Specification Inference Using Context-Free Language Reachability

Osbert Bastani
Stanford University
obastani@cs.stanford.edu

Saswat Anand
Stanford University
saswat@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

We present a framework for computing context-free language reachability properties when parts of the program are missing. Our framework infers candidate specifications for missing program pieces that are needed for verifying a property of interest, and presents these specifications to a human auditor for validation. We have implemented this framework for a taint analysis of Android apps that relies on specifications for Android library methods. In an extensive experimental study on 179 apps, our tool performs verification with only a small number of queries to a human auditor.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords program analysis; verification; specification inference

1. Introduction

Interprocedural analyses are the building blocks for many applications of program analysis, including program slicing [23], bug finding [1, 6, 18], and taint analysis [32]. One limitation of most interprocedural analyses is that they assume the entire program's source code can be analyzed. In practice, many programs call methods in libraries that may be written in lower-level languages, use dynamic features such as Java reflection, or do not have source code available for analysis. In fact, in our experience, for large systems it is unusual if any of these situations does *not* occur many times. Handling missing or hard-to-analyze code in a fully automatic way generally results in using either very pessimistic and imprecise assumptions or very optimistic and unsound assumptions.

One pragmatic approach to address these problems is to manually write specifications encoding the relevant behavior of the library (or hard-to-analyze) methods so that the library source code does not need to be analyzed. For this approach to work, it is critical that (i) the manual effort of writing the specifications is small, and (ii) the analysis produces sound results even if some specifications are missing. Prior work has addressed these issues to some extent: [3, 4, 7, 15, 16, 21, 22, 25, 33] have applied data mining techniques to infer specifications, which helps address (i), and [34] has developed techniques for inferring minimal sets of specifications needed for program verification, which addresses both (i) and (ii).

In this paper we consider the problem of inferring missing specifications in the context of solving a general context-free language (CFL) reachability problem. Many important interprocedural analyses can be formulated as CFL reachability problems, and our framework makes these algorithms more practical and easier to apply to large programs where parts of the code are either missing or too difficult to analyze statically.

We instantiate our framework for taint analysis of Android apps, and perform an extensive experimental study on 179 apps. Many of these apps have hundreds of thousands of bytecode instructions and thousands of calls to Android library methods. Instead of analyzing the Android library methods, we infer two kinds of specifications that help capture their taint behavior:

- missing taint flows between parameters and return values of library methods, and
- missing alias relations introduced by library methods.

Our work has three main contributions. First, we develop a general framework for describing potentially missing specifications for CFL reachability analyses (Section 4). In particular, the framework can be instantiated to compute the two kinds of specifications above, among others. Second, we present a specification inference algorithm based on this framework (Section 5) that produces sound results, infers accurate specifications, scales to large applications, and allows a human auditor to refine the results. Third, we evaluate an implementation (Section 6) of our specification inference framework with the aforementioned experimental study in which our tool infers a large collection of library specifications and verifies whether or not apps have taint flows (Section 7).

2. Motivating Example

We begin with an example that motivates the need for specification inference to verify reachability properties of partial programs. The technical development begins in Section 3.

The popularity of the Android framework (which is installed on over 1 billion phones) has led to a proliferation of Android malware. These malicious apps exhibit behaviors including theft of contact information, sending SMS text messages to premium phone numbers, and unauthorized location tracking. Many of the malicious behaviors exhibited by current Android malware can be described as the flow of sensitive data to untrusted recipients, such as location data flowing to an untrusted web server, or an untrusted phone number flowing to an SMS send request. In principle, standard static taint analysis can identify such malware.

For example, suppose we want to look for malware that leaks location data via SMS messages. Consider the Java-like code in

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676977

Figure 1. In principle, a static analysis can track the data returned by `getLatitude`, and determine that it is passed to `sendSMS`. Unfortunately, the Android library (written in Java, with calls to native code) is a classic example of how system libraries cause difficulties for static analyses. Android library methods frequently use reflection and native methods, making it very difficult to construct a precise call graph or to perform a sound and precise context sensitive points-to analysis. Examples of such problems in practice include:

- `System.arraycopy` is a native method.
- `Bundle.putLong` indirectly calls the native Android library method `Parcel.nativeWriteLong` (bundles are used to pass data to and receive data from the Android system).
- `GeoPoint.getLatitudeE6` is part of a closed-source Google library, so the source code is unavailable.

While implementations of the Android library methods may be missing, the net taint flows through these methods are generally simple. For example, we can summarize the net taint flows through `List.add`, `List.get`, and `Double.toString` as follows: (i) $\texttt{arg}_{\texttt{add}}$ may be *aliased* with $\texttt{ret}_{\texttt{get}}$ (i.e., they may point to the same object), and (ii) if taint flows to $\texttt{this}_{\texttt{toString}}$, then taint also flows to $\texttt{ret}_{\texttt{toString}}$. Here, $\texttt{ret}_{\texttt{fn}}$ is the return value of `fn`, $\texttt{this}_{\texttt{fn}}$ is the receiver of `fn`, and $\texttt{argName}_{\texttt{fn}}$ is the parameter passed to `fn` named `argName`. The library specifications in Figure 2 capture these properties. Note that the code in Figure 2 is only a specification for the `List` and `Double` classes (i.e., it soundly captures the information flows and aliasing introduced by their methods), and is not a valid implementation of the classes. There are two kinds of specifications:

1. An *alias specification* summarizes the potential aliases caused by a library method. These are written as short methods that do nothing except introduce the desired aliasing. For example, the specification for `List.add` means that calling `List.add` may cause $\texttt{arg}_{\texttt{add}}$ and $\texttt{this}_{\texttt{add}}\texttt{.val}$ to be aliased. Also, the specification for `List.get` says that calling `List.get` may cause $\texttt{this}_{\texttt{get}}\texttt{.val}$ and $\texttt{ret}_{\texttt{get}}$ to be aliased. Note that if $\texttt{this}_{\texttt{arg}}$ and $\texttt{this}_{\texttt{get}}$ are aliased, then these specifications cause $\texttt{arg}_{\texttt{add}}$ and $\texttt{ret}_{\texttt{get}}$ to be aliased.
2. A *taint specification* describes how taint flows through a library method. These specifications are expressed as `@Flow` annotations, where the taint on the first argument (either a parameter `arg`, a return value `ret`, the receiver `this`, or a special label such as `LOC` representing a taint source) is transfered to the second argument (similar to the first argument, except in this case a special label would represent a sink, e.g. `SMS`). For example, the specification of `Double.toString` means that if $\texttt{this}_{\texttt{toString}}$ is tainted, then $\texttt{ret}_{\texttt{toString}}$ is also tainted.

The specifications in Figure 2 enable the analysis to find the flow from the source label `LOC` to the sink label `SMS` in Figure 1. First, the return value $\texttt{ret}_{\texttt{getLatitude}}$ is tainted with the source label `LOC`. Second, this taint is passed to `lat`, which is stored in `list.val`. Third, the value is retrieved from `list` and stored in `latAlias`, before being converted into a string and passed as the `text` argument of the Android library method `sendSMS(String text)`. Finally, our specification says that the `text` argument of the method `sendSMS` is sent to the sink labeled `SMS`, so the code exhibits a flow from `LOC` to `SMS`.

Naïvely, we may expect that flow specifications are sufficient to capture all information flows. For example, we may consider using taint specifications that handle field accesses, and replace the alias specification for `List.add` with the flow specification `@Flow(arg, this.val)`. However, this specification is unsound—it fails to capture the flow from `LOC` to `SMS` in Figure 3:

```
1. Double lat = getLatitude();
2. List list = new List();
3. list.add(lat);
4. Double latAlias = list.get(0);
5. String latStr = latAlias.toString();
6. sendSMS(latStr);
```

Figure 1. A flow through the List and Double classes.

```
1. class List:
2.   Object val;
3.   void add(Object arg) { val = arg; }
4.   Object get(Integer index) { return val; }
5. class Double:
6.   @Flow(this, return)
7.   String toString() {}
8. class LocationManager:
9.   @Flow(LOC, return)
10.  static String getLatitude() {}
11.class SMS:
12.  @Flow(text, SMS)
13.  static void sendSMS(String text) {}
```

Figure 2. Specifications for Android library classes.

```
1. class Box: String f;
2. List list = new List();
3. Box box = new Box();
4. list.add(box);
5. Box boxAlias = list.get(0);
6. boxAlias.f = getLatitude();
7. sendSMS(box.f);
```

Figure 3. An information flow not captured by taint specifications.

`boxAlias.f` is tainted by `LOC`, so `list.val.f` is tainted (since `boxAlias` and `list.val` are aliased), but the proposed specification for `List.add` does not transfer this taint to `box.f` (even though taint flows to `box.f`). In general, we need alias specifications to precisely and soundly capture flows due to aliasing.

Manually writing library specifications is expensive: the Android library contains over 5000 classes, many exhibiting complex taint flow behavior. Typically, a human auditor must search the application for calls to potentially important library methods, and then manually write specifications for these methods. Even moderately large apps contain thousands of library method calls, but most of them are irrelevant to finding taint flows. Our experience is likely representative: over a one year period spent analyzing potential Android malware, we have written specifications for just 179 library classes, and we continue to find important new specifications.

Missing specifications can cause false negatives in the static analysis. For example, suppose we remove the library specification for `List.add` from Figure 2. Then the static analysis cannot find the flow from `LOC` to `SMS` (since the taint path between them is broken), causing a false negative. Unlike false positives, where the auditor has a list of flows to inspect in detail, false negatives are difficult to track down—ensuring that a tool has not produced a false negative such as the missing flow from `LOC` to `SMS` may require examining every library method call made by the application.

Intuitively, the visible application code contains useful information about the correct specifications for the library methods. For example, if we remove the specification for `List.add`, then there is still a flow from the `LOC` to $\texttt{arg}_{\texttt{add}}$, and a flow from $\texttt{this}_{\texttt{get}}\texttt{.val}$

to SMS. However, $\texttt{arg}_{\texttt{add}}$ and $\texttt{this}_{\texttt{add}}\texttt{.val}$ are no longer aliased, so the flow is broken. Only one additional assumption (i.e., that $\texttt{arg}_{\texttt{add}}$ is aliased with $\texttt{this}_{\texttt{add}}\texttt{.val}$) is needed to complete the flow.

This example motivates an approach to specification inference that searches for specifications that complete broken flows. Our approach proceeds in two steps. First, our tool finds *potential flows* by making worst-case assumptions about the missing specifications. Second, for each potential flow, our tool keeps track of which assumptions are sufficient to prove that the potential flow is a true flow, which we call *sufficient assumptions* for the potential flow. Finally, the tool proposes that these sufficient assumptions are true. These assumptions correspond to specifications that are the *inferred specifications* produced by the tool.

Continuing our example, our tool would find a potential flow from LOC to SMS by making worst-case assumptions about the specification for List.add, which includes introducing aliasing between $\texttt{arg}_{\texttt{add}}$ and $\texttt{this}_{\texttt{add}}\texttt{.val}$. This specification for List.add is a sufficient assumption for the potential flow from LOC to SMS, so it is inferred by our tool. Upon seeing this inferred specification, the human auditor can confirm that it is correct and thereby determine that the potential flow is a true flow.

In practice, there may be multiple sufficient assumptions for each potential flow. Our tool keeps track of a *minimal* set of sufficient assumptions—i.e., it looks for flows that are broken in the fewest places possible. The guiding principle is that *potential flows that require fewer assumptions are more likely to be real flows than potential flows that require more assumptions*. By extension, potential flows that produce the fewest inferred specifications are most likely to be correct and should be checked first by an auditor.

We use (interprocedural) taint analysis to motivate and validate our general framework. Taint analysis is a prototypical example of a program analysis that can be expressed as a CFL reachability problem and has many practical applications. Furthermore, it is built on top of points-to analysis, which is a key part of many interprocedural program analyses. One of our goals is to infer aliasing relationships potentially introduced by library methods that are missing specifications.

3. CFL Reachability for Explicit Taint Flows

In this section we present a static analysis for finding explicit taint flows and illustrate it using the example in Section 2. Our analysis performs CFL reachability on the portion of the code that is available, and uses specifications for the portion of the code that is unavailable. The specifications are usually manually generated, for example by a human auditor. Our understanding is that many practical systems have this design, and we do not claim that it is novel. However, as far as we know this approach is not well-documented in the literature, so we describe it in some detail. In Section 4, we generalize this analysis to one where some specifications (in addition to the code) are also unavailable (for example, because the auditor has only produced partial specifications), and the task is to infer the possible missing specifications.

Let $C = (U, \Sigma, P, T)$ be a context-free grammar (CFG), where U is the set of non-terminals, Σ is the set of terminals, P is the set of productions, and T is the start symbol. We assume C is normalized so that every production has at most two symbols on the right-hand side [19]. We write $A \stackrel{*}{\Rightarrow} \alpha$, where A is a non-terminal and α is a string of terminals and non-terminals, if α can be derived from A.

