Eventually Sound Points-To Analysis with Missing Code

ANONYMOUS AUTHOR(S)

Static analyses make the increasingly tenuous assumption that all source code is available for analysis; for example, large libraries often call into native code that cannot be analyzed. We propose a points-to analysis that initially makes optimistic assumptions about missing code, and then inserts runtime checks that report counterexamples to these assumptions that occur during execution. Our approach guarantees eventual soundness, which combines two guarantees: (i) the runtime checks are guaranteed to catch the first counterexample that occurs during any execution, in which case execution can be terminated to prevent harm, and (ii) only finitely many counterexamples ever occur, implying that the static analysis eventually becomes statically sound with respect to all remaining executions. We implement Optix, an eventually sound points-to analysis for Android apps, where the Android framework is missing. We show that the runtime checks added by Optix incur low overhead on real programs, and demonstrate how Optix improves a client information flow analysis for detecting Android malware.

1 INTRODUCTION

To guarantee soundness, static analyses often assume that all program source code is available for analysis. This assumption has become tenuous as programs increasingly depend on large libraries and frameworks that are prohibitively difficult to analyze. For example, mobile app stores can use static analysis to improve the quality of published apps by searching for malicious behaviors [Arzt et al. 2014; Feng et al. 2014; Fuchs et al. 2009; Gordon et al. 2015] or security vulnerabilities [Egele et al. 2013; Fahl et al. 2012; Mutchler et al. 2016]. However, Android apps depend extensively on the Android framework, which makes frequent use of native code and reflection, both of which are practical barriers to static analysis [Arzt et al. 2014; Feng et al. 2014; Gordon et al. 2015]. Furthermore, the Android framework contains deep call hierarchies, which can pose problems since static points-to analyses typically have limited context sensitivity [Bastani et al. 2018]. Therefore, the Android framework is often omitted from the static analysis [Bastani et al. 2015b; Clapp et al. 2015; Zhu et al. 2013], in which case we refer to it as missing. In any large software system, there are inevitably parts that are missing and cannot be handled soundly [Livshits et al. 2015].

When code is missing, one of the following desirable properties must be sacrificed: (i) soundness (by making optimistic assumptions), (ii) precision (by making pessimistic assumptions), or (iii) automation (by using human-written specifications that summarize missing code) [Bastani et al. 2015b; Zhu et al. 2013]. For many analyses, pessimistic assumptions are simply too imprecise, and losing soundness is a significant compromise. For example, consider malware detection—a security analyst must examine every potential malware, making false positives costly, but unsoundness can be exploited by a knowledgeable attacker to avoid detection.

Using specifications is a promising compromise—in principle, for a one-time cost of writing specifications, the precision of the analysis can be greatly improved without sacrificing soundness [Bastani et al. 2018]. Indeed, many static analysis systems rely on specifications to model missing code such as the Android framework [Arzt et al. 2014; Feng et al. 2014; Gordon et al. 2015], including systems used in production [Facebook 2017]. However, specifications are costly to write—even though a single specification typically only takes about a minute to write, oftentimes many thousands of specifications must be written. For example, the Android framework contains tens of thousands of methods, and even a single app may require several thousand specifications [Bastani
et al. 2015b]. Furthermore, handwritten specifications can be error prone [Heule et al. 2015] and need to be updated whenever the library changes.

Oftentimes, an effective strategy is to implement specifications as needed—in the malware example, large portions of libraries are typically irrelevant to the static analysis. Of course, determining which specifications are relevant can be very error prone. In particular, a very typical setting is that the static analysis system contains specifications for the most frequently used library functions, but specifications for the long tail of infrequently used library functions are missing [Bastani et al. 2015b, 2018; Clapp et al. 2015]. Typically, missing specifications are optimistically assumed to be empty, leaving open the possibility of false negatives. These tradeoffs can be alleviated, but not eliminated, by inferring specifications, e.g., automatically based on dynamic analysis [Bastani et al. 2018; Clapp et al. 2015; Heule et al. 2015] or interactively with a human analyst [Bastani et al. 2015b; Zhu et al. 2013].

