BEYOND DEDUCTIVE INFERENCE IN PROGRAM ANALYSIS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Osbert Bastani
November 2017

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(John Mitchell)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Percy Liang)

Approved for the Stanford University Committee on Graduate Studies

Abstract

As software becomes an increasingly critical part of our world, ensuring the safety and correctness of software becomes increasingly important. The goal of program analysis is to automatically test and verify programs to find bugs and improve software quality. However, in large software systems, there are inevitably parts of the system that are too difficult for the program analysis to handle. To scale to such systems, program analysis tools typically require *specifications* that summarize the behavior of these hard-to-analyze portions of the system. However, these specifications are often costly to produce, thus limiting the adoption of program analysis tools in practice.

In this thesis, we propose algorithms for *inferring* specifications. We design algorithms that extend the capabilities of existing specification inference algorithms to far more complex and challenging settings compared to existing work. In particular, our algorithms can infer specifications that have recursive structure, such as that of a finite-state automaton or even a context-free grammar. We implement our specification inference algorithms, and show that they can infer specifications that can be used by downstream program analyses that verify or test programs.

Acknowledgments

I thank my advisor Alex Aiken, who has taught me so much during my five years at Stanford, not only about program analysis, but about how to do good research. He has always found time to support and guide me despite his many obligations, helping me develop my ideas, editing my papers, and polishing my talks. More importantly, he has always inspired me to be ambitious in my work, and to tackle challenging problems where the path forward is not always in sight.

I thank Saswat Anand, for guiding me through the early years of my Ph.D. and teaching me so much about program analysis. I am still surprised when I realize that some of the insights he taught me are not so well known as they should be. I thank Percy Liang for teaching me everything I know about machine learning theory, and for always taking the time to think through my ideas, often clarifying and improving them in the process. I thank Isil Dillig, who took the time to give me much valuable advice on applying for academic jobs. I thank John Mitchell, who guided me during the first year of my Ph.D., and introduced me to the STAMP project. I also thank Clark Barrett and Mykel Kochenderfer for taking the time to serve on my defense committee.

I thank all my friends at Stanford—Tony Feng, Edward Gan, Carolyn Kim, Sindy Li, Armin Pourshafeie, Bharath Ramsundar, Rick Song, Jacob Steinhardt, Alex Zhai, and Joy Zhang—who have not only made the last five years incredibly fun, but also sat through long practice talks. I thank all my collaborators and colleagues—Lazaro Clapp, Yu Feng, Stefan Heule, Leo Lampropoulos, Ruben Martins, Patrick Mutchler, Aditya Nori, Manolis Papadakis, Rahul Sharma, Pratiksha Thaker, and Dimitrios Vytiniotis—who have made my Ph.D. enjoyable and much more productive.

I thank my parents for their love and support throughout my life, always ready to help me face whatever challenges I come across. Because they have always been so supportive of anything I do, I have never been afraid to take risks both in my research and in life. I thank my brother for being a great friend for almost my entire life. Finally, above all, I thank my wife and best friend, Hamsa Bastani, for her constant love and unwavering support, and for the countless hours she has spent listening to all my crazy ideas and giving her honest opinion, talking through what I need to do, motivating me to finish my work, proofreading everything I write, and cheering me up and helping me get back on my feet when things do not go so smoothly. I am eternally grateful that she made the Ph.D. journey with me, and that we will be embarking upon our next journey together.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Program analysis has become a vital part of the software development life-cycle due to its effectiveness at finding bugs and security vulnerabilities in large software systems [38, 16, 40, 103, 156, 106, 30, 21, 63, 33, 159]. Despite this progress, the use of program analysis tools has achieved limited adoption beyond relatively shallow bug detection methods and the application of verification techniques to some of the most safety critical systems.

An important obstacle for achieving widespread adoption is that there are often large upfront costs to using program analysis tools on a new codebase [21, 18]. Such costs arise because large systems typically contain components that cannot be analyzed by the tool, including (i) calls to functions implemented as native code, (ii) use of dynamic programming language features such as Java reflection, or (iii) dynamically loaded code. In fact, in our experience, for large systems it is unusual if any of these situations does *not* frequently occur. However, many program analyses assume that the entire program's source code is available for analysis.

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 [161]. An alternative, more pragmatic solution is to require that the human user (whom we refer to as the *analyst*) manually write *specifications* (also known as *summaries*, *annotations*, or *models*) describing the relevant behavior of the missing code so that it 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.

However, specifications can be time consuming for the analyst to provide. Therefore, a number of approaches to *inferring* specifications have been proposed. By the nature of the problem, these algorithms must leverage inference techniques beyond deductive inference used in a program analysis—otherwise, they would simply be part of the program analysis. Typically, these approaches depend on one of two kinds of inference techniques. First, they can use *inductive inference* to generalize a set of observations to a more general specification; these approaches can either use

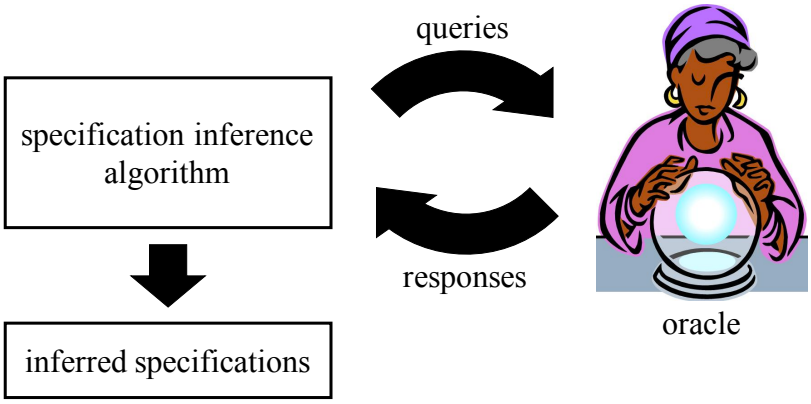


Figure 1.1: We design specifications inference algorithms that interact with an oracle to infer specifications.

observations from dynamic executions [9, 107, 8, 157, 128, 57, 126, 162, 112, 163, 19] or static information [86, 116, 131, 95, 20, 117, 22]. Alternatively, they can use *abductive inference* to identify the simplest specification that explains a set of observations; these approaches typically interact with a human analyst to refine the inferred specification [43, 161, 18, 5].

Thus far, specification inference algorithms have largely been limited to local properties of code, such as loop invariants [107, 128, 126], type annotations [20, 117], or simple interface specifications [8, 86, 95, 22]. While these tools have proven useful, more complex specifications are required for several widely-used program analyses, in particular, heap specifications are required for static points-to analysis [18], and program input grammars are required for grammar-based fuzzing [61, 19]. These specifications are differentiated by the presence of recursive or hierarchical structures. For example, a piece of code can exhibit complex aliasing patterns by storing objects in nested data structures, making it difficult for the specification inference algorithm to resolve how each part of the code contributes to the observed aliasing relation. Similarly, programs often have complex input languages defined by regular expressions or even context-free grammars.

In this thesis, we propose novel specification inference algorithms designed to infer specifications with complex hierarchical structure. At a high level, all the algorithms we propose use an active learning strategy where they interact with an *oracle* to obtain information about the true specifications. As the algorithm obtains new information, it adaptively selects new queries to ask the oracle to further prune the search space. The oracle can either be the human analyst using the analysis, or concrete executions of the program being analyzed. Figure 1.1 summarizes this high level approach.

In the first part of this thesis, we devise new algorithms that leverages interaction with the human analyst to infer specifications. We focus on inferring specifications for interprocedural static analyses, which have many practical applications such as automated software verification [16, 40, 33], bug finding [38, 3], and taint analysis [96, 155].

As a motivating example, consider the problem of deciding whether a given Android app is malicious. Typical Android malware exhibit behaviors including theft of contact information, sending SMS text messages to premium phone numbers, and unauthorized location tracking. Many of the malicious behaviors 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. As a consequence, information flow analysis has been proposed as a way of identifying Android malware [55, 51, 46, 14, 52].

In principle, a standard static information flow analysis, which we describe in Chapter 2, can identify such malware. Unfortunately, the Android framework (written in Java, with calls to native code) is a classic example of how system libraries cause difficulties for static analyses. Android framework 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 method `Parcel.nativeWriteLong` in the Android framework (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.

In Chapter 3, we propose a novel algorithm that infers specifications by interacting with the analyst. Our proposed inference algorithm formulates and issues queries to the analyst. These queries ask for the correct specification for a given function, which the analyst provides. The key advantage behind this approach is that it enables the analyst to implement specifications in a demand driven fashion. As we show in our evaluation, the number of queries issued to the analyst is fairly small, in particular, three orders of magnitude smaller than the number of specifications that must be written using a more naïve approach. While our problem formulation and algorithm are general and can be applied to any static analysis, we show that our framework can be used to infer particularly complex *points-to specifications* that describe the heap effects of calling functions.

Next, in Chapter 4, we consider the setting where the human responding to queries may be adversarial. In particular, the previous approach places the burden of writing specifications entirely on the analyst. This approach is feasible for writing specifications for code such as the Android framework, which the analyst can understand simply by reading the documentation. However, it may not be feasible if the missing code is in the Android app itself, since documentation may be unavailable and malicious code may be heavily obfuscated to prevent detection. Thus, we may want the developer to respond to the queries issued by our specification inference algorithm. To account for the possibility that the response may be adversarial, we then instrument the code to enforce that the response holds true. For example, if the developer claims that a certain line in the program

is unreachable, then we instrument the Android app to terminate if that line is ever about to be executed.

In the second part of this thesis, we devise new algorithms for inferring specifications that leverage observations from executions of the code. To infer more complex program properties, our algorithms employ *active learning* strategies, i.e., they adaptively choose the program inputs used in the executions as the search over the space of possible specifications proceeds. In Chapter 5, we show how this approach can be used to infer points-to specifications. In particular, we demonstrate that inferring points-to specifications can be reduced to a *language learning problem* [12, 110], which we solve using an active learning variant of an existing language learning algorithm.

Finally, in Chapter 6, we show that active learning strategies can be effective for inferring an even more complex class of program properties, namely, program input grammars. Documentation of program input formats, if available in a machine-readable form, can significantly aid many software analysis tools; our particular focus is on improving grammar-based fuzzers [97, 61, 75]. However, such documentation is often poor; for example, the specifications of Flex [145] and Bison [59] input syntaxes are limited to informal documentation. Even when detailed specifications are available, they are often not in a machine-readable form; for example, the specification for ECMAScript 6 syntax is 20 pages in Annex A of [45], and the specification for Java class files is 268 pages in Chapter 4 of [111].

The problem of automatically inferring program input grammars is particularly challenging. Program input grammars often exhibit complex recursive structure, especially when the program input language is context-free. However, the feedback from executing a program on a given input only reveals one bit of information—namely, whether that particular input is valid. We devise an algorithm combining ideas from program synthesis with an active learning strategy that guides the search over the space of possible grammars. We show how we can use our algorithm to perform grammar-based fuzzing in a way that covers 2× as many new lines of code compared to two baseline fuzzers.

The work in this dissertation was in collaboration with Alex Aiken, Saswat Anand, Percy Liang, and Rahul Sharma. The ideas discussed here appear in the conference papers [18, 17, 19].

Chapter 2

Background

In this chapter, we describe a static analysis for finding information flows. Our analysis performs context-free language (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. We do not claim that this design is novel, but to the best of our knowledge this approach is not well-documented in the literature, so we describe it here in detail.

We describe a standard flow-, path-, context-, and object-insensitive static analysis for computing *explicit* information flow analysis (also known as *taint analysis*) [123]. while our static analysis is context- and object-insensitive, it straightforwardly generalizes to an context- and object-sensitive analysis by using cloning [152]. The goal of information flow analysis is to determine whether sensitive information (e.g., location) is leaked outside of the system (e.g., to the Internet). It does so by computing whether specified *sink variables* in the program (e.g., the parameter of `sendHTTP`) may depend on specified *source variables* in the program (e.g., the return value of `getLocation`). If such a dependence exists, we say the parameter of `sendHTTP` is *tainted* by location information. Our focus is on a *sound* static analysis—if such a dependence exists, then it is computed by the analysis, but false positive information flows may also be reported.

Intuitively, explicit information flows only track information through data flows, ignoring information flows due to control flow depending on sensitive values. More precisely, explicit information flow analysis computes information flows assuming control flow is replaced with random (or nondeterministic) choices independent of the program state. We focus on explicit information flow analysis because it finds substantially fewer false positives compared to implicit information flow analysis, and furthermore is known to be useful for a wide range of tasks including finding many security vulnerabilities in web applications [96, 146, 135] and detecting most current Android malware [55, 51, 14]. Nevertheless, the techniques we develop generalize to any static analysis that can be expressed as a CFL reachability problem, which includes formulations of implicit information

flow analysis.

To analyze a program P, our static analysis uses *specifications* serving two distinct purposes. First, it requires a specification that encodes the property to be verified; in particular, this specification encodes which data in the program are sensitive and which functions in the program leak information out of the system. For simplicity, we focus on *safety properties*, which are specifications that assert that certain bad events (e.g., an information leak, or information corruption) never occur.

Second, our analysis consumes specifications that describe the semantics of functions. For example, in the case of information flow analysis, the specification for the `toString` method in the `Double` class might say something to the effect of

> In a call to `toString`, if the receiver is tainted, then the return value may be tainted.

Then, the static analysis would use this specification to propagate taint through the `toString` method, eliminating the need to analyze this method. More precisely, if the analyst provides a specification for a function, then the static analysis omits analyzing that function and instead assumes that the provided specification encodes the relevant semantics of the function.

Whereas the first kind of specification must be provided by the analyst, the second kind of specification is often optional—specifications are primarily designed to improve the scalability and precision of the static analysis. However, in cases where the function is missing (e.g., implemented in native code that cannot be analyzed, or dynamically loaded), then specifications are necessary as well. Finally, as we discuss below, specifications can also enable us to describe implicit flows that cannot be discovered by our static analysis. Thus, the analyst can provide specifications to capture an information flow that includes some implicit portions.

We begin by describing the points-to relation, which is an important intermediate relation computed by our analysis. Then, we describe the specifications used by our static information flow analysis, and we finally describe the rules for extracting the labeled graph G and CFG C from the input program P.

2.1 The Points-To Relation

An important intermediate relation computed by our static analysis is the *points-to relation* [11, 100, 152, 69], which computes what heap locations a reference variable may point to. In particular, points-to analysis enables our static analysis to resolve information flows due to aliasing, where two reference variables point to the same heap location.

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

```
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 2.1: An information 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.   @Src(LOC, return)
10.  static String getLatitude() {}
11.class SMS:
12.  @Sink(text, SMS)
13.  static void sendSMS(String text) {}
```

Figure 2.2: Specifications for Android library classes.

In points-to analysis, heap locations are abstractly represented by an allocation statement $o = (x \leftarrow X())$, which we call an *abstract object* $o \in \mathcal{H}$. Then, a *points-to edge* is a pair $x \hookrightarrow o \in \mathcal{V} \times \mathcal{H}$, which says that reference variable x may point to abstract object o. More precisely, this relation means that during some execution, x may point to a heap location containing a concrete object allocated at o. A *points-to analysis* is a static analysis that computes sets of points-to edges $\Pi \subseteq \mathcal{V} \times \mathcal{H}$.

Our static analysis uses a points-to analysis for Java programs formulated as a CFL reachability problem [137, 136]. In particular, the constructed labeled graph and CFG for points-to analysis is a subset of the graph G and CFG C constructed for our information flow analysis.

2.2 Specifications

In this section, we describe the specifications used by our static analysis. The source-sink specifications encode the safety property that we aim to verify; the remaining specifications are specifications designed to improve precision and scalability or handle missing code.

Source-sink specifications. The *source specifications* and the *sink specifications* specify the safety property that we seek to verify. More precisely, a source specification $\text{Src}(\ell, x)$ says that program variable x is tainted with source $\ell \in \mathcal{L}_{\text{source}}$, and $\text{Sink}(x, \ell)$ says that x is tainted with sink $\ell \in \mathcal{L}_{\text{sink}}$. Together, the source and sink specifications encode the safety property to be verified. In particular, the safety property says that no data in a source variable should ever flow to a sink variable. Sources and sinks must be annotated by the analyst using the annotations `@Source` and `@Sink`. For example, in Figure 2.2, the return value of `getLocation` is annotated as a source, and the parameter of `sendHTTP` is annotated as a sink.

Information flow specifications. The *flow specifications* describe how information flows from the parameters of a function to other parameters or to its return value. More precisely, a flow specification $\text{Flow}(x, y)$ says that if x is tainted, then y may be tainted as well. Here, x and y can be a parameter p_m, the return value r_m, or the receiver $\texttt{this}_m$; x may also be a label $\ell \in \mathcal{L}_{\text{source}}$ representing a source, and y may also be a label $\ell \in \mathcal{L}_{\text{sink}}$ representing a sink. Similar to source and sink specifications, flow specifications are provided by the `@Flow` annotations. In Figure 2.2, the `toString` method has an annotation specifying the flow specification $\text{Flow}(\texttt{this}, r_{\texttt{toString}})$. For example, the specification of `Double.toString` means that if $\texttt{this}_{\texttt{toString}}$ is tainted, then $r_{\texttt{toString}}$ is also tainted.

Points-to specifications. The *points-to specifications* summarize the potential points-to effects of a library function. These are written as short functions 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 $r_{\texttt{get}}$ 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 $r_{\texttt{get}}$ to be aliased.

Reachability specifications. The *reachability specifications* describe which functions in the program are reachable. These are simply annotations `@Reachable` indicating that a function is reachable. If a function is reachable, then the static analysis marks any function it calls as reachable as well; thus, only entry-points need to be annotated as being reachable. For example, in Figure 2.1, only the function `main` needs to be marked as reachable.

In general, the static analysis assumes that the `main` function is reachable. These specifications are required because the Android framework allows apps to register *callbacks*, which are functions in the app that should be called when certain system events occur. For example, the `onLocationChanged` callback is triggered if the user location changes. Callbacks must be marked as entry points since they are reachable, or else the static analysis will be unsound.

2.3 CFL Reachability

We assume our static analysis problem is formulated as a CFL reachability problem [119, 118, 99, 83, 85, 137, 136], which we describe in this section. Let $C = (\Sigma, U, P, T)$ 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 the form $A \to BC$. 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* $v \xrightarrow{\alpha} v' \in G$ 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

- A is a terminal and there exists $v \xrightarrow{A} v' \in G$, or
- there exists $v \xrightarrow{\alpha} v' \in G$ such that $A \stackrel{*}{\Rightarrow} \alpha$.

If $v \xrightarrow{T} v'$, we say v' is *C-reachable* from v. When C is unambiguous, we also use the notation $\overline{G} = G^C$. Now, we can efficiently solve the following graph reachability problem:

Definition 2.3.1 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 proceeds in two phases. In the first phase, the input program is converted into a graph $G = (V, E)$ and a CFG C. In the second phase, the CFL reachability problem for C and G is solved for some given sets of sources and sinks $V_{\text{source}}, V_{\text{sink}} \subseteq V$, typically by finding the transitive closure G^C using a standard dynamic programming algorithm [99]. In particular, G^C is computed as the (unique) minimal solution to the following constraint system:

- $\dfrac{e \in G}{e \in G^C}$
- $\dfrac{v \xrightarrow{B} v' \in G^C, \quad A \to B \in C}{v \xrightarrow{A} v' \in G^C}$
- $\dfrac{v \xrightarrow{B} v'' \xrightarrow{D} v' \in G^C, \quad A \to BD \in C}{v \xrightarrow{A} v' \in G^C}$

Finally, the safety property says that no source-sink paths exist in G^C. More precisely:

$$\bigwedge_{x \in V_{\text{source}}} \bigwedge_{y \in V_{\text{sink}}} (x \xrightarrow{T} y \notin G^C).$$

1. `v=new X()` $\Rightarrow o \xrightarrow{\text{New}} v$
2. `u=v` $\Rightarrow v \xrightarrow{\text{Assign}} u$
3. `u.f=v` $\Rightarrow v \xrightarrow{\text{Put}[f]} u$
4. `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 2.3: Program fact extraction rules for static information flow analysis. In Rule 8, $o_{v'}$ is a fresh vertex.

2.4 Static Explicit Information Flow Analysis

In this section, we describe the rules by which our static information flow analysis constructs the labeled graph G and the CFG C.

2.4.1 Constructing the Flow Graph

Given a program P, the *flow graph* for P is $G = (V, E)$, where $V = \mathcal{V} \cup \mathcal{H} \cup \mathcal{L}$, where $\mathcal{L} = \mathcal{L}_{\text{source}} \cup \mathcal{L}_{\text{sink}}$ is the set of source and sink labels. Figure 2.3 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 x on the left-hand side of the assignment. Rule 2 similarly encodes the flow when a reference variable x is assigned to another reference variable y. 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 and 6 add edges to G encoding the given source-sink specifications, and Rule 7 adds edges encoding the given information flow specifications. 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 points-to specifications for every method allocating a non-primitive return value (Rule 8 is discussed further below). Finally, Rule 9 allows us to express paths with

10. FlowsTo $\rightarrow$ New
11. FlowsTo $\rightarrow$ FlowsTo Assign
12. FlowsTo[f] $\rightarrow$ FlowsTo Put[f] $\overline{\text{FlowsTo}}$
13. FlowsTo $\rightarrow$ FlowsTo[f] FlowsTo Get[f]
14. SrcObj $\rightarrow$ SrcRef $\overline{\text{FlowsTo}}$
15. SrcObj $\rightarrow$ SrcObj FlowsTo RefRef $\overline{\text{FlowsTo}}$
16. SrcSink $\rightarrow$ 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 2.4: Productions for C_{flow}.

"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 $\texttt{lat} \xrightarrow{\text{Assign}} \texttt{this}^{\texttt{formal}}_{\texttt{add}} \xrightarrow{\text{Assign}} \texttt{this}_{\texttt{add}}$ and $r_{\texttt{toString}} \xrightarrow{\text{Assign}} r^{\texttt{formal}}_{\texttt{toString}} \xrightarrow{\text{Assign}}$ `latString`. This indirection ensures that function boundaries are clear.

Figure 2.5 shows the taint graph generated from the code in Figure 2.1 and the specifications in Figure 2.2 using the rules in Figure 2.3. 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

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

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.

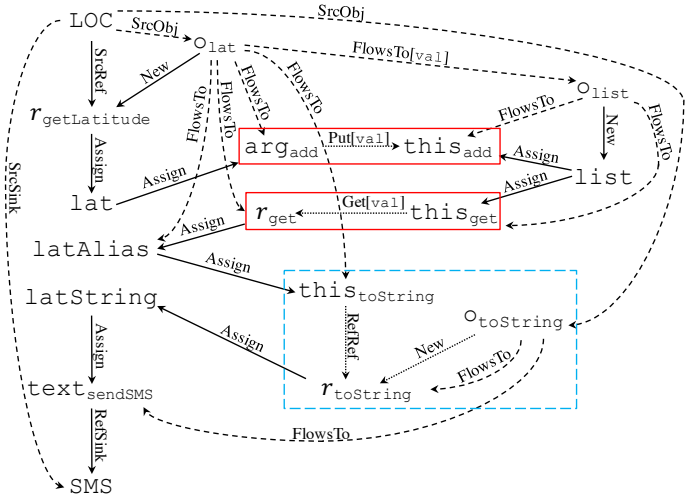


Figure 2.5: The flow graph G corresponding to the code in Figure 2.1 and the framework specifications in Figure 2.2. Solid edges are facts extracted from the code in Figure 2.1 (backwards edges added by Rule 9 are not shown). Dotted edges are facts extracted from the framework specifications in Figure 2.2. 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 2.4 (not all such edges are shown).

2.4.2 Constructing the CFG

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

$$\Sigma_{\text{flow}} = \{\text{New}, \text{Assign}, \text{SrcRef}, \text{RefSink}, \text{RefRef}\} \cup \{\text{Put}[f], \text{Get}[f] \mid f \in \mathcal{F}\}$$
$$U_{\text{flow}} = \{\text{FlowsTo}, \text{SrcObj}, \text{SrcSink}\} \cup \{\text{FlowsTo}[f] \mid f \in \mathcal{F}\}.$$

We also include symbols $\overline{\sigma}$ and $\overline{A}$ in Σ_{flow} and U_{flow}, respectively. The start symbol of C_{flow} is $T_{\text{flow}} = \text{SrcSink}$. The source vertices we consider are the 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 2.4. Rules 10-13 build the points-to relation $o \xrightarrow{\text{FlowsTo}} x$, which means that the reference variable $x \in \mathcal{U}$ may point to the abstract object $o \in \mathcal{O}$. For example, because Figure 2.5 contains the path

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

Rule 12 adds edge $o_{\texttt{lat}} \xrightarrow{\text{FlowsTo}[\texttt{val}]} o_{\texttt{list}}$. Here, $o_{\texttt{lat}}$ and $o_{\texttt{list}}$ are the abstract objects allocated to

`lat` and `list`, respectively. Then we have path

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

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

The backwards edge $\texttt{this}_{\texttt{add}} \xrightarrow{\overline{\text{FlowsTo}}} o_{\texttt{list}}$ in the example path above is added by Rule 17, which introduces a reversed edge $y \xrightarrow{\overline{A}} x$ for every non-terminal edge $x \xrightarrow{A} y$. 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 $o_{\texttt{list}} \xrightarrow{\text{FlowsTo}} \texttt{this}_{\texttt{add}}$ arises from the path of terminal edges

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

and the application of Rule 10 followed by Rule 11.

Because $r_{\texttt{getLatitude}}$ and `latAlias` can point to the same object, if one of them is tainted, then the other should be tainted as well. Instead of keeping track of taint on the reference variables, it is simpler to keep track of taint on the objects. In Figure 2.5, Rule 14 adds the edge $\texttt{LOC} \xrightarrow{\text{SrcObj}} o_{\texttt{lat}}$, which says that $o_{\texttt{lat}}$ is tainted. For this taint to flow to the sink, the analysis must be able to pass the taint to $o_{\texttt{toString}}$ (the object allocated to $r_{\texttt{toString}}$).

But here we encounter a problem: there is no explicit flow through the `toString` method in the `Double` class, 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. The flow specification on `toString` enables us to capture the information flow despite this portion of the flow being implicit. In particular, it says that if the receiver of `toString` is tainted, then the return value of `toString` is tainted as well.

One caveat is that for our analysis to propagate the taint using the specification for `toString`, it needs to apply Rule 15, which requires that the return value of `toString` point to an abstract object. However, since the static analysis omits analyzing `toString`, it appears that this return value does not point to any abstract object, in which case the static analysis would fail to propagate taint. Thus, Rule 8 from Figure 2.3 adds a *phantom object* to the graph, which is a (fresh) abstract object that is pointed to by the return value of a function with a analyst-provided specification. For example, Rule 8 adds $o_{\texttt{toString}} \xrightarrow{\text{New}} r_{\texttt{toString}}$, causing Rule 10 to add $o_{\texttt{toString}} \xrightarrow{\text{FlowsTo}} r_{\texttt{toString}}$.

Now we can capture the flow in Figure 2.1. In Figure 2.5, we have the path

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

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

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

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

2.5 Implementation

We have implemented this static information flow analysis for Android apps build on 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. 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 sources (location, device ID, SIM data, contacts, and calendar data) and sinks (network sockets, SMS messages, and user settings modifications). This baseline also includes specifications for important container objects including `ArrayList`, `LinkedList`, and `HashMap`.

In our implementation, program fact extraction is performed using the Chord platform [103] (which uses BDDBDDB as a backend [152]), modified to work with the Jimple intermediate representation provided by Soot [147]. In order to improve precision of our analysis, we extend the points-to rules (Rules 10-13 in Figure 2.4) 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.

Chapter 3

Interactive Specification Inference

In this chapter, we study the problem of interacting with a human analyst to infer specifications for a sound interprocedural static analysis. Such static analyses often have trouble analyzing programs that include large third-party libraries such as the Android framework. For example, these libraries often make significant use of code that is difficult to analyze, such as native code or dynamic programming language features such as Java reflection. Optimistically assuming these functions are no-ops can introduce unsoundness, whereas pessimistically making worst-case assumptions about the behaviors of these functions can substantially reduce precision and scalability.

We consider an approach where the analyst provides specifications that describe the behaviors of functions in the third-party library relevant to the static analysis. The static analysis processes the specifications in place of analyzing the library implementation, thereby substantially improving its precision and scalability. However, manually writing these specifications can be very time-consuming and error-prone [18]. In practice, the analyst typically writes specifications in a demand-driven fashion—given a new program, they try to identify all library functions that may be relevant to the static analysis, and implement specifications for just these functions. This strategy is effective in practice because in many settings, only a small number of functions are relevant to the static analysis, especially for a given program. The drawback is that if the analyst omits implementing a specification for a relevant library function, then the static analysis may become unsound.

We propose an algorithm for automating this process. Given a new program, our algorithm computes a set of library functions that are potentially relevant to the static analysis (possibly along with hypothetical specifications for these functions). Then, it requires that the analyst implement specifications for this set of functions. Depending on the responses of the analyst, it may identify new functions that may be relevant to the static analysis, so this process is iterated until it converges, i.e., no potentially relevant library functions remain. At this point, our algorithm guarantees that the results produced by the static analysis are sound, i.e., they are equivalent to the results obtained by running the static analysis with specifications provided for every library function.

```
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 3.1: 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 3.2: Specifications for Android framework classes.

Our algorithm can be used for any static analysis expressed as a context-free reachability problem. We apply our algorithm to infer specifications for the information flow analysis of Android apps described in Chapter 2, 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 framework methods.

Our work has three main contributions:

- We develop a general framework for describing potentially missing specifications for CFL reachability analyses (Section 3.2).
- We present a specification inference algorithm based on this framework that interacts with the analyst to infer relevant specifications (Section 3.3).
- We use our algorithm to infer a large collection of specifications for Android framework methods (Section 3.5).

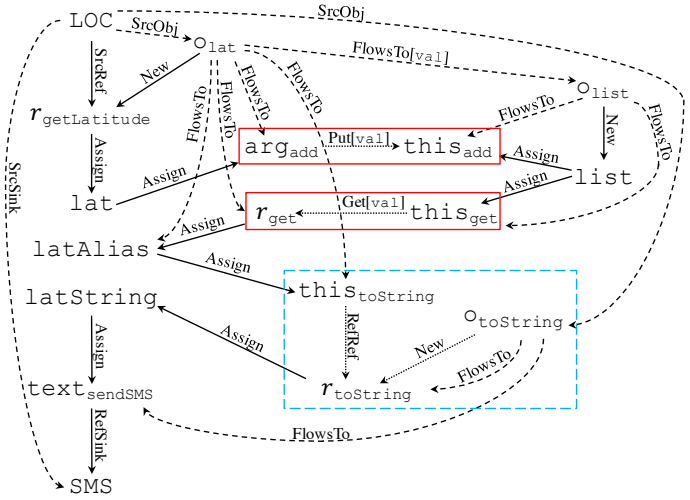


Figure 3.3: The flow graph G corresponding to the code in Figure 3.1 and the framework specifications in Figure 3.2. Solid edges are facts extracted from the code in Figure 3.1 (backwards edges added by Rule 9 are not shown). Dotted edges are facts extracted from the framework specifications in Figure 3.2. 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 2.4 (not all such edges are shown).

3.1 Overview

While implementations of the Android framework methods may be missing, the net information flows through these methods are generally simple. For example, consider the code in Figure 3.1. We can summarize the net information flows through `List.add`, `List.get`, and `Double.toString` as follows: (i) $\texttt{arg}_{\texttt{add}}$ may be aliased with $r_{\texttt{get}}$, and (ii) if taint flows to $\texttt{this}_{\texttt{toString}}$, then taint also flows to $r_{\texttt{toString}}$.

The specifications in Figure 3.2 enable the analysis to find the flow from the source `LOC` to the sink `SMS` in Figure 3.1. First, the return value $r_{\texttt{getLatitude}}$ is tainted with the `LOC` source. 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 framework method `sendSMS`. Finally, our specification says that the `text` parameter of the method `sendSMS` is sent to the `SMS` sink, 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 flow specifications that handle field accesses, and replace the points-to 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 3.4: `boxAlias.f` is tainted by `LOC`, so `list.val.f` is tainted (since `boxAlias` and `list.val` are aliased), but the

```
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 3.4: An information flow not captured by flow specifications.

proposed specification for `List.add` does not transfer this taint to `box.f` (even though taint flows to `box.f`). In general, we need points-to specifications to precisely and soundly capture flows due to aliasing.

Manually writing specifications is expensive: the Android framework contains over 5000 classes, many exhibiting complex taint flow behavior. Typically, the analyst must search the application for calls to potentially important framework methods, and then manually write specifications for these methods. Even moderately large apps contain thousands of framework method calls, but most of them are irrelevant to finding information 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 introduce false negatives into the static analysis results. For example, suppose we remove the specification for `List.add` from Figure 3.2. Then the static analysis cannot find the flow from `LOC` to `SMS` (since the information flow path between them is broken), causing a false negative. Unlike false positives, where the analyst 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 framework method call made by the app.

Intuitively, the visible application code contains useful information about the correct specifications for the framework 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 pessimistically making worst-case assumptions about the possible effects of 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 analyst 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 analyst.

3.2 Problem Statement

In this section, we formulate the problem of performing a sound and precise CFL reachability analysis when specifications are missing, along with the problem of inferring the missing specifications . Our formulation extends the CFL reachability framework for designing static analyses described in Chapter 2.

3.2.1 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 3.2.1 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'.

- $\mathcal{A}$ 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 information flow present in at least one of the possible completions of $\widehat{G}$, and the algorithm is precise if it does

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

Figure 3.5: An alternative (and incorrect) specification for `List.add`.

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 3.2.1 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 3.2.2 Suppose we are given the inputs as in Definition 3.2.1, 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 points-to and flow specifications.

3.2.2 $\mathcal{G}$ 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 functions. 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 functions 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 functions

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 flow graph G in Figure 3.3. Suppose the specification the method `List.add` in Figure 3.2 is missing, so the edge $\texttt{arg}_{\texttt{add}} \xrightarrow{\text{Put[val]}} \texttt{this}_{\texttt{add}}$ in Figure 3.3 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 function m, the only possible taint flows through m are from a parameter of m to a return value of m, or from one parameter to another parameter. We use $V_m \subseteq \widehat{V}$ to denote the vertices of $\widehat{G}$ corresponding to the 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 $\texttt{arg}_{\texttt{add}} \xrightarrow{\alpha} \texttt{this}_{\texttt{add}}$. Generalizing from this example, for a missing function m, it suffices to consider the family of graphs $\mathcal{G}$ consisting of all graphs containing $\widehat{G}$ as a subgraph with additional paths $w \xrightarrow{\alpha} w'$, where $w, w' \in V_m$.

For example, one completion of $\widehat{G}$ is the flow graph G in Figure 3.3, 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 3.5. 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 $\texttt{arg}_{\texttt{add}} \xrightarrow{\alpha} \texttt{this}_{\texttt{add.this}}$ 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 3.2.3 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 $w \xrightarrow{\alpha} w' \in G_{w,w'}$ satisfies $\alpha \in L(N)$. Conversely, for any $\alpha \in L(N)$, there exists a path $w \xrightarrow{\alpha} w' \in G_{w,w'}$.

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.

3.2.3 The Specification Inference Problem 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 points-to 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 $\mathcal{P}(G) = \{v_{\text{source}} \xrightarrow{\alpha} v_{\text{sink}} \in G \mid T \overset{*}{\Rightarrow} \alpha\}$ 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 3.3, we show how to reduce this problem to the shortest-path CFL reachability problem.

We are interested in inferring both flow and points-to specifications. Let $V_m = V_m^{\text{arg}} \cup \{r_m\}$, where V_m^{arg} is the set of parameters of a function m, and r_m is the return value of m. First, we infer missing flow specifications:

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

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.

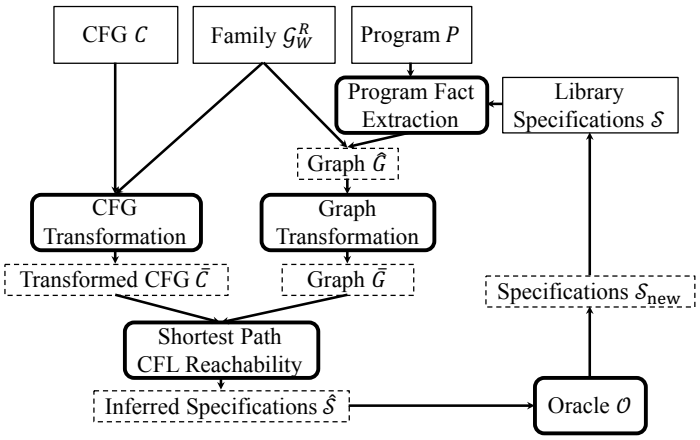


Figure 3.6: An overview of our specification inference system. The system infers specifications $\widehat{\mathcal{S}}$ and proposes them to the oracle $\mathcal{O}$ (i.e., the human analyst), who examines the proposals and generates a new set of specifications $\mathcal{S}_{\text{new}}$. Then $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}_{\text{new}}$, and the process repeats. Program fact extraction is described in Figure 2.3. 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 3.3.3 and Appendix 3.3.4. The specification refinement loop is performed by Algorithm 3.

Second, we infer missing points-to specifications:

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

The possible sequences of operations are bracketed by $(\text{Assign} + \overline{\text{Assign}})$ because allocation and field access operations cannot be performed on parameters and 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 Chapter 2).

3.3 Algorithms for Specification Inference

In this section we present an algorithm that solves the missing specifications problem stated in Definition 3.2.1 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 3.2.2 by using a shortest-path extension of the CFL reachability algorithm.

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 3.7: 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.

This allows us to construct an algorithm that interacts with a human analyst to produce results that are sound and precise with respect to G^*. An overview of our system is given in Figure 3.6.

3.3.1 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 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 3.3.1 Algorithm 1 is sound and precise for $\mathcal{G}_W^R$.

3.3.2 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 $n_i \overset{\alpha}{\dashrightarrow} m_i \in G_i$ such that $A \overset{*}{\Rightarrow} \alpha$. 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.

Algorithm 1 A sound and precise algorithm for $\mathcal{G}_W^R$. Here, $\mathcal{N}(s \xrightarrow{R} f)$ is defined in Figure 3.7.

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

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.

Algorithm 2 Optimized algorithm for $\mathcal{G}_W^R$. Here, $\mathcal{T}(G_i, C)$ applies the rules in Figure 3.8 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 $v \xrightarrow{T} v' \stackrel{?}{\in} (\text{PREPROCESSHELPER}(\widehat{G}, W))^{\overline{C}}$
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

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 3.8. 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 3.8; 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 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 $A_n \to BD_n \in \overline{C}$.

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 3.8: Productions for $\overline{C}$.

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

Figure 3.9: Examples of production rules added by Figure 3.8, along with the rules that generate them.

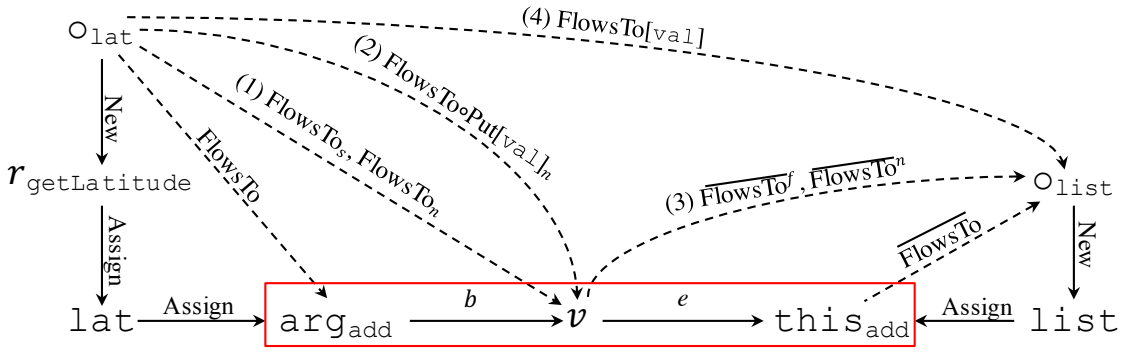


Figure 3.10: Algorithm 2 adds the dashed edges to Figure 3.3 if the specification for `List.add` is missing. We only show edges relevant to the production of the edge labeled (4).

(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 3.8. 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 3.8 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 $A \stackrel{*}{\Rightarrow} \beta$, 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 $A \stackrel{*}{\Rightarrow} \alpha\beta\gamma$. 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 $A \stackrel{*}{\Rightarrow} \alpha\beta\gamma$. This time, we only need to handle the case where the production is split at vertex v_i—i.e., $A \to BD$, where $B \stackrel{*}{\Rightarrow} \alpha\beta_1$ and $D \stackrel{*}{\Rightarrow} \beta_2\gamma$ (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 3.3.2 Algorithm 2 is sound and precise for $\mathcal{G}_W^R$.

We briefly discuss the complexity of Algorithm 2. The rules in Figure 3.8 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) [99].

As an example, consider $\mathcal{G}_{W_{\text{alias}}}^{R_{\text{alias}}}$ defined in Section 3.2. 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 3.9.

Consider the code in Figure 3.1, and suppose the specification for `List.add` is missing, so $W_{\text{alias}} = V_{\texttt{List.add}}$. In Figure 3.10, 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, since $\text{FlowsTo} \to \text{FlowsTo Assign}$).

2. This edge is added by $\text{FlowsTo} \circ \text{Put}[f]_n \to \text{FlowsTo}_n$ (Rule 1f, since $\text{FlowsTo} \circ \text{Put}[f] \to \text{FlowsTo Put}[f]$).

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, since $\overline{\text{FlowsTo}} \to \overline{\text{Assign}}\ \overline{\text{FlowsTo}}$).

4. This edge is added by $\text{FlowsTo}[f] \to \text{FlowsTo} \circ \text{Put}[f]_n\ \overline{\text{FlowsTo}}^n$ (Rule 4c, since $\text{FlowsTo}[f] \to \text{FlowsTo} \circ \text{Put}[f]\ \overline{\text{FlowsTo}}$).

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 $o_{\texttt{lat}} \xrightarrow{\text{FlowsTo}[\texttt{val}]} o_{\texttt{list}}$, it will add the edge $\texttt{LOC} \xrightarrow{\text{SrcSink}} \texttt{SMS}$ as a consequence of the productions in C_{taint}.