Let G be a directed graph such that the edges $v \xrightarrow{\sigma} v'$ in G are labeled with terminal symbols $\sigma \in \Sigma$. A *path* $v \overset{\alpha}{\dashrightarrow} v' \in G$ is a sequence of edges $v \xrightarrow{\sigma_1} w_1 \xrightarrow{\sigma_2} ... \xrightarrow{\sigma_k} v'$ such that $\alpha = \sigma_1 ... \sigma_k$. The transitive closure of G under C is the graph G^C such that $v \xrightarrow{A} v' \in G^C$ if and only if there exists $v \overset{\alpha}{\dashrightarrow} v' \in G$ such that $A \stackrel{*}{\Rightarrow} \alpha$. If $v \xrightarrow{T} v'$, we say v' is *C-reachable* from v.

DEFINITION 3.1. Given a CFG C, a graph $G = (V, E)$, and subsets of vertices $V_{\text{source}}, V_{\text{sink}} \subseteq V$, the *CFL reachability problem* is to determine whether there exist $v \in V_{\text{source}}$ and $v' \in V_{\text{sink}}$ such that v' is C-reachable from v.

Typically, formulating a program analysis as a CFL reachability problem involves converting the input program into a graph $G = (V, E)$ and a CFG C, and then solving the CFL reachability problem for C and G for some given sets of sources and sinks $V_{\text{source}}, V_{\text{sink}} \subseteq V$. This problem can be solved by finding the closure of G under C using a dynamic programming algorithm [19].

3.1 Explicit Taint Flows

We now describe a standard flow- and path-insensitive explicit taint analysis. *Explicit* taint analysis only tracks taint through data flows—i.e., it follows tainted data as it is copied in the course of program execution. We defer discussion of *implicit* flows, where the information transferred depends on control flow decisions, to Section 3.2. As a part of the explicit taint analysis, we perform points-to analysis to find taint flows due to aliasing. For Java, points-to analysis can be expressed as a CFL reachability problem [26]. We extend the points-to analysis to a taint analysis by including source and sink edges that pass taint into and out of the system.

A set of *taint specifications* $\mathcal{S}$ consists of three relations: the *source specifications* $\text{Src}(v, \ell)$, the *sink specifications* $\text{Sink}(v, \ell)$, and the *flow specifications* $\text{Flow}(v, v')$. Here, $v, v' \in G$ are formal method parameters or return values, and $\ell \in \mathcal{L} = \mathcal{L}_{\text{source}} \cup \mathcal{L}_{\text{sink}}$ is a *label* representing a taint source ($\ell \in \mathcal{L}_{\text{source}}$) or a sink ($\ell \in \mathcal{L}_{\text{sink}}$). The relations have the following semantics:

- $\text{Src}(v, \ell)$ says v is tainted with source $\ell \in \mathcal{L}_{\text{source}}$,
- $\text{Sink}(v, \ell)$ says v is passed to sink $\ell \in \mathcal{L}_{\text{sink}}$, and
- $\text{Flow}(v, v')$ says that any taint on v flows to v'.

Given a program P, the *taint graph* for P is $G = (V, E)$, where $V = \mathcal{O} \cup \mathcal{U} \cup \mathcal{L}$. Here, $\mathcal{O}$ is the set of abstract objects in the program (each of which uniquely corresponds to an object allocation site), $\mathcal{U}$ denotes the program's set of reference variables, and $\mathcal{L}$ is the set of labels introduced above. Figure 4 gives rules for generating an initial set of edges for the graph. Rules 1-4 handle primitive forms of statements. Rule 1 says that the contents of the abstract object o flow to the reference v on the left-hand side of the assignment (more formally, v may *point to* o). Rule 2 similarly encodes the flow when a reference variable v is assigned to another reference variable u. Rules 3 and 4 record the flows induced by field writes (or *puts*) and field reads (or *gets*) respectively; note that there is a distinct put/get operation for each field f.

Rules 5-7 formalize the description of the taint specifications $\mathcal{S}$ given above. Rule 5 (symbol SrcRef) says that a source taints a reference variable, Rule 6 (symbol RefSink) says that the contents of a reference variable flow to a particular sink, and Rule 7 (symbol RefRef) says that the contents of one reference variable flow to another reference variable. Rules 8 and 9 are technical devices. Intuitively, Rule 8 ensures at least one abstract object flows to the target reference variable of any flow specification, which eliminates the need to include alias specifications for every method allocating a non-primitive return value (Rule 8 is discussed further in Section 3.2). Finally, Rule 9 allows us to express paths with "backwards" edges by introducing a label $\overline{\sigma}$ to represent the reversal of an edge labeled σ.

We handle interprocedural taint flow as follows: arguments passed by the caller are assigned to formal parameters, which are

assigned to the corresponding references in the callee. Values returned by the callee are assigned to a formal return value, which is assigned to the corresponding reference in the caller. Vertices representing formal parameters and formal return values are added to G by the analysis, for example $\texttt{lat} \xrightarrow{\text{Assign}} \texttt{this}_{\texttt{add}}^{\texttt{formal}} \xrightarrow{\text{Assign}} \texttt{this}_{\texttt{add}}$ and $\texttt{ret}_{\texttt{toString}} \xrightarrow{\text{Assign}} \texttt{ret}_{\texttt{toString}}^{\texttt{formal}} \xrightarrow{\text{Assign}} \texttt{latString}$. This indirection ensures that method boundaries are clear.

Figure 6 shows the taint graph generated from the code in Figure 1 and the specifications in Figure 2 using the rules in Figure 4. For clarity, we have not included formal parameters and formal return values in the graph, but have assigned caller arguments directly to the corresponding callee references and callee return values directly to corresponding caller reference. The graph describes the explicit flows in the program. For example, the edges

$$\texttt{arg}_{\texttt{add}} \xrightarrow{\text{Put[val]}} \texttt{this}_{\texttt{add}} \xleftarrow{\text{Assign}} \texttt{list} \xrightarrow{\text{Assign}} \texttt{this}_{\texttt{get}} \xrightarrow{\text{Get[val]}} \texttt{ret}_{\texttt{get}}$$

capture the fact that any value stored in the list through the `List.add` method can potentially be the result of the `List.get` method. More precisely, the middle two edges show that the reference `list`, the receiver of `List.add`, and the receiver of `List.get` all potentially point to the same abstract object.

The next step is to identify the paths through the graph that correspond to explicit taint flows, which we specify using the CFG C_{taint} defined as follows (with $\mathcal{F}$ denoting the set of fields in P):

$$\begin{aligned}\Sigma_{\text{taint}} &= \{\text{New}, \text{Assign}, \text{SrcRef}, \text{RefSink}, \text{RefRef}\} \\ &\quad \cup \{\text{Put}[f], \text{Get}[f] \mid f \in \mathcal{F}\} \\ U_{\text{taint}} &= \{\text{FlowsTo}, \text{SrcObj}, \text{SrcSink}\} \cup \{\text{FlowsTo}[f] \mid f \in \mathcal{F}\}\end{aligned}$$

We also include symbols $\overline{\sigma}$ and $\overline{A}$ in Σ_{taint} and U_{taint}, respectively. The start symbol of C_{taint} is $T_{\text{taint}} = \text{SrcSink}$. The source vertices we consider are the taint source labels (i.e. $V_{\text{source}} = \mathcal{L}_{\text{source}}$), and the sink vertices we consider are the sink labels (i.e. $V_{\text{sink}} = \mathcal{L}_{\text{sink}}$).

The productions are shown in Figure 5. Rules 10-13 build the points-to relation $o \xrightarrow{\text{FlowsTo}} v$, which means that the reference variable $v \in \mathcal{U}$ may point to the abstract object $o \in \mathcal{O}$. For example, because Figure 6 contains the path

$$\texttt{o}_{\texttt{lat}} \xrightarrow{\text{FlowsTo}} \texttt{arg}_{\texttt{add}} \xrightarrow{\text{Put[val]}} \texttt{this}_{\texttt{add}} \xrightarrow{\overline{\text{FlowsTo}}} \texttt{o}_{\texttt{list}},$$

Rule 12 adds edge $\texttt{o}_{\texttt{lat}} \xrightarrow{\text{FlowsTo[val]}} \texttt{o}_{\texttt{list}}$. Here, $\texttt{o}_{\texttt{lat}}$ and $\texttt{o}_{\texttt{list}}$ are the abstract objects allocated to `lat` and `list`, respectively. Then we have path

$$\texttt{o}_{\texttt{lat}} \xrightarrow{\text{FlowsTo[val]}} \texttt{o}_{\texttt{list}} \xrightarrow{\text{FlowsTo}} \texttt{this}_{\texttt{get}} \xrightarrow{\text{Get[val]}} \texttt{ret}_{\texttt{get}}.$$

Therefore Rule 13 adds $\texttt{o}_{\texttt{lat}} \xrightarrow{\text{FlowsTo}} \texttt{ret}_{\texttt{get}}$, which causes Rule 11 to add $\texttt{o}_{\texttt{lat}} \xrightarrow{\text{FlowsTo}} \texttt{latAlias}$. This means that $\texttt{ret}_{\texttt{getLatitude}}$ and `latAlias` may point to the same abstract object $\texttt{o}_{\texttt{lat}}$, i.e. $\texttt{ret}_{\texttt{getLatitude}}$ and `latAlias` are aliased.

The backwards edge $\texttt{this}_{\texttt{add}} \xrightarrow{\overline{\text{FlowsTo}}} \texttt{o}_{\texttt{list}}$ in the example path above is added by Rule 17, which introduces a reversed edge $v' \xrightarrow{\overline{A}} v$ for every non-terminal edge $v \xrightarrow{A} v'$. In this way, Rule 17 plays the same role for non-terminal edges that Rule 9 plays for terminal edges. Note that the (forward, non-reversed) edge $\texttt{o}_{\texttt{list}} \xrightarrow{\text{FlowsTo}} \texttt{this}_{\texttt{add}}$ arises from the path of terminal edges

$$\texttt{o}_{\texttt{list}} \xrightarrow{\text{New}} \texttt{list} \xrightarrow{\text{Assign}} \texttt{this}_{\texttt{add}}$$

and the application of Rule 10 followed by Rule 11.

3.2 Implicit (Library) Taint Flows

Because $\texttt{ret}_{\texttt{getLatitude}}$ and `latAlias` can point to the same object, if one of them is tainted, then the other should be tainted as well.

1. $\texttt{v=new X()} \Rightarrow o \xrightarrow{\text{New}} v$
2. $\texttt{u=v} \Rightarrow v \xrightarrow{\text{Assign}} u$
3. $\texttt{u.f=v} \Rightarrow v \xrightarrow{\text{Put}[f]} u$
4. $\texttt{u=v.f} \Rightarrow v \xrightarrow{\text{Get}[f]} u$
5. $\text{Src}(v, \ell) \in \mathcal{S} \Rightarrow \ell \xrightarrow{\text{SrcRef}} v$
6. $\text{Sink}(v, \ell) \in \mathcal{S} \Rightarrow v \xrightarrow{\text{RefSink}} \ell$
7. $\text{Flow}(v, v') \in \mathcal{S} \Rightarrow v \xrightarrow{\text{RefRef}} v'$
8. $\exists v\ (\text{Flow}(v, v') \in \mathcal{S}) \Rightarrow o_{v'} \xrightarrow{\text{New}} v'$
9. $v \xrightarrow{\sigma} v' \Rightarrow v' \xrightarrow{\overline{\sigma}} v$ (where $\overline{\overline{\sigma}} = \sigma$)

Figure 4. Program fact extraction rules for taint analysis. In Rule 8, $o_{v'}$ is a fresh vertex.

10. $\text{FlowsTo} \rightarrow \text{New}$
11. $\text{FlowsTo} \rightarrow \text{FlowsTo Assign}$
12. $\text{FlowsTo}[f] \rightarrow \text{FlowsTo Put}[f]\ \overline{\text{FlowsTo}}$
13. $\text{FlowsTo} \rightarrow \text{FlowsTo}[f]\ \text{FlowsTo Get}[f]$
14. $\text{SrcObj} \rightarrow \text{SrcRef}\ \overline{\text{FlowsTo}}$
15. $\text{SrcObj} \rightarrow \text{SrcObj FlowsTo RefRef}\ \overline{\text{FlowsTo}}$
16. $\text{SrcSink} \rightarrow \text{SrcObj FlowsTo RefSink}$
17. $A \rightarrow A_1 ... A_k \Rightarrow \overline{A} \rightarrow \overline{A_k} ... \overline{A_1}$ (where $\overline{\overline{A}} = A$)

Figure 5. Productions for C_{taint}.