We propose a novel approach for detecting when missing specifications may be relevant to the analysis of a given program. In particular, our approach instruments a program running in production to detect whether the results of an optimistic static analysis may be unsound due to missing specifications. If unsoundness is detected, then execution can be terminated, or continued in a safe environment such as a sandbox. Thus, our approach dynamically ensures that no malicious functionality is executed. In particular, our soundness guarantees are similar to those provided by dynamic type systems or dynamic information flow control [Austin and Flanagan 2012; De Groef et al. 2012; Enck et al. 2014], except runtime exceptions are due to detected missing specifications rather than detected information flows or type errors.

Given a program (e.g., an Android app), we first run the static analysis with optimistic assumptions about missing specifications. If no errors are found, then the program is instrumented to detect counterexamples to the optimistic assumptions, and the instrumented program is published (e.g., on Google Play). If a counterexample is ever detected, then it is reported back to the publisher (e.g., Google), who can update their specifications and re-run the static analysis. In addition, execution may be terminated to prevent harm. With an appropriate instrumentation scheme, our approach satisfies three important properties:

- **Eventual soundness**: As soon as a counterexample occurs during execution, it is detected by the program instrumentation and reported to the static analysis. Furthermore, only finitely many counterexamples are ever reported.
- **Precision**: The analysis is at least as precise as having all specifications available.
- **Automation**: The analysis is highly automated.

The key property of interest is eventual soundness, which combines two guarantees. The first guarantee is similar to dynamic type checking: at the cost of some runtime overhead, we guarantee that unsoundness is detected as soon as it occurs, which enables the user to prevent damage (e.g., leaking of sensitive information) from occurring. In particular, eventually sound analyses are sound with respect to all executions observed so far.

The second guarantee is that with every reported counterexample, the specifications become progressively more complete. More precisely, suppose that a counterexample is reported for an execution e. Based on this information, the static analysis either discovers a bug (e.g., an information leak), in which case the program is repaired or removed, or continues to conclude that the program is safe given the updated specifications. In the latter case, for any subsequent execution identical to e, no counterexamples will be reported since the static analysis is now sound with respect to all behaviors exhibited in e and has concluded that all of these behaviors are safe.

Furthermore, note that we guarantee that only finitely many counterexamples are ever reported (this observation is a simple consequence of the fact that even in the worst case, there are only a...
finite number of potential counterexamples). Thus, the static analysis eventually becomes statically sound with respect to all subsequent executions. Even though we cannot detect when convergence is reached, it suggests that in practice, fewer and fewer counterexamples are reported over time. Indeed, we empirically observe this trend in our evaluation.

To design an eventually sound program analysis, we must design an instrumentation scheme that satisfies the eventual soundness property. Schemes satisfying eventual soundness or similar properties have been proposed for type checking [Flanagan 2006], resolving reflective call targets [Bodden et al. 2011], and determining reachable code [Bastani et al. 2015a]. However, in these settings, the schemes are relatively straightforward—e.g., to detect missing reflective call targets, it suffices to record the call target of each reflective call.

In this paper, we propose an eventually sound points-to analysis for Android apps, where the Android framework is missing. We focus on points-to analysis since it lies at the core of many static analyses, and we believe that eventually sound clients (e.g., static information flow analysis) can be designed around our analysis. In our setting, counterexamples are missing points-to edges that occur during an execution but are missing from the (optimistic) static analysis. Our main contribution is an eventually sound instrumentation scheme that detects and reports missing points-to edges. In contrast to previous settings, designing such a scheme for points-to analysis can be very challenging for two reasons:

- Naïvely using a dynamic points-to analysis to detecting counterexamples can incur huge overhead—for example, [Clapp et al. 2015] reports a 20× slowdown, and [Mock et al. 2001] reports a slowdown of two orders of magnitude.
- It is often not possible to insert runtime checks into missing code, e.g., in native code. Thus, we restrict our analysis to instrument only available code (in our case, the app code).

To address the first challenge, we leverage the fact that to be eventually sound, we do not need to report every counterexample that occurs during an execution. For example, if a potentially missing points-to edge $x \rightarrow o$ can only occur during an execution if another potentially missing points-to edge $y \rightarrow o$ first occurs, then we only need to monitor whether $y \rightarrow o$ occurs. By leveraging this property, we substantially reduce the amount of required instrumentation. For programs where instrumentation in performance-critical parts is required, the overhead can be further reduced by manually adding specifications summarizing the relevant missing code.