3.3.3 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 $\text{weight}(G)$, i.e. finding $G \in \mathcal{G}_W^R$ with source-sink path p of minimum $\text{weight}(p)$. To do so, we define a weight function on Σ by setting $\text{weight}(b) = \text{weight}(e) = \frac{1}{2}$, and $\text{weight}(\sigma) = 0$ for all other $\sigma \in \Sigma$. This extends to Σ^* by setting $\text{weight}(\sigma_1 ... \sigma_k) = \sum_{i=1}^{k} \text{weight}(\sigma_i)$. Note that for source-sink path $p = v_{\text{source}} \overset{\alpha}{\dashrightarrow} v_{\text{sink}} \in \mathcal{P}(\overline{G})$, $\text{weight}(p) = \text{weight}(\alpha)$. Consider the following:

Definition 3.3.3 Let $\overline{G}$ be the graph defined above. The *shortest-path CFL reachability problem* is to return the shortest path $p^* = \arg\min_{p \in \mathcal{P}(\overline{G})} \text{weight}(\alpha)$, or return $\emptyset$ if $\mathcal{P}(\overline{G}) = \emptyset$.

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

Knuth describes a generalization of Djikstra's algorithm to find the shortest string in a context-free grammar [82]. This algorithm generalizes to solving the shortest-path CFL reachability problem: we replace the worklist of edges in [99] with a heap of edges, where the priority of an edge is the weight of its shortest path; see Algorithm 4 in Section 3.3.4 for details. By using Algorithm 4 to compute the closure $\overline{G}^{\overline{C}}$, we find the source-sink path $p^* \in \mathcal{P}(\overline{G})$ 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 a human analyst. 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 3.3.4 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 analyst as an oracle $\mathcal{O}$. On input (w, w'), the analyst 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 3.3.5 Algorithm 3 computes $v_{\text{source}} \xrightarrow{T} v_{\text{sink}} \stackrel{?}{\in} (G^*)^C$.

3.3.4 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 [82] 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 [99]. 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 [99]).

3.4 Implementation

We have implemented the system described in Figure 3.6. Within our specification inference framework, we infer both flow and points-to specifications. For each inferred specification, we manually

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 3.11: Statistics on inferred Android framework specifications.

reference the Android framework documentation to determine if the specification is correct, and re-run the analysis with the updated specifications. We repeat this process until no new specifications are inferred.

When inferring points-to specifications, the large size of the static points-to sets makes it difficult to scale the inference algorithm. We implemented a demand-driven optimization. First, we perform the entire analysis using BDDBDDB 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.

Since our focus in on the long tail of missing specifications, we bootstrap our implementation of the specification inference framework with these baseline specifications. Also, our current implementation does not consider specifications involving static fields; as discussed in Section 3.2, 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.

Finally, to be fully sound, we would ideally infer flow and points-to specifications simultaneously. However, worst-case flow specifications introduce a large number of incorrect information 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 infer the two kinds of specifications separately in our experiments.

3.5 Evaluation

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 3.14(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 analyst to interactively run the analysis. The points-to specification inference algorithm successfully ran on 156 apps. The worst-case assumptions cause a substantial increase in the points-to relation size (see Figure 3.14(d)), which proved to be too large on the remaining 23 apps. Still, inferring

App	LOC	Sum.	Crit. Sum.	Tot. Sum.	Rounds	Acc.	Flows	Time	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	Points-To
andmj	239227	0	0	5229	1	N/A	0	1555	Points-To
ce667f	193518	0	0	5509	1	N/A	0	3234	Points-To
ConnectBot	136800	0	0	3293	1	N/A	0	3193	Points-To
05ed92	134235	1	1	12632	2	1.0	3	2254	Points-To
SMSBot	134230	15	3	4693	7	0.467	2	284.3	Points-To
yaaic	109286	0	0	4161	1	N/A	0	816.1	Points-To
ca70f4	81974	3	0	3307	3	0.0	0	45.57	Points-To
tomdroid	44478	0	0	3697	1	N/A	0	68.154	Points-To
a1d58b	41682	2	1	2285	3	0.5	1	7.847	Points-To

Figure 3.12: Specification inference results on large Android apps: the number of Jimple lines of code ("LOC"), the number of specifications proposed by our tool ("Sum."), the number of worst-case specifications ("Tot. Sum."), the number of critical specifications ("Crit. Sum."), the number of iterations with the analyst in Algorithm 3 ("Rounds"), the proportion of proposed specifications that are correct ("Acc."), the number of new information flows discovered ("Flows"), the running time in seconds ("Time"), and the specification type ("Type"). The accuracy is N/A if no specifications are inferred.

Class	Method	Specification	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)	Points-To
android.telephony.gsm.SmsMessage	java.lang.String getMessageBody()	(this, return)	Points-To
android.content.ContentValues	void put(java.lang.String, java.lang.String)	(arg2, this)	Points-To

Figure 3.13: Sample of inferred specifications. We show the class to which the method belongs ("Class"), the method signature ("Method"), the pair $(w, w') \in W$ returned by Algorithm 3 ("Specification"), and the specification type ("Type").

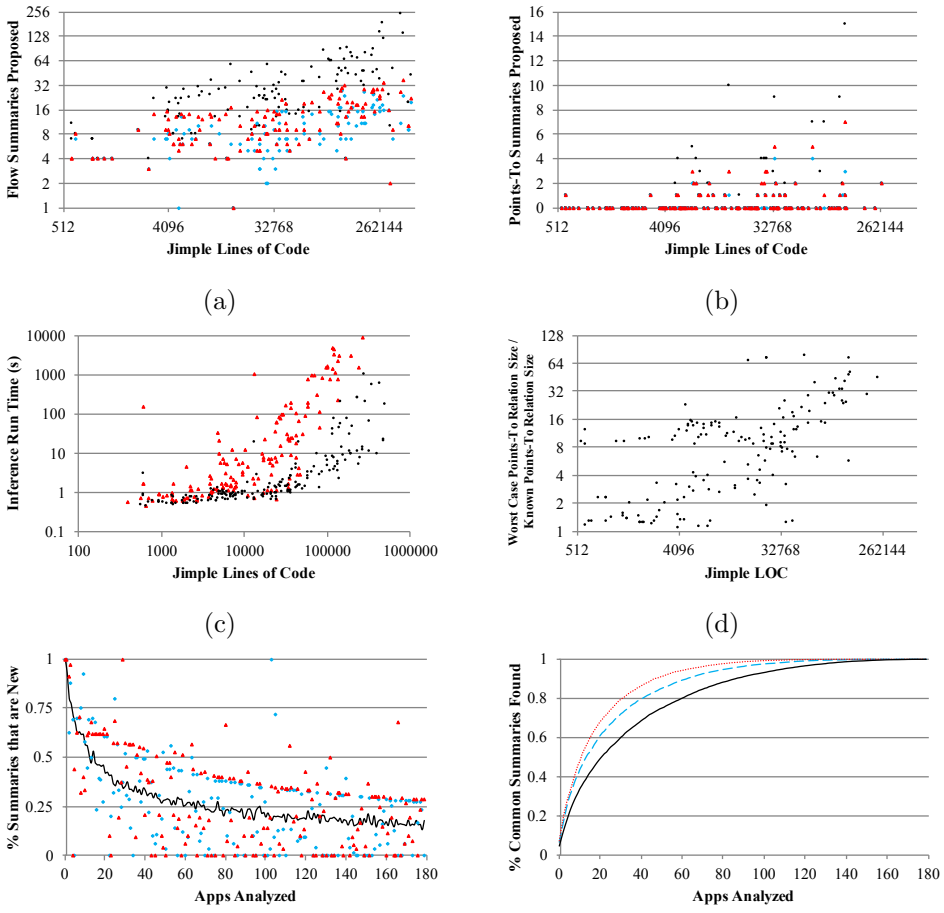


Figure 3.14: For (a) flow and (b) points-to: # specifications proposed (black, circle), # correct (red, triangle), and # critical (blue, diamond). (c) Run time of the flow (black, circle) and points-to (red, triangle) specification inference algorithms. (d) Ratio of worst-case points-to relation size 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.

points-to specifications runs in under 100 seconds for most apps with up to 100,000 lines of Jimple code.

Results for selected apps, including the largest four that ran successfully for each analysis, are shown in Figure 3.12. We show the number of specifications proposed by our tool, along with the number of worst-case specifications. Our tool may propose specifications that are correct (i.e., represent valid paths throughs framework 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, which is a lower bound for the number of specifications that must be proposed to achieve soundness. We call such specifications *critical specifications*, because they are the specifications that an analyst must examine in order to find all source-sink flows.

The number of inferred specifications is plotted as black circles in Figure 3.14(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 3.11. 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.

3.5.1 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 points-to specifications very accurate, in part because of type filters. We show some examples of inferred specifications in Figure 3.13.

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 parameters v and v' (or a parameter v and the 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 points-to 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.

3.5.2 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 3.14(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 specifications that the analyst 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 analyst'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 analystonly does a fraction of the work compared to starting from the baseline. The required work would drop further after processing more apps.

Figure 3.14(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 $c = 3$.

3.5.3 Verification

We ran Algorithm 3 to termination, i.e., until no new specifications for missing parts of the program could add any more information flows, which means that the remaining taint flows all occur in the original graph $\widehat{G}$. In the case of points-to specification inference, this also proves that no additional explicit information flows can occur. Figure 3.14(b) shows the number of specifications that had to be checked by an analyst to completely verify the absence of explicit information 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, information flows due to missing points-to specifications can still occur, and verification requires that the analyst supply all relevant points-to specifications. In practice, this analysis still discovers many framework methods that need points-to

specifications, since taint often flows forwards through these methods. Thus our tool finds many taint flows even if points-to specifications are missing.

There are currently two primary limitations to our tool. One we have already discussed: inferring both flow and points-to specifications simultaneously is too expensive for our tool on some apps. The second is that while we can infer missing flow and points-to 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 3.12, without our tool an analyst 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.

3.6 Conclusion

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 analyst 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 analyst to perform verification.

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($v \xrightarrow{\sigma} v', \text{weight}(\sigma)$)
    $I[v \xrightarrow{\sigma} v'] \leftarrow \emptyset$
  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($v \xrightarrow{D} v'', P_{\text{new}}$)
          $I[v \xrightarrow{D} v''] \leftarrow [v \xrightarrow{A} v', v' \xrightarrow{B} v'']$
        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($v'' \xrightarrow{D} v', P_{\text{new}}$)
          $I[v'' \xrightarrow{D} v'] \leftarrow [v'' \xrightarrow{B} v, v \xrightarrow{A} v']$
        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($v \xrightarrow{B} v', P_{\text{cur}}$)
        $I[v \xrightarrow{B} v'] \leftarrow [v \xrightarrow{A} v']$
      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

Chapter 4

Specification Inference with Untrusted Responses

In this chapter, we study the problem of interacting with a human user to infer specifications when the responses of the user are untrusted. We focus on the case of reachability summaries, which are used by the static reachability analysis to determine what code is reachable, which can be a major problem for static analyses [15, 37]—in practice, many false positives are flows through unreachable code. In our setting, this imprecision is caused both by an imprecise callgraph (due to virtual method calls) and by the lack of path sensitivity. Using sound assumptions about possible entry points of an Android app can also lead to imprecision. In our experiments, 92% of false positives were flows through unreachable code. Oftentimes, the unreachable code is found in large third-party libraries used by the app.

We are interested in the setting where the user of the static analysis (e.g., a security analyst at Google trying to identify and remove Android malware from Google Play) differs from the developer who wrote the program being analyzed (e.g., the developer of an Android app submitted to Google Play). Currently, the burden of identifying and discharging false positives is placed entirely on the analyst, despite the fact that the developer is most familiar with the app's code. Our goal is to shift some of this burden onto the developer. In particular, we want the developer to provide reachability specifications to the static analysis describing which methods in the app are reachable. These reachability specifications allow the static analysis to restrict its search space to reachable code, thereby reducing the false positive rate.

In practice, we envision that the developer will provide reachability specifications by supplying tests that exercise the app code—the specification we extract is that only tested code is reachable. We use tests to avoid the senario where a developer insists (either maliciously or to avoid effort) that everything is reachable, thereby wasting analyst time and eliminating the benefits of our approach.

Tests are executable, which means that the analyst can verify the correctness of the specifications. Using tests as specifications has a number of additional advantages. First, developers routinely write tests, so this approach both leverages existing tests and gives developers a familiar interface to the specification process. Second, concrete test cases can benefit the analyst in case the app must be manually examined. For our technical development, we assume that specifications are extracted from tests, though any method for obtaining correct reachability specifications suffices.

Of course, a malware developer can attempt to evade detection by specifying that the malicious code is unreachable. Our solution is simple: we enforce the developer-provided specifications by instrumenting the app to terminate if code not specified to be reachable (e.g., not covered by any of the developer-provided tests) is actually reached during runtime. The instrumented app is both consistent with the developer's specifications, and statically verified to be free of explicit information flows.

In practice, it may be difficult for developers to provide tests covering all reachable code. Therefore, we take an iterative approach to obtaining tests. To enforce the security policy, it is only necessary to terminate the app if it reaches untested code that may also lead to a malicious explicit information flow. Rather than instrument all untested program statements, we find a minimum size set of untested statements (called a *cut*) such that instrumenting these statements to terminate execution produces an app that is free of explicit information flows, and then propose this cut to the developer. If the developer finds the cut unsatisfactory, then the developer can provide new tests (or other reachability information) and repeat the process; otherwise, if the developer finds the cut satisfactory, then the cut is enforced via instrumentation as before. This process repeats until either a satisfactory cut is produced, or no satisfactory cut exists (in which case the analyst must manually review the app).

If the developer allows (accidentally or maliciously) reachable code to be instrumented, then it may be possible for the app to terminate during a benign execution. To make the process more robust against such failures, we can produce *multiple*, *disjoint* cuts. We then instrument the program to terminate only if at least one statement from every cut is reached during an execution of the app.

Our work has three main contributions:

- We formalize an interactive verification algorithm for producing verified apps using abductive inference (Section 4.2).
- For properties formulated in terms of CFL reachability, we reduce the abductive inference problem to an integer linear program (Section 4.3).
- We implement our framework (Section 4.4) for producing Android apps verified to be free of explicit information flows, and show that our approach scales to large Android apps, some with hundreds of thousands of lines of bytecode (Section 4.5).

```
1. void leak(boolean flag, String data) {
2.   // @Sink("sendHTTP.param", HTTP)
3.   if (flag) sendHTTP(data); }
4. @Entry("onCreate")
5. void onCreate() {
6.   // @Source(LOC, "getLocation.return")
7.   String loc = getLocation();
8.   Runnable runMalice = new Runnable() {
9.     void run() { leak(true, loc); }}
10.  Runnable runBenign = new Runnable() {
11.    void run() { leak(false, loc); }}
12.  runBenign.run(); }
```

Figure 4.1: An app $\mathcal{P}_{\text{onCreate}}$ for which the static analysis potentially finds a false positive information flow. The comment in line 2 indicates that the first argument of `sendHTTP` is a sink, and the comment in line 6 indicates that the return value of `getLocation` is a source.

4.1 Overview

Consider the Android app shown in Figure 4.1, which we call $\mathcal{P}_{\text{onCreate}}$. Suppose that a developer submits $\mathcal{P}_{\text{onCreate}}$ to the app store. The first step is to run an information flow analysis on $\mathcal{P}_{\text{onCreate}}$. We assume that the information sources and sinks are given (or inferred, see [95]). For simplicity, we assume that the goal is to prove that the user's location does not flow to the Internet:

$$\phi_{\text{flow}} = \nexists(\text{source-to-sink explicit information flow}).$$

In Figure 4.2, the dashed edges are edges added when computing the transitive closure of the app in Figure 4.2. For example, because we have edge $r_{\texttt{getLocation}} \xrightarrow{\overline{\text{New}}} o_{\text{location}}$, Rules 8 and 12 add the edge $r_{\texttt{getLocation}} \xrightarrow{\overline{\text{FlowsTo}}} o_{\text{location}}$. Also, because we have path

$$o_{\text{location}} \xrightarrow{\text{New}} r_{\texttt{getLocation}} \xrightarrow{\text{Assign}} \texttt{loc} \xrightarrow{\text{Assign}} \texttt{data} \xrightarrow{\text{Assign}} \texttt{text},$$

Rules 8 and 9 add edge $o_{\text{location}} \xrightarrow{\text{FlowsTo}} \texttt{text}$. Now we have path

$$\texttt{LOC} \xrightarrow{\text{SrcRef}} r_{\texttt{getLocation}} \xrightarrow{\overline{\text{FlowsTo}}} o_{\text{location}} \xrightarrow{\text{FlowsTo}} \texttt{text} \xrightarrow{\text{RefSink}} \texttt{HTTP},$$

from which Rule 11 adds $\texttt{LOC} \xrightarrow{\text{SrcSink}} \texttt{HTTP}$.

As discussed earlier, one major source of false positive information flows is unreachable code, so we remove parts of the program that are statically proven to be unreachable. However, statically computed reachability information can be very imprecise; we focus on imprecision due to the static callgraph. For example, using a callgraph generated by class hierarchy analysis, the analysis cannot determine that line 12 cannot call `runMalice.run`. Even with a more precise callgraph, the static

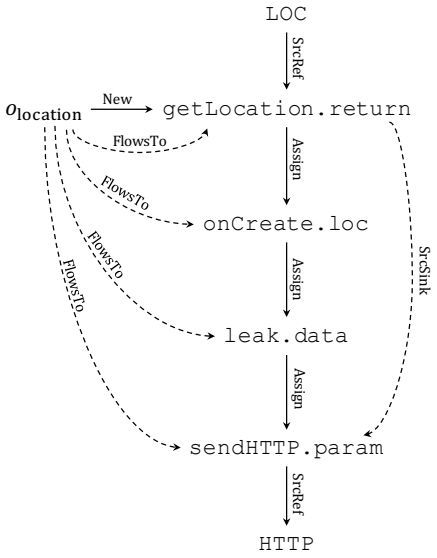


Figure 4.2: A part of the graph G for the code in Figure 4.1. Solid edges are edges extracted using the rules in Figure 4.1. Dashed edges are edges added by the rules in Figure 2.4. Backwards edges are omitted for clarity.

analysis may not be able to prove that `flag` is false in every execution. Hence, our information flow analysis finds a flow from `getLocation.return` to `sendHTTP.param`—i.e., it fails to prove ϕ_{flow}.

Our approach is to search for a *cut*, which is a subset of statements that can be removed from $\mathcal{P}_{\text{onCreate}}$ so that the resulting app $\mathcal{P}'_{\text{onCreate}}$ satisfies ϕ_{flow}. More formally, let S^* be the set of reachable program statements. Because our static analysis is sound, the set of reachable statements S computed by our static analysis overapproximates S^* (i.e., $S^* \subseteq S$). Let λ be a predicate of the form

$$\lambda = \bigwedge_{s \in E_\lambda} (s \notin S^*),$$

where $E_\lambda \subseteq S$. In other words, λ asserts that the subset E_λ of program statements are unreachable.

Then, a cut is a predicate λ such that if λ holds (i.e., every statement in $s \in E_\lambda$ is unreachable), then the security policy ϕ_{flow} (i.e., lack of explicit information flows) holds. In our example, any of the following choices for E_λ would allow the static analysis to prove ϕ_{flow} for $\mathcal{P}_{\text{onCreate}}$:

a. {`3.sendHTTP(data)`}

b. {`7.getLocation()`}

c. {`9.leak(true,loc)`}

We compute λ using *abductive inference* [43]: given facts χ (extracted from the app $\mathcal{P}$) and security policy ϕ, abductive inference computes a minimum size predicate λ such that (i) $\chi \wedge \lambda \models \phi$ (i.e., λ together with the known program facts χ suffice to prove the security policy ϕ) and (ii) $\text{SAT}(\chi \wedge \lambda)$ (i.e., λ is consistent with the known program facts χ). In our setting, we augment χ with facts extracted from the tests.

We propose a novel algorithm for solving the abductive inference problem for properties formulated in terms of context-free language (CFL) reachability. The security policy ϕ states that certain vertices in a graph representation of the program are unreachable. Our key insight is to formulate the CFL reachability problem as a constraint system, which we encode as an integer linear program. Finding minimum cuts in turn corresponds to a minimum solution for the integer linear program.

Once the cut λ has been computed, the app $\mathcal{P}$ is instrumented to ensure that λ holds, producing a verified app $\mathcal{P}'$:

$$\mathcal{P}' = \mathcal{P} - E_\lambda.$$

We can enforce λ by ensuring that the statements in E_{lambda} are unreachable in $\mathcal{P}'$. To enforce that a statement $s \in E_\lambda$ is unreachable, we terminate $\mathcal{P}_{\text{onCreate}}$ if execution reaches the program point immediately before s (i.e., if s is about to be executed). For example, our analysis produces $\mathcal{P}'_{\text{onCreate}} = \mathcal{P}_{\text{onCreate}} - E_\lambda$ by instrumenting $\mathcal{P}_{\text{onCreate}}$ to enforce that `3.sendHTTP(data)` is unreachable. By the definition of a cut, ϕ_{flow} provably holds for $\mathcal{P}'_{\text{onCreate}}$, so this app can be safely placed on the app store.

However, not all of the above choices for E_λ are desirable; for example, suppose our analysis infers $E_\lambda =$ (b)—then, $E_\lambda \cap S^* \neq \emptyset$, so removing E_λ from $\mathcal{P}_{\text{onCreate}}$ would result in a program that terminates during a valid execution. We call such a cut *invalid* (as opposed to a *valid* cut, which only removes unreachable statements; i.e., $E_\lambda \cap S^* = \emptyset$).

Thus, before publishing $\mathcal{P}'$, we show λ to the developer for inspection; if they determine that λ is invalid, then the developer returns a test that executes `onCreate`, showing that (b) is reachable. By requiring the the developer provides a test where the cut is invalid, we obtain a proof of the invalidity, which prevents the developer from automatically rejecting any cut. Upon executing `onCreate`, our system observes that (b) is reachable. Our system runs the inference algorithm to compute a new cut, this time prohibiting choice (b). The inference algorithm can return either (a) or (c). Suppose that this time, $E_\lambda =$ (a) is returned; then the developer accepts the cut E_λ because removing E_λ from $\mathcal{P}_{\text{onCreate}}$ does not remove any functionality.

Because (`3.sendHTTP(data)` $\notin S^*$) is true for $\mathcal{P}_{\text{onCreate}}$, we know $\mathcal{P}'_{\text{onCreate}}$ is semantically equivalent to $\mathcal{P}_{\text{onCreate}}$. Furthermore, the instrumentation in $\mathcal{P}'_{\text{onCreate}}$ incurs no runtime overhead, since it is unreachable.

There are three alternative scenarios:

- Our information flow analysis may prove ϕ_{flow} for $\mathcal{P}_{\text{onCreate}}$, so no instrumentation is needed.
- There may not exist a valid cut, in which case the analyst must manually inspect the app.
- Finally, the developer may incorrectly accept a cut that removes reachable code (i.e., an invalid cut). In this case, the instrumented app $\mathcal{P}'_{\text{onCreate}}$ may abort during usage, but safety is maintained.

Our experiments show that a valid cut exists for the majority of false positive information flows that occur in our benchmark. Furthermore, the number of interactions required to find a valid cut is typically small.

Finally, we extend this process to infer multiple disjoint cuts $\lambda_1, ..., \lambda_n$, and instrument $\mathcal{P}$ to terminate only if every λ_i fails. By using multiple cuts, we reduce the risk of incorrectly terminating the app if reachable code is cut.

4.1.1 Analyzing Callbacks

In addition to imprecision in the callgraph, another source of imprecision is whether to treat `runMalice.run` as a *callback*. Much of an Android app's functionality is executed via callbacks that are triggered when certain system events occur, so callbacks must be annotated as program entry points. The Android framework provides thousands of callbacks; however, many of these callbacks are poorly documented, which makes manually identifying and annotating callbacks a time consuming and error prone task. If a callback annotation is missing, then reachable code may be excluded from the analysis, introducing unsoundness.

On the other hand, every callback must override an Android framework method—we call any such method a *potential callback*. Of course, not every potential callback is a true callback; for example, any method overriding `Object.equals` is a potential callback but not a true callback. In our analysis, we make the sound assumption that *every* potential callback is a callback—that is, we conservatively overestimate the set of callbacks. We then infer a cut λ as before. For example, in Figure 4.1, the static analysis treats `runMalice.run` as a potential callback, and thus reports the flow of location data to the Internet. The abductive inference algorithm can return the cut

$$\lambda = (\texttt{3.sendHTTP(data)} \notin S^*)$$

which as before guarantees that the program is free of explicit information flows.

While more precise analyses such as [34] exist for soundly identifying callbacks, they are still overapproximations, and furthermore may be prone to false negatives (e.g., failing to handle native code). Our approach is both simple to implement and sound.

4.2 Interactive Verification

We model the imprecision of static analysis by categorizing program facts as *may-facts* and *must-facts*. May-facts are facts that the static analysis cannot prove are false for all executions. For example, (`3.sendHTTP(data)` $\in S^*$) is a may-fact for $\mathcal{P}_{\text{onCreate}}$. Conversely, must-facts χ are facts that are shown to hold for at least one concrete execution of $\mathcal{P}$. For example, since `7.getLocation()` is executed by running `onCreate`, this is a must-fact for $\mathcal{P}_{\text{onCreate}}$, i.e.

$$\chi = (\texttt{7.getLocation()} \in S^*) \wedge (...).$$

Our static analysis takes as input a cut λ that asserts that some may-facts are false; e.g., the predicate

$$\lambda_{9,11} = (\texttt{9.leak(true,loc)} \notin S^*) \wedge (\texttt{11.leak(false,loc)} \notin S^*)$$

is a cut with which the static analysis can verify ϕ_{flow} for $\mathcal{P}_{\text{onCreate}}$. These assumptions have a (finite) lattice structure $(\Lambda, \leq, \top, \bot)$, where $\lambda \leq \mu$ means: if $\chi \wedge \lambda \models \phi$ holds, then $\chi \wedge \mu \models \phi$ holds as well (i.e., μ makes stronger assumptions than λ). For example, (`9.leak(true,loc)` $\notin S^*) \leq \lambda_{9,11}$ because $\lambda_{9,11}$ makes stronger assumptions. The predicate $\bot$ corresponds to no assumptions (and is guaranteed to hold), and $\top$ corresponds to assuming that all may-facts are false.

We are interested in the setting where predicates correspond to sets of program statements:

- Predicates λ correspond to sets $E_\lambda \subseteq S$:

  $$\lambda = \bigwedge_{s \in E_\lambda} (s \notin S^*),$$

  where S is the set of program statements and S^* is the set of reachable program statements. In other words, λ asserts that statements $s \in E_\lambda$ are not reachable.

- Conjunction of predicates corresponds to set union:

  $$E_{\lambda_1 \wedge \lambda_2} = E_{\lambda_1} \cup E_{\lambda_2},$$

  i.e., two cuts hold simultaneously if all of the statements from both cuts are unreachable.

- Partial order corresponds to set inclusion:

  $$\lambda \leq \mu \text{ if } E_\lambda \subseteq E_\mu.$$

  In other words, smaller sets make fewer (therefore, weaker) assumptions.

- Top and bottom: $E_\top = S$ and $E_\bot = \emptyset$.

Given $\lambda \in \Lambda$, our static analysis tries to prove $\chi \wedge \lambda \models \phi$ (i.e., it tries to prove ϕ assuming λ). When can we hope to find a valid cut λ that helps the static analysis prove ϕ? Consider three cases:

1. The static analysis proves $\chi \wedge \bot \models \phi$. Since $\bot$ always holds, the static analysis has proven that ϕ holds.
2. The static analysis cannot prove $\chi \wedge \bot \models \phi$, but proves $\chi \wedge \top \models \phi$. In this case, we can search for a valid cut $\lambda \in \Lambda$ with which the static analysis can prove ϕ.
3. The static analysis cannot prove $\chi \wedge \top \models \phi$. This means that even making best-case assumptions, the static analysis fails to prove ϕ, so no cut $\lambda \in \Lambda$ can help the static analysis prove ϕ.

In the first case, the app is already free of malicious information flows. In the third case, the app must be sent to the analyst for manual analysis. The second case is our case of interest, which we describe in more detail in the subsequent sections.

4.2.1 Abductive Inference

Our goal is to find a valid cut $\lambda \in \Lambda$ with which the static analysis can verify that the policy ϕ holds. Our central tool will be a variant of abductive inference where the known-facts χ are extracted from tests:

Definition 4.2.1 Given must-facts χ extracted from dynamic executions, the *abductive inference problem* is to find a cut $\lambda \in \Lambda$ such that

$$\chi \wedge \lambda \models \phi \text{ and } \mathrm{SAT}(\chi \wedge \lambda). \tag{4.1}$$

Additionally, we constrain λ to be *minimal*, i.e. there does not exist $\mu \in \Lambda$ satisfying (4.1) such that $\mu < \lambda$.

Abductive inference essentially allows us to compute minimal specifications λ that are simultaneously consistent with the must-facts χ and verify the policy ϕ. In our setting, the abductive inference problem corresponds to finding a set E_λ such that:

- Removing E_λ from S suffices to prove ϕ_{flow} (i.e., removing E_λ from $\mathcal{P}$ guarantees that there are no source-sink flows in the resulting app $\mathcal{P}'$).
- E_λ is consistent with must-facts χ (i.e., E_λ does not contain any statements observed during a dynamic execution of $\mathcal{P}$).
- E_λ has minimum size.

We assume access to an oracle we can query to obtain the tests used to extract must-facts χ:

Definition 4.2.2 An *oracle* $\mathcal{O}$ is a function that, on input cut λ and program $\mathcal{P}$, returns a test T_{new} showing that λ does not hold for $\mathcal{P}$, or returns $\emptyset$ if λ holds for $\mathcal{P}$.

In our setting, the oracle is the developer and the inferred cut λ is shown to the developer as the set of statements E_λ to be removed from the program. If the developer is not satisfied with E_λ, then the developer can produce a new test case such that the extracted must-facts χ_{new} satisfy $\text{UNSAT}(\chi_{\text{new}} \wedge \lambda)$—i.e., χ_{new} shows that E_λ contains reachable statements, so λ is invalid. The tool updates the must-facts $\chi \leftarrow \chi \wedge \chi_{\text{new}}$ and reruns the inference procedure. This process repeats until the developer is satisfied with E_λ, upon which verification is complete. This process is performed by the refinement loop in function $\textsc{InteractiveCut}(\mathcal{P}, T)$ in Algorithm 5.

4.2.2 Instrumenting Cuts

Given cut λ, our framework produces

$$\mathcal{P}' \leftarrow \textsc{Instrument}(\mathcal{P}, \lambda),$$

where $\mathcal{P}'$ is instrumented to abort if λ is violated. The instrumentation guarantees that λ holds for $\mathcal{P}'$, so ϕ holds for $\mathcal{P}'$ as well (as long as ϕ is not related to termination properties of $\mathcal{P}'$). Furthermore, if λ holds for $\mathcal{P}$, then $\mathcal{P}$ and $\mathcal{P}'$ are semantically equivalent. The procedure is summarized in Algorithm 5, and an overview of the system (for callgraph specifications discussed in Section 4.3) is shown in Figure 4.3.

The properties ϕ we have in mind are security policies, for example the policy ϕ_{flow} that no malicious explicit information flow occurs, and abductive inference computes a cut E_λ such that removing E_λ from $\mathcal{P}$ produces an app $\mathcal{P}'$ with no malicious flows. The instrumentation enforces E_λ simply by terminating execution if $s \in E_\lambda$ is reached. In our example in Figure 4.1, we instrument $\mathcal{P}_{\texttt{onCreate}}$ to ensure that the cut $\lambda_3 = (\texttt{3.sendHTTP(data)} \notin S^*)$ holds. Then $\textsc{Instrument}(\mathcal{P}_{\texttt{onCreate}}, \lambda_3)$ adds instrumentation that terminates $\mathcal{P}_{\texttt{onCreate}}$ if `3.sendHTTP(data)` is reached.

4.2.3 Improving Precision Using Multiple Cuts

We can improve precision by computing multiple sufficient cuts, which must *all* fail before the instrumentation terminates $\mathcal{P}'$. In other words, we want $\lambda_1, ..., \lambda_n$ such that

$$\chi \wedge (\lambda_1 \vee ... \vee \lambda_n) \models \phi \text{ and } \forall i,\ \text{SAT}(\chi \wedge \lambda_i).$$

However, we need to avoid choosing $\lambda_1 = ... = \lambda_n$ (since then the predicates are correlated). To do so, we assume that Λ comes with a meet operator $\sqcap$, where $\lambda \sqcap \mu$ should mean "intersection of specifications λ and μ". We require that the predicates be disjoint—i.e., $\lambda_i \sqcap \lambda_j = \bot$ for all

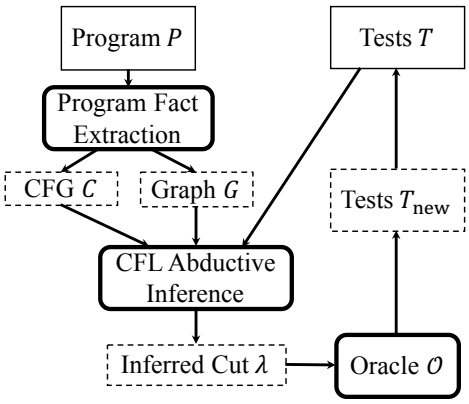


Figure 4.3: The interactive verification system. One iteration of the system proceeds as follows: (i) The system produces an inferred cut λ that suffices to prove absence of source-sink flows. (ii) The oracle $\mathcal{O}$ (which is the developer) either accepts λ, or generates a new test T_{new} showing that λ is invalid.

$1 \leq i < j \leq n$. This is stronger than requiring that the predicates λ_i are distinct, but maximizes the independence of the predicates, thus making it more likely that at least one of them holds.

In our setting, where predicates λ correspond to sets $E_\lambda \subseteq S$, the meet operator is intersection:

$$E_{\lambda \sqcap \mu} = E_\lambda \cap E_\mu.$$

This satisfies the requirement $\lambda \sqcap \mu \leq \lambda$ because $E_\lambda \cap E_\mu \subseteq E_\lambda$. Now, the condition $\lambda_i \sqcap \lambda_j = \bot$ says that our cuts E_{λ_i} should be non-intersecting: $E_{\lambda_i} \cap E_{\lambda_j} = \emptyset$.

We incrementally construct the predicates λ_i. Our first predicate is $\lambda_1 \leftarrow \textsc{Cut}(\phi, \chi, \Lambda)$. When computing λ_2, we need to ensure that $\lambda_1 \sqcap \lambda_2 = \bot$—i.e., we need to exclude every predicate in the *downward closure*

$$(\downarrow \{\lambda_1\}) = \{\mu \in \Lambda \mid \mu \leq \lambda_1\}$$

of λ_1 from consideration. To exclude these predicates, we add them to χ:

$$\chi_1 \leftarrow \chi \wedge \bigwedge_{\lambda \in (\downarrow \{\lambda_1\} - \{\bot\})} (\neg \lambda).$$

Now consider $\lambda_2 \leftarrow \textsc{Cut}(\phi, \chi_1, \Lambda)$. Let $\nu = \lambda_1 \sqcap \lambda_2$. Note that $\nu \in (\downarrow \{\lambda_1\})$, since $\nu \leq \lambda_1$. However, Cut returns λ_2 such that $\text{SAT}(\chi_1 \wedge \lambda_2)$, and $\chi_1 = (\neg \nu) \wedge (...)$ unless $\nu \notin (\downarrow \{\lambda_1\} - \{\bot\})$, so it must be the case that $\nu = \bot$.

Algorithm 5 Algorithm for interactively verifying $\mathcal{P}$. Here, the function CUT solves the abductive inference problem (algorithm described in Section 4.3), and the function EXTRACTFACTS constructs the must-facts χ.

```
procedure InteractiveCut(P, φ)
    T ← ∅
    while true do
        [χ, Λ] ←ExtractFacts(P, T)
        λ ←Cut(φ, χ, Λ)
        if λ = ∅ then
            return ∅
        end if
        T_new ← O(P, λ)
        if T_new = ∅ then
            return λ
        end if
        T ← T ∪ T_new
    end while
end procedure
procedure InteractiveVerify(P, φ)
    λ ←InteractiveCut(P, φ)
    return Instrument(P, λ)
end procedure
```

In general, after computing the first $i-1$ predicates $\{\lambda_1, ..., \lambda_{i-1}\}$, we compute

$$\chi_i \leftarrow \chi_{i-1} \wedge \bigwedge_{\mu \in (\downarrow\{\lambda_{i-1}\} - \{\bot\})} (\neg\mu),$$

and choose $\lambda_i \leftarrow$Cut(ϕ, χ_i, Λ). Algorithm 6 uses this procedure to compute $\alpha = \lambda_1 \vee ... \vee \lambda_n$. Note that at some point, the problem of computing CUT becomes infeasible, after which no new sufficient cuts can be computed.

In our setting,

$$\mu \in\downarrow \{\lambda\} \text{ if } E_\mu \subseteq E_\lambda.$$

To compute multiple cuts, we need an efficient way to compute the conjunction over $(\downarrow \{\lambda_i\} - \{\bot\})$. Note that

$$\bigwedge_{\mu \in (\downarrow\{\lambda_i\} - \{\bot\})} (\neg\mu) = \bigwedge_{s \in E_{\lambda_i}} (\neg\mu_{\{s\}}) = \bigwedge_{s \in E_{\lambda_i}} (s \in S^*).$$

To see the first equality, note that the conjunction on the right-hand side is over a subset of the conjunction on the left-hand side, so the left-hand side implies the right-hand side. Conversely, every $\mu \in (\downarrow \{\lambda_i\} - \{\bot\})$ can be expressed as a (nonempty) conjunction $\mu_{\{s_1\}} \wedge ... \wedge \mu_{\{s_m\}}$, where $s_1, ..., s_m \in E_{\lambda_i}$. Therefore $\neg\mu = (\neg\mu_{\{s_1\}}) \vee ... \vee (\neg\mu_{\{s_m\}})$, which is implied by the right-hand side.

Algorithm 6 Algorithm for computing multiple cuts.

procedure MULTIPLECUT(ϕ, χ, Λ, n)
  $\alpha \leftarrow$ false
  for all $1 \leq k \leq n$ **do**
    $\lambda_i \leftarrow$CUT(ϕ, χ, Λ)
    if $\lambda_i \neq \emptyset$ **then**
      $\alpha \leftarrow \alpha \vee \lambda_i$
      $\chi \leftarrow \chi \wedge \bigwedge_{\mu \in (\downarrow\{\lambda_i\} - \{\bot\})} (\neg\mu)$
    end if
  end for
  return α
end procedure

The resulting update rule is

$$\chi_i \leftarrow \chi_{i-1} \wedge \bigwedge_{s \in E_{\lambda_{i-1}}} (s \in S^*).$$

In other words, the next call to CUT assumes that every statement $s \in E_\lambda$ is in S^*. Since $\lambda_i \sqcap \lambda_j = \bot$ is equivalent to $E_{\lambda_i} \cap E_{\lambda_j} = \emptyset$, this condition is correctly enforced because χ is updated so that every statement that occurs in E_{λ_i} is prevented from occurring in E_{λ_j} (for $j > i$).

To instrument the program to enforce multiple cuts $\alpha = \lambda_1 \vee ... \vee \lambda_n$, we keep a global array of Boolean variables $[b_1, ..., b_n]$, all initialized to false. Whenever a predicate λ_i is violated, we update $b_i \leftarrow$ true. If $b_1 \wedge ... \wedge b_n$ ever becomes true, then all of the predicates λ_i have been violated and we terminate $\mathcal{P}'$.

4.3 Cuts for CFL Reachability

In this section, we describe an algorithm for performing interactive verification in the case of context-free language reachability. Let C be a context-free grammar, and let $G = (V, E)$ be a labeled graph constructed from a program $\mathcal{P}$. We assume for convenience that there is a single source $V_{\text{source}} = \{v_{\text{source}}\}$ and a single sink $V_{\text{sink}} = \{v_{\text{sink}}\}$. We consider policies ϕ of the form $\phi = (e^* \notin G^C)$, where $e^* = v_{\text{source}} \xrightarrow{T} v_{\text{sink}}$. This question can be answered in polynomial time [99]. However, the graph constructed by the static analysis in general is an approximation of the true graph $G^* = (V^*, E^*)$, i.e. $G^* \subseteq G$, which potentially introduces false positive source-sink paths.

In Section 4.2, we discuss predicates $E_\lambda \subseteq S$ corresponding to sets of program statements. In this section, we slightly modify notation and consider predicates $E_\lambda \subseteq E$ that correspond to sets of edges. All of our cuts are eventually converted to sets of statements—recall from Figure 2.3 that edges correspond directly to statements, except for edges of the form $v_{\text{source}} \xrightarrow{\text{SrcRef}} v$ and $v \xrightarrow{\text{RefSink}} v_{\text{sink}}$ that do not occur in our cuts.

The must-facts are predicates $(e \in G^*)$ where e is an edge certain to be in G^*, whereas the

may-facts are predicates ($e \in G^*$) where it is uncertain whether $e \in G^*$. Let $E_p \subseteq E$ be this set of *may-edges*; then $e \in E_p$ corresponds to may-fact ($e \in G^*$). Our goal is to infer specifications of the form

$$\lambda = \bigwedge_{e \in E_\lambda} (e \notin G^*)$$

where $E_\lambda \subseteq E_p$. In other words, λ specifies that the edges in E_λ are *not* in G^*. The partial ordering on the lattice Λ of specifications is $\lambda \leq \mu$ if $E_\lambda \subseteq E_\mu$; i.e., λ makes fewer assumptions about which edges are not in G^*.

In this setting, the abductive inference problem is to find a minimal subset $E_\lambda \subseteq E_p$ such that $\lambda \models \phi$ holds for $\mathcal{P}$. All else being equal, we prefer to find the smallest cuts possible, so we add the stronger constraint that $|E_\lambda|$ is minimized.

Definition 4.3.1 Let $G_\lambda = (V, E - E_\lambda)$; i.e. the subgraph of G with edges $e \in E_\lambda$ removed. A predicate $\lambda \in \Lambda$ is a *sufficient cut* if and only if $e^* \notin G_\lambda^C$. The *CFL reachability minimum cut problem* is to find a sufficient cut λ that minimizes $|E_\lambda|$—i.e., there does not exist any sufficient cut μ such that $|E_\mu| < |E_\lambda|$.

The CFL minimum cut problem is NP-hard—we give a proof in Appendix 4.6.