Instead of keeping track of taint on the reference variables, it is simpler to keep track of taint on the objects. In Figure 6, Rule 14 adds the edge $\texttt{LOC} \xrightarrow{\text{SrcObj}} \texttt{o}_{\texttt{lat}}$, which says that $\texttt{o}_{\texttt{lat}}$ is tainted. For this taint to flow to the sink, the analysis must be able to pass the taint to $\texttt{o}_{\texttt{ret}_{\texttt{toString}}}$ (the object allocated to $\texttt{ret}_{\texttt{toString}}$).

But here we encounter a problem: there is no explicit flow through `Double.toString`, because no data is copied from the input to the output of the method. Instead, there is an *implicit* flow through a sequence of look-ups converting digits in the double value to characters in the string. That is, information still flows from the input to the output of the method, but through control flow decisions rather than through explicit data flow.

Implicit flows are much more difficult to analyze precisely than explicit flows, because it is much harder to avoid tainting far too much of the program when analyzing implicit flows. Another way to handle implicit flows is to use explicit taint analysis but include appropriate specifications for methods that have implicit flows. We consider the set of specifications that pass taint from one argument to another argument or to the return value. In our specification $\mathcal{S}$, this behavior is described by the relation $\text{Flow}(v, v')$, where $v \in V$ is a library method parameter and $v' \in V$ is a library method parameter or return value. Such specifications require Rule 8 from Figure 4, which adds *phantom objects* to the graph.

Recall that we attach taint to objects, not references. Thus, no taint can flow to parts of the code that have no objects associated with them. If a library method does not have an alias specification, then the method will have no object associated with its return value. Rather than manually add alias specifications for every method with a non-primitive return object, Rule 8 in Figure 4 automatically adds a (fresh) abstract object that the return value points to. For example, Rule 8 adds $\texttt{o}_{\texttt{ret}_{\texttt{toString}}} \xrightarrow{\text{New}} \texttt{ret}_{\texttt{toString}}$, causing Rule 10 to add

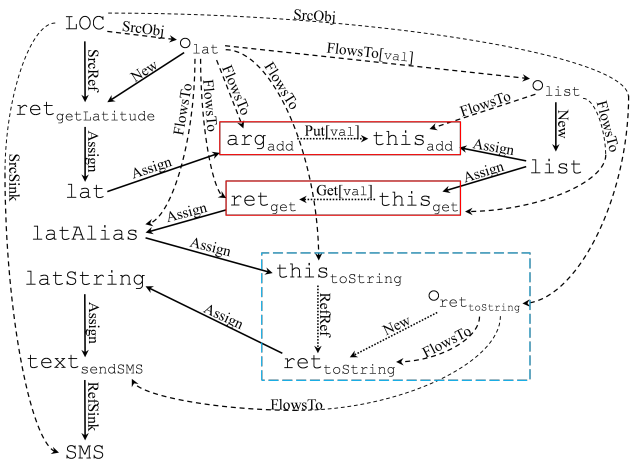

Figure 6. The taint graph G corresponding to the code in Figure 1 and the library specifications in Figure 2. Solid edges are facts extracted from the code in Figure 1 (backwards edges added by Rule 9 are not shown). Dotted edges are facts extracted from the library specifications in Figure 2. Edges corresponding to alias specifications are boxed in a solid red line, and edges corresponding to flow specifications are boxed in a dashed blue line. Dashed edges are edges added by productions in Figure 5 (not all such edges are shown).

$\texttt{o}_{\texttt{ret}_{\texttt{toString}}} \xrightarrow{\text{FlowsTo}} \texttt{ret}_{\texttt{toString}}$. Now we can capture the flow in Figure 1. In Figure 6, we have the path

$$\texttt{LOC} \xrightarrow{\text{SrcObj}} \texttt{o}_{\texttt{lat}} \xrightarrow{\text{FlowsTo}} \texttt{this}_{\texttt{toString}} \xrightarrow{\text{RefRef}} \texttt{ret}_{\texttt{toString}} \xrightarrow{\overline{\text{FlowsTo}}} \texttt{o}_{\texttt{ret}_{\texttt{toString}}}.$$

Rule 15 adds the edge $\texttt{LOC} \xrightarrow{\text{SrcObj}} \texttt{o}_{\texttt{ret}_{\texttt{toString}}}$. Now we have

$$\texttt{LOC} \xrightarrow{\text{SrcObj}} \texttt{o}_{\texttt{ret}_{\texttt{toString}}} \xrightarrow{\text{FlowsTo}} \texttt{text}_{\texttt{sendSMS}} \xrightarrow{\text{RefSink}} \texttt{SMS},$$

so Rule 16 adds the edge $\texttt{LOC} \xrightarrow{\text{SrcSink}} \texttt{SMS}$.

4. Problem Statement

In this section, we formulate the problem of performing a sound and precise CFL reachability analysis when some specifications are missing, along with the problem of inferring the missing specifications. Our formulation extends the framework defined in Section 3.

4.1 Missing Specifications CFL Reachability

Suppose we want to perform a CFL reachability analysis on a program P. Assume $G^* = (V^*, E^*)$ is the graph constructed from P with complete specifications. If specifications are missing, then the constructed graph $\widehat{G} = (\widehat{V}, \widehat{E})$ may be missing vertices and edges, i.e. $\widehat{V} \subseteq V^*$ and $\widehat{E} \subseteq E^*$. The goal is to perform a sound and precise worst-case analysis given some information about the missing vertices and edges. We encode the possible missing data as a family of graphs $\mathcal{G}$, where we only know that $G^* \in \mathcal{G}$.

DEFINITION 4.1. Suppose we are given $\widehat{G} = (\widehat{V}, \widehat{E})$, a set of sources $V_{\text{source}} \subseteq \widehat{V}$, a set of sinks $V_{\text{sink}} \subseteq \widehat{V}$, along with some family $\mathcal{G}$ of graphs such that for each $G \in \mathcal{G}$, $\widehat{G}$ is a subgraph of G, i.e. $\widehat{G} \subseteq G$. We call G a *completion* of $\widehat{G}$. Let $\mathcal{A} : V_{\text{source}} \times V_{\text{sink}} \to \text{Bool}$ be the result of a static analysis, where $\mathcal{A}(v, v') = \textit{true}$ indicates that taint flows from v to v'.

```
1. class List:
2.   void add(Object arg) {
3.     arg.f = arg;
4.     this.val = arg; }
```

Figure 7. An alternative (and incorrect) specification for `List.add`.

- $\mathcal{A}$ is *sound* if for every $v \in V_{\text{source}}$ and $v' \in V_{\text{sink}}$, $\mathcal{A}(v, v') =$ false if and only if there does not exist any $G \in \mathcal{G}$ such that $v \xrightarrow{T} v' \in G^C$.
- $\mathcal{A}$ is *precise* if for every $v \in V_{\text{source}}$ and $v' \in V_{\text{sink}}$, $\mathcal{A}(v, v') =$ true if and only if there exists $G \in \mathcal{G}$ such that $v \xrightarrow{T} v' \in G^C$.

The idea behind this definition is that an analysis is sound if it does not miss any taint flow present in at least one of the possible completions of $\widehat{G}$, and the algorithm is precise if it does not report any flows that do not occur in any completion of $\widehat{G}$. An analysis $\mathcal{A}$ solves the *missing specifications CFL reachability problem* for a family $\mathcal{G}$ if it is both sound and precise.

While Definition 4.1 captures the notion of performing a worst-case analysis that is sound and precise, we are also interested in keeping track of the assumptions that the worst-case analysis makes about missing specifications. In practice, many assumptions may produce the same results. Therefore we are interested in producing a minimal set of assumptions. Suppose we have a partial order $(\mathcal{G}, \leq)$, where $G_1 \leq G_2$ should mean that the graph G_1 makes at most as many assumptions as G_2. The definition of $\leq$ depends on the family $\mathcal{G}$. In addition to producing sound and precise results $\mathcal{A}$, we would like to produce a minimal $G \in \mathcal{G}$ (with respect to $\leq$) such that performing the CFL reachability analysis on G yields $\mathcal{A}$.

DEFINITION 4.2. Suppose we are given the inputs as in Definition 4.1, along with a partial order $(\mathcal{G}, \leq)$. We use the notation $e \overset{?}{\in} G$; this expression evaluates to true if $e \in G$ and false otherwise. The *CFL reachability specification inference problem* is to produce sound and precise results $\mathcal{A}$, along with *sufficient assumptions*, encoded as a graph $G \in \mathcal{G}$ satisfying $\mathcal{A}(v, v') = (v \xrightarrow{T} v' \overset{?}{\in} G^C)$ for every $v \in V_{\text{source}}$ and $v' \in V_{\text{sink}}$. Furthermore, we require that G is *minimal*, i.e. there does not exist sufficient assumptions $G' \in \mathcal{G}$ such that $G' < G$.

In the remainder of this section, we describe how we apply this framework to inferring alias and flow specifications.

4.2 $\mathcal{G}$ Using Regular Languages

To design an algorithm for solving a missing specifications CFL reachability problem, we must first specify the family $\mathcal{G}$ of graphs to which G^* may belong. Our goal is to define a family $\mathcal{G}$ that is simultaneously general, retains precision in practice, and admits efficient algorithms. We confine our presentation to inferring specifications for missing methods. This restriction is without loss of generality and is done to simplify notation and discussion throughout the rest of the paper.

One restriction we do make is that inferred specifications do not access static fields. Our algorithms in fact work without this restriction, but the results are almost always not useful. It is easy to see why: if there are at least two missing methods that can access static fields, it is possible for one to store a tainted value in a static field and the other to read it, whether or not these methods have anything else to do with each other. Furthermore, specifications involving static fields are rare: none of the specifications we have manually written have involved static fields.

Consider the taint graph G in Figure 6. Suppose the specification for the method `List.add` in Figure 2 is missing, so the edge

$\texttt{arg}_{\texttt{add}} \xrightarrow{\text{Put[val]}} \texttt{this}_{\texttt{add}}$ in Figure 6 is missing, giving us the graph $\widehat{G}$. Without access to static fields, the only way to complete a flow through $\widehat{G}$ is if there is a path connecting $\texttt{arg}_{\texttt{add}}$ to $\texttt{this}_{\texttt{add}}$. In general, for a method m, the only possible taint flows through m are from a parameter of m to m's return value, or from one parameter to another parameter. We use $V_m \subseteq \widehat{V}$ to denote the vertices of $\widehat{G}$ corresponding to the formal parameters and return value of m.

To be sound, we must assume that the specification of `List.add` could execute any sequence of operations. In other words, G consists of $\widehat{G}$ with some additional subgraph $G_{\texttt{arg}_{\texttt{add}},\texttt{this}_{\texttt{add}}}$ connecting $\texttt{arg}_{\texttt{add}}$ to $\texttt{this}_{\texttt{add}}$. Note that for any subgraph $G_{\texttt{arg}_{\texttt{add}},\texttt{this}_{\texttt{add}}}$ corresponding to a possible specification of `List.add`'s behavior, the only information relevant to the CFL reachability problem is the possible sequences of terminals $\alpha \in \Sigma^*$ that can occur along paths $\texttt{arg}_{\texttt{add}} \overset{\alpha}{\dashrightarrow} \texttt{this}_{\texttt{add}}$. Generalizing from this example, for a missing method m it suffices to consider the family of graphs $\mathcal{G}$ consisting of all graphs containing $\widehat{G}$ as a subgraph with additional paths $w \overset{\alpha}{\dashrightarrow} w'$, where $w, w' \in V_m$.

For example, one completion of $\widehat{G}$ is the taint graph G in Figure 6, which is just $\widehat{G}$ with the edge $\texttt{arg}_{\texttt{add}} \xrightarrow{\text{Put[val]}} \texttt{this}_{\texttt{add}}$ added back in. But there are other ways to complete $\widehat{G}$, even for this simple example. Consider the graph G' obtained when the specification for `List.add` is given in Figure 7. Then G' is $\widehat{G}$ with the additional path

$$\texttt{arg}_{\texttt{add}} \xrightarrow{\text{Put[f]}} \texttt{arg}_{\texttt{add}} \xrightarrow{\text{Put[val]}} \texttt{this}_{\texttt{add}}$$

In general, there may be infinitely many possible paths $\texttt{arg}_{\texttt{add}} \overset{\alpha}{\dashrightarrow} \texttt{this}_{\texttt{add.this}}$ because the sequence of operations α can be arbitrarily long. Thus, we need some compact way to represent an infinite language of strings; the regular languages are a natural choice. This discussion motivates our definition of the family $\mathcal{G}_W^R$:

DEFINITION 4.3. Let $W \subseteq \widehat{V} \times \widehat{V}$ and let R be a regular language over Σ. The family of graphs $\mathcal{G}_W^R$ contains the graph G if

- $\widehat{G} \subseteq G$
- If $(w, w') \in W$, then the nondeterministic finite automaton (NFA) N with the transition matrix given by the subgraph $G_{w,w'}$ satisfies $L(N) \subseteq L(R)$.