For the second challenge, note that because we use specifications, we are already unable to discover relationships about the missing code. For many clients, only relationships between variables in the available code are of interest—e.g., Android malware can be characterized by relationships between variables in the app code alone [Feng et al. 2014]. However, these relationships typically depend on relationships between variables in the missing code. For points-to analysis, we cannot observe when variables in the app might be aliased because they both point to the same object allocated in missing code. To address this issue, our analysis introduces proxy objects that correspond to concrete objects allocated in missing code, which enable us to soundly and precisely compute client relations that refer only to available code (e.g., aliasing and concrete types).

We implement our eventually sound points-to analysis in a tool called OPTIX, which analyzes Android apps treating the entire Android framework as missing code. We show that our instrumentation typically incurs low overhead—the median overhead is 4.3%, the overhead is less than 20% for more than 90% of the apps in our benchmark, and the highest is about 50%. The overhead of the outliers can be reduced as described above; in particular, only a few manually provided specifications are needed to reduce the overhead of the outliers to reasonable levels (see Section 8.1). Furthermore,
void main() { // program
String str = mkStr();
List list = new List(); // o_list
list.set(str);
Object data = list.get();
if(randBool()) {
    Object dataCopy = data;
    sendHttp(dataCopy); }}

String mkStr() { // library
    String libStr = new String(); // o_str
    return libStr; }

void sendHttp(String str) { // library
    ... }

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

Fig. 1. Program main (left) calls various library functions, for which the analyst provides specifications (right). Abstract objects o_list and o_str are labeled in comments.

We show that the instrumentation can be used to detect missing points-to specifications in the information flow client from [Feng et al. 2014], where specifications are used to model missing code. Empirically, we can detect missing specifications that are relevant to the information flow client. In summary, our contributions are:

• We propose an eventually sound points-to analysis for programs with calls to missing code that is also precise and automatic (Section 3). In particular, our analysis adds runtime instrumentation in the available code that detects and reports counterexamples, and can guarantee that soundness is never compromised (i.e., malicious functionality never gets executed) by terminating execution as soon as a counterexample is detected.

• We minimize instrumentation to reduce runtime overhead (Section 3) and introduce proxy objects to handle allocations in missing code (Section 4).

• We implement Optix, a points-to analysis for Android apps that treats the entire Android framework as missing.

• We show that the instrumentation overhead is manageable (Section 8), and that Optix can detect missing specifications relevant to the information flow client from [Feng et al. 2014].

The largest app in our benchmark has over 300K lines of Jimple code.

2 Overview

Consider the program in Figure 1. Suppose that a security analyst asks whether the program leaks the return value of mkStr to the Internet via a call to sendHttp, which requires knowing that str and dataCopy may be aliased. We use points-to analysis to determine which variables may be aliased. In particular, a points-to analysis computes points-to edge \( x \rightarrow o \) if variable \( x \) may point to a concrete object \( o \) allocated at allocation statement \( o \in O \) (called an abstract object) during execution. Two variables may be aliased if they may point to the same abstract object. Our example program exhibits points-to edges such as list \( \leftarrow o_{\text{list}} \), str \( \leftarrow o_{\text{str}} \), and dataCopy \( \leftarrow o_{\text{str}} \), so the points-to analysis concludes that str and dataCopy may be aliased.

Suppose that the library code is missing. For example, static analyses for Android apps often have difficulty analyzing Android framework code. In particular, the framework code makes substantial use of native code and reflection, which are too difficult to analyze, and are thus unsoundly ignored by most state-of-the-art static analyses [Arzt et al. 2014; Bastani et al. 2015b; Zhu et al. 2013], and because it uses deep call hierarchies, which can cause significant imprecision, since static points-to analyses often have limited context sensitivity [Bastani et al. 2018]. Thus, the Android framework is missing from the perspective of the static analyses.