4.3.1 Algorithms for CFL Reachability Cuts

We describe a reduction of the minimum cut problem to an integer linear program (ILP). The objective of the ILP is to minimize $|E_\lambda|$ over the set of predicates $\{\lambda \in \Lambda \mid e^* \notin G_\lambda^C\}$. We need to translate the constraints on λ into linear inequalities. To do so, we first recast the problem by introducing the Boolean variables $\delta_e = (e \notin G_\lambda^C) \in \{0, 1\}$ for every edge $e \in G^C$—i.e., $\delta_e = 1$ if removing E_λ from E causes e to be removed from G. We can recover E_λ given the values δ_e, i.e. $E_\lambda = \{e \in E_p \mid \delta_e = 1\}$. In this formulation, the objective is to minimize $|E_\lambda| = \sum_{e \in E_p} \delta_e$.

Recall that $e = v \xrightarrow{A} v' \in G_\lambda^C$ if there exists $e' = v \xrightarrow{B} v''$ and $e'' = v'' \xrightarrow{D} v'$ in G_λ^C such that $A \to BD \in C$ (we describe the case of binary productions—the case of unary productions is similar); we denote such a triple as $e \to e'e''$. Then $\delta_e \Rightarrow (\delta_{e'} \vee \delta_{e''})$ must hold—i.e., e is removed from G_λ^C only if either e' or e'' is removed from G_λ^C. Next, for the source-sink edge e^*, we add constraint $\delta_{e^*} = 1$, which enforces $(e^* \notin G_\lambda^C)$. Finally, we require that $\delta_e = 0$ for edges $e \in E - E_p$, since these edges cannot be removed from the graph. These constraints translate into linear inequalities:

1. Productions: $\delta_e \leq \delta_{e_1} + ... + \delta_{e_k}$ for every production $e \to e_1 ... e_k$ ($k \in \{1, 2\}$).
2. Remove the source-sink edge: $\delta_{e^*} = 1$.
3. Retain must-edges: $\delta_e = 0$ for every $e \in E - E_p$

The first set of constraints follows because $\delta_e = 1$ only if $\delta_{e_i} = 1$ for some $1 \leq i \leq k$.

The number of constraints generated by this approach is intractable for the typical ILP solver, so we introduce two optimizations to reduce the number of constraints. First, we construct the constraints in a top-down manner—i.e., we only include productions contained in some derivation of e^*. If an edge $e \in G^C$ is not contained in any derivation of e^*, then the presence of e in G^C_λ does not affect the presence of e^* in G^C_λ, so e can be ignored. This optimization is implemented by first processing all productions $e^* \to e_1...e_k$ ($k \in \{1, 2\}$); for every input e_i, we recursively add productions for e_i, which recursively adds every production in some derivation of e^*.

Second, any facts added to G^C produced from only the must-edges (i.e., from edges $e \in E - E_p$) are present in G^C_λ for every $\lambda \in \Lambda$. Note that the graph $G_\top = (V, E - E_p)$ contains no edges $e \in E_p$, so the edges $e \in G^C_\top$ are produced by must-facts alone. This means that we can first compute $G^C_\top$, and then only include variables δ_e for $e \in (G^C - G^C_\top)$. More precisely, consider a production $e \to e'e''$:

1. If $e', e'' \in G^C_\top$, then $e \in G^C_\top$, so we do not add any constraints.
2. If $e' \in G^C_\top$ but $e'' \notin G^C_\top$, then we treat this as the *unary* production $e \to e''$.
3. If $e', e'' \notin G^C_\top$, then we treat this as $e \to e'e''$ as before.

Algorithm 7 summarizes the procedure. The above discussion shows that the set E_p returned by Algorithm 7 solves CFL reachability minimum cut problem.

In practice, we include one additional constraint. For $\sigma \in \Sigma$, edges $e_1 = v \xrightarrow{\sigma} v'$ and $e_2 = v' \xrightarrow{\overline{\sigma}} v$ are distinct edges, but they are derived from the same program fact. To account for this, we impose the additional constraint $\delta_{e_1} = \delta_{e_2}$ for such pairs of edges.

For example, consider the graph in Figure 4.2. The graph shown in Figure 4.4 summarizes the possible derivations of the edge $\texttt{LOC} \xrightarrow{\text{SrcSink}} \texttt{HTTP}$ from the terminal edges; to distinguish this graph from G^C we refer to the edges of this graph as *arrows* and the vertices as *nodes*. There are two types of nodes—nodes corresponding to *productions* $e \to e_1...e_k$ (shown as black circles), and nodes corresponding to edges in G^C (shown as boxes containing the corresponding edge). Each production $e \to e_1...e_k$ has one incoming arrow from e, and one outgoing arrow to each of the edges $e_1, ..., e_k$. Let

$$E_p = \{v \xrightarrow{\text{Assign}} v' \mid v \text{ formal return value}\} \cup \{v \xrightarrow{\text{Assign}} v' \mid v' \text{ formal parameter}\}.$$

In other words, E_p is the set of edges corresponding to method invocations (recall that we treat each method invocation `x=foo(y)` as an assignment from argument `y` to formal parameter `foo.param` and an assignment from formal return value `foo.return` to the defined variable `x`).

Each production generates one constraint $\delta_e \leq \delta_{e_1} + ... + \delta_{e_k}$ in the ILP, though these constraints

Algorithm 7 This algorithm solves the CFL reachability minimum cut problem. Here, $\mathcal{S}$ maps variables δ_e to their value in the solution to the ILP.

procedure CFLCUT(C, G, E_p)
  $G^C \leftarrow$CLOSURE(C, G); $G^C_\top \leftarrow$CLOSURE($C, G - E_p$)
  $\mathcal{C} \leftarrow \{\delta_{e^*} = 1\}$
  $W \leftarrow [e^*]$; $X \leftarrow \{e^*\}$
  while $\neg W$.EMPTY() **do**
    $e \leftarrow W$.POP()
    for all $e \rightarrow e_1 ... e_k$ **do**
      $F \leftarrow \{e_i \mid e_i \notin G^C_\top\}$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{\delta_e \leq \sum_{e \in F} \delta_e\}$
      $W \leftarrow W$.CONCAT($[e \in F \mid e \notin X]$)
      $X \leftarrow X \cup \{e \in F \mid e \notin X\}$
    end for
  end while
  $\mathcal{S} \leftarrow$SOLVEILP($\min \sum_{e \in E_p} \delta_e, \mathcal{C}$)
  return $\{e \in E_p \mid \mathcal{S}(\delta_e) = 1\}$
end procedure

are simplified using the two optimizations described above. Figure 4.5 shows the constraints generated by Algorithm 7. Constraint 1 enforces that the SrcSink edge is in the cut. Constraint 2 enforces the production

$$(\texttt{LOC} \xrightarrow{\text{SrcSink}} \texttt{HTTP}) \Rightarrow (\texttt{LOC} \xrightarrow{\text{SrcRef}} r_{\texttt{getLocation}} \xrightarrow{\overline{\text{FlowsTo}}} o_{\text{location}} \xrightarrow{\text{FlowsTo}} \texttt{param} \xrightarrow{\text{RefSink}} \texttt{HTTP})$$

(where we used $\Rightarrow$ to denote the production), but the first, second, and fourth edges on the right-hand side of the production are in $G^C_\top$, so they are not included in the constraint. The third edge $o_{\text{location}} \xrightarrow{\text{FlowsTo}} \texttt{param}$ is produced from the three edges

$$r_{\texttt{getLocation}} \xrightarrow{\text{Assign}} \texttt{loc} \xrightarrow{\text{Assign}} \texttt{data} \xrightarrow{\text{Assign}} \texttt{param},$$

which is captured by Contraints 3-5.

4.4 Implementation

We have implemented the interactive verification algorithm for our static information flow analysis described in Chapter 2. We use the ILP solver SCIP [2]. For computing cuts, we use a variant of our information flow analysis that is not context-sensitive, but we compute a 2-CFA points-to analysis in BDDBDDB and use it to filter the points-to set we compute. More precisely, during the computation of the transitive closure of G, whenever an edge $e = o \xrightarrow{\text{FlowsTo}} v$ is produced, we check if the 2-CFA points-to set contains e. If not, we remove e from the graph and continue.

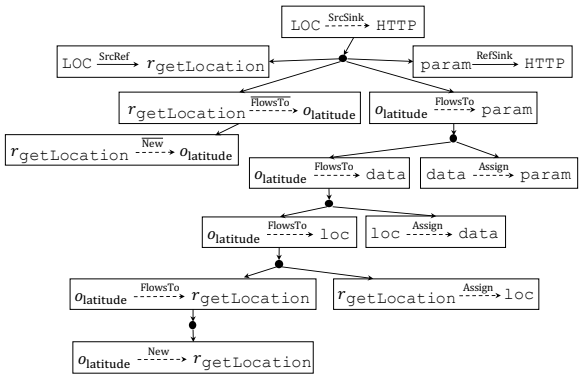


Figure 4.4: The derivation tree for the edge $r_{\texttt{getLocation}} \xrightarrow{\text{SrcSink}} \texttt{param}$ in the graph in Figure 4.2.

$$\max \left\{ \delta(r_{\texttt{getLocation}} \xrightarrow{\text{Assign}} \texttt{loc}) + \delta(\texttt{data} \xrightarrow{\text{Assign}} \texttt{param}) \right\}$$

subject to

1. $\delta(r_{\texttt{getLocation}} \xrightarrow{\text{SrcSink}} \texttt{param}) = 1$
2. $\delta(r_{\texttt{getLocation}} \xrightarrow{\text{SrcSink}} \texttt{param}) \leq \delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{param})$
3. $\delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{param}) \leq \delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{data}) + \delta(\texttt{data} \xrightarrow{\text{Assign}} \texttt{param})$
4. $\delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{data}) \leq \delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{loc}) + \delta(\texttt{loc} \xrightarrow{\text{Assign}} \texttt{data})$
5. $\delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} \texttt{loc}) \leq \delta(o_{\texttt{latitude}} \xrightarrow{\text{FlowsTo}} r_{\texttt{getLocation}}) + \delta(r_{\texttt{getLocation}} \xrightarrow{\text{Assign}} \texttt{loc})$

Figure 4.5: The integer linear program (ILP) corresponding to the productions shown in Figure 4.4.

App	LOC	Mal.	F/TP	E_p	$\lvert\{e^*\}\rvert$	$\lvert E_p\rvert$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{C}_{\text{opt}}\rvert$	$\frac{\lvert\mathcal{C}_{\text{opt}}\rvert}{\lvert\mathcal{C}\rvert}$	Time	$\lvert E_{\lambda_1}\rvert$	$\lvert E_{\lambda_2}\rvert$
411524	389K	Yes	TP	$E_p^{\text{p+r}}$	4	28K	144K	1982K	264K	0.13	64.4	6	7
0C2B78	322K	Yes	TP	E_p^{p}	3	23K	491K	5076K	956K	0.19	29.5	8	10
f7d928	258K	Yes	TP	E_p^{p}	4	48K	882K	18969K	1683K	0.089	663.2	11	25
tingshu	240K	Yes	TP	E_p^{p}	5	27K	655K	7530K	1280K	0.17	101.3	26	36
16677	200K	Yes	TP	$E_p^{\text{p+r}}$	4	40K	423K	6896K	809K	0.12	243.2	4	5
phone	198K	Yes	TP	E_p^{p}	3	8K	83K	968K	156K	0.16	11.9	11	11
583cc9	195K	Yes	TP	E_p^{p}	4	45K	792K	12575K	1526K	0.12	276.3	20	23
da8c48	190K	Yes	TP	E_p^{p}	1	5K	6K	166K	8K	0.051	0.2	1	1
4292c1	155K	Yes	TP	$E_p^{\text{p+r}}$	3	40K	1258K	10155K	2453K	0.24	92.5	1	1
5127eb	142K	Yes	TP	$E_p^{\text{p+r}}$	2	43K	289K	4579K	548K	0.12	427.0	5	5
1c2514	100K	Yes	TP	$E_p^{\text{p+r}}$	1	28K	347K	3182K	649K	0.20	181.7	3	4
wifi	98K	Yes	TP	$E_p^{\text{p+r}}$	3	31K	579K	6568K	1129K	0.17	281.3	8	16
browser	346K	No	FP	E_p^{p}	4	51K	669K	13718K	129K	0.094	32.1	7	19
00714C	248K	Yes	FP	E_p^{p}	4	51K	986K	16784K	1922K	0.11	118.1	21	25
highrail	247K	Yes	FP	$E_p^{\text{p+r}}$	3	39K	587K	9310K	1130K	0.12	451.5	9	9
flow	131K	No	FP	$E_p^{\text{p+r}}$	4	31K	409K	4759K	792K	0.17	48.8	11	12
calendar	125K	Yes	FP	$E_p^{\text{p+r}}$	4	31K	226K	3916K	430K	0.11	13.2	5	5
19780d	87K	Yes	FP	$E_p^{\text{p+r}}$	4	28K	244K	3742K	467K	0.13	894.7	21	22
aab740	86K	Yes	FP	$E_p^{\text{p+r}}$	4	28K	241K	3695K	461K	0.13	306.0	21	21
9d1da3	56K	Yes	FP	$E_p^{\text{p+r}}$	5	20K	217K	4321K	420K	0.097	16.9	4	4
018ee7	53K	Yes	FP	$E_p^{\text{p+r}}$	3	21K	148K	1952K	268K	0.098	9.8	5	5
ca70f4	44K	Yes	FP	$E_p^{\text{p+r}}$	3	10K	57K	456K	106K	0.23	10.4	3	4
battery	33K	Yes	FP	$E_p^{\text{p+r}}$	3	14K	98K	1076K	185K	0.17	377.1	12	13
7d43c8	27K	Yes	FP	$E_p^{\text{p+r}}$	4	9K	33K	368K	60K	0.16	5.6	3	4
Avg.	83K	–	–	–	2.32	12K	76K	2437K	338K	15.9	42.4	5.85	7.74

Figure 4.6: Statistics for some of the Android apps used in the experiments: the number of lines of Jimple bytecode ("LOC"), whether the app is malware ("Mal."), whether the app exhibited a true or false positive information flow ("F/TP"), the number of source-sink edges $|\{e^*\}| = |\{v_{\text{source}} \xrightarrow{T} v_{\text{sink}} \in G^C\}|$, the number of may-edges $|E_p|$, the number of variables $|\mathcal{V}|$ in the ILP, the unoptimized number of constraints $|\mathcal{C}|$ and the optimized number of constraints $|\mathcal{C}_{\text{opt}}|$, the percentage $|\mathcal{C}_{\text{opt}}|$ compared to $|\mathcal{C}|$, the running time of the ILP solver in seconds ("Time"), and the size of the first cut $|E_{\lambda_1}|$ and the second cut $|E_{\lambda_2}|$ (both on the first iteration of our algorithm). Where relevant, we give statistics for the largest ILP solved for the given app. Also, we include the average values over the entire corpus of 77 apps (where E_p is taken to be E_p^{p}).

4.5 Evaluation

We demonstrate the effectiveness of our approach by interactively verifying a corpus of 77 Android apps, including battery monitors, games, wallpaper apps, and contact managers. These apps are a combination of malware samples and a few benign apps obtained from a major security company. The malware in this corpus contain malicious functionalities that leak sensitive information (contact data, GPS location, and the device ID) to the Internet. We have ground truth on what information is leaked for each app. Our goal is to apply Algorithm 5 to produce apps proven not to leak sensitive information. The security policy is

$$\phi_{\text{flow}} = (v_{\text{source}} \xrightarrow{\text{SrcSink}} v_{\text{sink}} \notin G^{C_{\text{flow}}}$$

(with multiple source vertices v_{source} and sink vertices v_{sink}), where C_{flow} is the context-free grammar encoding the explicit information flow analysis described in Chapter 2.

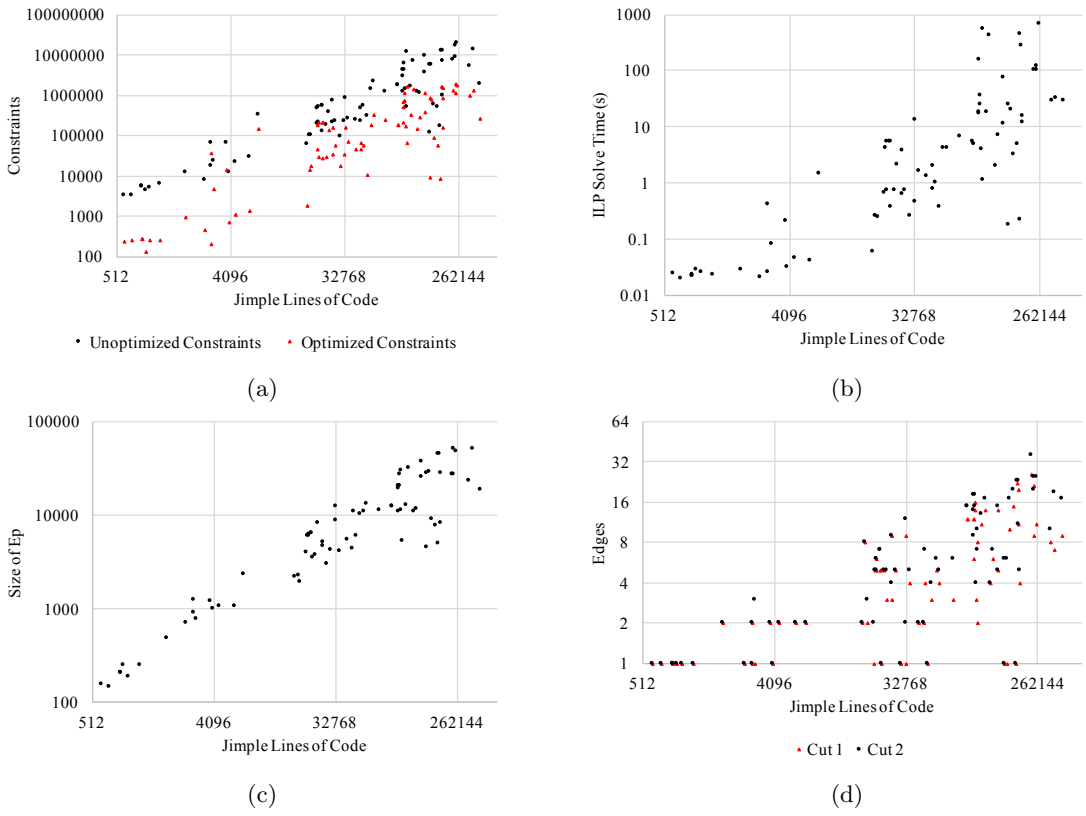


Figure 4.7: Statistics of the constraint system and resulting cuts for the corpus of 77 Android apps, plotted on a log-log scale: (a) number of unoptimized (black, circle) and optimized (red, triangle) constraints, (b) ILP solve time in seconds, (c) size of the search space E_p, (d) size of the first cut E_{λ_1} (red, triangle) and the second cut E_{λ_2} (black, circle).

As described in Section 4.1.1, we prune the program by removing provably unreachable statements before computing information flows. Also, we make worst-case assumptions about program entry points—i.e., we assume that every potential callback is an entry point (recall that a potential callback is any method in the application that overrides a method in the Android framework).

We consider cuts consisting of method invocation statements and return statements, since these statements determine interprocedural reachability. As described in Section 4.3, this corresponds to choosing $E_p = E_p^{\mathrm{p+r}}$, where

$$\begin{aligned} E_p^{\mathrm{p}} &= \{v \xrightarrow{\text{Assign}} v' \mid v' \text{ formal parameter}\} \\ E_p^{\mathrm{r}} &= \{v \xrightarrow{\text{Assign}} v' \mid v \text{ formal return value}\} \\ E_p^{\mathrm{p+r}} &= E_p^{\mathrm{p}} \cup E_p^{\mathrm{r}}. \end{aligned}$$

The cut λ asserts that certain edges in E_p cannot happen. If an edge in E_p^{p} or E_p^{r} is cut, then we add a statement `assert(false)` immediately before the corresponding method invocation statement.

To scale to some of the largest apps in our corpus, we needed to restrict our search space of cuts—for these apps we use $E_p = E_p^{\mathrm{p}}$ as the search space. Restricting the size of the search space can increase the size of the cuts (since the search space is strictly smaller), but in our experiments the cuts are still reasonably sized. For apps where our algorithm scaled using both E_p^{p} and $E_p^{\mathrm{p+r}}$, using E_p^{p} led an increase in cut size by at most a factor of about two (typically less).

In our first experiment, we run our tool on the corpus of apps and give statistics for the cuts we generate (Section 4.5.1). In our second experiment, we iteratively generate specifications that describe reachable code using Algorithm 5 (Section 4.5.2).

4.5.1 Inferring Cuts

We ran our tool on all the apps in our corpus. The results for twelve of the largest apps, along with all apps with false positives, are shown in Figure 4.6. We computed two cuts for each app, and include the sizes of each of these cuts in Figure 4.6. In our experience, additional cuts progressively became larger and less useful to examine (since the size of the search space reduces on every iteration), though in principle this process can safely be repeated until no new cuts can be produced—ϕ_{flow} continues to hold and having more cuts can only enlarge the set of allowed program behaviors.

We also include some statistics on the sizes of the constraint systems generated by Algorithm 7—these statistics are for the constraint system used to compute the first cut (which is the largest constraint system, though typically the size is similar for other runs). We have shown both the number of unoptimized constraints generated along with the number of constraints after applying the optimizations described in Section 4.3. Additionally, we include the average values over all 77 apps in the corpus (using $E_p = E_p^{\mathrm{p}}$ for consistency).

We have plotted some of these statistics in Figure 4.7 for all the apps in the corpus (again, using

$E_p = E_p^{\mathrm{p}}$ for consistency). In (a), we compare the size of the unoptimized constraint system to the size after applying optimizations. As can be seen, the optimized constraint system typically reduces the size by an order of magnitude ($\approx 10\times$). The unoptimized constraint systems typically proved to be intractable for the ILP solver to optimize, but with the optimizations the solver always terminated and finished fairly quickly. We also show the running time for the ILP in (b). As can be seen, our algorithm scales well to apps with hundreds of thousands of lines of Jimple bytecode.

In (c), we show the size of E_p^{p}—this gives a sense of the size of the search space of cuts, since there are $2^{|E_p^{\mathrm{p}}|}$ possible cuts. In (d), we show the sizes of the first two cuts produced. Most of the cuts have fewer than 16 edges, though the largest size for the first cut is 26 edges, and the largest size for the second cut is 36 edges. All of these cuts are sufficiently small so that the developer can easily verify whether the cut is valid. This suggests that the interactive verification process places little work on the developer; we further evaluate this workload in our second experiment.

4.5.2 Interactive Verification

In our second experiment, we manually carried out the procedure described in Algorithm 5 to produce verified apps $\mathcal{P}'$. Because the app developer is absent, we play the role of the developer. However, we are disadvantaged compared to the app developer: we only have the app bytecode, have superficial knowledge of the app's intended functionality, and lack access to the testing tools available to the developer. Furthermore, many of the apps crash when we try to run them due to incompatibilities with the Android emulator.

Thus, we provide the reachability information to our tool manually, determining which statements are reachable by reading the bytecode. The cuts are presented as a list of statements to be removed from the app, and we mark each statement as reachable or unreachable based on our inspection. For those apps that did not crash in the emulator (about half of the 12 apps) we also ran tests and found that reachability information was consistent with our specifications. In practice, we expect developers to write tests for Android apps using GUI testing frameworks such as Espresso [48].

We focused our efforts on producing cuts only for the false positives produced by our explicit information flow analysis. If the flow is a true positive, then no cut exists, so the analyst must necessarily inspect the app to determine whether it is malicious. As a consequence, in these cases little can be done to reduce the analyst workload.

The apps with false positive flows are shown in the second half of Figure 4.6. For each of these apps, we show the source of the false positive flow in Figure 4.8, and whether we determined that the cause of the false positive is due to unreachable code. These apps typically have other true positive flows—we include only sources that have false positive flows in ϕ_{flow} when performing the verification process (or else ϕ_{flow} would be false for the app).

In Figure 4.8, we show the results of our interactive verification process. We show two iterations of the process. For each iteration, we show the size of the cuts λ_1 and λ_2, along with the validity of

each cut. The inspection of the cuts proceeded until a valid cut was found, or it was determined that no cut was possible. After just two iterations of Algorithm 5, we succeeded in producing valid cuts for all apps with false positive explicit information flows, except for the app with a false positive not due to unreachable code. This means that only two interactions with the developer were necessary. The cuts remained small after the second iteration, which shows that the entire process is feasible for the developer to carry out. In the case of the final application (browser), because the false positive was not due to unreachable code, no valid cut can be produced by our method, which means that the app would be flagged for manual review.

To demonstrate how each step of Algorithm 5 contributes to verifying each app, Figure 4.9 plots the number of apps remaining to be verified at each step. As can be seen, the first cut on the first iteration alone clears many of the apps (6 out of 12), and the second cut on the first iteration clears an additional app. The first cut on the second iteration clears three of the remaining apps, and the second cut clears an additional app, leaving only one app that our process failed to verify.

Whereas the analyst would initially have had to analyze all 12 false positives, our approach reduces the analyst's workload to a single false positive. In our setting, this may not seem like a huge improvement, because the analyst still needs to analyze the true positive apps. However, our corpus of apps is heavily biased towards apps with malicious behaviors. In practice, the overwhelming majority of apps received by an app store are benign, which means that even a small false positive rate leads to a huge ratio of false positives to true positives that the analyst must analyze. We achieve a 92% reduction in the number of false positives that need to be discharged by the analyst, which enables the analyst to better focus effort.

While we cannot evaluate the workload required of the developer, we describe our own experience inspecting cuts. In 10 of the cases (including the invalid cut), the cuts were very easy to evaluate, taking only a few minutes, and we are very confident of the results. The remaining 2 cases were considerably more difficult, and took up to two hours each, leaving more room for error. This difficulty was primarily a consequence of code obfuscation. We believe that it would be significantly easier for the developer, who understands the app and has source code, to examine the cuts. While most cuts were in third-party libraries, the developer has knowledge of which library features they use, which should aid them in evaluating the correctness of the cut. Furthermore, developers often maintain high-coverage test suites, which we believe would also aid the process.

We found two sources of imprecision that led to the false positives. The first was the presence of a conditional to the following effect:

```
if (hasLocationPermission()) { leakLoc(); }
```

In cases where the app did not have permissions to access location, this caused the information flow analysis to report a false positive. The second was due to our sound assumption that every potential callback is an entry point, which caused unreachable code to be marked as reachable. In both cases, our algorithm can find cuts removing the unreachable code. In the case of the app for which no

App	v_{source}	Cause	Iteration 1				Iteration 2			
			$\lvert E_{\lambda_1} \rvert$	$\lvert E_{\lambda_2} \rvert$	I_1	I_2	$\lvert E_{\lambda_1} \rvert$	$\lvert E_{\lambda_2} \rvert$	I_1	I_2
browser	location	u.k.	5	6	No	No	5	None	No	No
00714C	contacts	u.r.	2	8	Yes	-	-	-	-	-
highrail	device ID	u.r.	1	2	Yes	-	-	-	-	-
flow	contacts	u.r.	2	3	Yes	-	-	-	-	-
calendar	location	u.r.	3	3	Yes	-	-	-	-	-
19780d	contacts	u.r.	1	1	No	No	2	9	Yes	-
aab740	contacts	u.r.	1	1	No	No	2	9	Yes	-
9d1da3	location	u.r.	4	4	No	No	5	5	No	Yes
018ee7	location	u.r.	4	4	Yes	-	-	-	-	-
battery	location	u.r.	8	8	No	Yes	-	-	-	-
ca7b26	location	u.r.	4	4	Yes	-	-	-	-	-
7d43c8	location	u.r.	3	4	No	No	4	4	Yes	-

Figure 4.8: Size and validity of cuts generated by Algorithm 7 for apps with false positive flows. "None" means no cut could be generated. For "Cause", "u.k." means the cause is unknown, and "u.r." means the information flow is unreachable. The values I_i indicate whether the ith cut is sufficient to prove the safety property ϕ_{flow}, i.e., $I_i = \chi \wedge \lambda_i \stackrel{?}{\models} \phi_{\text{flow}}$.

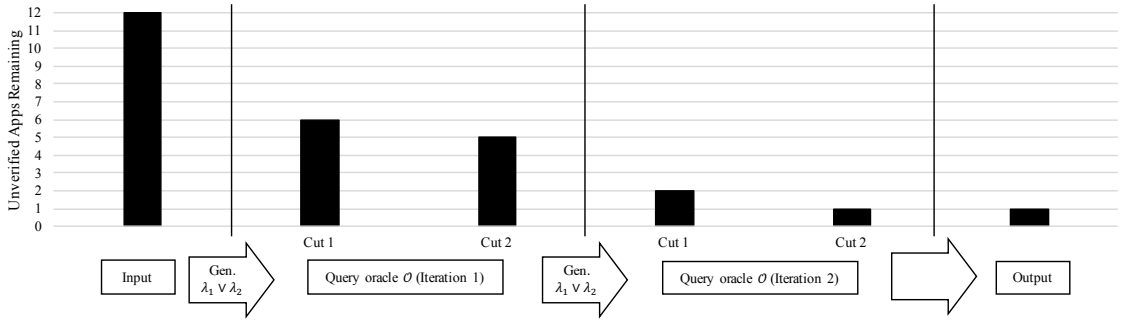


Figure 4.9: Visualization of how many apps are successfully verified at each step of the process. Algorithm 5 is run on each of the 12 input apps that have a false positive explicit information flow. The x-axis describes the various points in the process, and the y-axis describes the number of apps remaining to be verified at each point.

cut could be found, we believe the false positive was due to insufficient context sensitivity, not flows through unreachable code.

4.6 CFL Minimum Cut is NP-Hard

Theorem 4.6.1 The CFL minimum cut problem is NP-hard.

Proof: We prove the theorem by reducing the minimum vertex cover problem to the CFL minimum cut problem. Consider the minimum vertex cover problem for a given undirected graph $G = (V, E)$, where $V = \{v_1, ..., v_n\}$. We construct the following directed, labeled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and context-free grammar C such that a CFL minimum cut for $\mathcal{G}$ and C corresponds to a minimum vertex cover for G.

Context-free grammar C. The context-free grammar C is defined to be:

1. Alphabet $\Sigma = \{a, b, c, d\}$.
2. A single production $T \to abcbd$, where T is the start symbol.

Graph $\mathcal{G}$. The graph $\mathcal{G}$ is defined to be:

1. We have vertices $x^*, y^* \in \mathcal{G}$.
2. For each $i \in \{1, ..., n\}$, we have vertices $x_i, y_i \in \mathcal{G}$.
3. For each $i \in \{1, ..., n\}$, $\mathcal{E}$ contains the edges
$$x^* \xrightarrow{a} x_i \xrightarrow{b} y_i \xrightarrow{d} y^*.$$
4. For each edge $(v_i, v_j) \in E$, we have edges
$$y_i \xrightarrow{c} x_j, \qquad y_j \xrightarrow{c} x_i.$$

CFL minimum cut problem. Finally, the specification of the CFL minimum cut problem is as follows:

1. The source vertex is x^*.
2. The sink vertex is y^*.
3. The edges labeled b have weight 1.
4. All other edges have weight ∞.

CFL minimum cut $\Rightarrow$ vertex cover. First, we claim that given a cut $\mathcal{E}_{\text{cut}} = \{x_i \xrightarrow{b} y_i\}$, the corresponding vertices
$$V_{\text{cover}} = \{v_i \mid x_i \xrightarrow{b} y_i \in \mathcal{E}_{\text{cut}}\}$$
form a cover. To see this, note that for every edge $(v_i, v_j) \in E$, we have path
$$x^* \xrightarrow{a} x_i \xrightarrow{b} y_i \xrightarrow{c} x_j \xrightarrow{b} y_j \xrightarrow{d} y^*$$
in $\mathcal{E}$. By the definition of a cut, we know that
$$x_i \xrightarrow{b} y_i \in \mathcal{E}_{\text{cut}} \text{ or } x_j \xrightarrow{b} y_j \in \mathcal{E}_{\text{cut}}.$$
As a consequence, by the definition of V_{cover}, we have
$$v_i \in V_{\text{cover}} \text{ or } v_j \in V_{\text{cover}},$$

so V_{cover} is a vertex cover as claimed.

Vertex cover ⇒ CFL minimum cut. Conversely, we claim that given a cover $V_{\text{cover}} = \{v_i\}$, the corresponding edges

$$\mathcal{E}_{\text{cut}} = \{x_i \xrightarrow{b} y_i \mid v_i \in V_{\text{cover}}\}$$

form a cut. To see this, note that for every path

$$x^* \xrightarrow{a} x_i \xrightarrow{b} y_i \xrightarrow{c} x_j \xrightarrow{b} y_j \xrightarrow{d} y^*,$$

we have $(v_i, v_j) \in E$, so by the definition of a cover, we have

$$v_i \in V_{\text{cover}} \text{ or } v_j \in V_{\text{cover}}.$$

As a consequence, by the definition of $\mathcal{E}_{\text{cut}}$, we have

$$x_i \xrightarrow{b} y_i \in \mathcal{E}_{\text{cut}} \text{ or } x_j \xrightarrow{b} y_j \in \mathcal{E}_{\text{cut}}.$$

Furthermore, every CFL source-sink path in $\mathcal{G}$ has this form, so $\mathcal{E}_{\text{cut}}$ is a cut as claimed.

Finally, note that for both directions of the proof,

$$|V_{\text{cover}}| = |\mathcal{E}_{\text{cut}}|,$$

so in particular, a minimum cut corresponds to a minimum cover (and vice versa). □

4.7 Conclusion

Given a program $\mathcal{P}$ and a policy ϕ, our framework minimally instruments $\mathcal{P}$ to ensure that ϕ holds. This instrumentation is guaranteed to be consistent with given test cases, and furthermore the developer can interact with the process to produce suitable cuts. Our approach to handling false positives has the potential to make automated verification of the absence of explicit information flows a more practical approach for security analysts to produce safe and usable programs. We have applied this approach to verify the absence of malicious explicit information flows in a corpus of 77 Android apps. For 11 out of 12 false positives information flows we found, our tool produced valid cuts to enforce ϕ_{flow}.

In our experience, Android malware to date does not rely on sophisticated techniques to hide malicious behavior. We believe this is because such malware predominantly appears on third-party app stores where sophisticated security auditing (either manual or automatic) is unavailable. Android malware is likely to become more sophisticated over time, in which case the limitations

in our static analysis may be exploited. In particular, it may be interesting to study the following limitations to our current analysis:

- Implicit flows: While we do not take into account the possibility of implicit flows in the application [123], we can easily extend our technique to do so—we can include "transfer" edges in the analysis that pass taint from variables used in conditionals to variables used in branches.
- Exception analysis: Our analysis does not currently track flows due to exceptional control flow. There has been recent work on exception handling [27].
- Reflection: Our analysis cannot resolve method calls made using the Java reflection API, so we treat such calls as no-ops. There has been recent work on handling reflective method calls [23, 89].
- Missing models: Our information flow analysis depends on *information flow models* [18, 39, 161], which means that missing models can introduce unsoundness into our analysis. For the apps in our experiments, we have carefully searched for potential missing models.

For each of these settings, a key challenge is handling the high false positive rate from a sound analysis (implicit flows [80], exceptions [27], reflection [23], and missing models [18]). Our technique may therefore be particularly applicable to these settings, though the search space of cuts may need to be modified.

Chapter 5

Active Learning of Points-To Specifications

In this chapter, we propose an algorithm that leverages observations from concrete executions to infer points-to specifications. Two constraints make our problem substantially more challenging than previous algorithms for inferring specifications from concrete executions:

- Points-to effects cannot be summarized for a library function in isolation, e.g., in Figure 5.1, `set`, `get`, and `clone` all refer to the mutual field `f`.
- We may not be able to instrument library code, e.g., native code.

Now, suppose our algorithm proposes a candidate specification, and we want to check whether this candidate is "correct". More precisely, we want to ensure that the candidate is *admissible*, i.e., there is no strictly better specification. The first constraint says that the candidate must simultaneously summarize the points-to effects of `set`, `get`, and `clone`, and the second constraint says that we can only use input-output examples to check the admissibility.

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

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

Figure 5.1: An example of a program using the `Box` class in the library (right), and the implementation of the library functions `set`, `get`, and `clone` in the `Box` class.

We introduce *path specifications* to describe points-to effects of library code. Each path specification summarizes a single points-to effect of a combination of functions. An example is:

> For two calls `box.set(x)` and `box.get(0)`, the return value of `get` may alias `x`.

Path specifications have two desirable properties:

- We can check if a candidate path specification is admissible using input-output examples.
- A set of individually admissible path specifications is admissible as a whole.

These two properties imply that we can infer path specifications incrementally. In particular, we formulate the problem of inferring path specifications as a *language inference problem* [110], and we develop a language inference algorithm tailored to our problem instance. Our algorithm builds the language incrementally in two phases—it first infers a finite language of path specifications that it is certain are admissible (leveraging the two properties described above), and then inductively generalizes this language while trying to retain admissibility.

We implement our algorithm in a tool called Atlas[1], which infers path specifications for functions in Java libraries. In particular, we evaluate Atlas by using it to infer specifications for the Java Collections API, since this API contains many functions that exhibit complex points-to effects. Atlas infers the correct specifications for 97% of these functions. Previously, we had manually written points-to specifications for the Java Collections API—Atlas inferred 10× as many specifications.

We compare our specifications to handwritten specifications on a benchmark of 46 Android apps. Using these inferred specifications increases the precision of our static points-to analysis by 53% compared to analyzing the library code, and increases recall by 20% compared to using handwritten specifications. While the specifications synthesized by Atlas are incomplete, we show that using the inferred specifications achieves 76% recall for nontrivial points-to edges, including 100% recall for almost half the programs in our benchmark. Our contributions are:

- We introduce path specifications, and prove that they are sufficiently expressive to precisely model the library code when using a standard flow-insensitive points-to analysis.
- We formulate the problem of inferring path specifications as a language inference problem, and we design a language inference algorithm tailored to our problem instance.
- We implement our approach in Atlas, and use it to infer a large number of useful specifications for the Java Collections API.

5.1 Background

For the purposes of this chapter, we introduce a modified version of the grammar shown in Figure 2.4; in particular, we reformulate the points-to analysis to introduce a new intermediate relation. Our

[1] Atlas stands for AcTive Learning of Alias Specifications.

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

Figure 5.2: Productions for the context-free grammar C_{pt}. The start symbol of C_{pt} is FlowsTo.

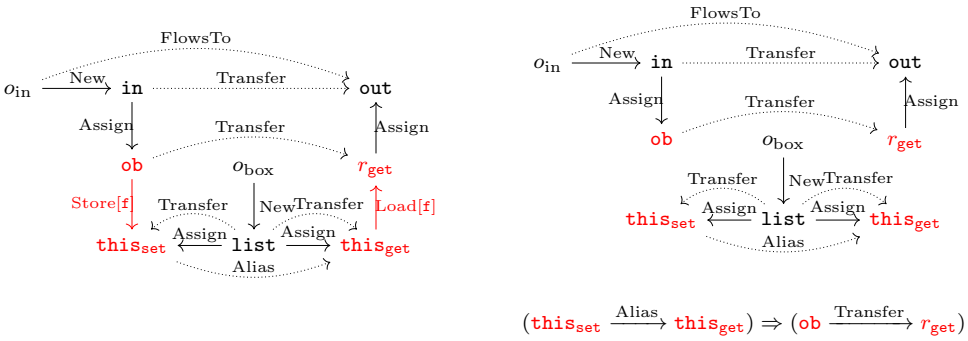


Figure 5.3: The solid edges are the graph G extract for the program `test` shown in Figure 5.1. In addition, the dashed edges are a few of the edges in $\overline{G}$ when computing the transitive closure. We omit backward edges (i.e., with labels $\overline{A}$) for clarity. Vertices and edges corresponding to library code are highlighted in red.

modified production rules are shown in Figure 5.2. The first production in Figure 5.2 constructs the *transfer* relation $x \xrightarrow{\text{Transfer}} y$, which says that x may be "indirectly assigned" to y. The second production constructs the "backwards" transfer relation. The third production constructs the *alias* relation $x \xrightarrow{\text{Alias}} y$, which says that x may alias y. The fourth production computes the points-to relation, i.e., $x \hookrightarrow o$ whenever $o \xrightarrow{\text{FlowsTo}} x \in \overline{G}$. For example, Figure 5.3 shows the computation of the transitive closure for the code shown in Figure 5.1.

5.2 Overview

Our algorithm infers a set of specifications that describe the behaviors of the library functions that are relevant to our static points-to analysis. It requires two inputs:

- **Library interface:** The type signature of each function in the library.
- **Blackbox access:** The ability to execute a library function on a chosen input and obtain the corresponding output.

Because we only have blackbox access to the library code, it is impossible to guarantee that the inferred specifications are both sound and precise. Instead, any inference algorithm must make

tradeoffs between these two properties. Our algorithm aims to ensure that the inferred path specifications S are *admissible*, which says that there are no "strictly better" path specifications S', i.e., the precision and recall of S are as good as those of S' and at least one is strictly better.

To this end, our algorithm infers specifications incrementally in two phases. In the first phase, our algorithm only infers specifications it is certain are admissible. In the second phase, it inductively generalizes this set of specifications, using a large number of tests to minimize the chance of inadmissibility. In our experiments, this phase does not infer any inadmissible specifications.

We define precision for path specifications with respect to Andersen's analysis [11], a context- and flow-insensitive points-to analysis, and to context- or object-sensitive extensions of this analysis based on cloning [152]. We show that path specifications can precisely model the library code when using Andersen's analysis; they are also compatible with other points-to analyses, but may be lose precision. For example, the path specifications for the `List` class describe the same points-to effects as the following code:

```
class List {
  Object f;
  void add(Object ob) { f = ob; }
  Object get(int i) { return f; } }
```

Andersen's analysis does not lose any precision by analyzing this code (or path specifications) instead of the true implementation of `List`, but a more precise static analysis may lose precision.