Here, $G_{w,w'}$ is the subgraph connecting w to w' (not including w or w'). This definition exploits the insight that we can think of the subgraph $G_{w,w'}$ as the transition graph of an NFA N with start state w and final state w'. Any path $w \overset{\alpha}{\dashrightarrow} w' \in G_{w,w'}$ satisfies $\alpha \in L(N)$. Conversely, for any $\alpha \in L(N)$, there exists a path $w \overset{\alpha}{\dashrightarrow} w' \in G_{w,w'}$.

In general, choosing $R = \Sigma^*$ will produce sound results, since this choice imposes no constraints on the allowed paths connecting w and w'. In practice, a more restrictive language may be chosen either to incorporate known constraints on potential specifications, or to trade some soundness for improved scalability.

4.3 Inferring Specifications for $\mathcal{G}_W^R$

We now formulate the corresponding missing specifications problem. Our goal is to infer specifications of the following form: there exists a path (or set of paths) connecting w and w'. As discussed above, both alias and flow specifications can be described in this manner. We need to define a partial order $\leq$ on $\mathcal{G}_W^R$ that captures the notion of making minimal assumptions about missing specifications. Because we are searching for source-sink paths, it is natural to define $\leq$ in terms of source-sink paths through G. To simplify notation, we assume there is a single source and a single sink, i.e. $V_{\text{source}} = \{v_{\text{source}}\}$ and $V_{\text{sink}} = \{v_{\text{sink}}\}$. Let $\mathcal{P}(G) = \{v_{\text{source}} \overset{\alpha}{\dashrightarrow} v_{\text{sink}} \in G \mid T \overset{*}{\Rightarrow} \alpha\}$ denote the set of all possible source-sink paths in G. We define the *weight* of a source-sink path $p \in \mathcal{P}(G)$ to be

$$\text{weight}(p) = \sum_{(w,w') \in W} (\#\text{ times } p \text{ passes through } G_{w,w'}).$$

In other words, the weight of a path p equals the number of assumptions $G_{w,w'} \neq \emptyset$ used along p. Note that if an assumption is used multiple times (i.e., p passes through $G_{w,w'}$ multiple times), then each use is counted separately. Define the weight of $G \in \mathcal{G}_W^R$ to be the minimum weight of any source-sink path in G: $\text{weight}(G) = \arg\min_{p \in \mathcal{P}(G)} \text{weight}(p)$. Now define $G_1 \leq G_2$ if $\text{weight}(G_1) \leq \text{weight}(G_2)$. In other words, we want to find $G \in \mathcal{G}_W^R$ with the source-sink path of lowest weight. In Section 5, we show how to reduce this problem to the shortest-path CFL reachability problem.

We are interested in inferring both flow and alias specifications. Let $V_m = V_m^{\text{arg}} \cup \{r_m\}$, where V_m^{arg} is the set of formal parameters of a method m, and r_m is the formal return value of m. First, we infer missing flow specifications (i.e., specifications $\text{Flow}(v, v')$, which describe implicit taint flows through library methods):

$$W_{\text{flow}} = \{(w, w') : w \in V_m^{\text{arg}} \text{ and } w' \in V_m\}$$
$$R_{\text{flow}} = \text{RefRef}$$

For example, we could use the family $\mathcal{G}_{W_{\text{flow}}}^{R_{\text{flow}}}$ to infer the specification for `Double.toString` if it were missing.

Second, we infer missing alias specifications (which describe alias relations potentially introduced by calls to methods such as `List.add` and `List.get`):

$$\begin{aligned} W_{\text{alias}} &= \{(w, w') : w, w' \in V_m\} \\ R_{\text{alias}} &= (\text{Assign} + \overline{\text{Assign}}) \\ &\quad (\text{New} + \text{Assign} + \text{Put}[f]_{f \in \mathcal{F}} + \text{Get}[f]_{f \in \mathcal{F}} \\ &\quad + \overline{\text{New}} + \overline{\text{Assign}} + \overline{\text{Put}[f]}_{f \in \mathcal{F}} + \overline{\text{Get}[f]}_{f \in \mathcal{F}})^* \\ &\quad (\text{Assign} + \overline{\text{Assign}}) \end{aligned}$$

The possible sequences of operations are bracketed by $(\text{Assign} + \overline{\text{Assign}})$ because allocation and field access operations cannot be performed on formal parameters and formal return values (since they are added to $\widehat{G}$ by the static analysis and do not correspond to references in the program, as described in Section 3).

5. Algorithms for Specification Inference

In this section we present an algorithm that solves the missing specifications problem stated in Definition 4.1 for $\mathcal{G}_W^R$. Next, we discuss an optimization that enables our algorithm to scale to large programs. Finally, we describe how to extend the algorithm to solve the specification inference problem stated in Definition 4.2 by using a shortest-path extension of the CFL reachability algorithm. This allows us to construct an algorithm that interacts with a human auditor to produce results that are sound and precise with respect to G^*. An overview of our system is given in Figure 8.

5.1 Algorithms for $\mathcal{G}_W^R$

Consider $(w, w') \in W$. Recall that every potential path $w \overset{\alpha}{\dashrightarrow} w'$ satisfies $\alpha \in L(R)$. To be sound and precise with respect to $\mathcal{G}_W^R$, it suffices to construct a subgraph connecting w and w' such that there is a path $w \overset{\alpha}{\dashrightarrow} w'$ through this subgraph if and only if $\alpha \in L(R)$.

The subgraphs that satisfy this property are the transition graphs for nondeterministic finite automata (NFAs) that accept $L(R)$. For every $(w, w') \in W$, Algorithm 1 constructs the transition graph $\mathcal{N}(s \xrightarrow{R} f)$ for one such NFA, and then adds this transition graph to $\widehat{G}$ to connect w to w', resulting in graph G'. Finally, we compute

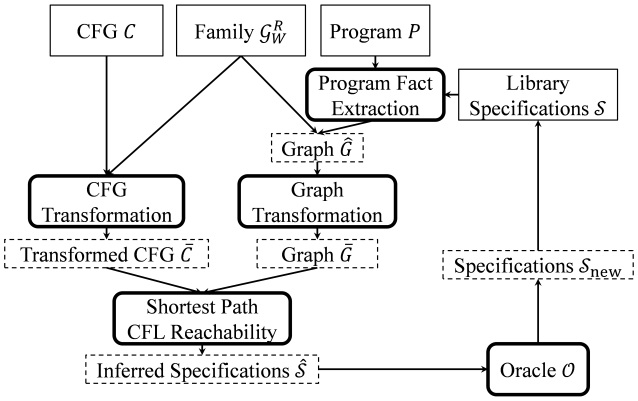

Figure 8. An overview of our specification inference system. The system infers specifications $\widehat{\mathcal{S}}$ and proposes them to the oracle $\mathcal{O}$ (i.e., the human auditor), who examines the proposals and generates a new set of specifications $\mathcal{S}_{\text{new}}$. Then $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}_{\text{new}}$, and the process repeats. Program fact extraction is described in Section 3. While not depicted here, C may depend on P. The graph transformation and CFG transformation are computed by Algorithm 2. The shortest-path CFL reachability algorithm is described in Section 5.3 and Appendix A. The specification refinement loop is performed by Algorithm 3.

1. $\mathcal{N}(v \xrightarrow{\sigma} v') = \{v \xrightarrow{\sigma} v'\}$ for all $\sigma \in \Sigma$
2. $\mathcal{N}(v \xrightarrow{R_1 R_2} v') = \mathcal{N}(v \xrightarrow{R_1} t) \cup \mathcal{N}(t \xrightarrow{R_2} v')$
3. $\mathcal{N}(v \xrightarrow{R_1 + R_2} v') = \mathcal{N}(v \xrightarrow{R_1} v') \cup \mathcal{N}(v \xrightarrow{R_2} v')$
4. $\mathcal{N}(v \xrightarrow{R_1^*} v') = \{v \xrightarrow{\epsilon} t\} \cup \mathcal{N}(t \xrightarrow{R_1} t) \cup \{t \xrightarrow{\epsilon} v'\}$

Figure 9. Given $v \xrightarrow{R} v'$, $\mathcal{N}$ constructs the transition graph for a NFA that accepts R with start state v and final state v'. In Rules 2 and 4, t is a fresh vertex.

the transitive closure $(G')^C$ of G' with respect to the context-free language C. The following correctness result follows from the correspondence between $L(R)$ and $\mathcal{N}(s \xrightarrow{R} f)$ described above.

THEOREM 5.1. Algorithm 1 is sound and precise for $\mathcal{G}_W^R$.

5.2 Optimizations

Consider the subgraph $G_i = \mathcal{N}(s_i \xrightarrow{R} f_i)$ constructed by Algorithm 1 for a pair $(w_i, w_i') \in W$ (where i is an index over pairs (w_i, w_i')). One issue scaling Algorithm 1 is that the CFL reachability algorithm may add a large number of edges that are only among the vertices within G_i. Any such *internal edge* $n_i \xrightarrow{A} m_i$, where n_i and m_i are vertices in G_i, is added whenever there is a path $n_i \xrightarrow{\alpha} m_i \in G_i$ such that $A \xRightarrow{*} \alpha$. The problem is that the subgraphs G_i are all isomorphic—thus, the same edges are recomputed many times by the standard CFL reachability algorithm. This observation suggests that we can benefit from precomputing the internal edges.

One convenient way to implement this optimization is to preprocess the grammar C instead of adding graphs to $\widehat{G}$. That is, we embed an (optimized) version of G_i in C. Since the graphs G_i are all isomorphic to one another, this embedding only needs to be performed once. Intuitively, such an transformation is possible because G_i encodes a regular language and C is context-free.

The essential idea is that for every $(w_i, w_i') \in W$, we replace G_i by a single vertex. Because we are only interested in summarizing the transitive closure of G_i with respect to C, we only need one vertex to represent the net effect of G_i, though this vertex may have many incident edges. More specifically, we modify $\widehat{G}$ in the following ways to define a new graph $\overline{G}$:

- We add a new vertex v_i to $\widehat{G}$; here the single vertex v_i will represent G_i.
- We add two new, distinct terminal symbols b and e to Σ, standing for "beginning G_i at w_i" and "exiting G_i at w_i'", respectively. These terminals are needed to mark in the modified grammar where we enter and exit G_i.
- We add the edges $w_i \xrightarrow{b} v_i \xrightarrow{e} w_i'$ to $\widehat{G}$.

We now turn to incorporating the transitions of each G_i into C, defining a new grammar $\overline{C}$. Let n_i, m_i, and r_i denote vertices in G_i (corresponding to NFA states n, m, and r, respectively). Let $v \in \widehat{V}$ (recall that $\widehat{V}$ is the set of vertices of $\widehat{G}$—i.e., all the vertices not in any G_i). Finally, G' is the graph constructed by Algorithm 1. We want C and $\overline{C}$ to correspond in the following way:

1. If there is an edge $v \xrightarrow{A} n_i \in (G')^C$, then there should be an edge $v \xrightarrow{A_n} v_i \in \overline{G}^{\overline{C}}$. Intuitively, the non-terminal A_n records that A was matched ending at the vertex in G_i corresponding to state n.
2. If there is an edge $n_i \xrightarrow{A} v \in (G')^C$, then there should be an edge $v_i \xrightarrow{A^n} v \in \overline{G}^{\overline{C}}$. Intuitively, the non-terminal A^n records that A was matched starting at the vertex in G_i corresponding to state n.
3. If there is an edge $n_i \xrightarrow{A} m_j \in (G')^C$ (that is *not* an internal edge), then there should be an edge $v_i \xrightarrow{A_m^n} v_j \in \overline{G}^{\overline{C}}$. Intuitively, the non-terminal A_m^n records that A was matched starting at the vertex in G_i corresponding to state n, and ending at the vertex in G_j corresponding to state m.

Finally, we need to define the productions for each of the additional non-terminals so that conditions 1-3 above are satisfied. We generate these additional productions of $\overline{C}$ from the productions in C and the C-closure of G_i using the rules in Figure 10. In the figure, we refer to vertices s_i and f_i in G_i introduced by Algorithm 1, which correspond to the start state s and end state e of the NFA, respectively.

We briefly explain Rules 1a-f (the rules for non-terminals of the form A_n) for producing productions of $\overline{C}$ in Figure 10; Rules 2a-f and 3a-h are similar.