For many clients (including static information flow analysis), it suffices to compute edges for visible variables \( x \in V_P \) in the available code; however, these edges often depend on relationships
in the missing code. Pessimistically assuming that missing code can be arbitrary is very imprecise, e.g., we may have \( \text{data} \leftarrow o_{\text{list}} \) in case the implementation of \text{get} is return this. Alternatively, optimistically assuming that missing code is empty can be unsound, for example, failing to compute \( \text{data} \leftarrow o_{\text{str}} \) and \( \text{dataCopy} \leftarrow o_{\text{str}} \). Such dynamic points-to edges that are not computed statically are missing.

A typical approach in practice is to provide specifications, which are code fragments that overapproximate the points-to behaviors of library functions. Examples of specifications are shown in Figure 1. For instance, because our static points-to analysis collapses arrays into a single field, we can overapproximate array of elements stored by the \text{List} class as a single field \( f \).

Suppose that the analyst has provided specifications for frequently used library functions such as \( \text{mkStr} \) and \text{sendHttpGet}, but a long tail of specifications remain missing, including those for \text{add} and \text{get}. Therefore, the (optimistic) static information flow analysis incorrectly concludes that \( \text{dataCopy} \) cannot point to \( \text{str} \), and that \( \text{mkStr} \) therefore does not leak to the Internet. Furthermore, dynamic information flow control cannot be applied since the missing code cannot be instrumented without modifying every end user’s Android installation.

Our analysis instruments the Android app to detect whether counterexamples to the optimistic assumption that every missing specification is empty; this instrumentation only inserts runtime checks in the available code. The instrumented app is published on Google Play. If the instrumentation observes that a counterexample occurs during an execution, then the counterexample is reported back to Google Play, which recomputes the static analysis to account for this new information.

Our example program \text{main} is instrumented to record the concrete objects pointed to by \text{libStr} and \text{data}. When the program is run:

- The variable \text{libStr} points to concrete object \( \bar{o}_{\text{str}} \), so our analysis concludes that \( \bar{o}_{\text{str}} \) is allocated at \( o_{\text{str}} \).
- The variable \text{data} points to \( o_{\text{str}} \), so our analysis concludes that \( \text{data} \leftarrow o_{\text{str}} \) and reports this counterexample.

Upon receiving this report, we add \( \text{data} \leftarrow o_{\text{str}} \) to the known counterexamples.

Given a new counterexample \( x \leftarrow o \), the static analysis at the very least learns that \( x \leftarrow o \) is a points-to edge that may occur. There are two ways in which the static analysis can generalize from this fact. First, it can compute additional missing points-to edges that are consequences of this fact according to the rules of the static analysis. For example, given the counterexample \( \text{data} \leftarrow o_{\text{str}} \), our static analysis additionally computes its consequence \( \text{dataCopy} \leftarrow o_{\text{str}} \), and determines that \( \text{str} \) and \( \text{dataCopy} \) may be aliased. Thus, the security analyst learns that the return value of \( \text{mkStr} \) may leak to the Internet, and can report any newly discovered bugs to the developer. In this case, the leak is discovered even if \( \text{randBool} \) returns false and the data is not leaked in that specific execution.

Second, the static analysis can also attempt to use specification inference to try and identify which missing specification may have been the “cause” of the missing points-to edge. By doing so, the static analysis generalizes the counterexample to eliminate unsoundness when analyzing future apps. In Section 5, we show how our tool leverages an existing specification inference algorithm to automatically infer candidate specifications that “explain” the counterexample. For example, given counterexample \( \text{data} \leftarrow o_{\text{str}} \), the specification inference algorithm would infer the specifications for \text{add} and \text{get} shown in Figure 1. One caveat is that the inferred specifications must be validated by a human, since it is impossible to guarantee that they are correct. We show in practice, the inference algorithm has high accuracy.

Next, we describe how our analysis instruments apps to detect missing points-to edges. Naïvely, we could use a dynamic points-to analysis, which instruments every allocation, assignment, load,
and store operation in the program to determine all of the dynamic points-to edges that occur during an execution. However, this approach requires far more instrumentation than necessary. In particular, suppose that multiple counterexamples occur during an execution; to be eventually sound, the instrumentation only has to detect the first one that occurs during execution. Leveraging this property enables us to substantially reduce the required instrumentation. For example, note that the missing points-to edge dataCopy ← ostr can only occur during execution where the missing points-to edge data ← ostr has already occurred. Furthermore, once data ← ostr has occurred, it is added to the static analysis, which computes dataCopy ← ostr as a consequence. Therefore, we never need to detect or report dataCopy ← ostr.