5.2.1 Path specifications

Our algorithm infers *path specifications* that summarize the effects of library code. A path specification is simply a sequence $s \in \mathcal{V}^*_{\text{path}}$, where $\mathcal{V}^*_{\text{path}}$ are variables in the library interface. For example, a path specification for the library functions `set` and `get` in Figure 5.1 is

$$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} \rightarrow \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}. \tag{5.1}$$

Here, $\texttt{this}_m$ and r_m denote the receiver and return value of library function m, respectively. The arrows in the path specification are for clarity; we can equivalently write this path specification as a sequence $\texttt{ob}\ \texttt{this}_{\texttt{set}}\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}$. Its meaning is the following logical formula:

$$(\texttt{this}_{\texttt{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}} \in \overline{G}) \Rightarrow (\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}} \in \overline{G}). \tag{5.2}$$

Intuitively, the notation $x \xrightarrow{A} y$ is an edge indicating that x and y satisfy relation A (e.g., they are aliased), and the graph $\overline{G}$ is the set of all such edges. Then, this formula says that if the receivers of `set` and `get` are aliased, then the parameter `ob` of `set` may be *transfered* to the return value of `get`. The transfer relation $x \xrightarrow{\text{Transfer}} y$ essentially encodes that x may be "indirectly assigned" to y. For example, in the code `z = x; y = z;`, `x` is transfered to `y`.

Path specifications have two key benefits. First, we can devise a test case P to check whether a given path specification s is admissible. The premise of s holds for P, but its conclusion only holds for P if the points-to effect specified by s is exhibited by the library code. Upon executing P, if we observe that the conclusion of s holds for P, then we have proven that s is admissible. Second, given a set S of path specifications, if we have determined that every path specification $s \in S$ is admissible, then we guarantee S is admissible as a whole—i.e., path specifications compose.

Continuing our example, the test case for the path specification (5.1) is the `test` function shown in Figure 5.1. Essentially, if we ignore the implementation of the `set` and `get` functions, then the premise of (5.2) holds for this program, but not the conclusion. Upon executing the program, if we see that the conclusion of (5.2) holds during execution, then we know that the behavior of the library functions specified by the path specification can occur, i.e., it is admissible.

5.2.2 Phase One: Sampling Positive Examples

Our algorithm initializes the set of inferred specifications to $S \leftarrow \emptyset$, and then repeats:

1. Propose a *candidate path specification* s.
2. Synthesize a test case that checks whether s is admissible.
3. Execute the test cases, and accept s (i.e., $S \leftarrow S \cup \{s\}$) if the test case passes.

By design, the synthesized test case passes only if s is admissible; therefore, at the end of the first phase, S remains admissible. However, the test cases may fail even if s is admissible—we cannot guarantee that no missed corner cases exist in the library code. Thus, test cases are designed heuristically to pass for the majority of admissible candidates.

5.2.3 Phase Two: Inductive Generalization

We show that path specifications can precisely model any library code with respect to Andersen's analysis (and its context- and object-sensitive extensions). However, the required set of path specifications may be infinitely large. Phase two inductively generalizes the finite set of specifications inferred in phase one to a description of a potentially infinite set of path specifications.

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

We devise a language inference algorithm based on RPNI [110]. Our algorithm proposes candidate inductive generalizations of S, and then checks the admissibility of each candidate using a large

number of test cases. Unlike phase one, a generalization may be inadmissible even if all the test cases pass; we show empirically that admissibility is maintained.

While our algorithm infers a regular set of path specifications, in general, the set of path specifications required to model the library code may not be regular. We find that regular sets of path specifications suffice to model library code occurring in practice (see Section 5.9.1).

For example, the path specifications for `set`, `get`, and `clone` functions are

$$\texttt{ob} \dashrightarrow \texttt{this}_{\texttt{set}} (\rightarrow \texttt{this}_{\texttt{clone}} \dashrightarrow r_{\texttt{clone}})^* \rightarrow \texttt{this}_{\texttt{get}} \dashrightarrow r_{\texttt{get}}. \tag{5.3}$$

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

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

Then, phase two would inductively generalize this specification to (5.3).

5.3 Path Specifications

In this section, we describe our specification language.

5.3.1 Motivation

Suppose that our static analysis could analyze the library implementation, and that by doing so, the extracted graph G contains additional paths

$$z_1 \overset{\beta_1}{\dashrightarrow} w_1, \ ..., \ z_k \overset{\beta_k}{\dashrightarrow} w_k$$

extracted from the library code, where the variables $z_1, w_1, ..., z_k, w_k$ are parameters and return values of library functions and $\beta_1, ..., \beta_k \in \Sigma_{\text{pt}}^*$. Furthermore, let $A_1, ..., A_{k-1}$ be nonterminals that satisfy $A \overset{*}{\Rightarrow} \beta_1 A_1 ... \beta_{k-1} A_{k-1} \beta_k$. In this case, while computing the transitive closure $\overline{G}$, if

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

then our static analysis would add edge $z_1 \xrightarrow{A} w_k$ to $\overline{G}$ as well.

However, since we cannot analyze the library implementation, the paths $z_i \overset{\beta_i}{\dashrightarrow} w_i$ are missing from G (and thus from $\overline{G}$), so the static analysis will not add the edge $z_1 \xrightarrow{A} w_k$ to $\overline{G}$. Therefore, we need a specification telling the analysis to add $z_1 \xrightarrow{A} w_k$ to $\overline{G}$ if all the edges $w_i \xrightarrow{A_i} z_{i+1}$ are in $\overline{G}$.

Library Code	Candidate Path Specifications	Generated Test Cases	

`void set(Object ob) { f = ob; }`
`Object get() { return f; }`
	$ob \dashrightarrow this_{set} \rightarrow this_{get} \dashrightarrow r_{get}$	`boolean test() {`
		`  Object in = new Object(); // o_in`
		`  Box box = new Box(); // o_box`
		`  box.set(in);`
		`  Object out = box.get();`
		`  return in == out; }`
			✓

`void set(Object ob) { f = ob; }` `Object get() { return g; }`	$\emptyset$	$\emptyset$	✓

`void set(Object ob) { f = ob; }`
`Object clone() { return f; }`
	$ob \dashrightarrow this_{set} \rightarrow this_{clone} \dashrightarrow r_{clone}$	`boolean test() {`
		`  Object in = new Object(); // o_in`
		`  Box box = new Box(); // o_box`
		`  box.set(in);`
		`  Object out = box.clone();`
		`  return in == out; }`
			✗

`void set(Object ob) { f = ob; }`
`Object get() { return f; }`
`Box clone() {`
`  Box b = new Box(); // ~o_clone`
`  b.f = f;`
` return b; }}
	$ob \dashrightarrow this_{set} (\rightarrow this_{clone} \dashrightarrow r_{clone})^* \rightarrow this_{get} \dashrightarrow r_{get}$	`boolean test0() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Object out = box0.get();`
		`  return in == out; }`
		
		`boolean test1() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Object out = box1.get();`
		`  return in == out; }`
		
		`boolean test2() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Box box2 = box1.clone();`
		`  Object out = box2.get();`
		`  return in == out; }`
		
		`...`
			✓

`void set(Object ob) { f = ob; }`
`Object get() {`
`  return f;`
`  return g;`
`  return h; }`
`Box clone() {`
`  Box b = new Box(); // ~o_clone`
`  b.g = f;`
`  b.h = g;`
` return b; }}
	$ob \dashrightarrow this_{set} \rightarrow this_{get} \dashrightarrow r_{get}$
	
	$+\ ob \dashrightarrow this_{set} \rightarrow this_{clone} \dashrightarrow r_{clone} \rightarrow this_{get} \rightarrow r_{get}$
	
	$+\ ob \dashrightarrow this_{set} \rightarrow this_{clone} \dashrightarrow r_{clone} \rightarrow this_{clone} \dashrightarrow r_{clone} \rightarrow this_{get} \dashrightarrow r_{get}$
		`boolean test0() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Object out = box0.get();`
		`  return in == out; }`
		
		`boolean test1() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Object out = box1.get();`
		`  return in == out; }`
		
		`boolean test2() {`
		`  Object in = new Object(); // o_in`
		`  Box box0 = new Box(); // o_box`
		`  box0.set(in);`
		`  Box box1 = box0.clone();`
		`  Box box2 = box1.clone();`
		`  Object out = box2.get();`
		`  return in == out; }`
			✓

Figure 5.4: Examples of hypothesized library implementations (left column), an equivalent set of path specifications (middle column), and the synthesized test cases to check the precision of these specifications (right column), with a check mark ✓ (indicating that the tests pass) or a cross mark ✗ (indicating that the tests fail).

For example, consider the library code in Figure 5.1. When analyzing the program `test` in the same figure with the library code available, the analysis includes the paths

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

In this case, we have

$$\text{Transfer} \overset{*}{\Rightarrow} \text{Store}[\texttt{f}]\ \text{Transfer}\ \text{Load}[\texttt{f}],$$

so we need a specification encoding the rule

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

This rule says that if the static analysis computes $\texttt{this}_{\texttt{set}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}} \in \overline{G}$, then it also computes $\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}} \in \overline{G}$. For example, this rule is applied in Figure 5.3 (right) to compute $\texttt{ob} \xrightarrow{\text{Transfer}} r_{\texttt{get}}$.

Path specifications are a language for expressing such rules. The middle column of Figure 5.4 shows examples of path specifications. In the first column, we show a hypothetical implementation of the library functions that has the same semantics as the corresponding path specification. In the last column, we show test cases that check the admissibility of the path specifications.

5.3.2 Syntax and Semantics

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

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

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

$$z_1 \dashrightarrow w_1 \rightarrow z_2 \dashrightarrow ... \dashrightarrow w_{k-1} \rightarrow z_k \dashrightarrow w_k. \tag{5.4}$$

Given path specification (5.4), for each $i \in [k]$, define the nonterminal A_i in the grammar C_{pt} to

be

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

Also, define the nonterminal A by

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

Then, the path specification corresponds to adding a rule

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

to the static points-to analysis. The corresponding rule also adds the backwards edge $w_k \xrightarrow{\overline{A}} z_1$ to $\overline{G}$, but we omit it for clarity. We refer to the premise of this rule as the premise of the path specification, and the conclusion of this rule as the conclusion of the path specification.

5.3.3 Admissibility

In this section, we formalize the notion of *admissibility*, which essentially says that a set S of path specifications is Pareto optimal in terms of precision and recall, i.e., there does not exist a set S' of path specifications that is strictly preferable to S.

Let $\overline{G}_*(P)$ denote the true set of relations for a program P (i.e., relations that hold dynamically). Furthermore, given path specifications S, let $\overline{G}(P, S)$ denote the points-to edges computed using S for P, let $\overline{G}_+(P, S) = \overline{G}(P, S) \setminus \overline{G}_*(P)$ be the false positive points-to edges computed, and $\overline{G}_-(P, S) = \overline{G}_*(P) \setminus \overline{G}(P, S)$ be the false negative points-to edges computed. We say S is *sound* if $\overline{G}_-(P, S) = \emptyset$ and *completely precise* if $\overline{G}_+(P, S) = \emptyset$.

Our notions of precision and recall are relative versions of the notions of complete precision and soundness, respectively. More precisely, given sets S and S' of path specifications, we say S has *higher or equal precision* than S' if for all programs P, $\overline{G}_+(P, S) \subseteq \overline{G}_+(P, S')$, i.e., S always produces fewer false positives than S'. Similarly, we say S has *higher or equal recall* than S' if for all programs P, $\overline{G}_-(P, S) \subseteq \overline{G}_-(P, S')$, i.e., S always produces fewer false negatives than S'. If for all programs P, S and S' compute the same relations, i.e., $\overline{G}(P, S) = \overline{G}(P, S')$, then we say S and S' are *equivalent*.

We say a set S of path specifications is *admissible* if, for any other set S' of path specifications, either S' does not have higher or equal precision than S or it does not have higher or equal recall

than S. In other words, there is no set of path specifications S' that is strictly better than S.

5.3.4 Checking Admissibility

Our algorithm needs to generate test cases that check whether a candidate path specification s is admissible. We describe sufficient conditions for a passing test case to prove admissibility, i.e., if the test case passes, then we guarantee that s is admissible. However, the test case may fail even if s is admissible. This property is inevitable since executions are underapproximations; we show empirically that if s is admissible, then the synthesized test case typically passes.

Definition 5.3.1 Let s be a path specification. We say a program P is a *potential witness* for s if:

- The conclusion ($e \in \overline{G}$) of s does not hold statically for P with empty specifications, i.e., $e \notin \overline{G}(P, \emptyset)$.
- The premise of s holds for P, i.e., $e \in \overline{G}(P, \{s\})$.
- For every set S of path specifications, if $e \in \overline{G}(P, S)$, then $S \cup \{s\}$ is equivalent to S.

We say P is a *witness* for s if furthermore the conclusion of s is a true relation of P, i.e., $e \in \overline{G}_*(P)$.

In other words, s is the most precise path specification that can compute e for P—for any set S of path specifications that can do so, adding s to S does not affect the semantics of S. In Figure 5.4, the test cases shown in the last column witness the corresponding path specifications.

Intuitively, if program P is a potential witness for path specification s with premise ψ and conclusion $\phi = (e \in \overline{G})$, then s is the only path specification that can be used by the static analysis to compute relation e for P. Therefore, if P witnesses s, then s is guaranteed to be admissible. More precisely, we have the following important result:

Theorem 5.3.2 For any set S of path specifications, if each $s \in S$ has a witness, then S is admissible.

Proof: Let S' be a set of path specifications. We need to show that (i) if S' has higher recall than S, then S has higher precision than S', and (ii) if S' has recall equal to S, then S has higher or equal precision than S'.

First, we claim that in either case, S' is equivalent to $S' \cup S$. To this end, consider a path specification $s \in S$ with conclusion ($e \in \overline{G}$) and witness P. We claim that if path specifications S' has higher or equal recall than S, then $S' \cup \{s\}$ is equivalent to S'. Since the static analysis is monotone, we have $e \in \overline{G}(P, \{s\}) \subseteq \overline{G}(P, S)$, so since S' has higher or equal recall than S, we have $e \in \overline{G}(P, S')$. By the definition of a witness, $S' \cup \{s\}$ is equivalent to S'. Thus, by induction, $S' \cup S$ is equivalent to S'.

Note that (ii) follows immediately, since S clearly has higher or equal precision than $S' \cup S$, so S has higher or equal precision than S' as well. To show (i), it remains to show that using S' computes a false positive edge that using S does not. Since S' has higher recall than S, there exists some program P and some edge e such that $e \in \overline{G}(P, S')$ but $e \notin \overline{G}(P, S)$. Let P' be the program "if False then P"; since our static analysis is flow-insensitive, we have $\overline{G}(P', S) = \overline{G}(P, S)$ and $\overline{G}(P', S') = \overline{G}(P, S')$, so $e \in \overline{G}(P', S')$ and $e \notin \overline{G}(P', S)$. But clearly e is a false positive for P', since P' does not exhibit any points-to edges. Thus, S' is less precise than S, as claimed. □

By Theorem 5.3.2, we can check whether a candidate path specification s is admissible by synthesizing a test case P that is a potential witness for s. If we execute the test case P and observe that the conclusion of s holds during the execution, then P is a witness for s, so s is admissible (though if P is not a witness of s, s may still be admissible). Finally, if S is the set of path specifications inferred by our algorithm, as long as each $s \in S$ has a witness, then S is admissible as well.

5.3.5 Equivalence to Library Implementations

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

Assumption 5.3.3 Let $\mathcal{F}_{\text{lib}}$ be fields accessed by the library and $\mathcal{F}_{\text{prog}}$ be fields accessed by the program, and let the *shared fields* be $\mathcal{F}_{\text{share}} = \mathcal{F}_{\text{lib}} \cap \mathcal{F}_{\text{prog}}$. We assume $\mathcal{F}_{\text{share}} = \emptyset$.

We can remove this assumption by having the static analysis treat accesses to library fields in the program as calls to getter and setter library functions. With this assumption, we have:

Theorem 5.3.4 Let $\overline{G}(P)$ be the points-to sets computed with the library code. Then, there exists S such that $\overline{G}(P, S)$ is sound and $\overline{G}(P, S) \subseteq \overline{G}(P)$.

We give a proof in Section 5.7. Note that the set S of path specifications may be infinite. This infinite blowup is unavoidable since we want the ability to test the admissibility of an individual path specification. In particular, the library implementation (e.g., the one shown on the fourth row of Figure 5.4) may exhibit effects that require infinitely many test cases to check admissibility.

5.3.6 Regular Sets of Path Specifications

Since the library implementation may correspond to an infinite set of path specifications, we need a mechanism for describing such sets. In particular, since a path specification is a sequence $s \in \mathcal{V}_{\text{path}}^*$, we can think of a set S of path specifications as a formal language $S \subseteq \mathcal{V}_{\text{path}}^*$ over the alphabet $\mathcal{V}_{\text{path}}$.

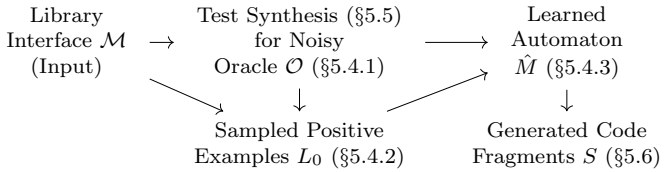


Figure 5.5: An overview of our specification inference system. The section describing each component is in parentheses.

Then, we can express an infinite set of path specifications using standard representations such as regular expressions or context-free grammars.

We make the empirical observation that the set of path specifications corresponding to the library implementation is a regular language. There is no particular reason that this fact should be true, but it holds empirically for all the Java library functions we have examined so far. For example, consider the library implementation shown in the first column of line four of Figure 5.4. This specification corresonds to the set of path specifications shown as a regular expression in the middle column of the same line (tokens in the regular expression are highlighted in blue for clarity).

One challenge is how to run our static points-to analysis with an infinite set of path specifications; we describe how to do so for the case of regular sets of path specifications in Section 5.6.

5.4 Specification Inference Algorithm

In this section, we describe our algorithm for inferring path specifications. Our system is summarized in Figure 5.5, which also shows the section where each component is described in detail.

5.4.1 Overview

Let the *target language* $S_* \subseteq \mathcal{V}^*_{\text{path}}$ be the set of all path specifications that have a witness. By Theorem 5.3.2, S_* is admissible. The goal of our algorithm is to infer a set of path specifications that approximates S_* as closely as possible.

Inputs. Our algorithm is given two inputs:

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

Using these two inputs, we construct the following two data structures.

Noisy oracle. Given a path specification s, the *noisy oracle* $\mathcal{O} : \mathcal{V}^*_{\text{path}} \to \{0, 1\}$ (i) always returns 0 if s is inadmissible, and (ii) ideally returns 1 if s is admissible (but may return 0). This oracle is implemented by synthesizing a potential witness P for s—if the conclusion of the specification holds upon executing P, then P is a witness for s so the oracle returns 1; otherwise, the oracle returns 0. We describe how we synthesize a witness for s in Section 5.5.

Positive examples. Phase one of our algorithm constructs a set of positive examples: it randomly samples candidate path specifications $s \sim \mathcal{V}^*_{\text{path}}$, and then uses $\mathcal{O}$ to determine whether each s is admissible. More precisely, given a set $S = \{s \sim \mathcal{V}^*_{\text{path}}\}$ of random samples, it constructs positive examples $S_0 = \{s \in L \mid \mathcal{O}(s) = 1\}$. We describe how we sample $s \sim \mathcal{V}^*_{\text{path}}$ in Section 5.4.2.

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

Definition 5.4.1 The *language inference problem* is to, given the noisy oracle $\mathcal{O}$ and the positive examples $S_0 \subseteq S_*$, infer a language $\hat{S}$ that approximates S_* as closely as possible.

In Section 5.4.3, we describe our algorithm for solving this problem. Our algorithm outputs a regular language $\hat{S} = \mathcal{S}(\hat{M})$, where $\hat{M}$ is a finite state automaton. For example, given

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

our language inference algorithm returns an automaton encoding the regular language

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

5.4.2 Sampling Positive Examples

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

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

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

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

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

Monte Carlo tree search. We can exploit the fact that certain choices $x \in \mathcal{T}(s)$ are much more likely to yield an admissible path specification than others. To do so, note that our search space is structured as a tree, where each vertex corresponds to a prefix in $\mathcal{V}_{\text{path}}^*$, the root corresponds to the prefix ϵ, edges are defined by $\mathcal{T}$, and leaves correspond to candidate path specifications.

We can sample $x \sim \mathcal{T}(s)$ using Monte Carlo tree search (MCTS) [28], a search algorithm that learns over time which choices are more likely to succeed. In particular, MCTS keeps track of a score $Q(s, x)$ for every visited $s \in \mathcal{V}_{\text{path}}^*$ and every $x \in \mathcal{T}(s)$. Then, the choices are sampled according to the distribution

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

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

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

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

5.4.3 Language Inference Algorithm

We modify RPNI [110] to leverage access to the noisy oracle. In particular, whereas RPNI takes as input a set of negative examples, we use the oracle to generate them on-the-fly. Our algorithm learns a regular language $\hat{S} = \mathcal{L}(\hat{M})$ represented by the (nondeterministic) finite state automaton (FSA) $\hat{M} = (Q, \mathcal{V}_{\text{path}}, \delta, q_{\text{init}}, Q_{\text{fin}})$, where Q is the set of states, $\delta : Q \times \mathcal{V}_{\text{path}} \to 2^Q$ is the transition function, $q_{\text{init}} \in Q$ is the start state, and $Q_{\text{fin}} \subseteq Q$ are the accept states. If there is a single accept state, we denote it by q_{fin}. We denote transitions $q \in \delta(p, \sigma)$ by $p \xrightarrow{\sigma} q$.

Our algorithm initializes $\hat{M}$ to be the FSA representing the finite language S_0. In particular, it initializes $\hat{M}$ to be the prefix tree acceptor [110], which is the FSA where the underlying transition graph is the prefix tree of S_0, the start state is the root of this prefix tree, and the accept states are the leaves of this prefix tree.

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

(i) replacing transitions

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

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

Our algorithm makes a single pass over all the states Q. We describe how a single step proceeds: let q be the state being processed in the current step, let Q_0 be the states that have been processed so far but not removed from Q, and let $\hat{M}$ be the current FSA. For each $p \in Q_0$, our algorithm checks whether merging q and p overgeneralizes the language, and if not, greedily performs the merge. More precisely, for each $p \in Q_i$, our algorithm constructs

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

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

For example, suppose our language learning algorithm is given a single positive example

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

Then, our algorithm constructs the finite state automaton

$$\hat{M}_0 = q_{\text{init}} \xrightarrow{\texttt{ob}} q_1 \xrightarrow{\texttt{this}_{\texttt{set}}} q_2 \xrightarrow{\texttt{this}_{\texttt{clone}}} q_3 \xrightarrow{r_{\texttt{clone}}} q_4 \xrightarrow{\texttt{this}_{\texttt{get}}} q_5 \xrightarrow{r_{\texttt{get}}} q_{\text{fin}}.$$

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

$$\hat{M}_1 = q_{\text{init}} \xrightarrow{\texttt{ob}} q_1 \xrightarrow{\texttt{this}_{\texttt{set}}} q_2 \xrightarrow{\texttt{this}_{\texttt{get}}} q_4 \xrightarrow{r_{\texttt{get}}} q_{\text{fin}}, \quad q_2 \xrightarrow{\texttt{this}_{\texttt{clone}}} q_3 \xrightarrow{r_{\texttt{clone}}} q_3$$

n

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

$$\begin{aligned}
&\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^0\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}\\
&\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^2\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}}\\
&\ldots\\
&\texttt{ob}\ \texttt{this}_{\texttt{set}}\ (\texttt{this}_{\texttt{clone}}\ r_{\texttt{clone}})^N\ \texttt{this}_{\texttt{get}}\ r_{\texttt{get}},
\end{aligned}$$

all of which are accepted by our noisy oracle $\mathcal{O}$. Therefore, our algorithm greedily accepts this merge and continues. The remaining merges fail, so our algorithm returns an FSA that equals $\hat{M}_1$.

5.5 Test Case Synthesis Algorithm

In this section, we describe our algorithm for synthesizing a test case to check correctness of a candidate path specification. For example, in Figure 5.6, the synthesized test case contains exactly the external edges in the candidate's premise:

$$\texttt{this}_{\texttt{add}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{clone}}, \quad r_{\texttt{clone}} \xrightarrow{\text{Transfer}} \texttt{this}_{\texttt{get}}.$$

Upon executing this test case, the candidate's conclusion

$$\texttt{in} \xrightarrow{\text{Transfer}} \texttt{out}$$

holds dynamically. Therefore, this test case witnesses the correctness of the given candidate.

Our algorithm first constructs a *skeleton* containing a call to each function in the specification. Then, it (i) fills in *holes* with variable names, (ii) initializes variables, and (iii) orders (or *schedules*) statements. The last step also adds a statement returning whether the candidate's conclusion holds.

There are certain constraints on the choices that ensure that the synthesized test case is a valid witness. Even with these constraints, a number of additional choices remain. Each choice produces a valid test case, but some of these test cases may not pass even if the candidate specification is correct. We describe the choices made by our algorithm, which empirically finds almost all correct candidate specifications.

5.5.1 Skeleton Construction

To witness correctness of the candidate path specification, the synthesized test case must exhibit *exactly* the external edges in its premise. In particular, the test case must include a call to each function in the candidate. Our algorithm constructs a *skeleton* consisting of these calls, for example, the skeleton on the second step of Figure 5.6. A symbol `??`, called a *hole*, is included for each

parameter and return value of each function call, and must be filled in with a variable name.

5.5.2 Filling Holes

The external edges in the candidate specification impose constraints on the arguments that should be used in each function call. In particular, the synthesized test case must exhibit every behavior encoded by the external edges in the candidate specification:

- **Alias:** For an aliasing edge $p_{m_i} \xrightarrow{\text{Alias}} p_{m_{i+1}}$, the algorithm has to ensure that the arguments p_{m_i} (passed to m_i) and $p_{m_{i+1}}$ (passed to m_{i+1}) are aliased.
- **Transfer:** For a transfer edge $r_{m_i} \xrightarrow{\text{Transfer}} p_{m_{i+1}}$, the algorithm has to use the return value of m_i as the argument passed to m_{i+1} (and similarly for backwards transfer edges $p_{m_i} \xrightarrow{\overline{\text{Transfer}}} r_{m_{i+1}}$).

For example, the holes in the skeleton in Figure 5.6 are filled so that the following premises are satisfied:

$$\texttt{this}_{\texttt{add}} \xrightarrow{\text{Alias}} \texttt{this}_{\texttt{clone}}, \quad r_{\texttt{clone}} \xrightarrow{\text{Transfer}} \texttt{this}_{\texttt{get}}.$$

One issue is that internal edges may be self-loops, in which case more than two parameters may need to be aliased. For example, consider the following candidate:

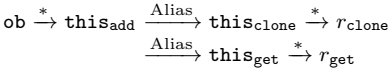
$$\begin{aligned} \texttt{ob} \xrightarrow{*} \texttt{this}_{\texttt{add}} &\xrightarrow{\text{Alias}} \texttt{this}_{\texttt{clone}} \xrightarrow{*} \texttt{this}_{\texttt{id}} \\ &\xrightarrow{\text{Alias}} \texttt{this}_{\texttt{get}} \xrightarrow{*} r_{\texttt{get}}. \end{aligned} \tag{5.5}$$

For the test case for this candidate, the three calls to `add`, `clone`, and `get` must all share the same receiver:

```
list.add(in);
List listClone = list.clone();
Object out = list.get(??);
```

Our algorithm partitions the holes into subsets that must be aliased—since aliasing is a transitive relation, every hole in a subset has to be aliased with every other hole in that subset. To do so, the algorithm constructs an undirected graph where the vertices are the holes, and an edge $(h, h') \in E$ connects two holes h and h' in the following cases:

- There is an external edge $w_{m_i} \xrightarrow{T} z_{m_{i+1}}$ in the candidate specification, where h is the hole corresponding to w_{m_i} and h' is the hole corresponding to $z_{m_{i+1}}$.
- There is an internal edge $p_{m_i} \xrightarrow{*} p_{m_i}$ in the candidate specification, where h is the hole corresponding to the p_{m_i} on the left-hand side and h' is the hole corresponding to the p_{m_i} on the right-hand side.



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

fill holes	`list.add(in);`
	`List listClone = list.clone();`
	`Object out = listClone.get(??);`

initialization & scheduling	`Object in = new Object();`
	`List list = new List()`
	`list.add(in);`
	`List listClone = list.clone();`
	`Object out = listClone.get(0);`
	`return in == out;`

Figure 5.6: Steps in the test synthesis algorithm (right) for a candidate path specification for `List` (left). Code added at each step is highlighted in blue. Scheduling is shown in the same line as initialization—it chooses the final order of the statements. This figure is a duplicate of Figure 5.6, and is shown here for clarity.

Then, our algorithm computes the connected components in this graph. For each connected component, the algorithm chooses a fresh variable name, and each hole in that connected component is filled with this variable name.

For example, for the candidate in Figure 5.6, our algorithm computes the following partitions:

$$\{\texttt{ob}\},\ \{\texttt{this}_{\texttt{add}}, \texttt{this}_{\texttt{clone}}\},\ \{r_{\texttt{clone}}, \texttt{this}_{\texttt{get}}\},\ \{r_{\texttt{get}}\},$$

and fills the corresponding holes with the variables names

$$\texttt{in},\ \texttt{list},\ \texttt{listClone},\ \texttt{out},$$

respectively. Similarly, for (5.5), we compute partitions

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

The variable names are the same as those chosen in Figure 5.6.

5.5.3 Variable Initialization

We describe primitive variables and reference variables separately. For the case of initializing reference variables, we describe two different strategies:

- **Null:** Whenever possible, initialize to `null`.
- **Instantiation:** Whenever possible, use constructor calls.

The first strategy ensures that the test case does not exhibit additional transfer and alias edges beyond those in the candidate specification. The second strategy may produce a test case that does

not witness correctness, since it may include spurious edges not in the premise of the candidate. However, certain functions require that some of their arguments are not null; for example, the `put` function in the `Hashtable` class. We show empirically that the second variant identifies a number of candidates missed by the first, and that these additional specifications are in fact correct.

Primitive initialization. We initialize all primitive variables with `0` (except characters, which are initialized as `'a'`). In our experience, the only important choice of primitive value is the `index` parameter passed to functions such as `get`, which retrieve data from collections. Choosing the `index = 0` retrieves the single object the test case previously added to the collection. Testing more primitive values is possible but has so far been unnecessary.

Reference initialization using null. Reference variables for which aliasing relations hold must be instantiated (unless they have already been initialized as the return value of a function call). Any other reference variable is initialized to `null`. For example, in Figure 5.6, the variables `list` and `out` must be instantiated, but `cloneList` has already been initialized as the return value of `clone`. In general, the test case we synthesize calls the constructor with the fewest number of arguments; primitive arguments are initialized as before, and reference arguments are initialized using `null`.

Reference initialization using instantiation. In this approach, we have to synthesize constructor calls when empty constructors are unavailable. For example, if the only constructor for the `List` class was `List(Object val)`, then we would have to initialize an object of type `Object` as well:

```
List list = new List(new Object());
```

We encode the problem of synthesizing a valid constructor call as a directed hypergraph reachability problem. A *directed hypergraph* is a pair $G = (V, E)$, where V is a set of vertices, and edges $e \in E$ have the form $e = (h, B)$, where $h \in V$ is the *head* of the edge, and $B \subseteq V$ is its *body*. For our purposes, B is a list rather than a set, and may contain a single vertex multiple times.

We construct a hypergraph $G = (V, E)$ where vertices correspond to classes, and edges to constructors:

- **Vertices:** A vertex $v \in V$ is a library class.
- **Edges:** An edge $e = (h, B) \in E$ is a constructor, where h is the class of the constructed object and B is the list of classes of the constructor parameters.

For convenience, we also include primitive types as vertices in G, along with an edge representing the "empty constructor", which returns the initialization value described above.

Now, a *path* T in the hypergraph $G = (V, E)$ is a finite tree with root $v_T \in V$ (called the *root* of the path), such that for each vertex $v \in T$, v and its (ordered) children $[v_1, ..., v_k]$ are an edge $e_{T,v} = (v, [v_1, ..., v_k]) \in E$. Note that for each leaf v of T, there must necessarily be an edge

$(v, []) \in E$, since v has no children. Also, we say a vertex $v \in V$ is *reachable* is there exists a path with root v.

In our setting, a path in our hypergraph G corresponds to a call to a constructor—for each vertex $\texttt{X} \in T$ with children $\texttt{X1}, ..., \texttt{Xk}$, we recursively define the constructor

$$C_T(\texttt{X}) = \texttt{new X(}C_T(\texttt{X1})\texttt{, ..., }C_T(\texttt{Xk})\texttt{)}.$$

Therefore, devising a constructor call to instantiate an object of type `X` amounts to computing a path in G with root `X`. Paths to every reachable vertex can be efficiently computed using a standard dynamic programming algorithm. Furthermore, we can add a weight w_e to each edge in $e \in E$. Then, the *shortest* path (i.e., the path minimizing the total weight $\sum_{v \in T} e_{T,v}$) can similarly be efficiently computed. We choose all weights $w_e = 1$ for each $e \in E$.

For example, suppose that the `List` class has a single constructor `List(Object val)`. Then, our algorithm constructs a hypergraph with two vertices and two edges:

$$V = \{\texttt{Object}, \texttt{List}\}$$
$$E = \{(\texttt{Object}, []),\ (\texttt{List}, [\texttt{Object}])\}.$$

Then, the path corresponding to `List` is the tree $T = \frac{\texttt{List}}{\texttt{Object}}$, which corresponds to the constructor call

```
new List(new Object())
```

used to instantiate variables of type `List`.

As with initializing primitive variables, multiple choices of constructor calls could be used, but selecting a single constructor suffices has been sufficient so far.

5.5.4 Statement Scheduling

Note that the test case now contains both function call statements as well as variable initialization statements added in the previous step. All the added variable initialization statements can be executed first, so it suffices to schedule the function call statements.

There are two kinds of constraints on scheduling function calls. First, edges in the candidate specification of the form

$$r_{m_i} \xrightarrow{\text{Transfer}} p_{m_{i+1}}$$

impose *hard constraints* on the schedule, since m_i must be called before m_{i+1} so its return value can be transfered to $p_{m_{i+1}}$ (edges of the form $p_{m_i} \xrightarrow{\overline{\text{Transfer}}} r_{m_{i+1}}$ impose hard constraints as well). For example, in Figure 5.6, the edge $r_{\texttt{clone}} \xrightarrow{\text{Transfer}} \texttt{this}_{\texttt{get}}$ imposes the hard constraint that the call to

`clone` must be scheduled before the call to `get`. Then, any of the following orderings is permitted:

$$[\texttt{add}, \texttt{clone}, \texttt{get}],\ [\texttt{clone}, \texttt{add}, \texttt{get}],\ [\texttt{clone}, \texttt{get}, \texttt{add}].$$

We use *soft constraints* to choose among schedules satisfying the hard constraints. Empirically, we observe that the order of the functions in the specification is typically the same as the order in which they must be called for the conclusion to be exhibited dynamically. More precisely, function m_i should be called before function m_j whenever $i < j$. In our example, the soft constraint says that `add` should be scheduled before both `clone` and `get`.

Our algorithm iteratively constructs a schedule $[i_1, ..., i_k]$ of the function calls $F = \{m_1, ..., m_k\}$. At iteration t, it selects the tth function call m_{i_t} from the remaining calls $F_t \subseteq F$. It does so greedily, by identifying the choices $G_t \subseteq F_t$ that satisfy the hard constraints, and then selecting $m_{i_t} \in G_t$ to be optimal according to the soft constraints. These conditions uniquely specify m_{i_t}, since our soft constraints are a total ordering.

Our algorithm keeps track of the remaining statements F_t as a directed acyclic graph (DAG), which includes an edge $m_i \to m_j$ for each hard constraint that m_i should be scheduled before m_j. Then, G_t is the set of roots of F_t. Furthermore, our algorithm maintains G_t as a priority queue, where the priority of m_i is i (the highest priority element in G_t is the element with the smallest index i).

We initialize $F_1 = F$; then, G_1 is the subset of vertices in F_1 without a parent. Updates are computed as follows:

1. The highest priority function call m_{i_t} in G_t is removed from both G_t and from F_t.
2. For each child m_i of m_{i_t} in F_t, we determine if m_i is now a root of F_t (i.e., none of its parents are in F_t).
3. For every child m_i that is now a root of F_t, we add m_i to G_t with priority i.

In Figure 5.6, F_1 has three vertices `add` (priority 1), `clone` (priority 2), and `get` (priority 3), and a single edge `clone` $\to$ `get`, and G_1 includes `add` and `clone`. Therefore, the selected schedule is $[\texttt{add}, \texttt{clone}, \texttt{get}]$.

5.5.5 Guarantees

First, we establish a general condition for P to be a potential witness:

Proposition 5.5.1 Let s be a path specification with premise $(e_1 \in \overline{G}) \wedge ... \wedge (e_k \in \overline{G})$. A program P is a potential witness of s if the set of edges $\{e_1, ..., e_k\}$ in the premise of s exactly equals

$$\left\{ w \xrightarrow{A} z \in \overline{G}(P, \emptyset) \mid w, z \in \mathcal{V}_{\text{lib}} \text{ and } A \in \{\text{Transfer}, \overline{\text{Transfer}}, \text{Alias}\} \right\}.$$

Proof: Let P be a potential witness for s, and suppose that the conclusion of s is $(e \in \overline{G})$. Let S be a set of path specifications that computes e for P, i.e., $e \in \overline{G}(P, S)$. We need to show that for any such S, $S \cup \{s\}$ is equivalent to S. Clearly, $S \cup \{s\}$ has higher or equal recall than S, so it suffices to show that it also has higher or equal precision than S. Consider an arbitrary program P'. Then, if s is used during the computation $\overline{G}(P, S \cup \{s\})$, then at that point, the premise of s holds for $\overline{G}$, i.e., $e_1, ..., e_k \in \overline{G}$. Since the graph for P is contained in the graph for P', and our static analysis is monotone, we have $e \in \overline{G}(P, S) \subseteq \overline{G}(P', S)$, i.e., e is computed without s. Thus, $\overline{G}(P', S \cup \{s\}) = \overline{G}(P', S)$, so $S \cup \{s\}$ equivalent to S as claimed. □

Then, we have the following guarantee for the test case synthesis algorithm:

Proposition 5.5.2 The test case P synthesized for path specification s is a potential witness for s.

Proof: (sketch) Let $s = z_1 \to w_1 \dashrightarrow ... \to z_k \dashrightarrow w_k$. Since the function calls are treated as no-ops by the static analysis (according to the definition of a potential witness), they do not add any edges to the extracted graph G except for assignments to and from parameters and return values. The only other edges in the graph G extracted from P are those corresponding to the allocation statements added to P in the initialization step.

First, we show that the edges in the premise of s are contained in $\overline{G}(P, \emptyset)$. For an edge $w_i \to z_{i+1}$, there are three possibilities—either $A_i = \text{Transfer}$, $A_i = \overline{\text{Transfer}}$, or $A_i = \text{Alias}$:

- **Case A_i = Transfer:** Then, w_i is a return value and z_{i+1} is a parameter. Then, the test case synthesis algorithm assigns the return value of m_i to the argument of m_{i+1}, i.e., the edges

$$w_i \xrightarrow{\text{Assign}} x \xrightarrow{\text{Assign}} z_{i+1} \in G,$$

  where G is the graph extracted from P. Therefore, we have $(w_i \xrightarrow{\text{Transfer}} z_{i+1}) \in \overline{G}(P, \emptyset)$.

- **Case $A_i = \overline{\text{Transfer}}$:** This case is analogous to the case $A = \text{Transfer}$.

- **Case A_i = Alias:** Then, w_i and z_{i+1} are both parameters. Then, w_i and z_{i+1} are both parameters. Then, the test case synthesis algorithm allocates a new object and passes it as a parameter to each m_i and m_{i+1}, i.e., the edges

$$o \xrightarrow{\text{New}} x \xrightarrow{\text{Assign}} w_i \in G \text{ and } o \xrightarrow{\text{New}} x \xrightarrow{\text{Assign}} z_{i+1} \in G.$$

  Therefore, we have $(w_i \xrightarrow{\text{Alias}} z_{i+1}) \in \overline{G}(P, \emptyset)$.

Second, consider all edges $w \xrightarrow{A_i} z$, where $w, z \in \mathcal{V}_{\text{lib}}$ and $A_i \in \{\text{Transfer}, \overline{\text{Transfer}}, \text{Alias}\}$, that are contained in the premise of s. By inspection, of the edges in G as described above, the only additional edges in $\overline{G}(P, \emptyset)$ of this form are:

- The self-loops $z_i \xrightarrow{\text{Transfer}} z_i$ and $w_i \xrightarrow{\text{Transfer}} w_i$ (since there is a production Transfer $\rightarrow \epsilon$ in the points-to grammar C_{pt}).
- The backward edges $z_{i+1} \xrightarrow{\overline{A_i}} w_i$ (when $A_i \in \{\text{Transfer}, \overline{\text{Transfer}}\}$).

If these edges were added to the premise of s for P, then by Proposition 5.5.1, we could conclude that P is a potential witness of s. However, these edges are in $\overline{G}(P, S)$ for any program P and any specifications S, so we can add them to the premise of s without affecting its semantics. From Definition 5.3.1, it follows that if P is a witness for s', and s' is equivalent to s, then P is a witness for s as well. Therefore, P is a witness for s as claimed. □

5.6 Static Points-To Analysis with Regular Sets of Path Specifications

In this section, we describe how to run our static points-to analysis in conjunction with a possibly infinite regular set S of path specifications (assumed to be represented as an FSA, i.e., $S = \mathcal{L}(\hat{M})$). In particular, our static analysis converts S to a set $\tilde{S}$ of *code fragment specifications*, which are replacements for the library code that have the same points-to effects as encoded by S.

Given path specifications S, our static analysis constructs *equivalent* code fragment specifications $\tilde{S}$, i.e., $\overline{G}(P, S) = \overline{G}(P, \tilde{S})$. In other words, $\tilde{S}$ has the same semantics as S with respect to our static points-to analysis. One detail in our definition of equivalence is that $\overline{G}(P, \tilde{S})$ may contain additional vertices corresponding to variables and abstract objects in the code fragment specifications; we omit these extra vertices and their relations at the end of the static analysis.