(a) Suppose we have edge $v \xrightarrow{A} w_i \in (G')^C$. Because we have edge $w_i \xrightarrow{\epsilon} s_i \in G'$, we produce $v \xrightarrow{A} s_i \in (G')^C$. In $\overline{G}^{\overline{C}}$, we

---

Algorithm 1 A sound and precise algorithm for $\mathcal{G}_W^R$. Here, $\mathcal{N}(s \xrightarrow{R} f)$ is defined in Figure 9.

function EXPANSION($C, \widehat{G}, W, R, v, v'$)
  return $v \xrightarrow{T} v' \stackrel{?}{\in} (\text{EXPANSIONHELPER}(\widehat{G}, W, R))^C$
end function
function EXPANSIONHELPER($\widehat{G}, W, R$)
  $G' \leftarrow \widehat{G}$
  for all $(w, w') \in W$ **do**
    // s and f are fresh vertices
    $G' \leftarrow G' \cup \{w \xrightarrow{\epsilon} s, f \xrightarrow{\epsilon} w'\} \cup \mathcal{N}(s \xrightarrow{R} f)$
  end for
  return G'
end function

1. Productions for A_n:

(a) $\dfrac{}{A_s \to Ab \in \overline{C}}$

(b) $\dfrac{s_i \xrightarrow{A} n_i \in G_i^C}{A_n \to b \in \overline{C}}$

(c) $\dfrac{n_i \xrightarrow{\epsilon} m_i \in G_i^C}{A_m \to A_n \in \overline{C}}$

(d) $\dfrac{A \to B \in C}{A_n \to B_n \in \overline{C}}$

(e) $\dfrac{A \to BD \in C}{A_n \to BD_n \in \overline{C}}$

(f) $\dfrac{A \to BD \in C \quad n_i \xrightarrow{D} m_i \in G_i^C}{A_m \to B_n \in \overline{C}}$

2. Productions for A^n:

(a) $\dfrac{}{A^f \to eA \in \overline{C}}$

(b) $\dfrac{n_i \xrightarrow{A} f_i \in G_i^C}{A^n \to e \in \overline{C}}$

(c) $\dfrac{m_i \xrightarrow{\epsilon} n_i \in G_i^C}{A^m \to A^n \in \overline{C}}$

(d) $\dfrac{A \to B \in C}{A^n \to B^n \in \overline{C}}$

(e) $\dfrac{A \to BD \in C}{A^n \to B^n D \in \overline{C}}$

(f) $\dfrac{A \to BD \in C \quad m_i \xrightarrow{B} n_i \in G_i^C}{A^m \to D^n}$

3. Productions for A_m^n:

(a) $\dfrac{}{A_s^n \to A^n b \in \overline{C}}$

(b) $\dfrac{}{A_n^f \to eA_n \in \overline{C}}$

(c) $\dfrac{n_i \xrightarrow{\epsilon} m_i \in G_i^C}{A_m^r \to A_n^r \in \overline{C}}$

(d) $\dfrac{m_i \xrightarrow{\epsilon} n_i \in G_i^C}{A_r^m \to A_r^n \in \overline{C}}$

(e) $\dfrac{A \to B \in C}{A_n^m \to B_n^m \in \overline{C}}$

(f) $\dfrac{A \to BD \in C}{A_n^m \to B^m D_n \in \overline{C}}$

(g) $\dfrac{A \to BD \in C \quad n_i \xrightarrow{D} m_i \in G_i^C}{A_m^r \to B_n^r \in \overline{C}}$

(h) $\dfrac{A \to BD \in C \quad m_i \xrightarrow{B} n_i \in G_i^C}{A_r^m \to D_r^n \in \overline{C}}$

4. Stitching productions:

(a) $\dfrac{}{A \to A_f e \in \overline{C}}$

(b) $\dfrac{}{A \to bA^s \in \overline{C}}$

(c) $\dfrac{A \to BD \in C}{A \to B_n D^n \in \overline{C}}$

(d) $\dfrac{A \to BD \in C}{A_n \to B_m D_n^m \in \overline{C}}$

(e) $\dfrac{A \to BD \in C}{A^n \to B_m^n D^m \in \overline{C}}$

(f) $\dfrac{A \to BD \in C}{A_m^n \to B_r^n D_m^r \in \overline{C}}$

Figure 10. Productions for $\overline{C}$.

Algorithm 2 Optimized algorithm for $\mathcal{G}_W^R$. Here, $\mathcal{T}(G_i, C)$ applies the rules in Figure 10 to C for the given graph G_i. Also, s and f are fresh vertices.

function PREPROCESS($C, \widehat{G}, W, R, v, v'$)
  $G_i \leftarrow \mathcal{N}(s \xrightarrow{R} f)$; $\overline{C} \leftarrow \mathcal{T}(G_i, C)$
  return $v \xrightarrow{T} v' \stackrel{?}{\in}$ (PREPROCESSHELPER($\widehat{G}, W$))$^{\overline{C}}$
end function
function PREPROCESSHELPER($\widehat{G}, W$)
  $\overline{G} \leftarrow \widehat{G}$
  for all $(w, w') \in W$ **do**
    // v is a fresh vertex
    $\overline{G} \leftarrow \overline{G} \cup \{w \xrightarrow{b} v, v \xrightarrow{e} w'\}$
  end for
  return $\overline{G}$
end function

1. FlowsTo$_n$ $\to$ FlowsTo$_s$ (Rule 1f)
2. FlowsTo$_n$ $\to$ FlowsTo$[f]_n$ (Rule 1f)
3. FlowsTo$_n$ $\to$ FlowsTo$[f]$ FlowsTo$_n$ (Rules 1e & 1f)
4. FlowsTo$[f]_n$ $\to$ FlowsTo$_n$ (Rule 1f)
5. FlowsTo$[f]_n$ $\to$ FlowsTo Put$[f]$ $\overline{\text{FlowsTo}}_n$ (Rule 1e)

Figure 11. Examples of production rules added by Figure 10, along with the rules that generate them.

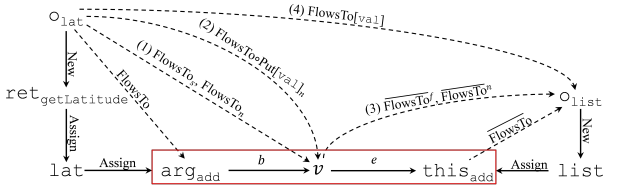

Figure 12. Algorithm 2 adds the dashed edges to Figure 6 if the specification for `List.add` is missing. We only show edges relevant to the production of the edge labeled (4).

have edges $v \xrightarrow{A} w_i \xrightarrow{b} v_i$, and we need to produce $v \xrightarrow{A_s} v_i$. This is achieved by the production $A_s \to Ab \in \overline{C}$.

(b) Suppose we have internal edge $s_i \xrightarrow{A} n_i \in G_i^C$. Because we have edge $w_i \xrightarrow{\epsilon} s_i \in G'$, we produce $w_i \xrightarrow{A} n_i \in (G')^C$. In $\overline{G}^{\overline{C}}$, we have edge $w_i \xrightarrow{b} v_i$ and we need to produce $w_i \xrightarrow{A_n} v_i$. This is achieved by the production $A_n \to b \in \overline{C}$.

(c) Suppose we have internal edge $n_i \xrightarrow{\epsilon} m_i \in G_i^C$ and edge $v \xrightarrow{A} n_i \in (G')^C$. Then we produce $v \xrightarrow{A} m_i \in (G')^C$. In $\overline{G}^{\overline{C}}$, we have edge $v \xrightarrow{A_n} v_i$, and we need to produce $v \xrightarrow{A_m} v_i$. This is achieved by the production $A_m \to A_n \in \overline{C}$.

(d) Suppose we have edge $v \xrightarrow{B} n_i \in (G')^C$ and production $A \to B \in C$. Then we produce $v \xrightarrow{A} n_i \in (G')^C$. In $\overline{G}^{\overline{C}}$, we have edge $v \xrightarrow{B_n} v_i$, and we need to produce $v \xrightarrow{A_n} v_i$. This is achieved by the production $A_n \to B_n \in \overline{C}$.

(e) Suppose we have edges $v \xrightarrow{B} v' \xrightarrow{D} n_i \in (G')^C$ and production $A \to BD \in C$. Then we produce $v \xrightarrow{A} n_i \in (G')^C$. In $\overline{G}^{\overline{C}}$, we have edges $v \xrightarrow{B} v' \xrightarrow{D_n} v_i$, and we need to produce $v \xrightarrow{A_n} v_i$. This is achieved by the production $A_n \to BD_n \in \overline{C}$.

(f) Suppose we have edge $v \xrightarrow{B} n_i \in (G')^C$, internal edge $n_i \xrightarrow{D} m_i \in G_i^C$, and production $A \to BD \in C$. Then we produce $v \xrightarrow{A} m_i \in (G')^C$. In $\overline{G}^{\overline{C}}$, we have edge $v \xrightarrow{B_n} v_i$, and we need to produce $v \xrightarrow{A_m} v_i$. This is achieved by the production $A_m \to B_n \in \overline{C}$.

Note that cases (e) and (f) correspond to two possibilities for binary productions, (e) handling the case where one edge is fully outside of G_i and (f) handling the case where one edge is fully inside G_i. In the case where only the middle vertex is in G_i and both endpoints are outside, then we need the "stitching production" (c) in Figure 10. We describe stitching productions (a), (b), and (c):

(a) Suppose we have edge $v \xrightarrow{A} f_i \in (G')^C$. Because we have edge $f_i \xrightarrow{\epsilon} w_i' \in G'$, we produce $v \xrightarrow{A} w_i' \in (G')^C$. In $\overline{G}^{\overline{C}}$, we have edges $v \xrightarrow{A_f} v_i \xrightarrow{e} w_i'$, and we need to produce $v \xrightarrow{A} w_i'$. This is achieved by the production $A \to A_f e \in \overline{C}$.

(b) Suppose we have edge $s_i \xrightarrow{A} v \in (G')^C$. Because we have edge $w_i \xrightarrow{\epsilon} s_i \in G'$, we produce $w_i \xrightarrow{A} v \in (G')^C$. In $\overline{G}^{\overline{C}}$, we have edges $w_i \xrightarrow{b} v_i \xrightarrow{A^s} v$, and we need to produce $w_i \xrightarrow{A} v$. This is achieved by the production $A \to bA^s \in \overline{C}$.

(c) Suppose we have edges $v \xrightarrow{B} n_i \xrightarrow{D} v' \in (G')^C$ and production $A \to BD \in C$. Then we produce $v \xrightarrow{A} v' \in (G')^C$. In $\overline{G}^{\overline{C}}$, we have edges $v \xrightarrow{B_n} v_i \xrightarrow{D^n} v'$, and we need to produce $v \xrightarrow{A} v'$. This is achieved by the production $A \to B_n D^n$.

The stitching productions (d), (e), and (f) are similar to (c).

Finally, we show that the rules given in Figure 10 are complete. As above, we focus on Rules 1a-f first. Note that we need to add an edge $v \xrightarrow{A_n} v_i$ whenever there exists a path $v \overset{\alpha}{\dashrightarrow} w_i$ and there exists $\beta \in \Sigma^*$ such that $A \overset{*}{\Rightarrow} \alpha\beta$ and $s_i \overset{\beta}{\dashrightarrow} n_i \in G_i$. In other words, α is the portion of the path in $\widehat{G}$ and β is the portion of the path in G_i, and the path ends at vertex $n_i \in G_i$. Consider the production that is the first step in the derivation of $A \overset{*}{\Rightarrow} \alpha\beta$:

- Case $A \to \epsilon$: then $\alpha = \epsilon$, so $v = w_i$, and we need to add edge $w_i \xrightarrow{A_n} v_i$. The fact that $\alpha = \epsilon$ also implies that $A \overset{*}{\Rightarrow} \beta$, so $s_i \xrightarrow{A} n_i \in G_i^C$. Hence this case is handled by Rule 1b.
- Case $A \to BD$: either α is a prefix of BD and β is a suffix of D (handled by Rule 1e), or α is a prefix of B and β is a suffix of BD (handled by Rule 1f).
- Case $A \to B$: then α is a prefix of B and β is a suffix of B, so this case is handled by Rule 1d.

Rule 1a is added to satisfy the semantics of the symbol $b \in \Sigma$. Finally, we have to consider ϵ transitions that occur in G_i^C—i.e., $n_i \xrightarrow{\epsilon} m_i \in G_i^C$ (these transitions are used in conjunction with the implicit productions $A \to \epsilon A$ and $A \to A\epsilon$). These transitions are handled by Rule 1c. Rules 2a-f and 3a-h follow similarly.

Next, we show that Rules 4a-f are complete. Note that we need to add an edge $v \xrightarrow{A} v'$ whenever there exist paths $v \overset{\alpha}{\dashrightarrow} w_i$ and $w_i' \overset{\gamma}{\dashrightarrow} v'$, and there exists $\beta \in L(R)$ such that $A \overset{*}{\Rightarrow} \alpha\beta\gamma$. Here, α and γ are the portions of the path in $\widehat{G}$, and β is the portion of the path in G_i. As before, we can consider production that is the first step in the derivation of $A \overset{*}{\Rightarrow} \alpha\beta\gamma$. This time, we only need to handle the case where the production is split at vertex v_i—i.e., $A \to BD$, where $B \overset{*}{\Rightarrow} \alpha\beta_1$ and $D \overset{*}{\Rightarrow} \beta_2\gamma$ (and $\beta = \beta_1\beta_2$); this is handled by Rule 4c. Rules 4d-f follow similarly when considering productions for A_n, A^n, and A_n^m. Finally, the semantics of the symbols b and e are handled by Rules 4b and 4a, respectively. While we only described the case where the path passes through a single pair (w_i, w_i'), the general case follows because the first step in the derivation can be split only at a single vertex v_i.