Another challenge with the instrumentation is how to handle allocations in missing code. For example, if the specification for mkStr were also missing, then our analysis cannot instrument LibStr to determine that δstr was allocated at ostr. Nevertheless, we can reason about such missing abstract objects based on observations in available code. In particular, suppose we instrument str and list. During execution, this instrumentation detects that str points to a concrete object δstr. Since δstr was not allocated at list, it must have been allocated in mkStr. We represent this fact by introducing a proxy object p_mkStr pointed to by the return value r_mkStr of mkStr. We discuss proxy objects in Section 4.

Finally, we describe an eventually sound points-to analysis, but more work is needed to ensure that the information flow client itself is eventually sound. We describe a candidate eventually sound information flow analysis in Section 9; evaluating this analysis is beyond the scope of our work.

3 EVENTUALLY SOUND POINTS-TO ANALYSIS

We describe our eventually sound points-to analysis, summarized in Figure 3.

3.1 Background and Assumptions

Consider a program P (whose code is available) containing calls to functions in a library L (whose code is missing). There are five kinds of statements: allocations (x ← X(), where X ∈ C is a class), assignments (x ← y, where x, y ∈ V_P are program variables), loads (x ← y.f, where f ∈ F is a field), stores (x.f ← y), and calls to library functions m ∈ M library (x ← m(y)). We omit control flow statements since our static analysis is flow-insensitive. We let p_m (resp., r_m) denote the parameter (resp., return value) of library function m. For convenience, we assume that each library function has exactly one argument, and that there are no functions in P.

Our static points-to analysis, described in Figure 2, is a standard flow- and context-insensitive analysis for computing points-to edges Π ⊆ V_P × O [Andersen 1994]; we describe how our results can be extended to context- and object-sensitive analyses with on-the-fly callgraph construction in Section 6.4. Rules 1-3 handle the semantics of each kind of statement. A function call x ← m(y) is
treated as an assignment of $y$ to the parameter $p_m$ and an assignment of the return value $r_m$ to $x$.

Rule 4 handles known counterexamples $\Pi_{\text{miss}} \subseteq \mathcal{V}_P \times O$.

We initially make three simplifying assumptions. First, we assume that library functions do not contain allocations; we remove this assumption in Section 4. Second, we make the disjoint fields assumption, which says that $\mathcal{F}_L \cap \mathcal{F}_P = \emptyset$, where $\mathcal{F}_L$ (resp., $\mathcal{F}_P$) are fields accessed by the library (resp., program), i.e., there are no shared fields $f \in \mathcal{F}_L \cap \mathcal{F}_P$. We discuss how to weaken this assumption in Section 6.1. Third, the programs we consider do not have callbacks; we discuss how to handle callbacks in Section 6.2.

### 3.2 Eventual Soundness

We first define soundness relative to an execution:

**Definition 3.1.** A points-to set $\Pi$ is sound relative to an execution $e$ if no counterexamples occur, i.e., there is no dynamic points-to edge $x \leftarrow o \notin \Pi$.

Consider a points-to analysis that for a sequence of instrumented executions $e_1, e_2, \ldots$ computes a sequence of points-to sets $\Pi_1, \Pi_2, \ldots$, both indexed by the natural numbers $i \in \mathbb{N}$. Here, $\Pi_i$ is computed as a function of the previous points-to set $\Pi_{i-1}$ and counterexample from $e_i$. Note that the instrumentation for $e_{i+1}$ can be chosen adaptively based on $\Pi_i$ and that $\Pi_{i+1} \subseteq \Pi_j$ if $i \leq j$.

**Definition 3.2.** The points-to analysis is eventually sound if for any sequence $e_1, e_2, \ldots$ of executions, (i) for any execution $e_i$, the instrumentation detects and reports the first counterexample that occurs during executing (if any), and (ii) there exists $n \in \mathbb{N}$ such that $\Pi_n$ is sound with respect to $e_i$ for every $i \geq n$.

Intuitively, the first property says that counterexamples are detected as soon as they occur (which ensures that any bugs or malicious behaviors are detected as soon as they occur), and the second property says that only a finite number of executions will report counterexamples.