5.6.1 Converting a Single Path Specification

For intuition, we begin by describing how to convert a single path specification

$$s = (z_1 \dashrightarrow w_1 \rightarrow ... \rightarrow z_k \dashrightarrow w_k)$$

into an equivalent set of code fragment specifications, where A_i = Alias for each i and z_1 is a parameter. Let the code fragment specifications $\tilde{S}$ corresponding to s be:

$$\begin{aligned} m_1 &= \{w_1.f_1 \leftarrow z_1\} \\ m_2 &= \{t_2 \leftarrow z_2.f_1, \quad w_2.f_2 \leftarrow t_2\} \\ &... \\ m_k &= \{w_k \leftarrow z_k.f_{k-1}\}, \end{aligned}$$

where $f_1, ...f_{k-1} \in \mathcal{F}$ are fresh fields and $t_2, ..., t_{k-1}$ are fresh variables. Then:

$$\text{(initial parameter)}\ \frac{q_{\text{init}} \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\ \ z = p_m,\ \ w \in \{p_m, r_m\}}{w.f_r \leftarrow z \in m} \qquad \text{(initial return)}\ \frac{q_{\text{init}} \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\ \ z = r_m,\ \ w \in \{p_m, r_m\}}{t \leftarrow X(),\ z \leftarrow t,\ w.f_r \leftarrow t \in m}$$

$$\text{(final parameter)}\ \frac{p \xrightarrow{z} q \xrightarrow{w} q_{\text{fin}} \in \hat{M},\ \ z = p_m,\ \ w = r_m}{w \leftarrow z.f_p \in m} \qquad \text{(final return)}\ \frac{p \xrightarrow{z} q \xrightarrow{w} q_{\text{fin}} \in \hat{M},\ \ z = r_m,\ \ w = r_m}{t \leftarrow X(),\ z.f_p \leftarrow t,\ w \leftarrow t \in m}$$

$$(A_i = \text{Alias})\ \frac{p \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\ \ z = p_m,\ \ w = p_m}{t \leftarrow z.f_p,\ w.f_r \leftarrow t \in m} \qquad (A_i = \text{Transfer})\ \frac{p \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\ \ z = p_m,\ \ w = r_m}{w\text{X}(),\ t \leftarrow z.f_p,\ w.f_r \leftarrow t \in m}$$

$$(A_i = \overline{\text{Transfer}})\ \frac{p \xrightarrow{z} q \xrightarrow{w} r \in \hat{M},\ \ \{z, w\} \subseteq \{p_m, r_m\}}{z \leftarrow X(),\ t \leftarrow w.f_r,\ z.f_p \leftarrow t \in m} \qquad \text{(initial final)}\ \frac{q_{\text{init}} \xrightarrow{z} q \xrightarrow{w} q_{\text{fin}} \in \hat{M},\ \ \{z, w\} \subseteq \{p_m, r_m\}}{w \leftarrow z \in m}$$

Figure 5.7: Rules for generating code fragment specifications from path specifications defined by a finite state automaton $\hat{M} = (Q, \mathcal{V}_{\text{path}}, \delta, q_{\text{init}}, Q_{\text{fin}})$, where for simplicity we assume $\hat{M}$ has a single accept state q_{fin}.

Proposition 5.6.1 We have $\overline{G}(P, \tilde{S}) = \overline{G}(P, \{s\}) \cup \overline{G}'(P, \tilde{S})$, where $\overline{G}'(P, \tilde{S})$ consists of the edges in $\overline{G}(P, \tilde{S})$ that refer to vertices corresponding to variables and abstract objects in $\tilde{S}$.

Proof: (sketch) First, we show that $\overline{G}(P, \{s\} \subseteq \overline{G}(P, \tilde{S})$. Suppose that the premise of s holds, i.e., $z_i \xrightarrow{A_i} w_{i+1} \in \overline{G}$ for each i. Then, the static analysis computes $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(P, \{s\})$; we need to show that $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(P, \tilde{S})$ as well. Note that we have

$$\begin{aligned}
&z_1 \xrightarrow{\text{Store}[f_1]} w_1 \xrightarrow{\text{Alias}} z_2 \xrightarrow{\text{Load}[f_1]} t_2 \in \overline{G}(P, \tilde{S}) \\
&t_2 \xrightarrow{\text{Store}[f_2]} w_2 \xrightarrow{\text{Alias}} z_3 \xrightarrow{\text{Load}[f_2]} t_3 \in \overline{G}(P, \tilde{S}) \\
&\ldots \\
&t_{k-1} \xrightarrow{\text{Store}[f_{k-1}]} w_{k-1} \xrightarrow{\text{Alias}} z_k \xrightarrow{\text{Load}[f_{k-1}]} w_k \in \overline{G}(P, \tilde{S}).
\end{aligned}$$

By induction, the static analysis computes $z_1 \xrightarrow{\text{Transfer}} t_i \in \overline{G}(P, \tilde{S})$ for each $i \in [k-1]$. Thus, the static analysis computes $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(P, \tilde{S})$, as claimed.

Next, we show the converse, i.e., that $\overline{G}(P, \tilde{S}) \subseteq \overline{G}(P, S) \cup \overline{G}'(P, \tilde{S})$. First, note that the only production with Store$[f]$ is

$$\text{Transfer} \rightarrow \text{Transfer Store}[f]\ \text{Alias Load}[f].$$

Since each f_i is a fresh field, there is only one edge labeled Store$[f_i]$ and only one edge labeled Load$[f_i]$. Thus, this production can only be triggered if (i) $z_i \xrightarrow{\text{Alias}} w_i \in \overline{G}(P, \tilde{S})$, and (ii) for some vertex x, $x \xrightarrow{\text{Transfer}} t_i \in \overline{G}(P, \tilde{S})$. If triggered, the static analysis adds an edge $x \xrightarrow{\text{Transfer}} t_{i+1}$ to $\overline{G}(P, \tilde{S})$. For $i = 1$, the only vertices x satisfying the second condition are $x = z_1$ and $x = t_1$. By

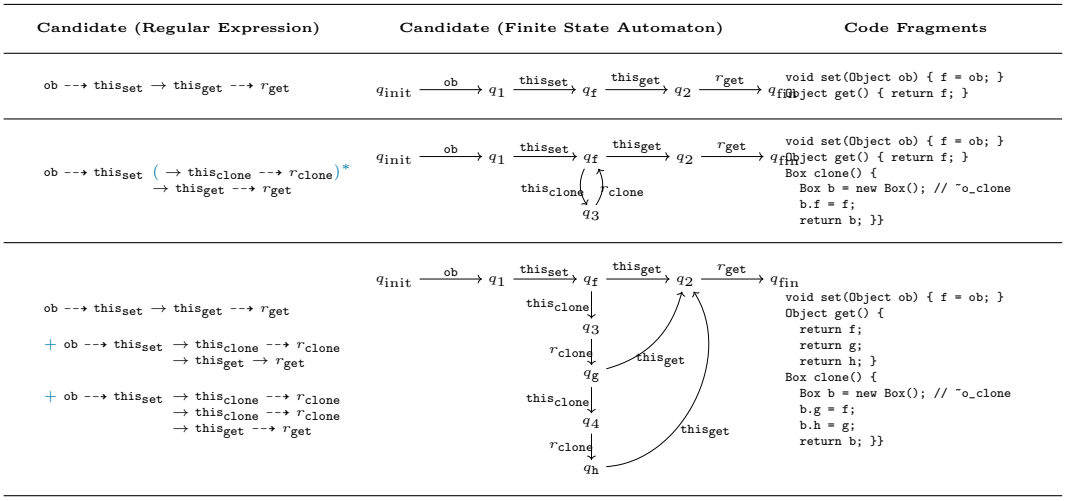


Figure 5.8: Examples of candidate code fragment specifications (left column), and the equivalent path specifications as a regular expression (middle column) and as a finite state automaton (right column).

induction, if $w_i \xrightarrow{\text{Alias}} z_{i+1} \in \overline{G}(P, \tilde{S})$ for each i, we have

$$z_1 \xrightarrow{\text{Transfer}} t_i \in \overline{G}(P, \tilde{S})$$
$$t_j \xrightarrow{\text{Transfer}} t_i \in \overline{G}(P, \tilde{S})$$

for each $j \leq i$. None of the t_i are part of an Assign edge except t_1 and t_k; for the latter, the production Transfer → Transfer Assign triggers and we get $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(P, \tilde{S})$. This edge is the only one in $\overline{G}(P, \tilde{S})$ that does not refer to vertices extracted from the code fragments, so the claim follows. □

5.6.2 Converting a Regular Set of Path Specifications

Our construction generalizes straightforwardly to constructing code fragment specifications from $\hat{M}$. For each state $q \in Q$, we introduce a fresh field $f_q \in \mathcal{F}$. Intuitively, transitions into q correspond to stores into f_q, and transitions coming out of q correspond to loads into f_q. In particular, we include statements in m according to the rules in Figure 5.7.

The following guarantee follows similarly to the proof of Proposition 5.6.1:

Proposition 5.6.2 We have $\overline{G}(P, \tilde{S}) = \overline{G}(P, S) \cup \overline{G}'(P, \tilde{S})$, where $\overline{G}'(P, \tilde{S})$ is defined as before.

In Figure 5.8, we show examples of path specifications (first column), the corresponding FSA (middle column), and the generated code fragment specifications. For example, in the second line,

the transitions

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

generate the specifications for `set` (the first two transitions, with field `f` $= f_{q_2}$) and `get` (the last two transitions), and the self-loop

$$q_2 \xrightarrow{\texttt{this}_{\texttt{clone}}} q_6 \xrightarrow{r_{\texttt{clone}}} q_2$$

generates the specification for `clone`.

5.7 Proof of Equivalence Theorem

We prove Theorem 5.3.4, relegating the proof of technical lemmas to Section 5.7.6.

5.7.1 Converting the Library Implementation to Path Specifications

First, we describe how to convert the library implementation into a set S of transfer and proxy object specifications. A specification of the form

$$z_1 \dashrightarrow w_1 \rightarrow \ldots \rightarrow z_k \dashrightarrow w_k.$$

is included in S if there exist paths

$$z_1 \overset{\beta_1}{\dashrightarrow} w_1, \quad \ldots, \quad z_k \overset{\beta_k}{\dashrightarrow} w_k$$

such that $A \overset{*}{\Rightarrow} \beta_1 \tilde{\alpha}_1 \ldots \tilde{\alpha}_{k-1} \beta_k$ in C_{pt}, where

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

and

$$\tilde{\alpha}_i = \begin{cases} \text{Assign} & \text{if } w_i = p_{m_i} \text{ and } z_{i+1} = r_{m_{i+1}} \\ \overline{\text{Assign}} & \text{if } w_i = r_{m_i} \text{ and } z_{i+1} = p_{m_{i+1}} \\ \overline{\text{New}}\ \text{New} & \text{if } w_i = p_{m_i} \text{ and } z_{i+1} = p_{m_{i+1}}. \end{cases}$$

Then, we prove that the conclusion of Theorem 5.3.4 holds for S constructed with this algorithm.

5.7.2 Proof Overview

Let $\overline{G}$ denote the points-to sets computed by running the static analysis with the library implementation available, and $\overline{G}(S)$ denote the points-to sets computed by running the static analysis with the path specifications S. We have to prove that $\overline{G} = \overline{G}(S)$; the direction $\overline{G}(S) \subseteq \overline{G}$ follows easily, since a path specification s is included in S exactly when the library implementation would imply the same logical formula as the semantics of s.

The challenging direction is to show that S is sound, i.e., $\overline{G} \subseteq \overline{G}(S)$. For simplicity, we focus on points-to edges $o \xrightarrow{\text{FlowsTo}} x$; the alias and transfer relations follow similarly. Suppose that $o \xrightarrow{\text{FlowsTo}} y \in \overline{G}(S)$; then, there must exist a path $o \xrightarrow{\text{New}} x \overset{\alpha}{\dashrightarrow} y$, where Transfer $\overset{*}{\Rightarrow} \alpha$. This path passes into and out of library functions, leading to a decomposition

$$x \overset{\alpha_0}{\dashrightarrow} z_1 \overset{\beta_1}{\dashrightarrow} w_1 \overset{\alpha_1}{\dashrightarrow} ... \overset{\beta_k}{\dashrightarrow} w_k \overset{\alpha_k}{\dashrightarrow} y, \tag{5.6}$$

where $\alpha = \alpha_0\beta_1\alpha_1...\beta_k\alpha_k$. This decomposition suggests that the following path specification may be applied to derive $x \xrightarrow{\text{Transfer}} y$:

$$z_1 \dashrightarrow w_1 \rightarrow ... \rightarrow z_k \dashrightarrow w_k. \tag{5.7}$$

At a high level, our proof has two parts. First, we prove the case where the segments of α in the program do not contain field accesses, i.e., $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, where

$$\begin{aligned}
\Sigma_{\text{free}} &= \{\text{Assign}, \overline{\text{Assign}}, \text{New}, \overline{\text{New}}\} \\
\Sigma_{\text{prog}} &= \{\text{Store}[f], \text{Load}[f], \overline{\text{Store}[f]}, \overline{\text{Load}[f]} \mid f \in \mathcal{F}_{\text{prog}}\} \\
\Sigma_{\text{lib}} &= \{\text{Store}[f], \text{Load}[f], \overline{\text{Store}[f]}, \overline{\text{Load}[f]} \mid f \in \mathcal{F}_{\text{lib}}\}.
\end{aligned}$$

Second, we show how "nesting" of fields allows us to reduce the general case to the case $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}} \cup \Sigma_{\text{prog}})^*$. In particular, by Assumption 5.3.3, the library field accesses and program field accesses do not match one another. As previously discussed, this assumption can be enforced by a purely syntactic program transformation where accesses to library fields in the program are converted into calls to getter and setter functions.

Consider a path of the form (5.6) such that $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$. We need to show that in this case, we derive the edge $x \xrightarrow{\text{Transfer}} y \in \overline{G}(S)$, where S is constructed as in Section 5.7.1. Our proof of this claim relies on two results. The first result says that for such a path, the conclusion of (5.7) holds when each w_i is connected to z_{i+1} by α_i:

Proposition 5.7.1 For any path of the form (5.6) such that $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$ we have (i) the case $w_i = r_i$ and $z_{i+1} = r_{i+1}$ cannot happen, and (ii) Transfer $\overset{*}{\Rightarrow} \beta_1\alpha_1\beta_2...\alpha_{k-1}\beta_k$.

As a consequence of this result, we know that the path specification (5.7) is contained in S. The

second result says that the premise of (5.7) holds for our case:

Proposition 5.7.2 For any path of the form (5.7) such that $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, we have

$$A_i \overset{*}{\Rightarrow} \alpha_i \quad (\forall i \in [k-1])$$
$$A_i \overset{*}{\Rightarrow} \alpha_i \quad (\forall i \in \{0, k\}).$$

Therefore, we can conclude that when running the static analysis using path specifications, we derive the conclusion of the path specification (5.7), i.e., $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(S)$. In summary, we have the following result:

Theorem 5.7.3 Theorem 5.3.4 holds for any $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$.

Proof: Consider an edge $x \xrightarrow{\text{Transfer}} y \in \overline{G}$ derived by the static analysis using the library implementation. We claim that this edge is derived by the static analysis when using path specifications, i.e., $x \xrightarrow{\text{Transfer}} y \in \overline{G}(S)$. By Proposition 5.7.1, we conclude that (5.7) is in S. Furthermore, by Proposition 5.7.2, the premise of (5.7) holds, so the static analysis derives its conclusion, i.e., $z_1 \xrightarrow{\text{Transfer}} w_k \in \overline{G}(S)$. Therefore, we have

$$x \xrightarrow{\text{Transfer}} z_1 \xrightarrow{\text{Transfer}} w_k \xrightarrow{\text{Transfer}} y \in \overline{G}(S),$$

so the static analysis derives $x \xrightarrow{\text{Transfer}} y \in \overline{G}(S)$, as claimed.

Now, we know that any points-to edge $o \xrightarrow{\text{FlowsTo}} y \in \overline{G}$ has the form $o \xrightarrow{\text{New}} x \xrightarrow{\text{Transfer}} y$. Since we have shown that $x \xrightarrow{\text{Transfer}} y \in \overline{G}(S)$, the static analysis also derives $o \xrightarrow{\text{FlowsTo}} y \in \overline{G}(S)$, so the result follows. □

In the remainder of the section, we introduce the technical machinery that enables us to reason about "equivalence" of the semantics of different sequences of statements. Then, we describe how we prove Propositions 5.7.1 & 5.7.2. Finally, we reduce Theorem 5.3.4 to Theorem 5.7.3.

5.7.3 Equivalent Semantics

Proving Propositions 5.7.1 & 5.7.2 requires reasoning about the *equivalence* of the semantics of sequences of statements in P. For example, to prove Proposition 5.7.1, we show that each α_i is "equivalent" to $\tilde{\alpha}_i$. Intuitively, for $\tilde{\alpha}_i = \text{Assign}$, we show that the sequence of statements represented by α_i exhibits the same semantics as a single assignment. For example, $y \leftarrow x,\ z \leftarrow y$ has the same points-to effects as $z \leftarrow x$ (assuming y is temporary). We leverage the correspondence established by formulating points-to analysis as context-free language reachability:

$$\text{sequence of statements} \ = \ \text{sequence } \alpha \in \Sigma^*.$$

For example, the first sequence of statements above corresponds to (Assign Assign), and the second to Assign.

Using this correspondence, we can reduce reasoning about sequences of statements with equivalent semantics to studying equivalence classes of strings $\alpha \in \Sigma^*$:

$$\text{equivalent sequences of statements} \quad = \quad \text{equivalence classes } [\alpha] \subseteq \Sigma^* \ .$$

In particular, $\alpha, \beta \in \Sigma^*$ are *equivalent* if

$$\gamma\alpha\delta \in \mathcal{L}(C_{\text{pt}}) \Leftrightarrow \gamma\beta\delta \in \mathcal{L}(C_{\text{pt}}) \quad (\forall \gamma, \delta \in \Sigma^*). \tag{5.8}$$

In other words, α can be used interchangeably with β in any string without affecting whether the string is contained in $\mathcal{L}(C_{\text{pt}})$. We use $[\alpha] = \{\beta \in \Sigma^* \mid \alpha \sim \beta\}$ to denote the equivalence class of $\alpha \in \Sigma^*$. Then, $[\alpha] = [\beta]$ if for any two paths

$$o \xrightarrow{\gamma} v \xrightarrow{\alpha} w \xrightarrow{\delta} x, \qquad o \xrightarrow{\gamma} v \xrightarrow{\beta} w \xrightarrow{\delta} x,$$

the first results in $x \hookrightarrow o$ if and only if the second does. For example, [Assign Assign] = [Assign].

Then, equivalence is compatible with sequencing:

Lemma 5.7.4 If $[\alpha] = [\alpha']$ and $[\beta] = [\beta']$, then $[\alpha\beta] = [\alpha'\beta']$.

Proof: By definition, $\gamma\alpha\beta\delta \Leftrightarrow \gamma\alpha'\beta\delta \Leftrightarrow \gamma\alpha'\beta'\delta$. □

In particular, Lemma 5.7.4 shows that sequencing is well-defined for equivalence classes:

$$[\alpha]\ [\beta] = [\alpha\beta], \tag{5.9}$$

since different choices $\alpha' \in [\alpha]$ and $\beta' \in [\beta]$ yield the same equivalence class, i.e., $[\alpha\beta] = [\alpha'\beta']$. Abstractly, Σ^* is a semigroup, with sequencing as the semigroup operation; then, Lemma 5.7.4 shows the equivalence relation is compatible with the semigroup operation, so the quotient $\Sigma/\sim$ is a semigroup with semigroup operation (5.9).

For convenience, we let ϕ denote an element of the equivalence class of strings such that

$$\text{for all } \gamma, \delta \in \Sigma^*, \quad \gamma\phi\delta \notin \mathcal{L}(C_{\text{pt}}). \tag{5.10}$$

In other words, $[\phi]$ describes sequences of statements that can never be completed to a valid flows-to path.

5.7.4 Proofs of Propositions 5.7.1 & 5.7.2

Now, we describe how to prove that under the conditions of Proposition 5.7.1, $[\alpha_i] = [\tilde{\alpha}_i]$, which suffices to prove the proposition. We focus on the case $\tilde{\alpha}_i = \text{Assign}$; the other cases are similar. We need the following technical lemma (we give a proof in Section 5.7.6):

Lemma 5.7.5 For any $\alpha \in \Sigma^*_{\text{free}}$, we have

$$[\text{Assign}]\ [\alpha]\ [\text{Assign}] \in \{[\text{Assign}],\ [\phi]\}.$$

With this lemma, since $\tilde{\alpha}_i = \text{Assign}$, $w_{m_i} = r_{m_i}$ and $z_{m_{i+1}} = p_{m_i}$, so the path $w_{m_i} \overset{\alpha_i}{\dashrightarrow} z_{m_{i+1}}$ has form

$$w_{m_i} = r_{m_i} \xrightarrow{\text{Assign}} y_i \overset{\alpha'_i}{\dashrightarrow} x_{i+1} \xrightarrow{\text{Assign}} p_{m_{i+1}} = z_{m_{i+1}},$$

where $\alpha_i = \text{Assign}\ \alpha'_i\ \text{Assign}$. By Lemma 5.7.5,

$$[\alpha_i] = [\text{Assign}]\ [\alpha'_i]\ [\text{Assign}] \in \{[\text{Assign}],\ [\phi]\}.$$

Since (New α) $\in \mathcal{L}(C_{\text{pt}})$, we cannot have $[\alpha_i] = [\phi]$, so

$$[\alpha_i] = [\text{Assign}] = [\tilde{\alpha}_i],$$

as claimed. We have also proven the claim in Proposition 5.7.2 that $A_i \overset{*}{\Rightarrow} \alpha_i$ (with $A_i = \text{Transfer}$) also follows. The other claims in Propositions 5.7.1 & 5.7.2 follow similarly. □

5.7.5 Reduction of Theorem 5.3.4 to Theorem 5.7.3

To handle field accesses, note that pairs of terminals $(\text{Store}[f], \text{Load}[f])$ and $(\overline{\text{Load}[f]}, \overline{\text{Store}[f]})$ in strings $\alpha \in \mathcal{L}(C_{\text{pt}})$ are matching. Therefore, can identify an inner-most nested pair (σ, τ) such that the string β between σ and τ contains no field accesses, i.e., $\beta \in \Sigma_{\text{free}}$. Furthermore, by Assumption 5.3.3, library field accesses and program field accesses do not match one another. In particular, the set of matching program field accesses is

$$\Delta_{\text{prog}} = \bigcup_{f \in \mathcal{F}_{\text{prog}}} \{(\text{Store}[f], \text{Load}[f]),\ (\overline{\text{Load}[f]}, \overline{\text{Store}[f]})\}.$$

Lemma 5.7.6 For any $\alpha \in \mathcal{L}(C_{\text{pt}})$, either $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, or there exists a pair of terminals $(\sigma, \tau) \in \Delta_{\text{prog}}$ such that $\alpha = \gamma\sigma\beta\tau\delta$, where $\gamma, \delta \in \Sigma^*$ and $\beta \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$.

The next step is to characterize $[\sigma\beta\tau]$:

Lemma 5.7.7 For any $(\sigma, \tau) \in \Delta_{\text{prog}}$ and $\beta \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$,

$$[\sigma]\ [\beta]\ [\tau] \in \{[\text{Assign}],\ [\phi]\}.$$

Finally, β must be an aliasing relation:

Lemma 5.7.8 For any $\beta \in \Sigma^*$,

$$[\text{Store}[f]]\ [\beta]\ [\text{Load}[f]] = [\text{Assign}] \Rightarrow [\beta] = [\overline{\text{New}}\ \text{New}]$$
$$[\overline{\text{Load}[f]}]\ [\beta]\ [\overline{\text{Store}[f]}] = [\text{Assign}] \Rightarrow [\beta] = [\overline{\text{New}}\ \text{New}].$$

Now, if $\alpha \in \Sigma^*_{\text{free}}$, we are done. Otherwise, putting the three lemmas together, we perform the following procedure:

1. By Lemma 5.7.6, we can write $\alpha = \gamma\sigma\beta\tau\delta$, where $(\sigma, \tau) \in \Delta_{\text{prog}}$ and $\beta \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, such that

$$y \overset{\gamma}{\dashrightarrow} v \overset{\sigma}{\rightarrow} w \overset{\beta}{\dashrightarrow} t \overset{\tau}{\rightarrow} u \overset{\delta}{\dashrightarrow} x.$$

2. By Lemma 5.7.7, $[\sigma]\ [\beta]\ [\tau] = [\text{Assign}]$.
3. By Lemma 5.7.8, $[\beta] = [\overline{\text{New}}\ \text{New}]$.
4. By Theorem 5.3.4, we have $w \xrightarrow{\text{Alias}} t \in \overline{G}(\tilde{S})$; therefore, $v \overset{\text{Transfer}}{\dashrightarrow} u \in \overline{G}(\tilde{S})$ as well.
5. Recursively apply the procedure to $\alpha' = \gamma\ \text{Assign}\ \delta$.

This procedure must terminate, since α has finitely many pairs of store and load statements. Theorem 5.3.4 follows. □

5.7.6 Proof of Technical Lemmas

We prove the technical lemmas used in Section 5.7.

Proof of Lemma 5.7.5 We first show the following lemma, which completely characterizes the subgroupoid of elements $\Sigma^*_{\text{free}} \subseteq \Sigma^*$:

Lemma 5.7.9 We have

$$
\begin{aligned}
[\text{Assign}]\ [\text{Assign}] &= [\text{Assign}] \\
[\text{Assign}]\ [\overline{\text{Assign}}] &= [\phi] \\
[\overline{\text{Assign}}]\ [\text{Assign}] &= [\phi] \\
[\overline{\text{Assign}}]\ [\overline{\text{Assign}}] &= [\overline{\text{Assign}}] \\
[\text{Assign}]\ [\overline{\text{New}}\ \text{New}] &= [\phi] \\
[\overline{\text{New}}\ \text{New}]\ [\text{Assign}] &= [\overline{\text{New}}\ \text{New}] \\
[\overline{\text{Assign}}]\ [\overline{\text{New}}\ \text{New}] &= [\overline{\text{Assign}}] \\
[\overline{\text{New}}\ \text{New}]\ [\overline{\text{Assign}}] &= [\phi] \\
[\overline{\text{New}}\ \text{New}]\ [\overline{\text{New}}\ \text{New}] &= [\phi].
\end{aligned}
$$

Proof: We show the first relation; the others follow similarly. First, we show that if γ Assign $\delta \in \mathcal{L}(C_{\text{pt}})$, then γ Assign Assign$\delta \in \mathcal{L}(C_{\text{pt}})$. There must exist a derivation

$$
\begin{aligned}
\text{FlowsTo} \Rightarrow ... &\Rightarrow \text{Transfer } u_\delta \\
&\Rightarrow \text{Transfer Assign } u_\delta \\
&\Rightarrow ... \\
&\Rightarrow \gamma \text{ Assign } \delta
\end{aligned}
$$

since the only production in C_{pt} containing the terminal symbol Assign is Transfer $\rightarrow$ Transfer Assign. Therefore, the following derivation also exists:

$$
\begin{aligned}
\text{FlowsTo} \Rightarrow ... &\Rightarrow \text{Transfer } \delta \\
&\Rightarrow \text{Transfer Assign } u_\delta \\
&\Rightarrow \text{Transfer Assign Assign } u_\delta \\
&\Rightarrow ... \\
&\Rightarrow \gamma \text{ Assign Assign } \delta,
\end{aligned}
$$

i.e., γ Assign Assign $\delta \in \mathcal{L}(C_{\text{pt}})$. By a similar argument, it follows that if γ Assign Assign $\delta \in \mathcal{L}(C_{\text{pt}})$, then γ Assign $\delta \in \mathcal{L}(C_{\text{pt}})$, so [Assign] [Assign] = [Assign]. $\square$

It follows directly that if $\alpha \in \Sigma^*_{\text{free}}$, then

$$[\alpha] \in \{[\phi],\ [\epsilon],\ [\text{Assign}],\ [\overline{\text{Assign}}],\ [\overline{\text{New}}\ \text{New}]\}.$$

In particular, for $\alpha' \in \Sigma^*_{\text{free}}$, [Assign] $[\alpha'] \in \{[\text{Assign}],\ [\phi]\}$, so the lemma follows by taking $\alpha' =$

α Assign. $\square$

Proof of Lemma 5.7.6 If we replace the terminal symbols $\sigma \in \Sigma_{\text{free}}$ with ϵ in C_{pt}, then C_{pt} is a parentheses matching grammar where each "open parentheses" Store$[f]$ (resp., $\overline{\text{Load}[f]}$) must be matched with a corresponding "closed parentheses" Load$[f]$ (resp., $\overline{\text{Store}[f]}$). Also, by Assumption 5.3.3, $\Sigma_{\text{lib}} \cap \Sigma_{\text{prog}} = \emptyset$.

Now, we prove by induction on the length of α. The base case $\alpha = \epsilon$ is clear. If $\alpha \in \Sigma^*$ does not contain a pair of matched parentheses $(\text{Store}[f], \text{Load}[f]) \in \Sigma^2_{\text{prog}}$, then $\alpha \in (\Sigma_{\text{free}} \cup \Sigma_{\text{lib}})^*$, so we are done. Otherwise, for any such pair of matched parentheses, we can express $\alpha = \gamma\sigma\alpha'\tau\delta$. By induction, the lemma holds for α', so we can write $\alpha = \gamma'\sigma'\beta'\tau'\delta'$ as in the lemma. Therefore, we have

$$\alpha = (\gamma\sigma\gamma')\sigma'\beta'(\tau'\delta'\tau\delta),$$

so the claim follows. $\square$

Proof of Lemma 5.7.7 We show the case $(\sigma, \tau) = (\text{Store}[f], \text{Load}[f])$, where $f \in \mathcal{F}_{\text{prog}}$; the case $(\sigma, \tau) = (\overline{\text{Load}[f]}, \overline{\text{Store}[f]})$ is similar. First, suppose that $\gamma\sigma\beta\tau\delta \in \mathcal{L}(C_{\text{pt}})$. Then, there must exist a derivation of form

$$\begin{aligned}
\text{FlowsTo} \Rightarrow ... &\Rightarrow u_\gamma\ \overline{\text{Transfer}}\ u_\delta \\
&\Rightarrow u_\gamma\ \text{Transfer}\ \sigma\ \text{Alias}\ \tau\ u_\delta \\
&\Rightarrow ... \\
&\Rightarrow \gamma\sigma\beta\tau\delta,
\end{aligned}$$

so the following derivation exists:

$$\begin{aligned}
\text{FlowsTo} \Rightarrow ... \Rightarrow u_\gamma\ \overline{\text{Transfer}}\ u_\delta & \\
&\Rightarrow u_\gamma\ \text{Transfer Assign}\ u_\delta \\
&\Rightarrow ... \\
&\Rightarrow \gamma\ \text{Assign}\ \delta.
\end{aligned}$$

The converse follows similarly, so the claim follows. $\square$

Proof of Lemma 5.7.8 We show two preliminary lemmas.

Lemma 5.7.10 We have

$$[\text{Store}[f]]\ [\overline{\text{New}}\ \text{New}]\ [\text{Load}[f]] = [\text{Assign}]$$
$$[\overline{\text{Load}[f]}]\ [\overline{\text{New}}\ \text{New}]\ [\text{Store}[f]] = [\overline{\text{Assign}}].$$

Proof: Suppose that γ Store$[f]$ $\overline{\text{New}}$ New Load$[f]$ $\delta \in \mathcal{L}(C_{\text{pt}})$. Then, we must have derivation

$$\begin{aligned}\text{FlowsTo} \Rightarrow ... &\Rightarrow u_\gamma\ \text{Transfer}\ u_\delta \\ &\Rightarrow u_\gamma\ \text{Store}[f]\ \text{Alias}\ \text{Load}[f]\ u_\delta \\ &\Rightarrow ... \\ &\Rightarrow \gamma\ \text{Store}[f]\ \alpha\ \text{Load}[f]\ \delta,\end{aligned}$$

so we also have derivation

$$\begin{aligned}\text{FlowsTo} \Rightarrow ... &\Rightarrow u_\gamma\ \text{Transfer}\ u_\delta \\ &\Rightarrow u_\gamma\ \text{Assign}\ u_\delta \\ &\Rightarrow ... \\ &\Rightarrow \gamma\ \text{Assign}\ \text{Load}[f]\ \delta.\end{aligned}$$

Thus, γ Assign $\delta \in \mathcal{L}(C_{\text{pt}})$. The converse follows similarly, as does the second claim. □

Lemma 5.7.11 For any $\beta \in \Sigma^* \setminus \{\epsilon\}$, we have

$$\begin{aligned}[\beta] = [\text{Assign}] &\Leftrightarrow \beta \in \mathcal{L}(C_{\text{pt}}, \text{Transfer}) \\ [\beta] = [\overline{\text{Assign}}] &\Leftrightarrow \beta \in \mathcal{L}(C_{\text{pt}}, \overline{\text{Transfer}}) \\ [\beta] = [\overline{\text{New}}\ \text{New}] &\Leftrightarrow \beta \in \mathcal{L}(C_{\text{pt}}, \text{Alias}).\end{aligned}$$

Proof: We first show the forward implication. If $[\beta] = [\text{Assign}]$, then New Assign $\in \mathcal{L}(C_{\text{pt}})$, so New $\beta \in \mathcal{L}(C_{\text{pt}})$. Therefore, there must exist a derivation

$$\text{FlowsTo} \Rightarrow \text{New Transfer}\ \Rightarrow ...\ \Rightarrow \text{New}\ \beta,$$

so $\beta \in \mathcal{L}(C_{\text{pt}}, \text{Transfer})$. The other two cases follow similarly. Now, we show the backward implication. Suppose that $\beta \in \mathcal{L}(C_{\text{pt}}, \text{Transfer})$. We prove by structural induction on the derivation of β from Transfer. Since $\beta \neq \epsilon$, β cannot have been produced by Transfer $\Rightarrow \epsilon$. If β is produced by Transfer $\rightarrow$ Transfer Assign, then $\beta = \beta'$Assign, where $\beta' \in \mathcal{L}(C_{\text{pt}}, \text{Transfer})$. By induction,

$[\beta'] = [\text{Assign}]$, so

$$[\beta] = [\beta'] \; [\text{Assign}] = [\text{Assign}] \; [\text{Assign}] = [\text{Assign}],$$

where the last step follows from Lemma 5.7.9. Next, if β is produced using the production

$$\text{Transfer} \rightarrow \text{Transfer Store}[f] \; \text{Alias Load}[f],$$

then $\beta = \beta' \; \text{Store}[f] \; \beta'' \; \text{Load}[f]$, where $\beta' \in \mathcal{L}(C_{\text{pt}}, \text{Transfer})$ and $\beta'' \in \mathcal{L}(C_{\text{pt}}, \text{Alias})$. By induction, $[\beta'] = [\text{Assign}]$ and $[\beta''] = [\overline{\text{New}} \; \text{New}]$, so

$$\begin{aligned} \beta &= [\beta'] \; [\text{Store}[f]] \; [\beta''] \; [\text{Load}[f]] \\ &= [\text{Assign}] \; [\text{Store}[f]] \; [\overline{\text{New}} \; \text{New}] \; [\text{Load}[f]] \\ &= [\text{Assign}], \end{aligned}$$

where the last step follows from Lemma 5.7.10 and Lemma 5.7.9. The remaining cases follow similarly. □

Now, suppose that $[\text{Store}[f]] \; [\beta] \; [\text{Load}[f]] = [\text{Assign}]$. Since

$$\text{New Store}[f] \; \overline{\text{New}} \; \text{New Load}[f] \in \mathcal{L}(C_{\text{pt}}),$$

we have

$$\text{New Store}[f] \; \beta \; \text{Load}[f] \in \mathcal{L}(C_{\text{pt}}),$$

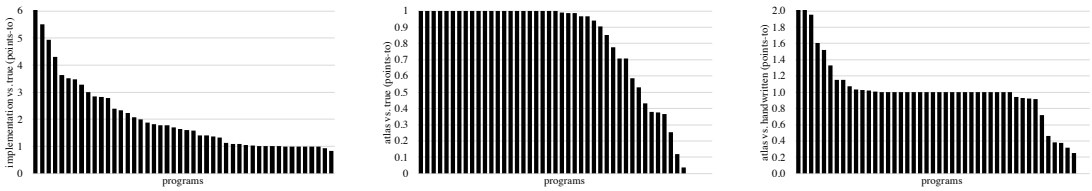
so the following derivation must exist:

$$\begin{aligned} \text{FlowsTo} &\Rightarrow \text{New Store}[f] \; \text{Alias Load}[f] \\ &\Rightarrow \ldots \\ &\Rightarrow \text{New Store}[f] \; \beta \; \text{Load}[f], \end{aligned}$$

i.e., $\beta \in \mathcal{L}(C_{\text{pt}}, \text{Alias})$. By Lemma 5.7.11, we have $[\beta] = [\overline{\text{New}} \; \text{New}]$. The second case follows similarly. □

5.8 Implementation

We have implemented our specification inference algorithm as Atlas. We use Atlas to infer specifications for our static analysis tool (a variant of Chord [103] modified to use Soot [147] as a



(a) (b) (c)

Figure 5.9: The ratio of nontrivial program points-to edges discovered using (a) ground truth specifications versus the Collections API implementation, (b) Atlas versus ground truth specifications, and (c) Atlas versus existing specifications. The ratios are sorted from highest to lowest for the 46 benchmark programs with nontrivial points-to edges. In (a) and (c), some values exceeded the graph scale.

backend) for Java and Android programs, which runs a 1-object-sensitive points-to analysis. Our tool omits analyzing the Android framework and the Java standard library, and instead analyzes user-provided code fragment specifications. Over two years, we have handwritten several hundred code fragment specifications, including many written specifically for our benchmark of programs.

In our evaluation, we focus on specifications for the Java 1.7 Collections API, in particular, for 31 classes that implement the `Collection` and `Map` interfaces. We focus on the Java Collections API since the functions it contains exhibit a variety of interesting points-to effects, which makes them a challenging target for inferring specifications. Indeed, the most complex points-to specifications we have written by hand for the entire Android framework are all for functions in this API. We can easily use our approach to infer specifications for the entire Android framework, but limit ourselves to these APIs to focus our evaluation—in particular, we provide ground truth specifications for a large fraction of the Java Collections API.

In total, there are four sets of code fragments for the Java Collections API that our tool can use:

- Specifications inferred by Atlas for the Colletions API.
- The 58 existing, handwritten specifications for the Collections API added to our system over the past two years (many written specifically for our benchmark).
- Ground truth specifications we wrote by hand for the 12 classes in the Collections API that are most frequently used by our benchmark—98.5% of calls to the Collections API target a function in one of these 12 classes.
- The class files comprising the actual implementation of the Collections API (developed by Oracle).

5.9 Evaluation

First, we evaluate the precision and recall of Atlas compared to both ground truth specifications and existing specifications. Second, we compare the points-to edges computed by our static points-to analysis using different code fragment specifications on a benchmark of 78 Android programs.

5.9.1 Specification Inference

We sampled a total of four million candidate path specifications (two million using each random sampling and MCTS), and run Atlas using the positive examples.

Positive examples: random sampling vs. MCTS. We sampled two million candidate path specifications using each sampling algorithm. Random sampling found 3,124 positive examples, whereas MCTS found 10,153. We aggregated all examples for a total of 11,613 positive examples.

Object initialization: null vs. instantiation. Each of the 11,613 positive examples passed the test case constructed using instantiation, but only 7,721 passed when using null initialization, i.e., instantiation finds 50% more specifications.

Inferred specifications. We inferred code fragment specifications for 733 functions; 591 included a non-proxy-object specification and 330 included a proxy-object specification.

Precision and recall. We examine the top 50 most frequently called functions in our benchmark (in total, accounting for 95% of the function calls). We count a specification as admissible if it is identical to the specification we would have written. For specifications with multiple statements, we count each statement fractionally. The recall of our algorithm was 97% (i.e., we inferred the admissible specification for 97% of the 50 functions) and the precision was 100% (i.e., each specification was as precise as the true specification).

Handwritten specifications. We inferred 92% of the 58 handwritten specifications; in addition, we infer an order of magnitude more new specifications (733 versus 58).

Discussion. Each of the 5 false negatives we examined was due to a false negative in our test case synthesis. For example, the function `subList(int,int)` in the `List` class requires a call of the form `subList(0, 1)` to retrieve the first object in the list. Similarly, the function `set(int,Object)` in the `List` class requires an object to already be in the list or it raises an index out-of-bounds exception.

5.9.2 Points-To Analysis

We evaluate different code fragment specifications using our 1-object-sensitive points-to analysis; this analysis is particularly sensitive to missing code fragment specifications, since many points-to edges depend on multiple specifications, and will not be computed if any of these specifications are missing. Furthermore, oftentimes a given program makes significant use of a small number of specifications, so a single missing specification can have a large effect. We show that nevertheless, Atlas can recover a large fraction of points-to edges. Overall, we make the following comparisons:

- We demonstrate the effectiveness of using specifications by showing that using ground truth specifications significantly decreases false positives compared to analyzing the actual implementation.
- We evaluate Atlas by comparing the specifications it infers to our ground truth specifications.
- We show that Atlas improves upon our existing, handwritten specifications, even though many of these specifications were written specifically for our benchmark.

To compare two sets of code fragments S and S', we replace the handwritten specifications for the Collections API with each S and S' and run our points-to analysis. We use $\Pi(S) \subseteq \mathcal{V} \times \mathcal{O}$ to denote the points-to edges computed using code fragments S, restricting to points-to edges in the program, i.e., $x \hookrightarrow o \in \Pi(S)$ only if x and o are in the program. Additionally, many points-to edges can be discovered without specifications—we disregard such *trivial* points-to edges $\Pi(\emptyset)$, where $\emptyset$ denotes the code fragments where each function is a no-op. Then, we report the ratio of number of nontrivial points-to edges discovered using S to the number discovered using S', i.e.,

$$R(S, S') = \frac{|\Pi(S) \setminus \Pi(\emptyset)|}{|\Pi(S') \setminus \Pi(\emptyset)|}.$$

We omit the 46 programs for which there are no nontrivial points-to edges, i.e., $\Pi(S) = \Pi(S') = \Pi(\emptyset)$. Finally, we focus on points-to edges since results for alias edges $x \xrightarrow{\text{Alias}} y$ (where both x and y are in the program) are very similar.

Benefit of specifications. We begin by showing the benefit of using specifications. In particular, we study the ratio $R(S_{\text{impl-12}}, S_{\text{true-12}})$ of analyzing the library implementation S_{impl} to analyzing the ground truth specifications S_{true}, both restricted to the 12 most frequently used classes. This ratio measures the number of false positives due to analyzing the implementation instead of using ground truth specifications, since every points-to edge computed using the implementation but not the ground truth specifications is a false positive. Figure 5.9 (a) shows this ratio $R(S_{\text{impl-12}}, S_{\text{true-12}})$. For a third of programs, the false positive rate is more than 100% (i.e., when $R \geq 2$), and for four programs, the false positive rate was more than 300% (i.e., $R \geq 4$). The average false positive rate

was 115.2%, and the median was 62.1%. Furthermore, for two programs, there were actually false negatives (i.e., $R < 1$) due to unanalyzable calls to native code.