We denote the subroutine constructing $\overline{C}$ by $\mathcal{T}$, i.e. $\overline{C} = \mathcal{T}(G_i, C)$. Note that any G_i can be used, since G_i (and hence G_i^C) is the same for every $(w_i, w_i') \in W$. Algorithm 2 calls $\mathcal{T}$ to obtain a new grammar $\overline{C}$. It then computes the transitive closure $\overline{G}^{\overline{C}}$. We have the following correctness result:

THEOREM 5.2. Algorithm 2 is sound and precise for $\mathcal{G}_W^R$.

We briefly discuss the complexity of Algorithm 2. The rules in Figure 10 are not recursive, so the number of productions in $\overline{C}$ is a constant multiple of the number of productions in C. Similarly, the graph $\overline{G}$ constructed by Algorithm 2 is a constant mulitple of the size of $\widehat{G}$. The complexity of Algorithm 2 is dominated by the complexity of computing the transitive closure $\overline{G}^{\overline{C}}$. This is $O(|G|^3|C|^3)$ (where $|G|$ is the number of vertices in G, and $|C|$ is the number of terminals and non-terminals in C) [19].

As an example, consider $\mathcal{G}_{W_{\text{alias}}}^{R_{\text{alias}}}$ defined in Section 4. Let

$$\Sigma_{\text{pt}} = \{\text{New}, \text{Assign}\} \cup \{\text{Get}[f'], \text{Put}[f'] \mid f' \in \mathcal{F}\}.$$

As before, we include symbols $\overline{\sigma}$ in Σ_{pt}. Recall that $R_{\text{alias}} = (\text{Assign} + \overline{\text{Assign}})\Sigma_{\text{pt}}^*(\text{Assign} + \overline{\text{Assign}})$. Then $\mathcal{N}(s \xrightarrow{R_{\text{alias}}} f)$ produces the transition graph for the NFA $N = (\{s, n, f\}, \delta, s, f)$, where $n \in \delta(s, \text{Assign})$, $n \in \delta(s, \overline{\text{Assign}})$, $n \in \delta(\sigma, n)$ for all $\sigma \in \Sigma_{\text{pt}}$, $f \in \delta(\text{Assign}, n)$, and $f \in \delta(\overline{\text{Assign}}, n)$. Productions for FlowsTo_n and $\text{FlowsTo}[f']_n$ ($f' \in \mathcal{F}$) are shown in Figure 11.

Consider the code in Figure 1, and suppose the specification for `List.add` is missing, so $W_{\text{alias}} = V_{\texttt{List.add}}$. In Figure 12, we show the following edges that are added by Algorithm 2:

1. This edge represents two edges: FlowsTo_s is added by $\text{FlowsTo}_s \to \text{FlowsTo}\ b$ (Rule 1a), and FlowsTo_n is added by $\text{FlowsTo}_s \to \text{FlowsTo}_n$ (Rule 1f, because $\text{FlowsTo} \to \text{FlowsTo Assign}$).
2. This edge is added by $\text{FlowsTo} \circ \text{Put}[f]_n \to \text{FlowsTo}_n$ (Rule 1f, because $\text{FlowsTo} \circ \text{Put}[f] \to \text{FlowsTo Put}[f]$).
3. This edge represents two edges: $\overline{\text{FlowsTo}}^f$ is added by $\overline{\text{FlowsTo}}^f \to e\ \overline{\text{FlowsTo}}$ (Rule 2a), and $\overline{\text{FlowsTo}}^n$ is added by $\overline{\text{FlowsTo}}^n \to \overline{\text{FlowsTo}}^f$ (Rule 2f, because $\overline{\text{FlowsTo}} \to \overline{\text{Assign}}\ \overline{\text{FlowsTo}}$).
4. This edge is added by $\text{FlowsTo}[f] \to \text{FlowsTo} \circ \text{Put}[f]_n\ \overline{\text{FlowsTo}}^n$ (Rule 4c, because $\text{FlowsTo}[f] \to \text{FlowsTo} \circ \text{Put}[f]\ \overline{\text{FlowsTo}}$).

We have used the production $\text{FlowsTo} \circ \text{Put}[\texttt{val}] \to \text{FlowsTo Put}[\texttt{val}]$ that comes from normalizing the CFG. Once Algorithm 2 adds the edge $\texttt{o}_{\texttt{lat}} \xrightarrow{\text{FlowsTo}[\texttt{val}]} \texttt{o}_{\texttt{list}}$, it will add the edge $\texttt{LOC} \xrightarrow{\text{SrcSink}} \texttt{SMS}$ as a consequence of the productions in C_{taint}.

5.3 Interactive Refinement

We extend Algorithm 2 to find *sufficient assumptions* for the edge $e' = v_{\text{source}} \xrightarrow{T} v_{\text{sink}}$ (recall that sufficient assumptions are encoded as graphs $G \in \mathcal{G}_W^R$ such that $\mathcal{A}(v_{\text{source}}, v_{\text{sink}}) = e' \overset{?}{\in} G^C$). If Algorithm 2 does not produce any source-sink edge e', then we simply return $\widehat{G}$. Otherwise, we record the inputs for each edge produced by Algorithm 2 when computing the closure $\overline{G}^{\overline{C}}$. Recursively searching through the inputs of e', we reconstruct a path $p = v_{\text{source}} \overset{\alpha}{\dashrightarrow} v_{\text{sink}}$ such that $T \overset{*}{\Rightarrow} \alpha$. We record the index of every pair of edges $w_i \xrightarrow{b} v_i \xrightarrow{e} w_i'$ that occurs along p, which we denote by $\mathcal{I}$. Then we add the corresponding graphs $G_i = \mathcal{N}(s_i \xrightarrow{R} f_i)$ to $\widehat{G}$, i.e. $G = \widehat{G} \cup \{G_i \mid i \in \mathcal{I}\}$. The resulting graph G has the desired property $e' \in G^C$.

There may exist multiple paths $p = v_{\text{source}} \overset{\alpha}{\dashrightarrow} v_{\text{sink}}$ such that $T \overset{*}{\Rightarrow} \alpha$, each of which may yield different sufficient assumptions G. We further extend Algorithm 2 to find minimal sufficient assumptions G for e'. Recall that this corresponds to minimizing weight(G), i.e. finding $G \in \mathcal{G}_W^R$ with source-sink path p of minimum weight(p). To do so, we define a weight function on Σ by setting weight(b) = weight$(e) = \frac{1}{2}$, and weight$(\sigma) = 0$ for all other $\sigma \in \Sigma$. This extends to Σ^* by setting weight$(\sigma_1 ... \sigma_k) = \sum_{i=1}^k$ weight(σ_i). Note that for source-sink path $p = v_{\text{source}} \overset{\alpha}{\dashrightarrow} v_{\text{sink}} \in \mathcal{P}(\overline{G})$, weight$(p)$ = weight(α). Consider the following:

DEFINITION 5.3. Let $\overline{G}$ be the graph defined above. The *shortest-path CFL reachability problem* is to return the shortest path $p^* = \arg\min_{p \in \mathcal{P}(\overline{G})}$ weight(α), or return $\emptyset$ if $\mathcal{P}(\overline{G}) = \emptyset$.

Knuth describes a generalization of Djikstra's algorithm to find the shortest string in a context-free grammar [14]. This algorithm generalizes to solving the shortest-path CFL reachability problem: we replace the worklist of edges in [19] with a heap of edges, where the priority of an edge is the weight of its shortest path (see Algorithm 4 in Appendix A for details). By using Algorithm 4 to compute the closure $\overline{G}^{\overline{C}}$, we find the source-sink path $p^* \in \mathcal{P}(\overline{G})$

Algorithm 3 Iterative refinement of results. Here, s and f are fresh vertices.

function ORACLEREFINE($C, \widehat{G}, W, R, v, v', \mathcal{O}$)
  $G_i \leftarrow \mathcal{N}(s \xrightarrow{R} f)$; $\overline{C} \leftarrow \mathcal{T}(G_i, C)$
  repeat
    $\overline{G} \leftarrow$PREPROCESSHELPER($\widehat{G}, W$)
    $p^* \leftarrow$SHORTESTPATH($\overline{C}, \overline{G}, v \xrightarrow{T} v'$)
    for all $w \xrightarrow{b} v \xrightarrow{e} w' \in p^*$ **do**
      $\widehat{G} \leftarrow \widehat{G} \cup \mathcal{O}(w, w')$
      $W \leftarrow W - \{(w, w')\}$
    end for
  until $v \xrightarrow{T} v' \notin \overline{G}^{\overline{C}}$ or weight(p^*) $= 0$
  return $v \xrightarrow{T} v' \stackrel{?}{\in} \overline{G}^{\overline{C}}$
end function

that passes through the fewest possible edges $w \xrightarrow{b} v \xrightarrow{e} w'$. Then the sufficient assumptions G constructed from p^* has minimum weight.

Finally, we describe an algorithm for interactively refining the static analysis results with the help of an auditor. Recall that the graph $\widehat{G}$ is missing some vertices and edges from G^*. Suppose we can query an oracle to obtain information about G^*:

DEFINITION 5.4. We say $\mathcal{O}$ is an *oracle* for G if for every $(w, w') \in W$, $\mathcal{O}(w, w') = G^*_{w,w'}$.

We use a human auditor as an oracle $\mathcal{O}$. On input (w, w'), the auditor examines the library documentation and return the true specification $G^*_{w,w'}$. The problem is to produce static analysis results that are sound and precise with respect to G^*, while making as few queries $\mathcal{O}(w, w')$ as possible.

Algorithm 3 solves this problem. It obtains the shortest path $p^* \in \mathcal{P}(\overline{G})$ for the edge $e' = v_{\text{source}} \xrightarrow{T} v_{\text{sink}}$ by calling $p^* \leftarrow$ SHORTESTPATH($\overline{C}, \overline{G}, e'$). Then the algorithm replaces every edge $w \xrightarrow{b} v \xrightarrow{e} w'$ in p^* with $\mathcal{O}(w, w')$. Algorithm 3 repeats this process until either weight(p^*) $= 0$, or until $\mathcal{P}(G') = \emptyset$. In the former case, the path p^* does not contain any symbols b or e, i.e. p^* does not pass through any potentially missing specifications. This proves that $p^* \in \widehat{G}$, i.e. $e' \in \widehat{G}^C \subseteq (G^*)^C$. In the latter case, because Algorithm 2 is sound, it only returns $p^* = \emptyset$ if there does not exist *any* $G \in \mathcal{G}^R_W$ such that $e' \in G$. Since we have assumed that $G^* \in \mathcal{G}^R_W$, this proves that $e' \notin (G^*)^C$. Therefore:

THEOREM 5.5. Algorithm 3 computes $v_{\text{source}} \xrightarrow{T} v_{\text{sink}} \stackrel{?}{\in} (G^*)^C$.

6. Implementation

We have implemented the system described in Figure 8 for Java, and specifically for the Android framework. A number of extensions are required beyond the essentials we have described, but these extensions do not introduce any new ideas. For example, we include rules for primitive variables that are essentially rules for reference variables without fields. Our current implementation does not consider specifications involving static fields; as discussed in Section 4, in our experience such specifications lead to many false positives as there are few constraints on the possible flows between static variables, and in practice there are few flows among them.

Prior to this work, we had manually written many specifications over a period of more than one year. These *baseline specifications* $\mathcal{S}$ cover 176 Android library classes, including a number of taint sources (location, device ID, SIM data, contacts, and calendar data) and a number of sinks (network sockets, SMS messages, and user settings modifications). This baseline also includes specifications for important container objects including ArrayList, LinkedList, and HashMap. Since our focus in on the long tail of missing specifications, we bootstrap our implementation of the specification inference framework with these baseline specifications.

Spec. Type	Flow	Alias
# Android apps	179	156
Total Correct Specifications Proposed	486	35
Total Specifications Proposed	1122	63
Overall Accuracy	0.433	0.556
Average # Specifications Proposed	23.8	0.813
Accuracy of Random Sample	0.140	N/A

Figure 13. Statistics on inferred Android library specifications.