**Definition 3.3.** The points-to analysis is precise if for every $i \in \mathbb{N}$, the points-to set $\Pi_i$ is a subset of the points-to set computed by analyzing the implementation of the missing code.

Note that while progress towards soundness is guaranteed, it is not possible to report how many sources of unsoundness remain at any point in time; in general, even if all program paths are executed, there may be missing points-to edges. For example, in the following code, suppose that `randInt` never evaluates to 0, the points-to edge $y \leftarrow o$ remains missing:

```java
void main() { // program
    Object foo(Object ob) { // library
        Object[] arr = new Object[2];
        arr[0] = ob;
        return arr[randInt()];
    }

    Object x = new Object(); // o
    Object y = foo(x); }
```

Despite the inability to quantify progress, the property of eventual soundness is useful, since it guarantees that only a finite number of counterexamples can possibly occur. In particular, this property implies that the number of counterexamples reported must decrease over time (eventually to zero). For example, suppose we try to construct an eventually sound interval analysis for a program $x \leftarrow m()$ with an integral variable $x$ by abstracting a set of counterexamples with the smallest interval that contains all the counterexamples. Such an analysis is not eventually sound. On the other hand, an analysis that abstracts counterexamples with $(-\infty, \infty)$ is sound and therefore (vacuously) eventually sound. Finally, the former analysis is eventually sound (but not precise) if after $n$ counterexamples, the analysis outputs $(-\infty, \infty)$.

Also, it is permissible for a counterexample to simply never occur in any execution, e.g., in the above code, if the call to `randInt` in `foo` always returns 1, then the counterexample $y \leftarrow o$ never
occurs during any execution. However, eventual soundness is still satisfied since soundness is
defined relative to the sequence of observed executions: if a counterexample exists but is never
observed, then the analysis is still sound for all executions that are observed.

3.3 Naïve Algorithm

We first describe a naïve eventually sound points-to analysis. Recall that we cannot compute points-
to edges for variables in missing code—our analysis only computes edges for visible variables in
the program.

Optimistic analysis. We use the static analysis in Figure 2 to compute static points-to edges
\( \Pi \), assuming that calls to library functions are no-ops—in particular, the set of counterexamples is
initially empty, i.e., \( \Pi_{\text{miss}} \leftarrow \emptyset \).

Runtime checks. A monitor is instrumentation added to a statement \( x \leftarrow * \) (where \( * \) stands for
any valid subexpression). After executing this statement, the monitor issues a report \( (x \leftarrow *, \bar{o}) \), i.e.,
it records the value of the concrete object \( \bar{o} \) pointed to by \( x \) after executing \( x \leftarrow * \). A monitoring
scheme \( M \) is a set of statements in the program to be monitored. Our goal is to design monitoring
schemes that satisfy the following property:

Definition 3.4. We say a monitoring scheme \( M \) is sound if for any execution, \( M \) reports the first
counterexamples that occurs (if any), and we say \( M \) is precise if it only reports counterexamples,
i.e., it does not report false positives.

Naïvely, it is sound and precise to monitor every variable \( x \in V_p \). Then, we can map each
concrete object \( \bar{o} \) to its allocation:

Definition 3.5. An abstract object mapping for an execution is a mapping \( \bar{o} \sim o \), where \( \bar{o} \) is a
concrete object allocated at abstract object \( o \).

For every report \( (x \leftarrow X(), \bar{o}) \), our analysis adds \( \bar{o} \sim o \) to the abstract object mapping, where
\( o = (x \leftarrow X()) \). Then, for every report \( (x \leftarrow *, \bar{o}) \) and \( \bar{o} \sim o \), we conclude that \( x \leftarrow o \) occurred
dynamically, and if missing, report it as a counterexample. In our example, our analysis monitors
libStr, detects that \( o_{\text{str}} \sim o_{\text{str}} \), and reports the counterexample data \( \leftarrow o_{\text{str}} \).

Updating the static analysis. We add every reported counterexample to \( \Pi_{\text{miss}} \). Our static
analysis in Figure 2 adds \( \Pi_{\text{miss}} \) to \( \Pi \) and computes the consequences of these added edges. Continuing
our example, our static analysis adds data \( \leftarrow o_{\text{str}} \) to \( \Pi \) (rule 4), and computes its consequence
dataCopy \( \leftarrow o_{\text{str}} \) (rule 2).