Finally, running time decreased by an average of 7.5%, and by 12.4% when restricted to analyses that ran for more than five minutes, even though we only analyzed 12 classes in the library implementation. In our experience, our points-to analysis is substantially more scalable than analyzing the Java standard library implementation.

Comparison to ground truth specifications. To show the quality of the specifications inferred by ATLAS, we study the ratio $R(S_{\text{atlas-12}}, S_{\text{true-12}})$ of using specifications inferred by ATLAS to using ground truth specifications, both restricted to the 12 most frequently used classes. We found that using ATLAS does not compute a single false positive points-to edge compared to using ground truth specifications, i.e., the precision of ATLAS is 100%. Then, the ratio $R(S_{\text{atlas-12}}, S_{\text{true}-12})$ measures the number of false negative points-to edges when using ATLAS compared to ground truth. Figure 5.9 (b) shows $R(S_{\text{atlas-12}}, S_{\text{true-12}})$. This number is almost one for more than half the programs, i.e., for almost half the programs, there are no false negatives. The median recall is 99.0%, and the average recall is 75.8%.

Improving upon handwritten specifications. To show how ATLAS can improve upon our existing, handwritten specifications, we study the ratio $R(S_{\text{atlas}}, S_{\text{hand}})$ of using specifications inferred by ATLAS to using the 58 existing specifications (on all 31 classes in the Collections API). This ratio compares the recall of ATLAS to that of our existing specifications—a higher ratio says that ATLAS has better recall, and a lower ratio says that our existing specifications have better recall. Figure 5.9 (c) shows $R(S_{\text{atlas}}, S_{\text{hand}})$. ATLAS finds a number of new points-to edges compared to the existing, handwritten specifications, despite the fact that many of the existing specifications were written specifically for this benchmark. On average, ATLAS discovers 20.1% new points-to edges. The median is 100%, i.e., ATLAS find the same number of points-to edges as our existing system.

Discussion. Our results show how the specifications inferred by ATLAS substantially improve recall compared to handwritten specifications. Oftentimes code is simply unavailable for analysis, e.g., due to native code, dynamically loaded code, or significant use of reflection. For such code, specifications are the only practical solution for precise and scalable static analysis. However, specifications are expensive and error prone to write—writing ground truth specifications for just 12 classes took one student more than a week of time, and bugs were discovered in the specifications during the course of our evaluation.

ATLAS is an automatic approach to generating specifications, and produces higher quality specifications compared to writing them by hand. Production systems often already require handwritten specifications to handle missing or hard-to-analyze code [49], but typically only provide specifications for the most frequently used functions. Tools like ATLAS that infer specifications for missing

code are crucial for improving the usability of static analysis.

5.10 Conclusion

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

Chapter 6

Synthesizing Program Input Grammars

In this chapter, we study the problem of automatically synthesizing grammars representing program input languages. Our goal is to extend some of the ideas for active learning of points-to summaries to infer more challenging specifications. We propose a grammar synthesis algorithm that is similar to the active language learning algorithm described in Chapter 5—it iteratively proposes candidate generalizations of the current language and then uses queries to an oracle to avoid choosing incorrect generalizations. However, instead of operating on finite state automata, the grammar synthesis algorithm we propose in this chapter operates on regular expressions and context-free grammars. In particular, program input languages are often context-free, so an algorithm for synthesizing program input languages must be able to learn recursive structure not present in points-to summaries.

Such a grammar synthesis algorithm has many potential applications. Our primary motivation is the possibility of using synthesized grammars with grammar-based fuzzers [97, 61, 75]. For example, such inputs can be used to find bugs in real-world programs [115, 102, 64, 158], learn abstractions [104], predict performance [77], and aid dynamic analysis [106]. Beyond fuzzing, a grammar synthesis algorithm could be used to reverse engineer input formats [76], in particular, network protocol message formats can help security analysts discover vulnerabilities in network programs [29, 154, 91, 92]. Synthesized grammars could also be used to whitelist program inputs, thereby preventing exploits [120, 142, 121].

Approaches to synthesizing program input grammars typically examine executions of the program, and then generalize these observations to a representation of valid inputs. These approaches can be either *whitebox* or *blackbox*. Whitebox approaches assume that the program code is available for analysis and instrumentation, for example, using dynamic taint analysis [76]. Such an approach is difficult when only the program binaries are available or when parts of the code (e.g., libraries)

are missing. Furthermore, these techniques often require program-specific configuration or tuning, and may be affected by the structure of the code. We consider the blackbox setting, where we only require the ability to execute the program on a given input and observe its corresponding output. Since the algorithm does not examine the program's code, its performance depends only on the language of valid inputs, and not on implementation details.

A number of existing language inference algorithms can be adapted to this setting [42]. However, we found them to be unsuitable for synthesizing program input grammars. In particular, L-Star [12] and RPNI [110], the most widely studied algorithms [148, 35, 58, 25, 36], were unable to learn or approximate even simple input languages such as XML, and furthermore do not scale even to small sets of seed inputs. Surprisingly, we found that L-Star and RPNI perform poorly even on the class of regular languages they target.

The problem with these algorithms is that despite having theoretical guarantees, they depend on assumptions that do not hold in the setting of learning program input grammars. For example, they typically avoid overgeneralizing by relying on an "oracle" to provide negative examples that are used by the algorithm to identify and remove overly general portions of the language. However, these oracles are not available in our setting—e.g., L-Star obtains such examples from an equivalence oracle, and RPNI obtains them "in the limit". They likewise assume that positive examples exercising all interesting behaviors are provided by this oracle. In our setting, the needed positive and negative examples are difficult to find, and existing algorithms consistently overgeneralize (e.g., return Σ^*) or undergeneralize (e.g., return $\emptyset$). Additionally, despite having polynomial running time, they can be very slow on our problem instances. To the best of our knowledge, other existing grammar inference algorithms are either impractical [87, 42] or make assumptions similar to L-Star and RPNI [78].

This chapter presents the first practical algorithm for synthesizing program input grammars in the blackbox setting. Our algorithm synthesizes a context-free grammar $\hat{C}$ encoding the language L_* of valid program inputs, given

- A small set of *seed inputs* $E_{\text{in}} \subseteq L_*$ (i.e., examples of valid inputs). Typically, seed inputs are readily available—in our evaluation, we use small test suites that come with programs or examples from documentation.
- Blackbox access to the program executable to answer *membership queries* (i.e., whether a given input is valid).

Our algorithm adopts a high-level design commonly used by language learning algorithms (e.g., RPNI)—it starts with the language containing exactly the given positive examples, and then incrementally generalizes this language, using negative examples to avoid overgeneralizing. Our algorithm avoids the shortcomings of existing algorithms in two ways:

- It considers a much richer set of potential generalizations, which addresses the issue of omitted positive examples.

- It generates negative examples on the fly to avoid overgeneralizing, which addresses the issue of omitted negative examples.

In particular, our algorithm constructs a series of increasingly general languages using *generalization steps*. Each step first proposes a number of candidate languages that generalize the current language, and then uses carefully crafted membership queries to reject candidates that overgeneralize. Our algorithm considers candidates that (i) add repetition and alternation constructs characteristic of regular expressions, (ii) induce recursive productions characteristic of context-free grammars, in particular, parentheses matching grammars, and (iii) generalize constants in the grammar.

We implement our approach in a tool called GLADE,[1]. We conduct an extensive empirical evaluation of GLADE (Section 6.7), and show that GLADE substantially outperforms both L-Star and RPNI, even when restricted to synthesizing regular expressions. Furthermore, we show that GLADE successfully synthesizes input grammars for real programs, which can be used to fuzz test those programs. In particular, GLADE automatically synthesizes a program input grammar, and then uses the synthesized grammar in conjunction with a standard grammar-based fuzzer (described in Section 6.7.3) to generate new test inputs. Many fuzzing applications require valid inputs, for example, differential testing [158]. We show that when restricted to generating valid inputs, GLADE increases line coverage compared to both a naïve fuzzer and a production fuzzer afl-fuzz [159]. Our contributions are:

- We introduce an algorithm for synthesizing program input grammars from seed inputs and blackbox program access (Section 6.2). Our algorithm first learns regular properties such as repetitions and alternations (Section 6.3), and then learns recursive productions characteristic of matching parentheses grammars (Section 6.4).
- We implement our grammar synthesis algorithm in a tool called GLADE, and show that GLADE outperforms two widely studied language learning algorithms, L-Star and RPNI, in our application domain (Section 6.7.2).
- We use GLADE to fuzz test programs, showing that it increases the number of newly covered lines of code using valid inputs by up to 6× compared to two baseline fuzzers (Section 6.7.3).

6.1 Problem Formulation

Suppose we are given a program that takes inputs in Σ^*, where Σ is the input alphabet (e.g., ASCII characters). We let $L_* \subseteq \Sigma^*$ denote the *target language* of valid program inputs; typically, L_* is a highly structured subset of Σ^*. Our goal is to synthesize a language $\hat{L}$ approximating L_* from blackbox program access and seed inputs $E_{\text{in}} \subseteq L_*$. We represent blackbox program access as an

[1] GLADE stands for Grammar Learning for AutomateD Execution.

- Target language $\mathcal{L}(C_{\text{XML}})$, where the context-free grammar C_{XML} has terminals $\Sigma_{\text{XML}} = \{\texttt{a}, ..., \texttt{z}, \texttt{<}, \texttt{>}, \texttt{/}\}$, start symbol A_{XML}, and production

$$A_{\text{XML}} \rightarrow (\texttt{a} + ... + \texttt{z} + \texttt{<a>} A_{\text{XML}} \texttt{})^*$$

- Oracle $\mathcal{O}_{\text{XML}}(\alpha) = \mathbb{I}[\alpha \in \mathcal{L}(C_{\text{XML}})]$
- Seed inputs $E_{\text{XML}} = \{\alpha_{\text{XML}}\}$, where $\alpha_{\text{XML}} =$ `<a>hi</a>`

Figure 6.1: A context-free language $\mathcal{L}(C_{\text{XML}})$ of XML-like strings, along with an oracle $\mathcal{O}_{\text{XML}}$ for this language and a seed input α_{XML}.

oracle $\mathcal{O}$ such that $\mathcal{O}(\alpha) = \mathbb{I}[\alpha \in L_*]$ (here, $\mathbb{I}$ is the indicator function, so $\mathbb{I}[\mathcal{C}]$ is 1 if $\mathcal{C}$ is true and 0 otherwise). In particular, we run the program on input $\alpha \in \Sigma^*$, and conclude that α is a valid input (i.e., $\alpha \in L_*$) if the program does not print an error message. Access to the oracle is crucial to avoid overgeneralizing, e.g., rejecting $\hat{L} = \Sigma^*$, whereas the seed inputs give a starting point from which to generalize.

As a running example, suppose the program input language is the XML-like grammar C_{XML} shown in Figure 6.1. We use + to denote alternations and $*$ (the Kleene star) to denote repetitions. Terminals that are part of regular expressions or context-free grammars are highlighted in blue. Given seed input α_{XML} and oracle $\mathcal{O}_{\text{XML}}$, our goal is to synthesize a language $\hat{L}$ approximating $L_* = \mathcal{L}(C_{\text{XML}})$.

Ideally, we would learn L_* exactly, i.e., $\hat{L} = L_*$, but it is impossible to guarantee exact learning [66]. Instead, we want $\hat{L}$ to be a good approximation of L_*. To measure the approximation quality, we require probability distributions over L_* and $\hat{L}$. In Section 6.7.1, we define the distributions we use in detail. Briefly, we convert the context-free grammar into a *probabilistic context-free grammar*, and use the distribution induced by sampling strings in this probabilistic grammar. Then, we measure the quality of $\hat{L}$ as follows:

Definition 6.1.1 Let $\mathcal{P}_{L_*}$ and $\mathcal{P}_{\hat{L}}$ be probability distributions over L_* and $\hat{L}$, respectively. The *precision* of $\hat{L}$ is $\Pr_{\alpha \sim \mathcal{P}_{\hat{L}}}[\alpha \in L_*]$ and the *recall* of $\hat{L}$ is $\Pr_{\alpha \sim \mathcal{P}_{L_*}}[\alpha \in \hat{L}]$ (here, $\alpha \sim \mathcal{P}$ denotes a random sample from $\mathcal{P}$).

For high precision, a randomly sampled string $\alpha \sim \mathcal{P}_{\hat{L}}$ must be valid with high probability, i.e., $\alpha \in L_*$. For high recall, $\hat{L}$ must contain a randomly sampled valid string $\alpha \sim \mathcal{P}_{L_*}$ with high probability. Both are desirable: $\hat{L} = \{\alpha_{\text{in}}\}$ has perfect precision but typically low recall, whereas $\hat{L} = \Sigma^*$ has perfect recall but typically low precision. Finally, while the synthesized language $\hat{L}$ is context-free, it is often possible for $\hat{L}$ to approximate L_* with high precision and recall even if L_* is not context-free (e.g., L_* is context-sensitive).

Algorithm 8 Our grammar synthesis algorithm. Given seed input $\alpha_{\text{in}} \in L_*$ and oracle $\mathcal{O}$ for L_*, it returns an approximation of L_*.

```
procedure LearnLanguage(α_in, O)
    L̂_current ← {α_in}
    while true do
        M ← ConstructCandidates(L̂_current)
        L̃_chosen ← ∅
        for all L̃ ∈ M do
            S ← ConstructChecks(L̂_current, L̃)
            if CheckCandidate(S, O) then
                L̃_chosen ← L̃
                break
            end if
        end for
        if L̃_chosen = ∅ then
            return L̂_current
        end if
        L̂_current ← L̃_chosen
    end while
end procedure
procedure CheckCandidate(S, O)
    for all α ∈ S do
        if O(α) = 0 then
            return false
        end if
    end for
    return true
end procedure
```

6.2 Overview

In this section, we give an overview of our grammar synthesis algorithm (summarized in Algorithm 8). We consider the case where E_{in} consists of a single seed input $\alpha_{\text{in}} \in L_*$; an extension to multiple seed inputs is given in Section 6.5.1. Our algorithm starts with the language $\hat{L}_1 = \{\alpha_{\text{in}}\}$ containing only the seed input, and constructs a series of languages

$$\{\alpha_{\text{in}}\} = \hat{L}_1 \Rightarrow \hat{L}_2 \Rightarrow ...,$$

where $\hat{L}_{i+1}$ results from applying a *generalization step* to $\hat{L}_i$. On one hand, we want the languages to become successively larger (i.e., $\hat{L}_i \subseteq \hat{L}_{i+1}$); on the other hand, we want to avoid overgeneralizing (ideally, the newly added strings $\hat{L}_{i+1} \setminus \hat{L}_i$ should be contained in L_*). Our framework returns the current language $\hat{L}_i$ if it is unable to generalize $\hat{L}_i$ in any way. Figure 6.2 shows the series of languages constructed by our algorithm for the example in Figure 6.1. Steps R1-R9 (detailed in Section 6.3) generalize the initial language $\hat{L}_1 = \{\alpha_{\text{XML}}\}$ by adding repetitions and alternations. Steps C1-C2 (detailed in Section 6.4) add recursive productions.

We now describe generalization steps at a high level.

Step	Language	Candidates	Checks
R1	$[\texttt{<a>hi}]_{\text{rep}}$	⋆ $([\texttt{<a>hi}]_{\text{alt}})^*$ $([\texttt{<a>hi</a}]_{\text{alt}})^*[\texttt{>}]_{\text{rep}}$ … $\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{}]_{\text{rep}}$ …	{ϵ ✓, <a>hi<a>hi ✓} {<a>hi</a ✗, <a>hi</a<a>hi ✗} … {<a> ✓, <a>hihi ✓} …
R2	$([\texttt{<a>hi}]_{\text{alt}})^*$	$([\texttt{<}]_{\text{rep}} + [\texttt{a>hi}]_{\text{alt}})^*$ … ⋆ $([\texttt{<a>hi}]_{\text{rep}})^*$	{< ✗, a>hi ✗} … ∅
R3	$([\texttt{<a>hi}]_{\text{rep}})^*$	$(([\texttt{<a>hi</a}]_{\text{alt}})^*[\texttt{>}]_{\text{rep}})^*$ … ⋆ $(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{}]_{\text{rep}})^*$ …	{<a>hi</a ✗, <a>hi</a<a>hi ✗} … {<a> ✓, <a>hihi ✓} …
R4	$(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{}]_{\text{rep}})^*$	$(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*([\texttt{}]_{\text{alt}})^*)^*$ … $(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{</a}([\texttt{>}]_{\text{alt}})^*)^*$ ⋆ $(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{})^*$	{<a>hi ✗, <a>hi ✗} … {<a>hi</a ✗, <a>hi> ✗} ∅
R5	$(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{})^*$	⋆ $(\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}})^*\texttt{})^*$ $(\texttt{<a>}([\texttt{hi}]_{\text{rep}})^*\texttt{})^*$	{<a>h ✓, <a>i ✓} ∅
R6	$(\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}})^*\texttt{})^*$	⋆ $(\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{rep}})^*\texttt{})^*$	∅
R7	$(\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{rep}})^*\texttt{})^*$	⋆ $(\texttt{<a>}([\texttt{h}]_{\text{rep}} + \texttt{i})^*\texttt{})^*$	∅
R8	$(\texttt{<a>}([\texttt{h}]_{\text{rep}} + \texttt{i})^*\texttt{})^*$	⋆ $(\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$	∅
R9	$(\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$	–	–
C1	$\begin{pmatrix} A'_{\text{R1}} \to (\texttt{<a>}A'_{\text{R3}}\texttt{})^* \\ A_{\text{R3}} \to (\texttt{h} + \texttt{i})^* \end{pmatrix}, \{(A'_{\text{R1}}, A'_{\text{R3}})\}$	⋆ $\begin{pmatrix} A \to (\texttt{<a>}A\texttt{})^* \\ A \to (\texttt{h} + \texttt{i})^* \end{pmatrix}, \emptyset$ $\begin{pmatrix} A'_{\text{R1}} \to (\texttt{<a>}A'_{\text{R3}}\texttt{})^* \\ A_{\text{R3}} \to (\texttt{h} + \texttt{i})^* \end{pmatrix}, \emptyset$	{hihi ✓, <a><a>hi<a>hi ✓} ∅
C2	$\begin{pmatrix} A \to (\texttt{<a>}A\texttt{})^* \\ A \to (\texttt{h} + \texttt{i})^* \end{pmatrix}, \emptyset$	–	–

Figure 6.2: The generalization steps taken by our algorithm given seed input α_{XML} and oracle $\mathcal{O}_{\text{XML}}$. The initial language $\{\alpha_{\text{XML}}\}$ is generalized to a regular expression in steps R1-R9. The resulting regular expression is translated to a context-free grammar, which is further generalized in steps C1-C2. The candidates at each step are shown in order of preference, with the most preferable on top (ellipses indicate omitted candidates). Checks for each candidate are shown; a green check mark ✓ indicates that the check passes and a red cross ✗ indicates that it fails. A star ⋆ is shown next to the selected candidate.

Candidates. The ith generalization step first constructs *candidate* languages $\tilde{L}_1, ..., \tilde{L}_n$, with the goal of choosing $\hat{L}_{i+1}$ to be the candidate that increases recall the most without sacrificing precision. To ensure candidates can only increase recall, we consider *monotone* candidates $\tilde{L} \supseteq \hat{L}_i$. Furthermore, the candidates are ranked from most preferable ($\tilde{L}_1$) to least preferable ($\tilde{L}_n$). Figure 6.2 shows the candidates considered for our running example. They are listed in order of preference, with the top candidate being the most preferred. In steps R1-R9, the candidates add a single repetition or alternation to the current regular expression; in steps C1-C2, the candidates try to equate nonterminals in the current context-free grammar.

Checks. To ensure high precision, we want to avoid overgeneralizing. Ideally, we want to select a candidate that is *precision-preserving*, i.e., $\tilde{L} \setminus \hat{L}_i \subseteq L_*$. In other words, all strings added to the candidate $\tilde{L}$ (compared to the current language $\hat{L}_i$) are contained in the target language L_*. However, we only have access to a membership oracle for L_*, so it is typically impossible to prove that a given candidate $\tilde{L}$ is precision-preserving—we would have to check $\mathcal{O}(\alpha) = 1$ for every $\alpha \in \tilde{L} \setminus \hat{L}_i$, but this set is often infinite.

Instead, we carefully choose a finite number of heuristic *checks* $S \subseteq \tilde{L} \setminus \hat{L}_i$. Then, our algorithm rejects $\tilde{L}$ if $\mathcal{O}(\alpha) = 0$ for any $\alpha \in S$. Alternatively, if all checks pass (i.e., $\mathcal{O}(\alpha) = 1$), then $\tilde{L}$ is *potentially precision-preserving*. Since the candidates are ranked in order of preference, we choose the first potentially precision-preserving candidate. Figure 6.2 shows examples of checks our algorithm

constructs.

6.3 Phase One: Regular Expression Synthesis

We describe the first phase of generalization steps, which generalize the seed input into a regular expression.

6.3.1 Candidates

In phase one, the current language is represented by a regular expression annotated with extra data: substrings of terminals $\alpha = \sigma_1 ... \sigma_k$ may be enclosed in square brackets, i.e., $[\alpha]_\tau$, where $\tau \in \{\text{rep}, \text{alt}\}$. These annotations indicate that the bracketed substring in the current regular expression can be generalized by adding either a repetition (if $\tau = \text{rep}$) or an alternation (if $\tau = \text{alt}$). The seed input α_{in} is automatically annotated as $[\alpha_{\text{in}}]_{\text{rep}}$. Then, each generalization step selects a single bracketed substring $[\alpha]_\tau$ and generates candidates based on *decompositions* of α (i.e., an expression of α as a sequence of substrings $\alpha = \alpha_1 ... \alpha_k$):

- **Repetitions:** If generalizing $P[\alpha]_{\text{rep}}Q$, for each decomposition $\alpha = \alpha_1\alpha_2\alpha_3$ such that $\alpha_2 \neq \epsilon$, generate

$$P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q.$$

- **Alternations:** If generalizing $P[\alpha]_{\text{alt}}Q$, for each decomposition $\alpha = \alpha_1\alpha_2$, where $\alpha_1 \neq \epsilon$ and $\alpha_2 \neq \epsilon$, generate

$$P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q.$$

In both cases, the candidate $P\alpha Q$ is also generated. For example, in Figure 6.2, step R1 selects $[\texttt{<a>hi}]_{\text{rep}}$ and applies the repetition rule.

The candidates are monotonic:

Proposition 6.3.1 Each candidate constructed in phase one of our algorithm is monotone.

Proof: There are two cases:

- **Repetitions:** Every candidate has form (omitting bracketed substrings) $R' = P\alpha_1\alpha_2^*\alpha_3Q$, where the current language is $R = P\alpha Q$ and $\alpha = \alpha_1\alpha_2\alpha_3$. Since $\alpha \in \mathcal{L}(\alpha_1\alpha_2^*\alpha_3)$, it is clear that

$$\mathcal{L}(R) = \mathcal{L}(P\alpha Q) \subseteq \mathcal{L}(P\alpha_1\alpha_2^*\alpha_3 Q) = \mathcal{L}(R').$$

- **Alternations:** Every candidate has form (omitting bracketed substrings) $R' = P(\alpha_1 + \alpha_2)Q$, where the current language is $R = P\alpha Q$ and $\alpha = \alpha_1\alpha_2$. Note that an bracketed expression $[\alpha]_{\text{alt}}$ always occurs within a repetition, so the candidate has form

$$\begin{aligned} R' &= ...(... + (\alpha_1 + \alpha_2) + ...)^*... \\ &= ...(... + (\alpha_1 + \alpha_2)^* + ...)^*..., \end{aligned}$$

  so since $\alpha \in (\alpha_1 + \alpha_2)^*$, we have

$$\mathcal{L}(R) = \mathcal{L}(P\alpha Q) \subseteq \mathcal{L}(P(\alpha_1 + \alpha_2)^* Q) = \mathcal{L}(R').$$

The result follows. □

We briefly describe the intuition behind these rules. In particular, we define a *meta-grammar*[2] $\mathcal{C}_{\text{regex}}$, which is a context-free grammar whose members $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ are regular expressions. The terminals of $\mathcal{C}_{\text{regex}}$ are $\Sigma_{\text{regex}} = \Sigma \cup \{+, *\}$, where $+$ denotes alternations and $*$ denotes repetitions. The nonterminals are $\mathcal{V}_{\text{regex}} = \{T_{\text{rep}}, T_{\text{alt}}\}$, where T_{rep} corresponds to repetitions (and is also the start symbol) and T_{alt} corresponds to alternations. The productions are

$$\begin{aligned} T_{\text{rep}} &::= \beta \mid T_{\text{alt}}^* \mid \beta T_{\text{alt}}^* \mid T_{\text{alt}}^* T_{\text{rep}} \mid \beta T_{\text{alt}}^* T_{\text{rep}} \\ T_{\text{alt}} &::= T_{\text{rep}} \mid T_{\text{rep}} + T_{\text{alt}} \end{aligned}$$

where $\beta \in \Sigma^* - \{\epsilon\}$ ranges over nonempty substrings of α_{in}.

Consider the series of regular expressions $R_1 \Rightarrow ... \Rightarrow R_n$ in phase one. For each regular expression, we can replace each bracketed substring $[\alpha]_\tau$ with the nonterminal T_τ. Doing so produces a derivation in $\mathcal{C}_{\text{regex}}$, for example, steps R1-R9 in Figure 6.2 correspond to the derivation:

$[\texttt{<a>hi}]_{\text{rep}}$	T_{rep}
$\Rightarrow ([\texttt{<a>hi}]_{\text{alt}})^*$	$\Rightarrow T_{\text{alt}}^*$
$\Rightarrow ([\texttt{<a>hi}]_{\text{rep}})^*$	$\Rightarrow T_{\text{rep}}^*$
$\Rightarrow (\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{}]_{\text{rep}})^*$	$\Rightarrow (\texttt{<a>}T_{\text{alt}}^* T_{\text{rep}})^*$
$\Rightarrow (\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}T_{\text{alt}}^*\texttt{})^*$
$\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}(T_{\text{rep}} + T_{\text{alt}})^*\texttt{})^*$
$\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{rep}})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}(T_{\text{rep}} + T_{\text{rep}})^*\texttt{})^*$
$\Rightarrow (\texttt{<a>}([\texttt{h}]_{\text{rep}} + \texttt{i})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}(T_{\text{rep}} + \texttt{i})^*\texttt{})^*$
$\Rightarrow (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$	$\Rightarrow (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$

In fact, this correspondence goes backwards as well:

[2] We use the term *meta-grammar* to distinguish $\mathcal{C}_{\text{regex}}$ from the context-free grammars we synthesize.

Proposition 6.3.2 For any derivation $T_{\text{rep}} \overset{*}{\Rightarrow} R$ in $\mathcal{C}_{\text{regex}}$ (where $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$), there exists $\alpha_{\text{in}} \in \mathcal{L}(R)$ such that R can be derived from α_{in} via a series of generalization steps

$$\{\alpha_{\text{in}}\} = R_1 \Rightarrow ... \Rightarrow R_n = R$$

Proof: Consider the derivation of a regular expression $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$:

$$T_{\text{rep}} = \eta_1 \Rightarrow ... \Rightarrow \eta_n = R.$$

We prove that for each i, there is a series of generalization steps

$$R_i \Rightarrow R_{i+1} \Rightarrow ... \Rightarrow R_n = R$$

such that each R_j (for $i \leq j \leq n$) maps to η_j in the way defined in Section 6.3.1 (i.e., by replacing $[\alpha]_\tau$ with T_τ); we express this mapping as $\eta_j = \overline{R_j}$. The result follows since for $i = 1$, we get $[\alpha]_{\text{rep}} = R_1 \Rightarrow ... \Rightarrow R_n = R$, so we can take $\alpha_{\text{in}} = \alpha$.

We prove by (backward) induction on the derivation. The base case $i = n$ is trivial, since $\eta_n \in \mathcal{L}(\mathcal{C}_{\text{regex}})$, so we can take $R_n = \eta_n$ since $\overline{R_n} = R_n = \eta_n$. Now, suppose that we have a series of generalization steps $R_{i+1} \Rightarrow ... \Rightarrow R_n = R$ that satisfies the claimed property. It suffices to show that we can construct R_i such that $R_i \Rightarrow R_{i+1}$ is a generalization step and $\overline{R_i} = \eta_i$. Consider the following cases for the step $\eta_i \Rightarrow \eta_{i+1}$ in the derivation:

- Step $\mu T_{\text{rep}} \nu \Rightarrow \mu \beta T^*_{\text{alt}} T_{\text{rep}} \nu$: Then, we must have

  $$R_{i+1} = P\alpha_1[\alpha_2]_{\text{alt}}[\alpha_3]_{\text{rep}}Q,$$

  where $\overline{P} = \mu$, $\overline{Q} = \nu$, and $\alpha_1 = \beta$. Also, since R_{i+1} is valid, we have $\alpha_1, \alpha_2, \alpha_3 \neq \epsilon$. Therefore, we can take

  $$R_i = P[\alpha]_{\text{rep}}Q,$$

  where $\alpha = \alpha_1\alpha_2\alpha_3 \neq \epsilon$. The remaining productions for T_{rep} are similar. In particular, the assumption that $\beta \neq \epsilon$ in these derivations is needed to ensure that $\alpha \neq \epsilon$.

- Step $\mu T_{\text{alt}} \nu \Rightarrow \mu(T_{\text{rep}} + T_{\text{alt}})\nu$: Then, we must have

  $$R_{i+1} = P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q,$$

  where $\overline{P} = \mu$ and $\overline{Q} = \nu$. Also, since R_{i+1} is valid, we have $\alpha_1, \alpha_2 \neq \epsilon$. Therefore, we can take

  $$R_i = P[\alpha]_{\text{alt}}Q,$$

where $\alpha = \alpha_1\alpha_2 \neq \epsilon$. The remaining production for T_{alt} is similar.

The result follows. □

Furthermore, $\mathcal{L}(\mathcal{C}_{\text{regex}})$ almost contains every regular expression:

Proposition 6.3.3 For any regular language L_*, there exist $R_1, ..., R_m \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ such that $L_* = \mathcal{L}(R_1 + ... + R_m)$.

Proof: We slightly modify $\mathcal{C}_{\text{regex}}$, by introducing a new nonterminal T_{regex}, taking T_{regex} to be the start symbol, and adding productions

$$T_{\text{regex}} ::= \bar{\epsilon} \mid T_{\text{alt}} \mid \bar{\epsilon} + T_{\text{alt}},$$

where $\bar{\epsilon} \in \Sigma_{\text{regex}}$ is a newly introduced terminal denoting the regular expression for the empty language. This modification has two effects:

- Now, regular expressions $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ can have top-level alternations.
- Furthermore, the top-level alternation can explicitly include the empty string $\bar{\epsilon}$ (e.g., $R = \bar{\epsilon}+\texttt{a}$).

As described in Section 6.3.1, the first modification can be addressed by using multiple inputs (see Section 6.5.1), which allows our algorithm to learn top-level alternations. The second modification can be addressed by including a seed input $\bar{\epsilon} \in E_{\text{in}}$, in which case phase one of our algorithm synthesizes $\bar{\epsilon}$ (since there is nothing for it to generalize).

Now, let the context-free grammar $\tilde{C}_{\text{regex}}$ be a standard grammar for regular expressions:

$$T ::= \beta \mid TT \mid T + T \mid T^*. \tag{6.1}$$

It suffices to show that for any $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$, there exists $R' \in \mathcal{L}(\tilde{C}_{\text{regex}})$ such that $\mathcal{L}(R) = \mathcal{L}(R')$ (which we express as $R \equiv R'$).

First, we prove the result for $\mathcal{C}^{\epsilon}_{\text{regex}}$, which is identical to $\mathcal{C}_{\text{regex}}$ except that we allow $\beta = \epsilon$. Let $R \in \mathcal{L}(\tilde{C}_{\text{regex}})$. Suppose that either $R = S_1 + S_2$, $R = S_1S_2$, or $R = \beta$. We claim that we can express R as

$$\begin{aligned} R &\equiv X_1 + ... + X_n \\ X_i &= Y_{i,1}...Y_{i,k_i} \qquad (1 \leq i \leq n) \end{aligned} \tag{6.2}$$

where either $Y_{i,j} = \beta$ or $Y_{i,j} = W^*_{i,j}$ for each i and j. Consider two possibilities:

- Suppose R can be expressed in the form (6.2), but $Y_{i,j} = Z_1 + Z_2$. Then

$$\begin{aligned} X_i &= Y_{i,1}...Y_{i,j}...Y_{i,k_i} \\ &= Y_{i,1}...(Z_1 + Z_2)...Y_{i,k_i} \\ &\equiv Y_{i,1}...Z_1...Y_{i,k_i} + Y_{i,1}...Z_2...Y_{i,k_i} \end{aligned}$$

  which is again in the form (6.2).

- Suppose R has the form (6.2), but $Y_{i,j} = Z_1 Z_2$. Then

$$X_i = Y_{i,1}...Y_{i,j}...Y_{i,k_i} = Y_{i,1}...Z_1 Z_2...Y_{i,k_i}$$

  which is again in the form (6.2).

Note that either $R = S_1 + S_2$ or $R = S_1 S_2$, so R starts in the form (6.2). Therefore, we can repeatedly apply the above two transformations until $Y_{i,j} = \beta$ or $Y_{i,j} = W_{i,j}^*$ for every i and j. This process must terminate because the parse tree for R is finite, so the claim follows.

Now, we construct $R' \in \mathcal{L}(\mathcal{C}^{\epsilon}_{\text{regex}}, T_{\text{alt}})$ such that $R \equiv R'$ by structural induction. First, suppose that either $R = S_1 + S_2$, $R = S_1 S_2$, or $R = \beta$. Then we can express R in the form (6.2). By induction, $W_{i,j} \equiv W'_{i,j}$ for some $W'_{i,j} \in \mathcal{L}(\mathcal{C}^{\epsilon}_{\text{regex}}, T_{\text{alt}})$ for every i and j. By the definition of T_{rep}, we have $X_i \in \mathcal{L}(\mathcal{C}^{\epsilon}_{\text{regex}}, T_{\text{rep}})$, so by the definition of T_{alt}, we have $R \in \mathcal{L}(\mathcal{C}^{\epsilon}_{\text{regex}}, T_{\text{alt}})$, so the inductive step follows.

Alternatively, suppose $R = S^*$. If $S = S_1^*$, then $R \equiv S_1^*$, so without loss of generality assume $S = S_1 + S_2$, $S = S_1 S_2$, or $S = \beta$, so by the previous argument, we have $S \in \mathcal{L}(\mathcal{C}^{\epsilon}_{\text{regex}}, T_{\text{alt}})$. Since $T_{\text{alt}} ::= T_{\text{rep}}$ and $T_{\text{rep}} ::= T^*_{\text{alt}}$, we have $R \in \mathcal{L}(\mathcal{C}^{\epsilon}_{\text{regex}}, T_{\text{alt}})$, so again the inductive step follows. Finally, since $T ::= T_{\text{alt}}$, we have $R \in \mathcal{L}(\mathcal{C}^{\epsilon}_{\text{regex}})$.

Now, we modify the above proof to show that as long as $\epsilon \notin \mathcal{L}(R)$, we have $R \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{alt}})$. As before, we proceed by structural induction. Suppose that either $R = S_1 + S_2$, $R = S_1 S_2$, or $R = \beta$, so we can express R in the form (6.2). First, consider the case $Y_{i,j} = \beta$; if $\beta = \epsilon$, we can remove $Y_{i,j}$ from X_i unless $k_i = 1$. However, if $Y_{i,j} = \beta = \epsilon$ and $k_i = 1$, whence $X_i = \epsilon$ so $\epsilon \in \mathcal{L}(R)$, a contradiction; hence, we can always drop $Y_{i,j}$ such that $Y_{i,j} = \epsilon$. For the remaining $Y_{i,j} = \beta$, we have $Y_{i,j} \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{rep}})$ by the definition of $\mathcal{C}_{\text{regex}}$.

Second, consider the case $Y_{i,j} = Z^*_{i,j}$. Let $Z'_{i,j}$ be a regular expression such that $\mathcal{L}(Z'_{i,j}) = \mathcal{L}(Z_{i,j}) - \{\epsilon\}$, and note that

$$Y_{i,j} = Z^*_{i,j} \equiv (Z'_{i,j})^*.$$

By induction, we know that $Z_{i,j} \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{alt}})$, so $Y'_{i,j} = (Z'_{i,j})^* \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{rep}})$ by the definition of $\mathcal{C}_{\text{regex}}$.

For each X_i, we remove every $Y_{i,j} = \beta = \epsilon$ and replace every $Y_{i,j} = Z_{i,j}^*$ with $Y'_{i,j} = (Z'_{i,j})^*$ to produce $X'_i \equiv X_i$. By definition of $\mathcal{C}_{\text{regex}}$, we have $X_i \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{rep}})$, so $R \in \mathcal{L}(\mathcal{C}_{\text{regex}}, T_{\text{alt}})$ as claimed; now, the case $R = S^*$ follows by the same argument as before.

For any R such that $\epsilon \in \mathcal{L}(R)$, we can write $R = \epsilon + S$ where $\epsilon \notin \mathcal{L}(S)$ and apply the above argument to S. Since $T ::= \epsilon + T_{\text{alt}}$ is a production in $\mathcal{C}_{\text{regex}}$, we have shown that $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ for any regular expression R. □

This proposition says that phase one can synthesize almost any regular language L_*, assuming the "right" sequence of generalization steps is taken. Our extension to multiple inputs in Section 6.5.1 extends this result to any regular language. However, the space of all regular expressions is too large to search exhaustively. We sacrifice completeness for efficiency—our algorithm greedily chooses the first candidate according to the candidate ordering described in Section 6.3.2.

The productions in $\mathcal{C}_{\text{regex}}$ are unambiguous, so each regular expression $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ has a single valid parse tree. This disambiguation allows our algorithm to avoid considering candidate regular expressions multiple times.

6.3.2 Candidate Ordering

The candidate ordering is a heuristic designed to maximize the generality of the regular expression synthesized at the end of phase one. We use the following ordering for candidates constructed by phase one generalization steps:

- **Repetitions:** If generalizing $P[\alpha]_{\text{rep}}Q$, among

$$P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q,$$

  we first prioritize shorter α_1, since α_1 is not further generalized. Second, we prioritize longer α_2—for example, in step R3 of Figure 6.2, if we instead chose candidate `<a>`$([\texttt{h}]_{\text{alt}})^*[\texttt{i}\texttt{}]_{\text{rep}}$, then we would synthesize (`<a>h`*`i`*`</a>`)*, which is less general than step R9.

- **Alternations:** If generalizing $P[\alpha]_{\text{alt}}Q$, among

$$P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q,$$

  we prioritize shorter α_1—for example, in step R5 of Figure 6.2, if we instead chose candidate (`<a>`$([\texttt{hi}]_{\text{rep}})^*$`</a>`)*, then step R6 would instead be (`<a>`$([\texttt{hi}]_{\text{rep}})^*$`</a>`)*, which is less general than the one we obtain.

In either case, the final candidate $P\alpha Q$ is ranked last. Note that candidate repetitions and candidate alternations can be ordered independently—each generalization step considers only repetitions (if the chosen bracketed string has form $[\alpha]_{\text{rep}}$) or only alternations (if it has form $[\alpha]_{\text{alt}}$).

6.3.3 Check Construction

We describe how phase one of our algorithm constructs checks $S \subseteq \tilde{L} \setminus \hat{L}_i$. Each check $\alpha \in S$ has form $\alpha = \gamma\rho\delta$, where ρ is a *residual* capturing the portion of $\tilde{L}$ that is generalized compared to $\hat{L}_i$, and (γ, δ) is a *context* capturing the portion of $\tilde{L}$ which is in common with $\hat{L}_i$. More precisely, suppose the current language is $P[\alpha]_\tau Q$, where $[\alpha]_\tau$ is chosen to be generalized, and the candidate language is $PR_\alpha Q$, i.e., α is generalized to R_α. Then, a residual $\rho \in \mathcal{L}(R_\alpha) \setminus \{\alpha\}$ captures how R_α is generalized compared to the substring α, and a context (γ, δ) captures the semantics of the expressions (P, Q).

We may want to choose $\gamma \in \mathcal{L}(P)$ and $\delta \in \mathcal{L}(Q)$. However, P and Q may not be regular expressions. For example, on step R5 in Figure 6.2, $P =$ "(`<a>`", $\alpha =$ "`hi`", and $Q =$ "`</a>`)*" (the expressions are quoted to emphasize the placement of parentheses). Instead, P and Q form a regular expression when sequenced together, possibly with a string α' in between, i.e., $P\alpha'Q$. We want contexts (γ, δ) such that

$$\gamma\alpha'\delta \in \mathcal{L}(P\alpha'Q) \quad (\forall \alpha' \in \Sigma^*). \tag{6.3}$$

Then, the constructed check $\alpha = \gamma\rho\delta$ satisfies

$$\gamma\rho\delta \in \mathcal{L}(P\rho Q) \subseteq \mathcal{L}(PR_\alpha Q),$$

where the first inclusion follows from (6.3) and the second inclusion follows since $\rho \in \mathcal{L}(R_\alpha)$. We discard α such that $\alpha \in \mathcal{L}(\hat{L}_i)$ to obtain valid checks $\alpha \in \tilde{L} \setminus \hat{L}_i$.

Next, we explain the construction of residuals and contexts. Our algorithm generates residuals as follows:

- **Repetitions:** For current language $P[\alpha]_{\text{rep}}Q$ and candidate $P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q$, generate residuals $\alpha_1\alpha_3$ and $\alpha_1\alpha_2\alpha_2\alpha_3$.
- **Alternations:** For current language $P[\alpha]_{\text{alt}}Q$ and candidate $P(\alpha_1 + \alpha_2)Q$, generate residuals α_1 and α_2.

