPL*.

[4] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. In *POPL*.

[5] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. University of Cophenhagen.

[6] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* (1987).

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*.

[8] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Interactively verifying absence of explicit information flows in Android apps. In *OOPSLA*.

[9] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification inference using context-free language reachability. In *POPL*.

[10] Osbert Bastani, Lazaro Clapp, Saswat Anand, Rahul Sharma, and Alex Aiken. 2017. Eventually Sound Points-To Analysis with Missing Code. *arXiv preprint arXiv:1711.03436* (2017).

[11] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *PLDI*.

[12] Nels E Beckman and Aditya V Nori. 2011. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*.

[13] Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: mining explicit information flow specifications from concrete executions. In *ISSTA*.

[14] Facebook. 2017. Adding models. (2017). http://fbinfer.com/docs/adding-models.html

[15] Manuel Fähndrich, Jeffrey S Foster, Zhendong Su, and Alexander Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*.

[16] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE*.

[17] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. Scandroid: Automated security certification of android. (2009).

[18] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified synthesis: automatically learning the x86-64 instruction set. In *PLDI*.

[19] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing models for opaque code. In *FSE*.

[20] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S Foster, and Armando Solar-Lezama. 2016. Synthesizing framework models for symbolic execution. In *ICSE*.

[21] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *ECML*.

[22] John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. In *PLDI*.

[23] John Kodumal and Alexander Aiken. 2005. Banshee: A scalable constraint-based analysis toolkit. In *SAS*.

[24] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: Inferring the specification within. In *OSDI*.

[25] Percy Liang and Mayur Naik. 2011. Scaling abstraction refinement via pruning. In *PLDI*.

[26] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. In *PLDI*.

[27] David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *TCS* (2000).

[28] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*.

[29] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *PLDI*.

[30] Jeremy W Nimmer and Michael D Ernst. 2002. Automatic generation of program specifications. In *ISSTA*.

[31] José Oncina and Pedro García. 1992. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition* (1992).

[32] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *PLDI*.

[33] Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* (1998).

[34] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* (2003).

[35] Rahul Sharma and Alex Aiken. 2014. From invariant checking to invariant inference using randomized search. In *CAV*.

[36] Rahul Sharma, Aditya V Nori, and Alex Aiken. 2012. Interpolants as classifiers. In *CAV*.

[37] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *OOPSLA*.

[38] Olin Shivers. 1991. *Control-flow analysis of higher-order languages*. Ph.D. Dissertation. Citeseer.

[39] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *ISSTA*.

[40] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *PLDI*.

[41] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*.

[42] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *OOPSLA*.

[43] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java bytecode optimization framework. In *CASCON*.

[44] John Whaley and Monica Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*.

[45] Robert P Wilson and Monica S Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *PLDI*.

[46] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*.

[47] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *PLDI*.

[48] Haiyan Zhu, Thomas Dillig, and Isil Dillig. 2013. Automated inference of library specifications for source-sink property verification. In *APLAS*.