In our implementation, program fact extraction is performed using the Chord platform [20] modified to work with the Jimple intermediate representation provided by Soot [30]. In order to improve precision of our analysis, we extend the points-to rules (Rules 10-13 in Figure 5) so that they are context sensitive—more precisely, we use a 1-CFA points-to analysis. Additionally, because Java is type safe, we use type filters in the points-to analysis (i.e., a reference of type T can only point to an object of type T' if T' is a subtype of T). Our solver detects and discards points-to edges that are not consistent with the type constraints of the program. Our shortest-path CFL solver is based on the CFL solver in [19], modified with the algorithm from [14] to compute shortest paths; details are given in Appendix A.

Within our specification inference framework we implement both flow $(R_{\text{flow}}, W_{\text{flow}})$ and alias $(R_{\text{alias}}, W_{\text{alias}})$ specifications. For each inferred specification, we manually reference the Android library documentation to determine if the specification is correct, and rerun the analysis with the updated specifications. We repeat this process until no new specifications are inferred.

When inferring alias specifications, the large size of the sound points-to analysis makes it difficult to scale the inference algorithm. We implemented a demand-driven optimization. First, we perform the entire analysis using BDDBDDB [31] as the solver. However, BDDBDDB cannot compute shortest paths. Instead, we use the results from BDDBDDB to prune irrelevant edges (i.e., edges that do not contribute to a source-sink path) from $\overline{G}$. Finally, we recompute the analysis using our shortest-path CFL solver.

Ideally we would infer alias and flow specifications simultaneously. However, worst-case flow specifications introduce a large number of incorrect taint flows, causing the demand-driven optimization to fail to eliminate enough edges for the specification inference algorithm to scale for some benchmarks. As a result, we perform the two analyses separately in our experiments. Nevertheless, inferring alias specifications alone is already sufficient to be sound with respect to finding explicit taint flows.

7. Experimental Results

We ran our tool on a corpus of 179 Android apps. Our results are for the optimized version of our specification inference algorithm, i.e. Algorithm 2, since preliminary experiments with Algorithm 1 did not scale even to apps of moderate size. The running time for one iteration of Algorithm 3 is plotted in Figure 16(c). The flow specification inference algorithm ran on all 179 apps, running in fewer than 10 seconds per iteration on average for most apps, which is fast enough to allow a human auditor to interactively run the analysis. The alias specification inference algorithm successfully ran on 156 apps. The worst-case assumptions cause a substantial increase in the points-to relation size (see Figure 16(d)), which proved to be too large on the remaining 23 apps. Still, inferring alias specifications runs in under 100 seconds for most apps with up to 100,000 lines of Jimple code.

App Name	Jimple LOC	# Spec.	# Critical Spec.	Total # Spec.	Rounds	Accuracy	Flows	Run Time (s)	Spec. Type
411524	497178	43	20	6478	5	0.5116	42	178.6	Flow
APG-M	471943	20	9	5553	5	0.5	16	22.30	Flow
browser	421026	139	24	10361	16	0.2662	48	607.3	Flow
0C2B78	395053	244	11	4726	78	0.1066	222	10.38	Flow
highrail	265875	142	15	4242	18	0.1690	64	12.20	Flow
iwz	263014	38	17	4937	5	0.5789	38	14.24	Flow
ce667f	193518	48	26	4388	5	0.6042	44	10.38	Flow
ConnectBot	136800	4	4	3454	2	1.0	2	2.357	Flow
yaaic	109286	0	0	5440	1	N/A	0	3.769	Flow
tomdroid	44478	14	5	3029	6	0.3571	2	1.607	Flow
highrail	265875	2	2	5167	2	1.0	1	9623	Alias
andmj	239227	0	0	5229	1	N/A	0	1555	Alias
ce667f	193518	0	0	5509	1	N/A	0	3234	Alias
ConnectBot	136800	0	0	3293	1	N/A	0	3193	Alias
05ed92	134235	1	1	12632	2	1.0	3	2254	Alias
SMSBot	134230	15	3	4693	7	0.467	2	284.3	Alias
yaaic	109286	0	0	4161	1	N/A	0	816.1	Alias
ca70f4	81974	3	0	3307	3	0.0	0	45.57	Alias
tomdroid	44478	0	0	3697	1	N/A	0	68.154	Alias
a1d58b	41682	2	1	2285	3	0.5	1	7.847	Alias

Figure 14. Specification inference results on large Android apps. *Critical specifications* are both correct and lie along a true source-sink path. "Total # specifications" is the number of worst-case specifications, "Rounds" is the number of iterations in Algorithm 3, "Accuracy" is the proportion of proposed specifications that are correct, and "Flows" is the number of new flows discovered (accuracy is N/A if no specifications are inferred).

Class	Method Signature	Specification	Spec. Type
com.google.android.maps.GeoPoint	int getLatitudeE6()	(`this`, `return`)	Flow
java.lang.Double	double parseDouble(java.lang.String)	(`arg1`, `return`)	Flow
org.json.JSONObject	org.json.JSONObject getJSONObject(java.lang.String)	(`this`, `return`)	Alias
android.telephony.gsm.SmsMessage	java.lang.String getMessageBody()	(`this`, `return`)	Alias
android.content.ContentValues	void put(java.lang.String, java.lang.String)	(`arg2`, `this`)	Alias

Figure 15. Sample of inferred specifications. "Specification" is the pair $(w, w') \in W$ returned by Algorithm 3.

Results for selected apps, including the largest four that ran successfully for each analysis, are shown in Figure 14. We show the number of specifications proposed by our tool (# Spec.), along with the number of worst-case specifications (Total # Spec.). Our tool may propose specifications that are correct (i.e., represent valid paths throughs library methods), but do not contribute to a correct source-sink path in the program. Therefore, we also show the number of specifications that are both correct *and* contribute to a true source-sink flow (# Critical Spec.), which is a lower bound for the number of specifications proposed. We call such specifications *critical specifications*, because they are the specifications that an auditor must examine in order to find all source-sink flows. The number of inferred specifications is plotted as black circles in Figure 16(a) and (b), along with the number of inferred specifications that are correct (plotted as red triangles), and the number of critical specifications (plotted as blue diamonds). For readability, (a) is a log-log plot, and the x-axis of (b) is log-scale. The accuracy of the aggregated specifications are shown in Figure 13. Note that the accuracy is directly correlated with the manual labor required by the oracle: higher accuracy means that the oracle will have to examine fewer incorrect specifications.

7.1 Specification Inference Accuracy

Our first experiment demonstrates the accuracy of the specifications inferred. For each app, we ran our inference algorithm with the baseline specifications $\mathcal{S}$. The inferred alias specifications are very accurate, in part because of type filters. We show some examples of inferred specifications in Figure 15.

We compare our results to randomly chosen specifications. We randomly chose 50 possible flow specifications in the following way: randomly choose a method, randomly choose a pair of formal parameters v and v' (or a formal parameter v and the formal return value v'), and propose the flow specification $v \xrightarrow{\text{RefRef}} v'$. The accuracy of a specification randomly chosen in this way is only 0.140, whereas the overall accuracy of the specifications inferred by our tool is 0.433.

The number of specifications proposed, which grows roughly linearly with app size, is very manageable. It is usually a small multiple of the number of critical specifications. For alias specification inference, each app produced fewer than 20 proposals, each of which could be checked in under a minute. Significantly more flow specifications are inferred, but these are even faster to check. All but five of the apps required fewer than 100 proposals. In total, the tool helped discover hundreds of new specifications and flows, a task that we estimate would have taken weeks without the tool.

7.2 Specification Aggregation

Our second experiment demonstrates how our tool can be used to quickly build a useful collection of library specifications. Consider analyzing the apps in some arbitrary order and aggregating specifications along the way; that is, the ith app is analyzed using all of the correct specifications discovered in analyzing the first $i-1$ apps. Intuitively, the most frequently used methods should have their specifications discovered relatively early in the process and we should subsequently benefit from already having those specifications and not needing to infer them again.

Figure 16(e) shows the proportion of new specifications proposed by Algorithm 3 with aggregation to the number of specifications proposed when Algorithm 3 starts from the baseline. The red, triangle series shows, for each app, the percentage of new specifica-

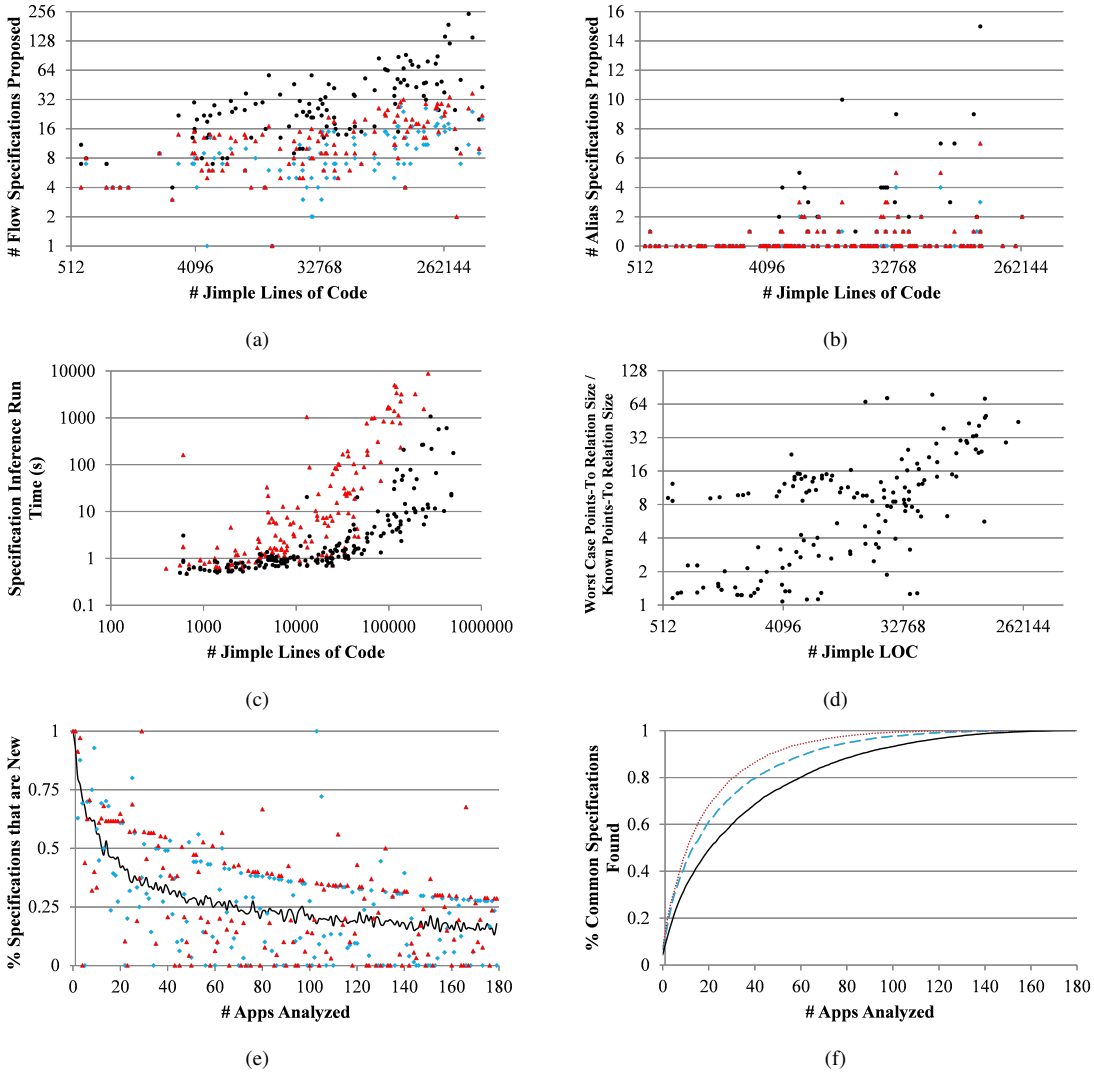

Figure 16. For (a) flow and (b) alias: # specifications proposed (black, circle), # correct (red, triangle), and # critical (blue, diamond). (c) Run time of the flow (black, circle) and alias (red, triangle) specification inference algorithms. (d) Ratio of worst-case points-to relation size to known points-to relation size. (e) Ratio of # specifications with aggregation to # specifications from baseline, averaged over 100 random orders (black line), and for two different random orders (red triangle, blue diamond). (f) Proportion of common specifications proposed, for $c = 2$ (black, solid), 3 (blue, dashed), and 4 (red, dotted), averaged over 100 random orders.

tions that the auditor must examine. After the 100th app, more than 2/3 of the needed specifications are already known, and for many apps no new specifications are needed. The blue, diamond series shows the same effect from processing the apps in a different random order. The black line shows the average number of new specifications over 100 such runs (varying the order of the apps each time); as can be seen, the auditor's workload for a new app with aggregation approaches about 20% of that without aggregation. The red and blue series give a sense of the considerable variance, but the overall trend is clear: regardless of chosen order, the proportion of new specifications quickly becomes small and the auditor only does a fraction of the work compared to starting from the baseline. The required work would drop further after processing more apps.