Next, our algorithm associates a context (γ, δ) with each bracketed string $[\alpha]_\tau$. The context for the initial bracketed string $[\alpha_{\text{in}}]_{\text{rep}}$ is (ϵ, ϵ). After each generalization step, contexts for new bracketed substrings are generated:

- **Repetitions:** For current language $P[\alpha]_{\text{rep}}Q$, where $[\alpha]_{\text{rep}}$ has context (γ, δ), and candidate $P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q$, the context generated for the new bracketed substring $[\alpha_2]_{\text{alt}}$ is $(\gamma\alpha_1, \alpha_3\delta)$, and for $[\alpha_3]_{\text{rep}}$ is $(\gamma\alpha_1\alpha_2, \delta)$.
- **Alternations:** For current language $P[\alpha]_{\text{alt}}Q$, where $[\alpha]_{\text{alt}}$ has context (γ, δ), and candidate $P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q$, the context generated for the new bracketed substring $[\alpha_1]_{\text{rep}}$ is $(\gamma, \alpha_2\delta)$,

and for $[\alpha_2]_{\text{alt}}$ is $(\gamma\alpha_1, \delta)$.

For example, on step R3, the context for $[\texttt{<a>hi}]_{\text{rep}}$ is (ϵ, ϵ). The residuals for candidate $(([\texttt{<a>hi</a}]_{\text{alt}})^*[\texttt{>}]_{\text{rep}})^*$ are `<a>hi</a` and `<a>hi</a>>`; since the context is empty, these residuals are also the checks, and they are rejected by the oracle, so the candidate is rejected. On the other hand, the residuals (and checks) for the chosen candidate $(\texttt{<a>}([\texttt{hi}]_{\text{alt}})^*[\texttt{}]_{\text{rep}})^*$ are `<a></a>` and `<a>hihi</a>`, which are accepted by the oracle. For the new bracketed string $[\texttt{hi}]_{\text{alt}}$, the algorithm constructs the context $(\texttt{<a>}, \texttt{})$, and for the new bracketed string $[\texttt{}]_{\text{rep}}$, the algorithm constructs the context $(\texttt{<a>hi}, \epsilon)$.

Similarly, on step R5, the context for $[\texttt{hi}]_{\text{alt}}$ is $(\texttt{<a>}, \texttt{})$. The residuals constructed for the chosen candidate $(\texttt{<a>}([\texttt{h}]_{\text{rep}} + [\texttt{i}]_{\text{alt}})^*\texttt{})^*$ are `h` and `i`, so the constructed checks are `<a>h</a>` and `<a>i</a>`. Our algorithm constructs the context $(\texttt{<a>}, \texttt{i})$ for the new bracketed string $[\texttt{h}]_{\text{rep}}$ and the context $(\texttt{<a>h}, \texttt{})$ for the new bracketed string $[\texttt{i}]_{\text{alt}}$.

We have the following result, which ensures that the constructed checks are valid (i.e., belong to $\tilde{L} \setminus \hat{L}_i$):

Proposition 6.3.4 The contexts constructed by phase one generalization steps satisfy (6.3).

Proof: We prove by induction. The initial context (ϵ, ϵ) for $[\alpha_{\text{in}}]_{\text{rep}}$ clearly satisfies (6.3). Next, assume that the context (γ, δ) for the current language satisfies (6.3). There are two cases:

- **Repetitions:** Suppose that the current language is $R = P[\alpha]_{\text{rep}}Q$ and the candidate is $R' = P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{alt}}$. Then, the context constructed for $[\alpha_2]_{\text{alt}}$ is $(\gamma', \delta') = (\gamma\alpha_1, \alpha_3\delta)$. Also, let $P' =$ "$P\alpha_1($" and $Q' =$ "$)^*\alpha_3 Q$", so $R' = P'\alpha_2 Q'$. Then, for any $\alpha' \in \Sigma^*$, we have

$$\begin{aligned}\gamma'\alpha'\delta' = \gamma\alpha_1\alpha'\alpha_3\delta &\in \mathcal{L}(P\alpha_1\alpha'\alpha_3 Q)\\ &\subseteq \mathcal{L}(P\alpha_1(\alpha')^*\alpha_3 Q)\\ &= \mathcal{L}(P'\alpha' Q'),\end{aligned}$$

  where the first inclusion follows by applying (6.3) to the context (γ, δ) with $\alpha_1\alpha'\alpha_3 \in \Sigma^*$. Therefore, the context (γ', δ') satisfies (6.3). Similarly, the context constructed for $[\alpha_3]_{\text{rep}}$ is $(\gamma', \delta') = (\gamma\alpha_1\alpha_2, \delta)$. Also, let $P' = P\alpha_1\alpha_2^*$ and $Q' = Q$, so $R' = P'\alpha_3 Q'$. Then, for any $\alpha' \in \Sigma^*$, we have

$$\begin{aligned}\gamma'\alpha'\delta' = \gamma\alpha_1\alpha_2\alpha' &\in \mathcal{L}(P\alpha_1\alpha_2\alpha' Q)\\ &\subseteq \mathcal{L}(P\alpha_1\alpha_2^*\alpha' Q)\\ &= \mathcal{L}(P'\alpha' Q'),\end{aligned}$$

  where the first inclusion follows by applying (6.3) to the context (γ, δ) with $\alpha_1\alpha_2\alpha' \in \Sigma^*$. Therefore, the context (γ', δ') satisfies (6.3).

Step	Chosen Generalization		Productions	Language $\mathcal{L}(\hat{C}, A_i)$
R1	$[\texttt{<a>hi}]^{\text{R1}}_{\text{rep}}$	$\Rightarrow ([\texttt{<a>hi}]^{\text{R2}}_{\text{alt}})^*$	$\{A_{\text{R1}} \to A'_{\text{R1}},\ A'_{\text{R1}} \to \epsilon + A'_{\text{R1}} A_{\text{R2}}\}$	$(\texttt{<a>(h + i)*})^*$
R2	$[\texttt{<a>hi}]^{\text{R2}}_{\text{alt}}$	$\Rightarrow [\texttt{<a>hi}]^{\text{R3}}_{\text{rep}}$	$\{A_{\text{R2}} \to A_{\text{R3}}\}$	$\texttt{<a>(h + i)*}$
R3	$[\texttt{<a>hi}]^{\text{R3}}_{\text{rep}}$	$\Rightarrow \texttt{<a>}([\texttt{hi}]^{\text{R5}}_{\text{alt}})^*[\texttt{}]^{\text{R4}}_{\text{rep}}$	$\{A_{\text{R3}} \to \texttt{<a>} A'_{\text{R3}} A_{\text{R4}},\ A'_{R3} \to \epsilon + A'_{R3} A_{R5}\}$	$\texttt{<a>(h + i)*}$
R4	$[\texttt{}]^{\text{R4}}_{\text{rep}}$	$\Rightarrow \texttt{}$	$\{A_{\text{R4}} \to \texttt{}\}$	$\texttt{<a>}$
R5	$[\texttt{hi}]^{\text{R5}}_{\text{alt}}$	$\Rightarrow [\texttt{h}]^{\text{R8}}_{\text{rep}} + [\texttt{i}]^{\text{R6}}_{\text{alt}}$	$\{A_{\text{R5}} \to A_{\text{R8}} + A_{\text{R6}}\}$	$\texttt{h} + \texttt{i}$
R6	$[\texttt{i}]^{\text{R6}}_{\text{alt}}$	$\Rightarrow [\texttt{i}]^{\text{R7}}_{\text{rep}}$	$\{A_{\text{R6}} \to A_{\text{R7}}\}$	$\texttt{i}$
R7	$[\texttt{i}]^{\text{R7}}_{\text{rep}}$	$\Rightarrow \texttt{i}$	$\{A_{\text{R7}} \to \texttt{i}\}$	$\texttt{i}$
R8	$[\texttt{h}]^{\text{R8}}_{\text{alt}}$	$\Rightarrow \texttt{h}$	$\{A_{\text{R8}} \to \texttt{h}\}$	$\texttt{h}$
R9	–		–	–

Figure 6.3: The productions added to $\hat{C}_{\text{XML}}$ corresponding to each generalization step are shown. The derivation shows the bracketed subexpression $[\alpha]^i_\tau$ (annotated with the step number i) selected to be generalized at step i, as well as the subexpression to which $[\alpha]^i_\tau$ is generalized. The language $\mathcal{L}(\hat{C}, A_i)$ (i.e., strings derivable from A_i) equals the subexpression in $\hat{R}$ that eventually replaces $[\alpha]^i_\tau$. As before, steps that select a candidate that strictly generalizes the language are bolded (in the first column).

- **Alterations:** Suppose that the current language is $R = P[\alpha]_{\text{alt}}Q$ and the candidate is $R' = P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q$. Then, the context constructed for $[\alpha_1]_{\text{rep}}$ is $(\gamma', \delta') = (\gamma, \alpha_2\delta)$. Also, let $P' =$ "$P($" and $Q' =$ "$+\alpha_2)Q$", so $R' = P'\alpha_2 Q'$. Then, for any $\alpha' \in \Sigma^*$, we have

$$\begin{aligned}\gamma'\alpha'\delta' = \gamma\alpha'\alpha_2\delta &\in \mathcal{L}(P\alpha'\alpha_2 Q)\\ &= \mathcal{L}(P(\alpha' + \alpha_2)^* Q)\\ &= \mathcal{L}(P(\alpha' + \alpha_2)Q)\\ &= \mathcal{L}(P'\alpha' Q'),\end{aligned}$$

  where the inclusion follows by applying (6.3) to the context (γ, δ) with $\alpha'\alpha_2 \in \Sigma^*$, and the equality on the third line follows as in the proof of Proposition 6.3.1.

The claim follows. □

6.3.4 Computational Complexity

Let n be the length of the seed input α_{in}. In phase one, our algorithm considers at most $O(n^2)$ repetition candidates (since each of the n^2 substrings of α_{in} is considered at most once), and $O(n^3)$ alternation candidates (since at most $O(n)$ alternation candidates are considered per discovered repetition). Examining each candidate takes constant time (assuming each query to $\mathcal{O}$ takes constant time), so the complexity of phase one is $O(n^3)$. In our evaluation, we show that our algorithm is quite scalable.

6.4 Phase Two: Recursive Properties

The second phase of generalization steps learn recursive properties of program input languages that cannot be represented using regular expressions. Consider the regular expression $(\texttt{<a>(h+i)*})^*$

obtained at the end of phase one in Figure 6.2, which can be written as $\hat{R}_{\text{XML}} = (\texttt{<a>}R_{\text{hi}}\texttt{})^*$, where $R_{\text{hi}} = (\texttt{h} + \texttt{i})^*$. Since every regular language is also context-free, we can begin by translating $\hat{R}_{\text{XML}}$ to the context-free grammar

$$\{A_{\text{XML}} \to (\texttt{<a>}A_{\text{hi}}\texttt{})^*,\ A_{\text{hi}} \to (\texttt{h} + \texttt{i})^*\}.$$

Then, we can equate the nonterminals A_{XML} and A_{hi} to obtain the context-free grammar $\hat{C}_{\text{XML}}$:

$$\{A \to (\texttt{<a>}A\texttt{})^*,\ A \to (\texttt{h} + \texttt{i})^*\},$$

which does not overgeneralize, since $\mathcal{L}(\hat{C}_{\text{XML}}) \subseteq \mathcal{L}(C_{\text{XML}})$. Furthermore, $\mathcal{L}(\hat{C}_{\text{XML}})$ is not regular, as it contains the language of matching tags <a> and .

In general, phase two of algorithm first translates the synthesized regular expression $\hat{R}$ into a context-free grammar $\hat{C}$. Then, each generalization step considers equating a pair (A, B) of nonterminals in $\hat{C}$, where A and B correspond to *repetition subexpressions* of $\hat{R}$, which are subexpressions R of $\hat{R}$ of the form $R = R_1^*$. The restriction to equating repetition subexpressions is empirically motivated—in practice, recursive constructs can typically also be repeated, e.g., in matching parentheses grammars, so constraining the search space reduces the potential for imprecision without sacrificing recall. In our example, A_{XML} corresponds to repetition subexpression $\hat{R}_{\text{XML}}$, and A_{hi} corresponds to repetition subexpression R_{hi}, so our algorithm considers equating A_{XML} and A_{hi}.

In the remainder of this section, we first describe how we translate regular expressions to context-free grammars, and then describe phase two candidates and checks.

6.4.1 Translating $\hat{R}$ to a Context-Free Grammar

Our algorithm translates the regular expression $\hat{R}$ to a context-free grammar $\hat{C} = (V, \Sigma, P, T)$ such that $\mathcal{L}(\hat{R}) = \mathcal{L}(\hat{C})$ and subexpressions in $\hat{R}$ correspond to nonterminals in $\hat{C}$. Intuitively, the translation follows the derivation of $\hat{R}$ in the meta-grammar $\mathcal{C}_{\text{regex}}$ (described in Section 6.3.1). First, the terminals in $\hat{C}$ are the program input alphabet Σ. Next, the nonterminals V of $\hat{C}$ correspond to generalization steps, additionally including an auxiliary nonterminal for steps that generalize repetition nodes:

$$V = \{A_i \mid \text{step } i\} \cup \{A'_i \mid \text{step } i \text{ generalizes } P[\alpha]_{\text{rep}}Q\}.$$

The start symbol is A_1. Finally, the productions are generated according to the following rules:

- **Repetition:** If step i generalizes current language $P[\alpha]_{\text{rep}}Q$ to $P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q$, we generate productions

$$A_i \to \alpha_1 A'_i A_k, \quad A'_i \to \epsilon + A'_i A_j,$$

where j is the step that generalizes $[\alpha_2]_{\text{alt}}$ and k is the step that generalizes $[\alpha_3]_{\text{rep}}$. Intuitively, these productions are equivalent to the "production" $A_i \to \alpha_1 A_j^* A_k$.

- **Alternation:** If step i generalizes $P[\alpha]_{\text{alt}}Q$ to $P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q$, we include production $A_i \to A_j + A_k$, where j is the step that generalizes $[\alpha_1]_{\text{rep}}$ and k is the step that generalizes $[\alpha_2]_{\text{alt}}$.

For example, Figure 6.3 shows the result of the translation algorithm applied to the generalization steps in the first phase of Figure 6.2 to produce a context-free grammar $\hat{C}_{\text{XML}}$ equivalent to $\hat{R}_{\text{XML}}$. Here, steps R1 and R3 handle the semantics of repetitions, step R5 handles the semantics of the alternation, steps R2 and R6 only affect brackets so they are identities, and steps R4, R7, and R8 are constant expressions. Furthermore, $\mathcal{L}(\hat{C}, A_i)$ is the language of strings matched by the subexpression that eventually replaces the bracketed substring $[\alpha]_\tau$ generalized on step i; this language is shown in the last column of Figure 6.3.

The auxiliary nonterminals A'_i correspond to repetition subexpressions in $\hat{R}$—if step i generalizes $[\alpha]_{\text{rep}}$ to $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$, then $\mathcal{L}(\hat{C}, A'_i) = \mathcal{L}(R^*)$, where R is the subexpression to which $[\alpha_2]_{\text{alt}}$ is eventually generalized. In our example, A'_{R1} corresponds to $\hat{R}_{\text{XML}} = (\texttt{<a>}(\texttt{h} + \texttt{i})^*\texttt{})^*$, and A'_{R3} corresponds to $R_{\text{hi}} = (\texttt{h} + \texttt{i})^*$.

For conciseness, we redefine $\hat{C}_{\text{XML}}$ to be the equivalent context-free grammar with start symbol A'_{R1} and productions

$$A'_{\text{R1}} \to (\texttt{<a>} A'_{\text{R3}} \texttt{})^*, \quad A'_{\text{R3}} \to (\texttt{h} + \texttt{i})^*$$

where the Kleene star implicitly expands to the productions described in the repetition case.

6.4.2 Candidates and Ordering

The candidates considered in phase two of our algorithm are *merges*, which are (unordered) pairs of nonterminals (A'_i, A'_j) in $\hat{C}$, where i and j are generalization steps of phase one. Recall that these nonterminals correspond to repetition subexpressions in $\hat{R}$. In particular, associated to $\hat{C}$ is the set M of all such pairs of nonterminals. In Figure 6.2, the regular expression $\hat{R}_{\text{XML}}$ on step R9 is translated into the context-free grammar $\hat{C}_{\text{XML}}$ on step C1, with its corresponding set of merges M_{XML} containing just $(A'_{\text{R1}}, A'_{\text{R3}})$.

Each phase two generalization step selects a pair $(A'_i, A'_j) \in M$ and considers two candidates (in order of preference):

- The first candidate $\tilde{C}$ equates A'_i and A'_j by introducing a fresh nonterminal A and replacing all occurrences of A'_i and A'_j in $\hat{C}$ with A.
- The second candidate equals the current language $\hat{C}$.

In either case, the selected pair is removed from M. The candidates are monotone since equating two nonterminals can only enlarge the generated language.

For example, in step C1 of Figure 6.2, the candidate $(A'_{\text{R1}}, A'_{\text{R3}})$ is removed from M_{XML}; the first candidate is constructed by equating A'_{R1} and A'_{R3} in $\hat{C}_{\text{XML}}$ to obtain

$$\tilde{C}_{\text{XML}} = \{A \to (\texttt{<a>}A\texttt{})^*,\ A \to (\texttt{h} + \texttt{i})^*\},$$

where $\mathcal{L}(\tilde{C}_{\text{XML}})$ is not regular. The chosen candidate is $\hat{C}'_{\text{XML}} = \tilde{C}_{\text{XML}}$, since the checks (described in Section 6.4.3) pass. On step C2, M is empty, so our algorithm returns $\hat{C}'_{\text{XML}}$. In particular, $\hat{C}'_{\text{XML}}$ equals $\mathcal{L}(C_{\text{XML}})$, except the characters $\texttt{a} + ... + \texttt{z}$ are restricted to $\texttt{h} + \texttt{i}$. In Section 6.5.2, we describe an extension that generalizes characters in $\hat{C}'_{\text{XML}}$.

Finally, we formalize the intuition that equating $(A'_i, A'_j) \in M$ corresponds to merging repetition subexpressions, which says that equating $(A'_i, A'_j) \in M$ merges R and R' in $\hat{R}$:

Proposition 6.4.1 Let regular expression $\hat{R}$ translate to context-free grammar $\hat{C}$. Suppose that nonterminal A_i in $\hat{C}$ corresponds to repetition subexpression R, so $\hat{R} = PRQ$, and A_j to R', so $\hat{R} = P'R'Q'$. Let $\tilde{C}$ be obtained by equating A_i and A_j in $\hat{C}$. Then, $\mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C})$ (and symmetrically, $\mathcal{L}(P'RQ') \subseteq \mathcal{L}(\tilde{C})$).

Proof: (sketch) We show that if we merge two nonterminals $(A'_i, A'_j) \in M$ by equating them in the context-free grammar $\hat{C}$ (translated from $\hat{R}$) to obtain $\tilde{C}$, then the repetition subexpressions R in $\hat{R} = PRQ$ (corresponding to A'_i) and R' in $\hat{R} = P'R'Q'$ (corresponding to A'_j) are merged; i.e., $\mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C})$ and $\mathcal{L}(P'RQ') \subseteq \mathcal{L}(\tilde{C})$. While we prove the result for the translation $\hat{C}$ of $\hat{R}$, note that (i) subsequent merges can only enlarge the generated language, and (ii) the order in which merges are performed does not affect the final context-free grammar, so the result holds for any step of phase two of our algorithm.

Note that equating two nonterminals $(A'_i, A'_j) \in M$ in $\hat{C}$ is equivalent to adding productions $A'_i \to A'_j$ and $A'_j \to A'_i$ to $\hat{C}$. Therefore, Proposition 6.4.1 shows that both $\mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C})$ and $\mathcal{L}(P'RQ') \subseteq \mathcal{L}(\tilde{C})$. It suffices to show that adding $A'_i \to A'_j$ to $\hat{C}$ results in the context-free grammar $\tilde{C}$ satisfying $\mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C})$ (intuitively, this is a one-sided merge that only merges $\hat{R}'$ into $\hat{R}$, not vice versa).

We use the fact that our algorithm for translating a regular expression to a context-free grammars works more generally for any regular expression $R \in \mathcal{L}(\mathcal{C}_{\text{regex}})$ derived from T_{rep} in according to the meta-grammar $\mathcal{C}_{\text{regex}}$. In particular, if we consider the series of generalization steps

$$\alpha_{\text{in}} = R_1 \Rightarrow ... \Rightarrow R_n = \hat{R},$$

we get a corresponding derivation

$$T^{(1)}_{\text{rep}} = \eta_1 \Rightarrow ... \Rightarrow \eta_n = \hat{R}$$

in $\mathcal{C}_{\text{regex}}$ as described in Section 6.3.1. Similarly to the labels on bracketed strings in the series of generalization steps, we label each nonterminal in the derivation with the index at which it is expanded. For example, for the derivation corresponding to the the series of generalization steps in Figure 6.3 is

$$\begin{aligned}
&T^{(1)}_{\text{rep}} \\
&\Rightarrow (T^{(2)}_{\text{alt}})^* \\
&\Rightarrow (T^{(3)}_{\text{rep}})^* \\
&\Rightarrow (\texttt{<a>}(T^{(5)}_{\text{alt}})^* T^{(4)}_{\text{rep}})^* \\
&\Rightarrow (\texttt{<a>}(T^{(5)}_{\text{alt}})^* \texttt{})^* \\
&\Rightarrow (\texttt{<a>}(T^{(8)}_{\text{rep}} + T^{(6)}_{\text{alt}})^* \texttt{})^* \\
&\Rightarrow (\texttt{<a>}(T^{(8)}_{\text{rep}} + T^{(7)}_{\text{rep}})^* \texttt{})^* \\
&\Rightarrow (\texttt{<a>}(T^{(8)}_{\text{rep}} + \texttt{i})^* \texttt{})^* \\
&\Rightarrow (\texttt{<a>}(\texttt{h} + \texttt{i})^* \texttt{})^*
\end{aligned}$$

Now, each nonterminal A_i is associated to step i in the derivation, and we add productions for A_i depending on step i in the derivation (and auxiliary nonterminals A'_i if step i in the derivation expands nonterminal T_{rep} in the meta-grammar):

- Step $\mu T^{(i)}_{\text{rep}} \nu \Rightarrow \mu\beta(T^{(j)}_{\text{alt}})^* T^{(k)}_{\text{rep}}\nu$: We add productions $A_i \rightarrow \beta A'_i A_k$ and $A'_i \rightarrow \epsilon \mid A'_i A_j$.
- Step $\mu T^{(i)}_{\text{alt}} \nu \Rightarrow \mu(T^{(j)}_{\text{rep}} + T^{(k)}_{\text{alt}})\nu$: We add production $A_i \rightarrow A_j \mid A_k$.

Now, consider step i in the derivation, where productions for A_i and A'_i were added to $\hat{C}$. Then, step i of the derivation has form

$$\mu T^{(i)}_{\text{rep}} \nu \Rightarrow \mu\beta(T^{(j)}_{\text{alt}})^* T^{(k)}_{\text{rep}}\nu.$$

We can assume without loss of generality that we expand $T^{(i)}_{\text{rep}}$ last; i.e., $\mu = \overline{\mu} = P$ and $\nu = \overline{\nu} = Q$

do not contain any nonterminals. Therefore, the derivation has form

$$\begin{aligned}(\eta_1 = T^{(1)}_{\text{rep}}) &\Rightarrow ... \\ &\Rightarrow (\eta_i = PT^{(i)}_{\text{rep}}Q) \\ &\Rightarrow (\eta_{i+1} = P\beta(T^{(j)}_{\text{alt}})^* T^{(k)}_{\text{rep}}Q) \\ &\Rightarrow ... \\ &\Rightarrow (\eta_n = PRQ).\end{aligned}$$

Now, note that the following derivation is also in $\mathcal{C}_{\text{regex}}$:

$$\begin{aligned}(\eta_1 = T^{(1)}_{\text{rep}}) &\Rightarrow ... \\ &\Rightarrow (\eta_i = PT^{(i)}_{\text{rep}}Q) \\ &\Rightarrow (\eta'_{i+1} = P\beta'(T^{(j')}_{\text{alt}})^* T^{(k')}_{\text{rep}}Q) \\ &\Rightarrow ... \\ &\Rightarrow \eta'_{n'} = PR'Q\end{aligned}$$

since R' can be derived from T_{rep}. Note that $\hat{R}' = PR'Q$ is exactly the regular expression produced by this derivation. Then, let $\hat{C}'$ be the context-free grammar obtained by applying our translation algorithm to $\hat{R}'$ using this derivation.

Note that $\hat{C}'$ has the same productions as $\hat{C}$, except the productions for A_i in $\hat{C}$ (i.e., all productions added on step i of the derivation and after) have been replaced with productions A_i in $\hat{C}'$ such that $\mathcal{L}(\hat{C}', A_i) = \mathcal{L}(R')$. Since $\mathcal{L}(R') \subseteq \mathcal{L}(\tilde{C}, A_i)$, and the nonterminals involved in the productions for A_i do not occur in $\tilde{C}$, it is clear that adding the productions for A_i in $\hat{C}'$ to $\tilde{C}$ does not modify $\mathcal{L}(\tilde{C})$. By construction, the other productions in $\hat{C}'$ are in $\hat{C}$, so they are also in $\tilde{C}$. Therefore, $\mathcal{L}(\hat{C}') \subseteq \mathcal{L}(\tilde{C})$. The result follows, since $\mathcal{L}(\hat{C}') = \mathcal{L}(\hat{R}') = \mathcal{L}(PR'Q)$. □

6.4.3 Check Construction

Consider the candidate $\tilde{C}$ obtained by merging $(A'_i, A'_j) \in M$ in the current language $\hat{C}$, where A'_i corresponds to repetition subexpression R and A'_j to R'. Suppose that step i generalizes $P[\alpha]_{\text{rep}}Q$ to $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$, and step j generalizes $[\alpha']_{\text{rep}}$ to $\alpha'_1([\alpha'_2]_{\text{alt}})^*[\alpha'_3]_{\text{rep}}$. Note that $([\alpha_2]_{\text{alt}})^*$ is eventually generalized to the repetition subexpression R in $\hat{R}$, and $([\alpha'_2]_{\text{alt}})^*$ is eventually generalized to R' in $\hat{R}$.

Our algorithm constructs the check $\gamma\rho'\delta$, where $\rho' = \alpha'\alpha' \in \mathcal{L}(R')$ is a residual for R', and (γ, δ) is the context for $([\alpha_2]_{\text{alt}})^*$. This check satisfies

$$\gamma\rho'\delta \in \mathcal{L}(PR'Q) \subseteq \mathcal{L}(\tilde{C}),$$

where the first inclusion follows by the property (6.3) for contexts described in Section 6.3.3, and the second inclusion follows from Proposition 6.4.1. A similar argument to Proposition 6.3.4 shows that this context satisfies property (6.3).

The check $\gamma\rho'\delta$ tries to ensure that R' can be substituted for R without overgeneralizing, i.e., $\mathcal{L}(PR'Q) \subseteq L_*$. Our algorithm similarly generates a second check trying to ensure that R can be substituted for R', i.e., $\mathcal{L}(P'RQ) \subseteq L_*$.

For example, in Figure 6.2, the context for the repetition subexpression $\hat{R}_{\text{XML}}$ = (`<a>`(`h` + `i`)*`</a>`)* is (ϵ, ϵ), and the residual for R_{hi} is `hihi`, so the constructed check is `hihi`. Similarly, the context for R_{hi} is (`<a>`, `</a>`) and the residual for $\hat{R}_{\text{XML}}$ is `<a>hi</a><a>hi</a>`, so the constructed check is `<a><a>hi</a><a>hi</a></a>`.

6.4.4 Learning Matching Parentheses Grammars

To demonstrate the expressive power of merges, we show that they can represent the following class of generalized matching parentheses grammars:

Definition 6.4.2 A *generalized matching parentheses grammar* is a context-free grammar $C = (V, \Sigma, P, S_1)$, with

$$V = \{S_1, ..., S_n, R_1, ..., R_n, R'_1, ..., R'_n\}$$

and productions

$$S_i \to (R_i(S_{i_1} + ... + S_{i_{k_i}})^* R'_i)^*,$$

where for $1 \leq i \leq n$, R_i, R'_i are regular expressions over Σ.

In other words, R_i and R'_i are pairs of matching parentheses, except that they are allowed to be regular expressions, e.g., XML tags. They may also match the empty string ϵ, e.g., to permit unmatched open parentheses. Then, the valid matched parentheses strings matched by the grammars $S_{i_1}, ..., S_{i_{k_i}}$ can occur between R_i and R'_i. In particular, the XML-like grammar shown in Figure 6.1 is a generalized matching parentheses grammar, where the "parentheses" are `<a>` and `</a>`. We have the following result, which says that phase two of our algorithm at least allows us to learn the common class of generalized matching parentheses grammars:

Proposition 6.4.3 For any generalized matching parentheses grammar C, there exists a regular expression R and merges M over R such that letting C' be the grammar obtained by transforming R into a context-free grammar and performing the merges in M, we have $\mathcal{L}(C) = \mathcal{L}(C')$.

Proof: (sketch) Let C be a generalized matching parentheses grammar. Suppose that non-terminal S_i $(1 \leq i \leq n)$ corresponds to production

$$S_i \rightarrow R_i(S_{i_1} + ... + S_{i_{k_i}})^* R'_i.$$

First, we need to identify a context such that S_i can occur in a derivation in C; in particular, we want to construct a derivation of the form

$$\begin{aligned} S_0 = S_{i,1} &\Rightarrow R_{i,1} S_{i,2} R'_{i,1} \\ &\Rightarrow R_{i,1} R_{i,2} S_{i,3} R'_{i,2} R'_{i,1} \\ &\Rightarrow ... \\ &\Rightarrow R_{i,1} ... R_{i,h_i} S_i R'_{i,h_i} ... R'_{i,1}. \end{aligned}$$

To do so, we construct a directed graph G with vertices $\{S_1, ..., S_n\}$ and edges $S_i \rightarrow S_j$ whenever the production for S_i has form

$$S_i \rightarrow R_i(... + S_j + ...)^* R'_i.$$

In other words, an edge indicates that S_j is contained in a derivation of S_i. Then, we can constructed the desired derivation using a spanning tree rooted at S_1, in particular, by examining the path

$$S_1 = S_{i,1} \rightarrow ... \rightarrow S_{h_i} \rightarrow S_i$$

from S_1 to S_i in this spanning tree. Note that if no path exists, then S_i cannot occur in any derivation of S_1.

Now, for each pair of regular expressions R_i and R'_i $(1 \leq i \leq n)$, let $\alpha_i \in \mathcal{L}(R_i R'_i) \subseteq \mathcal{L}(C, S_i)$. Then, let

$$\begin{aligned} X_i &= R_{i,1} ... R_{i,h_i} Y_i R'_{i,h_i} ... R'_{i,1} \\ Y_i &= (R_i(\alpha^*_{i_1} + ... + \alpha^*_{i_{k_i}}) R'_i)^*. \end{aligned}$$

Intuitively, X_i is constructed according to the derivation of S_1 containing S_i, and Y_i is constructed using the production for S_i. In paricular, by construction, $\mathcal{L}(X_i) \subseteq \mathcal{L}(C)$.

Consider the following regular expression:

$$\begin{aligned} X &= X_1 + ... + X_n \\ M &= \{(Y_i, \alpha^*_{j_k}) \mid i = j_k\}. \end{aligned}$$

We claim that translating X and M into a context-free grammar yields a grammar C' such that $\mathcal{L}(C) = \mathcal{L}(C')$. First, we show that each production in C is also in C', which implies that $\mathcal{L}(C) \subseteq \mathcal{L}(C')$. In particular, note that the translation algorithm introduces exactly one nonterminal for each Y_i, since two repetition nodes Y_i and Y_j are never merged together, and every other repetition node in X is merged with a Y_i node. Let S'_i be the nonterminal introduced for Y_i; since each α_{i_j} is merged with Y_{i_j}, the production added to C' is

$$S'_i \rightarrow (R_i(S'_{i_1} + ... + S'_{i_{k_i}})^* R'_i)^*,$$

which is equivalent to the production for S_i in C.

Next, we show that $\mathcal{L}(X) \subseteq \mathcal{L}(C)$. First, note that by construction, $\mathcal{L}(X_i) \subseteq \mathcal{L}(C)$ for each $1 \leq i \leq n$, so $\mathcal{L}(X) \subseteq \mathcal{L}(C)$. Second, applying each merge in M does not affect this invariant, since Y_i and $\alpha^*_{j_k}$ can both be replaced with $S_i = S_{j_k}$. Therefore, $\mathcal{L}(C) = \mathcal{L}(C')$. □

6.4.5 Computational Complexity

The complexity of phase two is $O(n^4)$, where n is the length of the seed input α_{in}, since each pair of repetition subexpressions is a merge candidate, and as shown in Section 6.3.4, there are at most $O(n^2)$ repetition candidates. Therefore, the overall complexity is $O(n^4)$.

6.5 Extensions

In this section, we discuss two extensions to our algorithm.

6.5.1 Multiple Seed Inputs

Given multiple seed inputs $E_{\text{in}} = \{\alpha_1, ..., \alpha_n\}$, our algorithm first applies phase one separately to each α_i to synthesize a corresponding regular expression $\hat{R}_i$. Then, it combines these into a single regular expression $\hat{R} = \hat{R}_1 + ... + \hat{R}_n$ and applies phase two to $\hat{R}$. Repetition subexpressions in different components $\hat{R}_i$ of $\hat{R}$ may be merged. A useful optimization is to construct $\hat{R}$ incrementally—if we have $\alpha_i \in \mathcal{L}(\hat{R}_1 + ... + \hat{R}_{i-1})$, then α_i can be skipped.

6.5.2 Character Generalization

After phase one, we include a *character generalization* phase that generalizes terminals in the synthesized regular expression $\hat{R}$. At each generalization step, the algorithm selects a terminal string $\alpha = \sigma_1...\sigma_k$ in $\hat{R}$, i.e., $\hat{R} = P\alpha Q$, and a terminal σ_i in α, and a different terminal $\sigma \in \Sigma$ such that $\sigma \neq \sigma_i$, and considers two candidates. First, $\tilde{R} = P\sigma_1...\sigma_{i-1}(\sigma + \sigma_i)\sigma_{i+1}...\sigma_k Q$ replaces σ_i with

$(\sigma_i + \sigma)$. Second, the current language $\hat{R}$. Each such generalization is considered exactly once in this phase.

For the first candidate, we construct residual $\rho = \sigma$. Every terminal string α in $\hat{R}$ was added by generalizing $[\alpha']_{\text{rep}}$ to $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$, where $\alpha = \alpha_1$. Supposing that the context for $[\alpha']_{\text{rep}}$ is (γ, δ), we construct context $(\gamma\sigma_1...\sigma_{i-1}, \sigma_{i+1}...\sigma_k\alpha_3\delta)$. The generated checks are $\gamma\rho\delta$.

For example, in the regular expression $\hat{R}_{\text{XML}}$ output by phase one in Figure 6.2, our algorithm considers generalizing each terminal in `<a>`, `h`, `i`, and `</a>` to every (different) terminal $\sigma \in \Sigma$. Generalizing `<` to `a` is ruled out by the check `aa>hi</a>`. Alternatively, `h` is generalized to `a` since the generated checks `<a>ai</a>` and `<a>a</a>` pass. Eventually, $\hat{R}_{\text{XML}}$ generalizes to

$$\hat{R}'_{\text{XML}} = (\texttt{<a>}((\texttt{a} + ... + \texttt{z}) + (\texttt{a} + ... + \texttt{z}))^*\texttt{})^*,$$

which phase two generalizes to the grammar $\hat{C}'_{\text{XML}}$:

$$\left\{ \begin{array}{l} A \rightarrow (\texttt{<a>}A\texttt{})^*, \\ A \rightarrow ((\texttt{a} + ... + \texttt{z}) + (\texttt{a} + ... + \texttt{z}))^* \end{array} \right\}.$$

In particular, $\mathcal{L}(\hat{C}'_{\text{XML}}) = \mathcal{L}(C_{\text{XML}})$.

6.6 Discussion

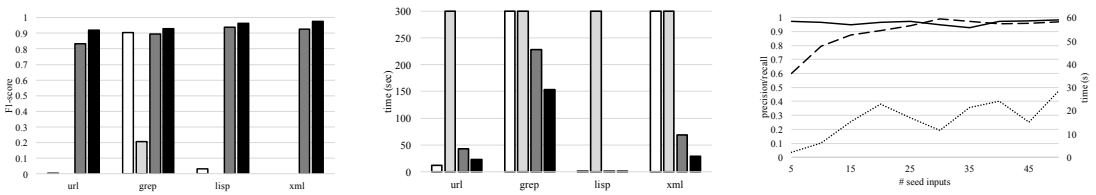
Phases of GLADE**.** We have described GLADE as proceeding in three phases, but the distinction is primarily for purposes of clarity. More precisely, the character generalization phase can equivalently be performed at any time. Phase two (the merging phase) depends on phase one to identify candidate repetition subexpressions to merge, but these phases could be interleaved if desired.

Limitations. The greedy search strategy is necessary for GLADE to efficiently search the space of languages. However, the cost of greediness is that suboptimal grammars may be synthesized (i.e., only generating a subset of the target language), even if all selected candidates are precise. For example, consider extending the XML grammar shown in Figure 6.1 with the production

$$A_{\text{XML}} \rightarrow \texttt{<a/>}.$$

Given the seed input

$$\alpha_{\text{in}} = \texttt{<a><a/>},$$



(a) (b) (c)

Figure 6.4: We show (a) the F_1 score, and (b) the running time of L-Star (white), RPNI (light grey), GLADE omitting phase two (dark grey), and GLADE (black) for each of the four test grammars C. The algorithms are trained on 50 random samples from the target language $L_* = \mathcal{L}(C)$. In (c), for the XML grammar, we show how the precision (solid line), recall (dashed line), and running time (dotted line) of GLADE vary with the number of seed inputs $|E_{\text{in}}|$ (between 0 and 50). The y-axis for precision and recall is on the left-hand side, whereas the y-axis for the running time (in seconds) is on the right-hand side.

phase one of GLADE synthesizes the regular expression

$$(\texttt{<a>}(\texttt{><a/>})^*\texttt{>})^*,$$

which is a valid subset of L_{XML}. However, in phase two of GLADE, the two repetition nodes

$$(\texttt{><a/>})^* \quad \text{and} \quad (\texttt{<a>}(\texttt{><a/>})^*\texttt{>})^*$$

cannot be merged, since the check `><a/>` is invalid. Ideally, GLADE would instead synthesize the regular expression

$$(\texttt{<a>}(\texttt{<a/>})^*\texttt{})^*,$$

in phase one, in which case the two repetition nodes

$$(\texttt{<a/>})^* \quad \text{and} \quad (\texttt{<a>}(\texttt{<a/>})^*\texttt{})^*$$

are successfully merged in phase two. GLADE fails to do so because of the greedy nature of phase one. If GLADE is instead provided with the seed inputs

$$\{\texttt{<a/>}, \texttt{<a>hi}\},$$

then it would successfully recover the target language.

Intuitively, the greedy strategy employed by GLADE works best when the target language has fewer nondeterministic constructs (as is the case with many program input languages in practice, e.g., to ensure efficient parsing). Such grammars are less likely to have multiple incompatible candidates

at each generalization step, ensuring that GLADE rarely makes suboptimal choices.

6.7 Evaluation

We implement our grammar synthesis algorithm in a tool called GLADE, which synthesizes a context-free grammar $\hat{C}$ given an oracle $\mathcal{O}$ and seed inputs $E_{\text{in}} \subseteq L_*$. In our first experiment, we compare GLADE to widely studied language inference algorithms, and in our second experiment, we evaluate the ability of GLADE to learn useful approximations of real program input grammars for a fuzzing client. We note that the only grammar used to guide the design our algorithm is the XML grammar, and no other grammar was used for this purpose. GLADE is implemented in Java, and all experiments are run on a 2.5 GHz Intel Core i7 CPU.

6.7.1 Sampling Context-Free Grammars

We describe how we randomly sample a string α from a context-free grammar C. The ability to sample implicitly defines a probability distribution $\mathcal{P}_{\mathcal{L}(C)}$ over $\mathcal{L}(C)$, which we use to measure precision and recall as in Definition 6.1.1. We also use random samples in our grammar-based fuzzer in Section 6.7.3. To describe our approach, we more generally describe how to sample $\alpha \sim \mathcal{P}_{\mathcal{L}(C,A)}$ (which is the language of strings that can be derived from nonterminal A using productions in C). To do so, we convert the context-free grammar $C = (V, \Sigma, P, S)$ to a *probabilistic context-free grammar*. For each nonterminal $A \in V$, we construct a discrete distribution $\mathcal{D}_A$ of size $|P_A|$ (where $P_A \subseteq P$ is the set of productions in C for A). Then, we randomly sample $\alpha \sim \mathcal{P}_{\mathcal{L}(C,A)}$ as follows:

- Randomly sample production $(A \to A_1 ... A_k) \sim \mathcal{D}_A$.
- If A_i is a nonterminal, recursively sample $\alpha_i \sim \mathcal{P}_{\mathcal{L}(C,A_i)}$; otherwise, if A_i is a terminal, let $\alpha_i = A_i$.
- Return $\alpha = \alpha_1 ... \alpha_k$.

For simplicity, we choose $\mathcal{D}_A$ to be uniform.

6.7.2 Comparison to Language Inference

In our first experiment, we show that GLADE can synthesize simple input grammars with much better precision and recall compared to two widely studied language inference algorithms, L-Star [12] and RPNI [110], both implemented using `libalf` [24]. We also compare to a variant of GLADE with phase two omitted, which restricts GLADE to learning regular languages, which shows that the benefit of GLADE is not just its ability to synthesize non-regular properties.

Grammar	Target Language L_*	Synthesized Grammar $\hat{L}$
URL	$A \rightarrow$ `http`$(\epsilon + $ `s`$)$`://`$(\epsilon + $ `www.`$)[\ldots]^*.[\ldots]^*$	$A \rightarrow$ `http://`$B^*.C^* +$ `https://`$B^*.C^*$ $+$`http://www.`$B^*.C^* +$ `https://www.`$B^*.C^*$ $B \rightarrow [\ldots]^*$ $C \rightarrow [\ldots]^*$
Grep	$A \rightarrow ([\ldots] +$ `\(`A`\)`$)^*$	$A \rightarrow ([\ldots]^* + ($`\(`$(A^*)^*$`\)`$)^*)^*$
Lisp	$A \rightarrow$ `(`$[\ldots][\ldots]^*(\_\_^*([\ldots][\ldots]^* + A))^*$`)`	$A \rightarrow$ `(`$([\ldots]^*[\ldots]((\_^*\_A)^*\_^*\_)^*)^*[\ldots]^*[\ldots]$`)`
XML	$A \rightarrow$ `<a`$(\_\_^*[\ldots][\ldots]^*$`="`$[\ldots]^*$`")`*`>`$(A + [\ldots])^*$`</a>`	$A \rightarrow$ `<a`$(\_^*\_[\ldots]^*[\ldots]^*$`="`$[\ldots]^*$`")`$^*B^*$`>`$[\ldots]^*$`</a>` $B \rightarrow$ `>`$[\ldots]^*$`<a`$(\_^*\_[\ldots]^*[\ldots]^*$`="`$[\ldots]^*$`")`$^*B^*$`>`$[\ldots]^*$`</a` $+$`>`$[\ldots]^*$`<a>`$[\ldots]^*$`</a`

Figure 6.5: Examples of context-free grammars that are synthesized by GLADE for the given target languages. The symbol ␣ denotes a space. For clarity, character ranges with large numbers of characters are denoted by [...].