Figure 16(f) shows the proportion of *common* specifications that are identified after analyzing each number of apps. We say a specification is common if it is proposed by the tool for at least $c \in \{2, 3, 4\}$ apps. In this graph, the black, solid curve corresponds to $c = 2$; the blue, dashed curve corresponds to $c = 3$; and the red, dotted curve corresponds to $c = 4$. Each line shows the average proportion over 100 random permutations. The tool quickly picks up a large fraction of the common specifications, reaching more than 82% after just a quarter of the apps have been analyzed in the case $c = 3$.

7.3 Verification

We ran Algorithm 3 to termination, i.e., until no new specifications for missing parts of the program could add any more taint flows, which means that the remaining taint flows all occur in the original graph $\widehat{G}$. In the case of alias inference, this also proves that no additional explicit taint flows can occur. Figure 16(b) shows the number of specifications that had to be checked by an auditor to completely verify the absence of explicit taint flows in an app. This number is very reasonable (at most 15), showing that the tool makes verification of large apps practical. In the case of flow specification inference, taint flows due to missing alias specifications can still

occur, and verification requires that the auditor supply all relevant alias specifications. In practice, this analysis still discovers many library methods that need alias specifications, since taint often flows forwards through these methods. Thus our tool finds many taint flows even if alias specifications are missing.

There are currently two primary limitations to our tool. One we have already discussed: inferring both alias and flow specifications simultaneously is too expensive for our tool on some apps. The second is that while we can infer missing flow and alias specifications, we still require a complete list of the possible sources and sinks in the program to be able to find flows at all. While manually annotating sources and sinks is a much easier problem (by orders of magnitude) than finding flows, it would still be useful to consider how to provide automatic assistance in discovering sources and sinks in large apps.

As can be seen from the total number of potential specifications shown in Figure 14, without our tool an auditor would have to examine a huge number of potential specifications. Even if many of these can be easily eliminated, our experience has been that without the aid of our tool, performing verification on moderately sized apps can take hours or even days, and performing verification on large apps is almost impossible.

8. Related Work

Sound analysis with missing specifications. There has been previous work that shares our goal of inferring specifications for verification based on abductive inference [19]. There are several differences in the two approaches. First, we handle the general class of CFL reachability problems, whereas [34] addresses standard graph reachability problems. Second, abductive inference is a very general tool and solving abductive inference problems is NP-hard. Our algorithm, which is tailored to specification inference, runs in polynomial time. As a result, our system appears to scale considerably better, and we are able to conduct a much larger experiment on many more apps than [34]. There has also been prior work on sound call graph generation [2]. Their work constructs a placeholder library that exhibits every possible behavior that can affect the call graph. Our technique, in addition to being more general, actually proposes library specifications and allows an auditor to interactively refine analysis results.

Specification inference. There has been work using data mining to infer specifications [3, 4, 7, 15, 16, 21, 22, 25, 33]. Several of these techniques use dynamic traces to propose specifications [4, 33], while others apply domain knowledge and statically infer specifications from the source code [7, 15, 16, 22, 25]. Our tool differs in that it applies to CFL reachability problems. Furthermore, our tool produces sound results, which enables interactive verification.

CFL reachability for program analyses. A large number of program analyses have been expressed as CFL reachability problems, for example points-to analysis [26, 28], many set constraint problems [11, 13, 19], various interprocedural analyses [23, 24], and type qualifier inference [10]. Our work makes these techniques more applicable for whole-program analysis by providing a practical and sound approach to dealing with missing or hard-to-analyze portions of the program. Our work makes use of ideas for combining context-free reachability with additional regular language properties such as [12].

Applications of program analysis to security. Work on taint analysis for Android includes SCanDroid [9], which statically tracks taint flows between applications, TaintDroid [8], which is a dynamic system that performs real time monitoring, and FlowDroid [5], which uses static analysis to find information leaks. Static analysis has also been applied to finding vulnerabilities in web applications [17, 27, 29, 32].

Algorithm 4 Algorithm for computing shortest-path CFL reachability.

function SHORTESTPATH(G, C, e)
  $G^C \leftarrow$GRAPH(); $I \leftarrow$MAP(); $H \leftarrow$HEAP()
  for all $v \xrightarrow{\sigma} v' \in G$ **do**
    H.UPDATE($v \xrightarrow{\sigma} v'$, weight($\sigma$))
    $I[v \xrightarrow{\sigma} v'] \leftarrow \emptyset$
  end for
  while $\neg H$.EMPTY() **do**
    $[v \xrightarrow{A} v', P_{\text{cur}}] \leftarrow H$.DELETEMIN()
    $G^C \leftarrow G^C \cup \{v \xrightarrow{A} v'\}$
    for all $D \rightarrow AB \in C$ **do**
      for all $v' \xrightarrow{B} v'' \in G^C$ **do**
        $P_{\text{new}} \leftarrow P_{\text{cur}} + H$.PRIORITY($v' \xrightarrow{B} v''$)
        if $P_{\text{new}} < H$.PRIORITY($v \xrightarrow{D} v''$) **then**
          H.UPDATE($v \xrightarrow{D} v''$, P_{new})
          $I[v \xrightarrow{D} v''] \leftarrow [v \xrightarrow{A} v', v' \xrightarrow{B} v'']$
        end if
      end for
    end for
    for all $D \rightarrow BA \in C$ **do**
      for all $v'' \xrightarrow{B} v \in G^C$ **do**
        $P_{\text{new}} \leftarrow H$.PRIORITY($v'' \xrightarrow{B} v$)+$P_{\text{cur}}$
        if $P_{\text{new}} < H$.PRIORITY($v'' \xrightarrow{D} v'$) **then**
          H.UPDATE($v'' \xrightarrow{D} v'$, P_{new})
          $I[v'' \xrightarrow{D} v'] \leftarrow [v'' \xrightarrow{B} v, v \xrightarrow{A} v']$
        end if
      end for
    end for
    for all $B \rightarrow A \in C$ **do**
      if $P_{\text{cur}} < H$.PRIORITY($v \xrightarrow{B} v'$) **then**
        H.UPDATE($v \xrightarrow{B} v'$, P_{cur})
        $I[v \xrightarrow{B} v'] \leftarrow [v \xrightarrow{A} v']$
      end if
    end for
  end while
  if $e \in I$ **then**
    return GETPATH(I, e)
  end if
  return $\emptyset$
end function
function GETPATH(I, e)
  if $I[e] = \emptyset$ **then**
    return $[e]$
  end if
  $[e_1, ..., e_k] \leftarrow I[e]$
  return GETPATH(e_1)+...+GETPATH(e_k)
end function

9. Conclusions

We have developed a general framework that applies to any program analysis formulated as a CFL reachability problem. Our framework allows us to perform a sound analysis by inferring missing specifications, and furthermore allows an auditor to interactively refine the results. We have demonstrated the quality of the specifications inferred by our tool on a corpus of 179 real-world Android apps. Our results show that our tool can both help build large collections of specifications very efficiently, and make it practical for an auditor to perform verification.

A. Shortest-Path CFL Reachability

The shortest-path algorithm in Algorithm 4 generalizes Knuth's algorithm for finding shortest strings in CFLs to computing shortest-path CFL reachability. Essentially, Knuth's algorithm [14] builds

on an algorithm for determining emptiness of a context-free grammar: it adds a heap that keeps track of the shortest sequence of terminals that can be derived from each non-terminal symbol. Similarly, Algorithm 4 generalizes the algorithm for computing CFL reachability described in [19]. In the pseudocode, arrays are denoted as $[x_1, ..., x_k]$, and addition of arrays is defined to be $[x_1, ..., x_k] + [y_1, ..., y_h] = [x_1, ..., x_k, y_1, ..., y_h]$.

We introduce a heap H that keeps track of the shortest path for each edge $v \xrightarrow{A} v'$. The current priority of the edge $v \xrightarrow{A} v'$ is the length of the current shortest path. If a shorter path is found, then the heap is updated with the new path and the new priority. At every iteration of the algorithm, the lowest priority edge $v \xrightarrow{A} v'$ (with priority P_{cur}) is removed from the heap, added to G^C, and then processed. Note that once this happens, there can be no way of producing $v \xrightarrow{A} v'$ with lower priority: every subsequent edge removed from H must have priority at least P_{cur}, so any edge produced while processing such an edge must also have priority at least P_{cur}. Since every possible way of producing $v \xrightarrow{A} v'$ is considered, the shortest path is correctly identified.

The heap H supports the following operations: UPDATE(e) updates the priority of edge e (and adds e to H if $e \notin H$), EMPTY() returns true if the heap contains no edges, PRIORITY(e) returns the current priority of edge e, and DELETEMIN() removes the lowest priority element in the heap and returns it (along with its current priority). We assume that PRIORITY returns ∞ for edges not yet added to H, and 0 for edges already removed from H. The complexity of Algorithm 4 is $O(|G|^3|C|^3(\log|G| + \log|C|))$ because of the additional cost of updating the heap (as before, $|G|$ is the number of vertices in G, and $|C|$ is the number of terminals and non-terminals in C).

The shortest path itself is stored in a map I, which keeps track of the edges $[e_1, ..., e_k]$ (where $k \in \{0, 1, 2\}$) used to produce $v \xrightarrow{A} v'$. The shortest path itself is reconstructed by recursively querying the shortest path for each edge in $I[v \xrightarrow{A} v']$.

Algorithm 4 does not handle edges labeled with the empty string ϵ, or productions $A \to \epsilon$. In order to handle the former, our solver introduces a fresh terminal symbol $\widehat{\epsilon}$, replaces every edge $v \xrightarrow{\epsilon} v'$ with $v \xrightarrow{\widehat{\epsilon}} v'$, and adds productions $A \to \widehat{\epsilon}A$ and $A \to A\widehat{\epsilon}$ for every non-terminal A in the input grammar. The latter is handled by adding self loops $v \xrightarrow{A} v$ for every $v \in V$ and every $A \in U$ such that $A \to \epsilon \in C$ (see [19]).

Acknowledgments

This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, P. Hawkins. An overview of the Saturn project. In *PASTE*, 43-48, 2007.

[2] K. Ali, O. Lhoták. Averroes: whole-program analysis without the whole program. In *ECOOP*, 2013.

[3] R. Alur, P. Černý, P. Madhusudan, W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, 2005.

[4] G. Ammons, R. Bodík, J. Larus. Mining specifications. In *POPL*, 2002.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, P. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, 2014.

[6] T. Ball, S. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[7] N. Beckman, A. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, 2011.

[8] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, A. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[9] A. P. Fuchs, A. Chaudhuri, J. S. Foster. SCanDroid: automated security certification of Android applications. In *IEEE Symposium on Security and Privacy*, 2010.

[10] D. Greenfieldboyce, J. S. Foster. Type qualifier inference in Java. In *OOPSLA*, 2007.

[11] J. Kodumal, A. Aiken. Banshee: a scalable constraint-based analysis toolkit. In *SAS*, 2005.

[12] J. Kodumal, A. Aiken. Regularly annotated set constraints. In *PLDI*, 2007.

[13] J. Kodumal, A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, 2004.

[14] D. Knuth. A generalization of Dijkstra's algorithm. In *Information Processing Letters*, 6(1):1-5, 1977.

[15] T. Kremenek, P. Twohey, G. Back, A. Ng, D. Engler. From uncertainty to belief: inferring the specification within. In *OSDI*, 2006.

[16] B. Livshits, A. V. Nori, S. K. Rajamani, A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.

[17] B. Livshits, M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[18] B. Livshits, M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *FSE*, 2003.

[19] D. Melski, T. Reps. Interconvertibility of a class of set constraints and context-free language reachability. In *Theoretical Computer Science*, 248(1):29-98, 2000.

[20] M. Naik, A. Aiken, J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[21] J. W. Nimmer, M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, 2002.

[22] M. K. Ramanathan, A. Grama, S. Jagannathan. Static specification inference using predicate mining. In *PLDI*, 2007.

[23] T. Reps. Program analysis via graph reachability. In *ILPS*, 1997.

[24] T. Reps, S. Horwitz, M. Sagiv. Precise interprocedural data flow analysis via graph reachability. In *POPL*, 1995.

[25] S. Shoham, E. Yahav, S. Fink, M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA*, 2007.

[26] M. Sridharan, D. Gopan, L. Shan, R. Bodik. Demand-driven points-to analysis for Java. In *OOPSLA*, 2005.

[27] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, R. Berg. F4F: taint analysis of framework-based web applications. In *OOPSLA*, 2011.

[28] M. Sridharan, R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, 2006.

[29] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

[30] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan. Soot: a Java bytecode optimization framework. In *CASCON*, 1999.

[31] J. Whaley, M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *OOPSLA*, 2004.

[32] Y. Xie, A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[33] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.

[34] H. Zhu, T. Dillig, I. Dillig. Automated inference of library specifications for source-sink property verification. In *APLAS*, 2013.