Grammars. We manually wrote four grammars encoding valid inputs for various programs:

- A regular expression for matching URLs [138].
- A grammar for the regular expression accepted as input by GNU Grep [60]
- A grammar for a simple Lisp parser [109], including support for quoted strings and comments.
- A grammar for XML parsers [150], including all XML constructs (attributes, comments, CDATA sections, etc.), except that only a fixed number of tags are included (to ensure that the grammar is context-free).

Methods. For each grammar C, we sampled 50 seed inputs $E_{\text{in}} \subseteq L_* = \mathcal{L}(C)$ using the technique in Section 6.7.1, and implemented an oracle $\mathcal{O}$ for L_*. Then, we use each algorithm to learn L_* from E_{in} and $\mathcal{O}$. Since the algorithms sometimes cannot scale to all 50 inputs, we incrementally give the seed inputs to the algorithms until they time out (after 300 seconds), and use the last language successfully learned without timing out.

L-Star. Angluin's L-Star algorithm learns a regular language $\hat{R}$ approximating the target language L_*. It takes as input a membership oracle and an *equivalence oracle* $\mathcal{O}_E$; given a candidate regular language $\hat{R}$, $\mathcal{O}_E$ accepts $\hat{R}$ if $\mathcal{L}(\hat{R}) = L_*$, and returns a counterexample otherwise. In our experiments, there is no way to check equivalence with the target language (i.e., the program input language). Instead, we use the variant in [12] where the equivalence oracle $\mathcal{O}_E$ is implemented by randomly sampling strings to search for counter-examples; we accept $\hat{R}$ if none are found after 50 samples.

RPNI. RPNI learns a regular language $\hat{R}$ given both positive examples E_{in} and negative examples E_{in}^-. As negative examples, we sample 50 random strings not in L_*.

Results. We estimate the precision of $\hat{C}$ by $\frac{|E_{\text{prec}} \cap L_*|}{|E_{\text{prec}}|}$, where E_{prec} consists of 1000 random samples from $\mathcal{L}(\hat{C})$, and estimate the recall of $\hat{C}$ by $\frac{|E_{\text{rec}} \cap \mathcal{L}(\hat{C})|}{|E_{\text{rec}}|}$, where E_{rec} consists of 1000 random samples from L_*, and report the F_1-score $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. The F_1 score is a standard metric combining

precision and recall—achieving high F_1 score requires both high precision and high recall. We also report the running time of each algorithm, which is timed out at 300 seconds. We average all results over five runs. Figure 6.4 shows (a) the F_1-score and (b) the running time of each algorithm; (c) shows how the precision, recall, and running time of GLADE vary with the number of samples in E_{in}.

Performance of GLADE. With just the 50 given training examples, GLADE was able to learn each grammar with an F_1-score of nearly 1.0, meaning that both precision and recall were nearly 100%. These results strongly suggest that GLADE learns most of the true structure of L_*. Finally, as can be seen from Figure 6.4 (c), GLADE performs well even with few samples, and its running time likewise scales well with the number of samples. The performance of GLADE with phase two omitted (i.e., P1 in Figure 6.4) continues to substantially outperform L-Star and RPNI.

Phases of GLADE. As can be seen in Figure 6.4 (a), GLADE consistently performs 5-10% better than P1—i.e., the majority of the improvement of GLADE over existing algorithms is due to the active learning strategy, and the remainder is due to the ability to induce context-free grammars.

Furthermore, a consequence of our optimization when using multiple inputs (see Section 6.5.1), GLADE is actually faster than P1—because GLADE generalizes better than P1, it uses fewer samples in E_{in}, thereby reducing the running time. We performed the same experiment using GLADE with the character generalization phase removed (but including both phases one and two). This variant of GLADE consistently performed similar but slightly worse than P1 both in terms of F_1-score and running time, so we omit results.

Comparison to L-Star and RPNI. L-Star performs well for the Grep grammar, but essentially fails to learn the other grammars, achieving either very small precision or very small recall. RPNI performs even worse, failing to learn any of the languages. L-Star guarantees exact learning only when a true equivalence oracle is available. Similarly, RPNI has an "in the limit" learning guarantee, i.e., for any enumeration of all strings $\alpha_1, \alpha_2, ... \in \Sigma^*$, it eventually learns the correct language. Both of these learning guarantees require following examples:

- **Positive:** Exercise all transitions in the minimal DFA.
- **Negative:** Reject all incorrect generalizations.

These examples are assumed to be provided either by the equivalence oracle (for L-Star) or in the given examples E_{in} and E_{in}^- (for RPNI).

However, in our setting, the equivalence oracle is unavailable to the L-Star algorithm and must be approximated using random sampling, so its theoretical guarantees may not hold. Indeed, random sampling rarely provides the needed examples—for example, in most runs of L-Star, at most two

Program	Lines of Code	Lines in E_{in}	Time (min.)
sed	2K	3	0.25
flex	6K	15	1.83
grep	12K	4	0.17
bison	13K	14	4.91
xml	123K	7	2.30
ruby	120K	80	229.00
python	128K	267	269.00
javascript	156K	118	113.00

Figure 6.6: For each program, we show lines of program code, the lines of seed inputs E_{in}, and running time of GLADE.

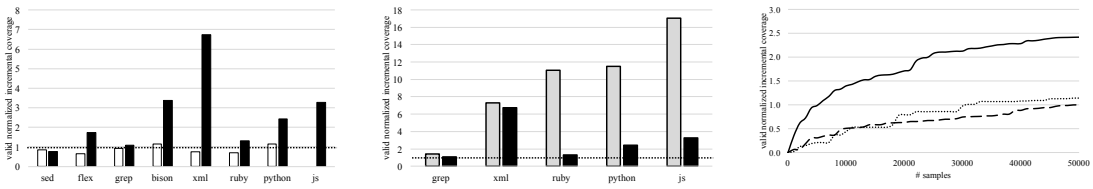
calls to the equivalence oracle found counterexamples. Similarly, for RPNI, the given examples are typically incomplete, so its theoretical guarantees likewise may not hold.

Furthermore, because these algorithms are designed to learn when the guarantees hold, they do not provide any mechanisms for recovering from failure of the assumptions, and instead fail dramatically. For example, if a terminal appears in L_* but not in any seed input in E_{in}, then the language learned by RPNI does not contain any strings with this terminal. In contrast, GLADE incorporates generalization steps that enable it to generalize beyond behaviors in the given examples, and its carefully selected checks often provide the counterexamples needed to avoid overgeneralizing.

Additionally, while polynomial, the running times of L-Star and RPNI are very long. The long running time of L-Star is not because L_* is non-regular, instead, we observe that L-Star algorithm issues a large number of membership queries on each of its iterations. In our setting, L-Star often could not even learn a four state automaton.

Examples. Figure 6.5 shows examples of grammars synthesized by GLADE for the target language shown and a small set of representative seed inputs. The target languages are substantially simplified fragments of the grammars used in this experiment (to ensure clarity); the synthesized grammars are correspondingly simplified.

The structure of a synthesized grammar sometimes differs from the structure of the grammar defining the target language, even if they generate the same language. Such discrepancies occur because GLADE obtains no information about the internal representation of the target language. For example, consider the synthesized XML grammar. In a more natural grammar, the character > at the front of the production for B would instead appear in the production for A, and the corresponding > in the production for A would instead appear at the end of the production for B; however, this modification does not affect the generated language.



(a) (b) (c)

Figure 6.7: In (a) we show the normalized incremental coverage restricted to valid samples for the naïve fuzzer (black dotted line), afl-fuzz (white), and GLADE (black). In (b), we show the same metric for the naïve fuzzer (black dotted line) and GLADE (black); grey represents either a handwritten fuzzer (for Grep and the XML parser) or a large test suite (for Python, Ruby, and Javascript). In (c), we compare the valid normalized incremental coverage of GLADE (solid) to the naïve fuzzer (dashed) and afl-fuzz (dotted) as the number of seed inputs varies (all values are normalized by the final coverage of the naïve fuzzer).

6.7.3 Comparison to Fuzzers

For fuzzing applications such as differential testing [158], it is useful to obtain a large number of grammatically valid samples that exercise different functionalities of the given program. GLADE is perfectly suited to automatically generating such inputs. Given blackbox access $\mathcal{O}$ to a program with input language L_* and seed inputs $E_{\text{in}} \subseteq L_*$, GLADE automatically synthesizes a context-free grammar $\hat{C}$ approximating L_*. Then, GLADE uses a standard grammar-based fuzzer that takes as input the synthesized grammar $\hat{C}$ and the seed inputs E_{in}, and randomly generates new inputs $\alpha \in \mathcal{L}(\hat{C})$ that can be used to test the program; we give details below.

In our application to fuzzing, it is acceptable for $\hat{C}$ to be an approximation—high precision suffices to ensure that most generated inputs are valid, and high recall ensures that most program behaviors have a chance of being executed.

We compare GLADE to two baseline fuzzers (described below) on the task of generating valid test inputs, and show that GLADE consistently performs significantly better.

Grammar-based fuzzer. GLADE first synthesizes a context-free grammar $\hat{C}$ approximating the target language L_* of valid program inputs. Our grammar-based fuzzer, based on standard techniques [75], takes as input the synthesized context-free grammar $\hat{C}$ and the seed inputs E_{in}. To generate a single random input, our grammar-based fuzzer first uniformly selects a seed input $\alpha \in E_{\text{in}}$ and constructs the parse tree for α according to $\hat{C}$. Second, it performs a series of n modifications to α, where n is chosen uniformly between 0 and 50. A single modification is performed as follows:

- Randomly choose a node N of the parse tree of α.
- Decompose $\alpha = \alpha_1\alpha_2\alpha_3$ where α_2 is represented by the subtree with root N.
- Letting A be the nonterminal labeling N, randomly sample $\alpha' \sim \mathcal{P}_{\mathcal{L}(C,A)}$, and return $\alpha_1\alpha'\alpha_3$.

Afl-fuzz. Our first baseline fuzzer is a production fuzzer developed at Google [159], and is widely used due to its minimal setup requirements and state-of-the-art quality. It systematically modifies the input example (e.g., bit flips, copies, deletions, etc.). Unlike GLADE, afl-fuzz requires that the program be instrumented to obtain branch coverage for each execution—it uses this information to identify when an input α causes the program to execute new paths. It adds such inputs α to a worklist, and iteratively applies its fuzzing strategy to each input in the worklist. This monitoring allows it to incrementally discover deeper code paths. To run afl-fuzz on multiple inputs E_{in}, we fuzz each input $\alpha \in E_{\text{in}}$ in a round-robin fashion.

Naïve fuzzer. We implement a second baseline fuzzer, which is not grammar aware. It randomly selects a seed input $\alpha \in E_{\text{in}}$ and performs n random modifications to α, where n is chosen randomly between 0 and 50. A single modification of α consists of randomly choosing an index i in $\alpha = \sigma_1 ... \sigma_k$, and either deleting the terminal σ_i or inserting a randomly chosen terminal $\sigma \in \Sigma$ before σ_i.

Programs. We set up each fuzzer on eight programs that include front-ends of language interpreters (Python, Ruby, and Mozilla's Javascript engine SpiderMonkey), Unix utilities that take structured inputs (Grep, Sed, Flex, and Bison), and an XML parser. We were unable to setup afl-fuzz for Javascript, showing that even production fuzzers can have setup difficulties when they require code instrumentation. For interpreters (e.g., the Python interpreter), we focus on fuzzing just the parser (e.g., the Python parser) since the input grammar of the interpreter contains elements such as variable and function names, use-before-define errors, etc., that are out of scope for our grammar synthesis algorithm. To fuzz the parser, we "wrap" the input inside a conditional statement, which ensures that the input is never executed. For example, we convert the Python input (`print 'hi'`) to the input (`if False: print 'hi'`). Then, syntactically incorrect inputs are rejected, but inputs that are syntactically correct but possibly have runtime errors are accepted.

Seed inputs. To fuzz a program, we use a small number of seed inputs $E_{\text{in}} \subseteq L_*$ that capture interesting semantics of the target language L_*. These seed inputs were obtained either from documentation and tutorials or from small test suites that came with the program.

Methods. Coverage is difficult to interpret because a large amount of code in each program is unreachable due to configuration, test code that cannot be executed, and other unused functionality. Therefore, we use a relative measure of coverage to evaluate performance. As before, all results are averaged over five runs.

For each program and fuzzer, we generate 50,000 samples $E \subseteq \Sigma^*$ by running the fuzzer on the program. First, we restrict E to valid inputs, i.e., $E \cap L_*$. In particular, the *valid coverage* of E,

computed using gcov, is

$$\frac{\#(\text{lines covered by } E \cap L_*)}{\#(\text{lines coverable})}.$$

Next, the *valid incremental coverage* of E is the percentage of code covered by valid inputs in E, ignoring those already covered by the seed inputs E_{in} (thereby measuring the ability to discover inputs that execute new code paths):

$$\frac{\#(\text{lines covered by } E \cap L_* \text{ but not covered by } E_{\text{in}})}{\#(\text{lines coverable but not covered by } E_{\text{in}})}.$$

Finally, to enable comparison across programs, the *valid normalized incremental coverage* normalizes the incremental coverage by a baseline E_{base}:

$$\frac{\text{valid incremental coverage of } E}{\text{valid incremental coverage of } E_{\text{base}}}.$$

In particular, we use samples from the naïve fuzzer as E_{base}.

Results. In Figure 6.6, we show various statistics for the eight programs we use and for the corresponding seed inputs E_{in}. We also show the time GLADE needed to synthesize an approximation of the program input grammar. In Figure 6.7 (a), we show the valid normalized incremental coverages of the various fuzzers. In (b), for five of our programs, we show a proxy for the "upper bound" in coverage that is achievable—for Grep and the XML parser, we show the valid normalized incremental coverage achieved by our handwritten grammars, and for Python, Ruby, and Javascript, we show the valid normalized incremental coverage of a large test suite (each more than 100,000 lines of code). In (c), we show how coverage varies with the number of samples for Python.

Comparison to baselines. As can be seen from Figure 6.7 (a), GLADE (black) is effective at generating valid inputs that exercise new code paths, significantly outperforming both the naïve fuzzer (black dotted line) and afl-fuzz (white) except on Grep (where it only performs slightly better) and Sed (where it actually performs slightly worse). Since these programs have a relatively simple input format, using a grammar-based fuzzer is understandably less effective. For the remaining six programs, our grammar-based fuzzer performs between 1.3 and 7 times better than the naïve fuzzer.

Comparison to proxy for the upper bound. Figure 6.7 (b) compares GLADE (black bars) to a proxy for the upper bound of coverage, i.e., handwritten grammars or large test suites (grey bars). For Grep, both GLADE and the naïve fuzzer achieve coverage close to the handwritten grammar. For the XML parser, GLADE significantly outperforms the naïve fuzzer, achieving coverage close to the handwritten grammar. For Python and Javascript, GLADE is able to recover a significantly

```
<a>
  \%
  <a QE="{>_">
    C
    <a    _="#">
      ">q(+_[s:?>^0+
      <a  _eD="{@">
        :"<a>. q</a>1+%
      </a>
      y<!--       y-->y
    </a>
    _<a>x</a>y
  </a>
  xy<?q  xy?>xy<?xV <?By_![?>x
</a>
```

Figure 6.8: An example of a valid sample from the grammar synthesized by GLADE for the XML parser. For clarity, the string has been formatted with additional whitespace.

larger fraction of the upper bound compared to the naïve fuzzer. However, a sizable gap remains, which is expected since the test suites are very large (each having at least 100,000 lines of code) and are specifically designed to test the respective programs. We provided fewer seed inputs for Ruby, which explains why GLADE outperformed the naïve fuzzer by a smaller amount (about 30%).

Coverage over time. Figure 6.7 (c) shows how the valid normalized incremental coverage varies with the number of samples. GLADE (solid) quickly finds a number of high-coverage inputs that the other fuzzers cannot, and continues to find more inputs that execute new lines of code.

Examples. The synthesized grammars are too large to show. Instead, as an example, a fragment of the synthesized XML grammar is

$$A \to \texttt{<a}\sqcup^*\sqcup[...]^*[...]\texttt{="}[...]^*\texttt{"}B^*\texttt{>}[...]^*\texttt{}$$
$$B \to \texttt{>}[...]^*\texttt{<a}\sqcup^*\sqcup[...]^*[...]\texttt{="}[...]^*\texttt{"}B^*\texttt{>}[...]^*\texttt{</a}$$
$$+\,\texttt{>}[...]^*\texttt{<a>}[...]^*\texttt{</a}.$$

This grammar is identical to the synthesized XML grammar shown in Figure 6.5, except that attributes cannot be repeated. In particular, GLADE learns that attributes cannot be repeated since XML semantics requires that different attributes have different names—for example, the input string `<a a="" a=""></a>` is invalid. Therefore, repeating the attribute would lead to overgeneralization, so this construct is rejected by GLADE. Indeed, this constraint on attribute names is not a context-free property, so as expected, GLADE learns a subset of the XML input language.

Figure 6.8 shows an example of a valid sample from the grammar synthesized by GLADE for the

XML parser. As can be seen, the sample contains many XML constructs, including nested tags, attributes, comments, and processing instructions.

Synthesis. Finally, our approach uses machinery related to some of the recent work on programming by example—in particular, a systematic search guided by a meta-grammar. This approach has been used to synthesize string [68], number [132], and table [70] transformations (and combinations thereof [113, 114]), as well as recursive programs [53, 6] and parsers [88]. Unlike these approaches, our approach exploits an oracle to reject invalid candidates.

6.8 Conclusion

We have presented Glade, the first practical algorithm for inferring program input grammars, and demonstrated its value in an application to fuzz testing. We believe Glade may be valuable beyond fuzzing, e.g., to generate whitelists of inputs or to reverse engineer input formats.

Chapter 7

Related Work

7.1 Specification Inference for Static Analysis

Interactive specification inference. There has been previous work on interacting with a human analyst to infer specifications [43, 161]. [43] proposes an algorithm for inferring program invariantes by interacting with the analyst. In particular, they use abductive inference to infer a minimal-sized invariant that is sufficient to verify a postcondition; they propose this invariant to the analyst, who can either accept it (in which case the program is verified) or reject it (in which case the process is repeated).

The work closest to our own is [161], which uses abductive inference to interactively infer specifications for library code [161]. There are several differences in the two approaches. First, we handle the general class of CFL reachability problems, whereas [161] 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 [161].

There has also been work on interacting with the analyst to refine the results of a static analysis [98]. In particular, the analyst can mark certain outputs of the static analysis as false positives. Then, they use a probabilistic model equivalent to a Markov logic network to generalize this data to remove additional false positives. Unlike our approach, they do not guarantee soundness at the end of the interactive process, and they furthermore do not generalize the learned information beyond a single program.

Inferring specifications from executions. There has been working on mining executions for specifications [9, 107, 8, 157, 128, 104, 108, 57, 126, 162, 39, 74, 112, 163, 73, 79]. These techniques have been used to infer both kinds of specifications described in Chapter 2, namely, (i) specifications

that describe desired behavior and (ii) specifications that describe behavior assumed to hold.

For the first kind of specifications, [9, 8] study the problem of inferring specifications that should hold for clients of an API interface, such as "the exception E should not be raised". These specifications take the form of state machines, e.g., a file should be opened and then closed. [157] extends these ideas to the case where the execution traces are imperfect and may exhibit bugs.

For the second kind of specifications, [107] leverages the idea that statically checking the correctness of specifications is easier than devising them to begin with. For example, in general, checking the correctness of loop invariants is decideable, but inferring a loop invariant is undecideable. Thus, they first use test cases to exercise the code and use data mining techniques to find specifications (in particular, program invariants) that are consistent with the behaviors observed during execution. Then, they use a static checker to check which specifications are correct. There has been work extending these ideas by using machine learning to infer invariants [129, 127, 126, 112]; these approaches can additionally use counterexamples provided by the static checker to refine the inferred specifications. These ideas have also been applied to learn other kinds of specifications, including refinement types [162] and shape invariants [163]. Similar approaches have also been used to prove program termination properties [108].

The work most closely related to our own uses dynamic executions to infer specifications that summarizes properties of library functions, including information flow specifications [39], specifications for x86 instructions [73], specifications for callback control flow [79], and even full implementations of the library functions [74]. To the best of our knowledge, we are the first to study the use of active learning strategies to infer specifications in the blackbox setting, as well as the first to incorporate ideas from program synthesis by examples to infer program properties.

Finally, [65] uses must-facts extracted from guided dynamic executions to avoid spending effort trying to discharge true positives. In contrast, we use must-facts extracted from tests as specifications for may-facts—i.e., we *enforce* that may-facts not observed in tests are invalid using instrumentation, and use abductive inference to minimize the amount of instrumentation required.

Inferring specifications from static information. There has been work on mining specifications from static information using machine learning, both of the first kind of specification described above [86, 116, 131, 95, 20, 117] and of the second [44, 160, 5, 22]. For the first kind of specifications, there has been work on using probabilistic graphical models to mine interface specifications [86], sources, sinks, and sanitizers for taint analysis [95], and type-state specifications [20], and type annotations [117]. Finally, [5] uses abductive inference to infer function preconditions.

For the second kind of specifications, there has been work on using abductive inference to infer program invariants [44] and abstractions [160], and on using extensions of decision tree learning algorithms to infer transfer functions for static analysis [22]. A approach related to abductive inference has been proposed for choosing sanitizer placement [94].

7.2 Program Analysis

Dynamic safety. Instrumenting programs to ensure safety properties is well-studied, for example to enforce type safety [72, 105] and to ensure control-flow integrity [1]. Our work applies similar principles to ensure the integrity of information flows, which is more challenging because information flows are global properties. In [23], instrumentation is guided by testing: only reflective calls observed during execution are permitted. Their instrumentation issues warnings to the user for potential unsoundness in the static analysis. Finally, there has been work on modifying programs to coerce potentially problematic inputs into acceptable forms manually specified by the user [120, 121].

Dynamic taint tracking has been applied to produce programs that terminate execution upon violation of the security policy, for example, for Android apps [46] and for a web browser [41]. To the best of our knowledge, existing approaches to enforce information flow policies require instrumenting the entire program (or modifying the runtime environment). In contrast, our approach uses very minimal instrumentation, and often places that instrumentation in unreachable code where it will have zero runtime cost. Other approaches for restricting app behaviors have been proposed, for example [122], but the policies enforced are local (e.g., disallowing calls to certain library methods).

Callgraph analysis. There has been prior work on generating sound callgraphs for library code without analyzing the library code [7], which is related to our problem of inferring reachability specifications. Their work constructs a placeholder library that exhibits every possible behavior that can affect the call graph. Our technique for inferring reachability specifications, in addition to being more general, actually proposes library specifications and allows an auditor to interactively refine analysis results. Similar work has inferred callback specifications by analyzing the code [34]; however, this approach exhibits both false positives and false negatives.

CFL reachability. A large number of program analyses have been expressed as CFL reachability problems, for example points-to analysis [137, 136], various interprocedural analyses [119, 118], and type qualifier inference [67]. 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 CFL reachability with additional regular language properties such as [84].

The set constraints formalism, which has very efficient and scalable solutions [85], can also encode most popular CFL reachability problems [99, 83]. Because of the formulation as constraints, in this formalism it is possible to analyze partial programs or modules separately and then combine the solutions [85]. Andersen-style flow-insensitive points-to analysis has been one of the most studied applications for set constraints [50, 141, 71, 4, 84]. However, none of these works address the issue of analyzing missing code.

Applications to security. Work on information flow analysis for Android includes SCanDroid [55], which statically tracks taint flows between applications, TaintDroid [46], which is a dynamic system that performs real time monitoring, and FlowDroid [14], which uses static analysis to find information leaks. Information flow properties have been used to find real examples of Android malware [51, 47, 52]. Static analysis has also been applied to finding vulnerabilities in web applications [96, 155, 146, 135].

The approach in [47] shares our goal of moving the burden of verification to the (possibly adversarial) developer; they require the developer to annotate the source code with information types to guide the auditing process. Compared to [47], our approach does not require source code (commercial app stores typically do not have access to source code) and leverages existing test suites to produce specifications rather than requiring annotations specific to information flow.

Static points-to analysis. There is a large literature on static points-to analysis [130, 11, 153, 50, 100], including formulations based on CFL reachability [118, 137]. Recent work has focused on improving context-sensitivity [152, 136, 90, 160, 133]. Using specifications in conjunction with these analyses can improve precision, scalability, and even soundness.

One alternative is to use demand driven static analyses to avoid analyzing the entire library code [137]; however, these approaches are not designed to work with missing code, and furthermore do not provide much benefit for demanding clients that require analyzing a substantial fraction of the library code.

7.3 Language Learning

Mining input formats. The work most closely related to our own is [76], which uses dynamic taint analysis to trace the flow of each input character, and uses this information to reconstruct the input grammar. More broadly, there has been work on reverse engineering network protocol message formats [29, 154, 91, 92], though these papers focus on learning and understanding the structure of given inputs rather than learning a grammar; for example, [29] looks for variables representing the internal parser state to determine the protocol, and [91] constructs syntax trees for given inputs. All of these techniques rely on static and dynamic analysis methods intended to reverse engineer parsers of specific designs.

In contrast, our approach is fully blackbox and depends only on the language accepted by the program, not the specific design of the program's parser. In addition, our approach can be used when the program cannot be instrumented, for instance, to learn the input format for a remote program. Finally, the programs we consider have more complex input formats than most previously examined programs.

Learning theory. There has been a line of work in learning theory (often referred to as *grammar induction* or *grammar inference*) aiming to learn a grammar from either examples or oracles (or both); see [42] for a survey. The most well known algorithms are L-Star [12] and RPNI [110]. These algorithms have a number of applications including model checking [58], model-assisted fuzzing [35, 36], verification [148], and specification inference [25]. To the best of our knowledge, our work is the first to focus on the application of learning common program input languages from positive examples and membership oracles.

Additionally, [87] discusses approaches to learning context-free grammars, including from positive examples and a membership oracle. As they discuss, these algorithms are often either slow [134] or do not generalize well [81].

Bayesian language learning. A related line of work aims to learn probabilistic grammars from examples alone [140, 139]. These algorithms study a different setting than ours, in particular, they are given access to positive (and sometimes negative) examples, but do not assume access to a membership oracle. These algorithms typically identify frequently occurring patterns that are likely to correspond to nonterminals in the grammar. More precisely, these algorithms are typically Bayesian learning algorithms that operate by putting a prior over the space of grammars, and then computing the most likely grammar conditioned on the given examples. To achieve statistically significant results, these algorithms require a large number of input examples.

In contrast, our algorithm leverages access to the membership oracle, enabling it to use actively generated examples to determine which patterns are actually in the grammar. Therefore, our algorithm works well even when only a few seed inputs are available. While it may be possible to modify existing Bayesian language learning algorithms to fit this setting, to the best of our knowledge, no such active learning variants of these algorithms have been proposed.

Additionally, whereas this literature aims to learn a probabilistic grammar, our grammar synthesis algorithm learns a deterministic grammar. The difference is how we measure approximation quality—in particular, even though our definitions of precision and recall require distributions over L_* and $\hat{L}$, they still measure the approximation quality of $\hat{L}$ deterministically, i.e., the predicates $\alpha \in L_*$ and $\alpha \in \hat{L}$ are binary rather than probabilistic.

Blackbox fuzzing. Numerous approaches to automated test generation have been proposed; we refer to [10] for a survey. Approaches to fuzzing (i.e., random test case generation) broadly fall into two categories: whitebox (i.e., statically inspect the program to guide test generation) and blackbox (i.e., rely only on concrete program executions). Blackbox fuzzing has been used to test software for several decades; for example, [124] randomly tests COBOL compilers and [115] generated random inputs to test parsers. An early application of blackbox fuzzing to find bugs in real-world programs was [102], who executed Unix utilities on random byte sequences to discover crashing inputs. Subsequently, there have been many approaches using blackbox fuzzing with dynamic analysis to find

bugs and security vulnerabilities [54, 143, 101]; see [144] for a survey. Finally, afl-fuzz [159] is almost blackbox, requiring only simple instrumentation to guide the search.

Whitebox fuzzing. Approaches to whitebox fuzzing [64, 13] typically build on *dynamic symbolic execution* [62, 125, 31, 30, 32]; given a concrete input example, these approaches use a combination of symbolic execution and dynamic execution to construct a constraint system whose solutions are inputs that execute new program branches compared to the given input. It can be challenging to scale these approaches to large programs [56]. Therefore, approaches relying on more imprecise input have been studied; for example, taint analysis [56], or extracting specific information such as a checksum computation [151].

Grammar-based fuzzing. Many fuzzing approaches leverage a user-defined grammar to generate valid inputs, which can greatly increase coverage. For example, blackbox fuzzing has been combined with manually written grammars to test compilers [93, 158]; see [26] for a survey. Such techniques have also been used to fuzz interpreters; for example, [75] develops a framework for grammar-based testing and applies it to find bugs in both Javascript and PHP interpreters.

Grammar-based approaches have also been used in conjunction with whitebox techniques. For example, [61] fuzzes a just-in-time compiler for Javascript using a handwritten Javascript grammar in conjunction with a technique for solving constraints over grammars, and [97] combines exhaustive enumeration of valid inputs with symbolic execution techniques to improve coverage. In [144], Chapter 21 gives a case study developing a grammar for the Adobe Flash file format. Our approach can complement existing grammar-based fuzzers by automatically generating a grammar.

Finally, there has been some work on inferring grammars for fuzzing [149], but focusing on simple languages such as compression formats. To the best of our knowledge, our work is the first targeted at learning complex program input languages that contain recursive structure, e.g., XML, regular expression formats, and programming language syntax.

Chapter 8

Conclusion

Specification inference is a promising approach to improving the usability of program analysis in practice. Specifications are crucial for modeling parts of programs that are too difficult for a static program analysis to handle, such as native code or dynamic language features. By either interacting with the human analyst or leveraging observations from concrete executions, the algorithms we have described can infer high quality specifications that can be used by client program analyses to test and verify programs. In particular, we have proposed a novel algorithm, based on abductive inference, that interacts with a human analyst to infer specifications, and shown that it quickly converges to the true specifications. We have extended interactive specification inference to the case when the analyst is not trusted, by instrumenting the code to enforce responses. In addition, we have proposed novel algorithms that infer specifications, based on inductive inference, that use active learning strategies to infer complex hierarchical specifications. These algorithms propose candidate specifications, and then actively query an oracle to determine which candidates may be correct.

There are two important directions for further research. First, for interactive specification inference, a key challenge is understanding what kinds of queries can feasibly be answered by a human analyst. We have focused on simple properties of library functions, which are well documented, and reachability properties, which are easily known by the developer. More complex static analyses may have intermediate relations that the human analyst is unable to reason about, making it more challenging to interactively infer specifications. Second, our algorithms for actively inferring specifications from executions are both domain specific. We believe that these techniques are much more generally applicable. An important research direction is devising a general framework for inferring specifications for blackbox code.

Going forward, the techniques we have developed in this thesis can be extended to infer specifications for a much wider range of program analyses. We hope that by reducing the cost of writing specifications, these downstream program analyses will become cost-effective enough to use in a much wider range of practical applications.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, pages 340–353, 2005.

[2] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[3] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *PASTE*, pages 43–48, 2007.

[4] Alex Aiken, Jeffrey S Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI*, pages 129–140, 2003.

[5] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, pages 789–801, 2016.

[6] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.

[7] Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *ECOOP*, pages 378–400, 2013.

[8] Rajeev Alur, Pavol Černỳ, Parthasarathy Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPl*, pages 98–109, 2005.

[9] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

[10] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[11] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[12] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[13] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in dynamic web applications. In *ISSTA*, pages 261–272, 2008.

[14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, pages 259–269, 2014.

[15] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

[16] Thomas Ball and Sriram K Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[17] Osbert Bastani, Saswat Anand, and Alex Aiken. Interactively verifying absence of explicit information flows in android apps. In *OOPSLA*, pages 299–315, 2015.

[18] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, pages 553–566, 2015.

[19] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *PLDI*, pages 95–110, 2017.

[20] Nels E Beckman and Aditya V Nori. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, pages 211–221, 2011.

[21] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *CACM*, 53(2):66–75, 2010.

[22] Pavol Bielik, Veselin Raychev, and Martin Vechev. Learning a static analyzer from data. In *CAV*, pages 233–253, 2017.

[23] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, pages 241–250, 2011.

[24] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R Piegdon. libalf: The automata learning framework. In *CAV*, pages 360–364, 2010.

[25] Matko Botinčan and Domagoj Babić. Sigma*: Symbolic learning of input-output specifications. In *POPL*, pages 443–456, 2013.

[26] Abdulazeez S Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information and software technology*, 39(9):617–625, 1997.

[27] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: better together. In *ISSTA*, pages 1–12, 2009.

[28] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[29] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *CCS*, pages 317–329, 2007.

[30] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[31] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. In *CCS*, pages 322–335, 2006.

[32] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *CACM*, 56(2):82–90, 2013.

[33] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. *NFM*, 15:3–11, 2015.

[34] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[35] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security*, pages 139–154, 2011.

[36] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *OOPSLA*, pages 623–640, 2013.

[37] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. Using meta-level compilation to check flash protocol code. In *ASPLOS*, pages 59–70, 2000.

[38] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, 2001.

[39] Lazaro Clapp, Saswat Anand, and Alex Aiken. Modelgen: mining explicit information flow specifications from concrete executions. In *ISSTA*, pages 129–140, 2015.

[40] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *ESOP*, 2005.

[41] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *CCS*, pages 748–759, 2012.

[42] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[43] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *PLDI*, pages 181–192, 2012.

[44] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456, 2013.

[45] ECMA International. *Standard ECMA-262: ECMA 2015 Language Specification*. 6 edition, June 2015.

[46] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *TOCS*, 32(2):5, 2014.

[47] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. Collaborative verification of information flow for a high-assurance app store. In *CCS*, pages 1092–1104, 2014.

[48] Espresso. `https://developer.android.com/training/testing/ui-testing/espresso-testing.html`, 2017.

[49] Facebook. Adding models, 2017.

[50] Manuel Fähndrich, Jeffrey S Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, 1998.

[51] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE*, pages 576–587, 2014.

[52] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *NDSS*, 2017.

[53] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239, 2015.

[54] Justin E Forrester and Barton P Miller. An empirical study of the robustness of windows nt applications using random testing. In *WSS*, pages 59–68, 2000.

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

[56] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE*, pages 474–484, 2009.

[57] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *CAV*, pages 69–87, 2014.

[58] Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic learning of component interfaces. In *SAS*, pages 248–264, 2012.

[59] GNU. Gnu bison. `https://www.gnu.org/software/bison`, 2014.

[60] GNU Grep. `https://www.gnu.org/software/grep/manual`, 2016.

[61] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

[62] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[63] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *CACM*, 55(3):40–44, 2012.

[64] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166, 2008.

[65] Patrice Godefroid, Aditya V Nori, Sriram K Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56, 2010.

[66] E Mark Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[67] David Greenfieldboyce and Jeffrey S Foster. Type qualifier inference for java. In *OOPSLA*, pages 321–336, 2007.

[68] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[69] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.

[70] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.

[71] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *PLDI*, pages 254–263, 2001.

[72] Fritz Henglein. Global tagging optimization by type inference. In *LFP*, pages 205–215, 1992.

[73] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: automatically learning the x86-64 instruction set. In *PLDI*, pages 237–250, 2016.

[74] Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: Computing models for opaque code. In *FSE*, pages 710–720, 2015.

[75] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security*, pages 445–458, 2012.

[76] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *ASE*, pages 720–725, 2016.

[77] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *NIPS*, pages 883–891, 2010.

[78] Hiroki Ishizaka. Polynomial time learnability of simple deterministic languages. *Machine Learning*, 5(2):151–164, 1990.

[79] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *ICSE*, pages 156–167, 2016.

[80] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Cant live with em, cant live without em. In *ICISS*, pages 56–70, 2008.

[81] Bruce Knobe and Kathleen Knobe. A method for inferring context-free grammars. *Information and Control*, 31(2):129–146, 1976.

[82] Donald E Knuth. A generalization of dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[83] John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. In *PLDI*, pages 207–218, 2004.

[84] John Kodumal and Alex Aiken. Regularly annotated set constraints. In *PLDI*, pages 331–341, 2007.

[85] John Kodumal and Alexander Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, pages 218–234, 2005.

[86] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, pages 161–176, 2006.

[87] Lillian Lee. Learning of context-free languages: A survey of the literature. *Technical Report TR-12-96, Harvard University*, 1996.

[88] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *PLDI*, pages 565–574, 2015.

[89] Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. In *SAS*, pages 162–180, 2015.

[90] Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *PLDI*, pages 590–601, 2011.

[91] Zhiqiang Lin and Xiangyu Zhang. Deriving input syntactic structure from execution. In *FSE*, pages 83–93, 2008.

[92] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Reverse engineering input syntactic structure from program execution and its applications. *IEEE Transactions on Software Engineering*, 36(5):688–703, 2010.

[93] Christian Lindig. Random testing of c calling conventions. In *AADEBUG*, pages 3–12, 2005.

[94] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *POPL*, pages 385–398, 2013.

[95] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, pages 75–86, 2009.

[96] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security*, pages 18–18, 2005.

[97] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *ASE*, pages 134–143, 2007.

[98] Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. A user-guided approach to program analysis. In *FSE*, pages 462–473, 2015.

[99] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *TCS*, 248(1):29–98, 2000.

[100] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *TOSEM*, pages 1–11, 2002.

[101] Barton P Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *RT*, pages 46–54, 2006.

[102] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *CACM*, 33(12):32–44, 1990.

[103] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[104] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from tests. In *POPL*, pages 373–386, 2012.

[105] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *TOPLAS*, 27(3):477–526, 2005.

[106] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[107] Jeremy W Nimmer and Michael D Ernst. Automatic generation of program specifications. In *ISSTA*, pages 229–239, 2002.

[108] Aditya V Nori and Rahul Sharma. Termination proofs from tests. In *FSE*, pages 246–256, 2013.

[109] Peter Norvig. `http://norvig.com/lispy.html`, 2010.

[110] José Oncina and Pedro García. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition*, 5(99-108):15–20, 1992.

[111] Oracle America, Inc. *The Java^{TM} Virtual Machine Specification*. 7 edition, July 2011.

[112] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *PLDI*, pages 42–56, 2016.

[113] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *PLDI*, pages 408–418, 2014.

[114] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *OOPSLA*, pages 107–126, 2015.

[115] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[116] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *PLDI*, pages 123–134, 2007.

[117] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *POPL*, pages 111–124, 2015.

[118] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11):701–726, 1998.

[119] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[120] Martin Rinard. Acceptability-oriented computing. In *OOPLSA*, pages 221–239, 2003.

[121] Martin C. Rinard. Living in the comfort zone. In *OOPSLA*, pages 611–622, 2007.

[122] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. Firedroid: Hardening security in almost-stock android. In *ACSAC*, pages 319–328, 2013.

[123] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[124] Richard L Sauder. A general test data generator for cobol. In *AIEE-IRE (Spring)*, pages 317–323, 1962.

[125] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *FSE*, pages 263–272, 2005.

[126] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, pages 88–105, 2014.

[127] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592, 2013.

[128] Rahul Sharma, Aditya V Nori, and Alex Aiken. Interpolants as classifiers. In *CAV*, pages 71–87, 2012.

[129] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *OOPSLA*, pages 391–406, 2013.

[130] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[131] Sharon Shoham, Eran Yahav, Stephen J Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[132] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651, 2012.

[133] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *PLDI*, pages 485–495, 2014.

[134] Ray J Solomonoff. A new method for discovering the grammars of phrase structure languages. In *Information Processing*, 1960.

[135] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: taint analysis of framework-based web applications. In *OOPSLA*, pages 1053–1068, 2011.

[136] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI*, pages 387–400, 2006.

[137] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *OOPSLA*, pages 59–76, 2005.

[138] Stack Overflow. `http://stackoverflow.com/questions/3809401/what-is-a-good-regular-expression-to-match-a-url`, 2010.

[139] Andreas Stolcke. *Bayesian learning of probabilistic language models*. PhD thesis, University of California, Berkeley, 1994.

[140] Andreas Stolcke and Stephen Omohundro. Inducing probabilistic grammars by bayesian model merging. *Grammatical inference and applications*, pages 106–118, 1994.

[141] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *POPL*, pages 81–95, 2000.

[142] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL*, pages 372–382, 2006.

[143] Michael Sutton and Adam Greene. The art of file format fuzzing. In *Blackhat USA conference*, 2005.

[144] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[145] The Flex Project. Flex: The fast lexical analyzer. `http://flex.sourceforge.net`, 2008.

[146] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

[147] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *CASCON*, page 13, 1999.

[148] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Learning to verify safety properties. In *ICFEM*, pages 274–289, 2004.

[149] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. Experiences with model inference assisted fuzzing. In *WOOT*, 2008.

[150] W3C. `https://www.w3.org/TR/2008/REC-xml-20081126`, 2008.

[151] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, pages 497–512, 2010.

[152] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[153] Robert P Wilson and Monica S Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, pages 1–12, 1995.

[154] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. Automatic network protocol analysis. In *NDSS*, volume 8, pages 1–14, 2008.

[155] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security*, pages 179–192, 2006.

[156] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *TOPLAS*, 29(3):16, 2007.

[157] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE*, pages 282–291, 2006.

[158] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *PLDI*, pages 283–294, 2011.

[159] Michal Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl`, 2015.

[160] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *PLDI*, pages 239–248, 2014.

[161] Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *APLAS*, pages 290–306, 2013.

[162] He Zhu, Aditya V Nori, and Suresh Jagannathan. Learning refinement types. In *ICFP*, pages 400–411, 2015.

[163] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *PLDI*, pages 491–507, 2016.