Guarantees. Let \( \Pi^* \) be the points-to edges computed using \( \Pi_{\text{miss}}^* = \Pi^*_{\text{miss}} \), where \( \Pi^*_{\text{miss}} \) is the
set of all missing points-to edges. Then, our analysis is:

- Eventually sound: Since we monitor every variable and abstract object, we are guaranteed
to detect any counterexample, including the first to occur during execution. Furthermore,
since \( \Pi^*_{\text{miss}} \) is finite, only finitely many counterexamples are ever reported; thus, \( \Pi \) is sound
for all executions following the last reported counterexample.
- Precise: Any sound set of points-to edges \( \Pi' \) must contain the missing points-to edges \( \Pi^*_{\text{miss}} \).
Therefore, \( \Pi_{\text{miss}} \subseteq \Pi^*_{\text{miss}} \subseteq \Pi' \), so \( \Pi \subseteq \Pi^* \subseteq \Pi' \).
- Automatic: Our static analysis requires no human input.

In our example, the static points-to set \( \Pi \) is sound after the counterexample data \( \leftarrow o_{\text{str}} \) is reported,
since the static analysis then computes the remaining missing points-to edge dataCopy \( \leftarrow o_{\text{str}} \).

### Data Structure

<table>
<thead>
<tr>
<th>monitors</th>
<th>Reports</th>
<th>Abstract Object Mapping</th>
<th>Missing Points-to Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>(allocation) $M_{\text{alloc}} = O_P$</td>
<td>(allocation) $(x \leftarrow X(), \ x \leftarrow \delta) \in R_{\text{alloc}}$</td>
<td>(program abstract objects) $(x \leftarrow X(), \delta) \in R_{\text{alloc}}$</td>
<td>$(x \leftarrow m(y), \ x \leftarrow \delta) \in R_{\text{call}}$</td>
</tr>
<tr>
<td>(function call) $x \leftarrow m(y)$</td>
<td>(function call) $x \leftarrow m(y)$</td>
<td>(proxy objects) $(s, \delta) \not\in R_{\text{alloc}}$, $(x_1 \leftarrow m_1(y_1), \delta) \in R_{\text{call}}$, ...</td>
<td>$(x \leftarrow m(y), \delta) \in R_{\text{call}}$, $\delta \sim p = {m_1, \ldots}$</td>
</tr>
<tr>
<td>$x \leftarrow m(y)$</td>
<td></td>
<td></td>
<td>$x \leftarrow o \in \Pi_{\text{miss}}$</td>
</tr>
</tbody>
</table>

---

**Fig. 3.** Given a program $P$, Optix adds monitors to $P$. It uses reports issued by these monitors during executions to compute the counterexamples $\Pi_{\text{miss}}$, which the static analysis in Figure 2 uses to compute optimistic points-to edges $\Pi \subseteq V_P \times (O_P \cup \mathcal{P})$.

### 3.4 Optimized Monitoring

We now describe how to reduce monitoring.

**Restricting to function calls.** Recall that monitoring dataCopy is unnecessary—the missing points-to edge dataCopy $\leftarrow o_{\text{str}}$ is computed by the static analysis once data $\leftarrow o_{\text{str}}$ is reported, so it suffices to monitor data. In general, it suffices to monitor function calls and allocations:

**Proposition 3.6.** The monitoring scheme $M_{\text{min}} = M_{\text{alloc}} \cup M_{\text{call}}$ is sound and precise, where $M_{\text{alloc}} = O$ and $M_{\text{call}} = \{x \leftarrow m(y) \mid m \in M\}$.

We give a proof in Appendix A.1. Figure 3 shows our algorithm using $M_{\text{min}}$.

**Restricting to leaked abstract objects.** We can further reduce the size of $M_{\text{alloc}}$. In particular, library functions can only access abstract objects reachable from the parameter $y$ of a call $x \leftarrow m(y)$, which implies that the return value $r_m$ can only point to such an abstract object $o$. Thus, it suffices to restrict $M_{\text{alloc}}$ to include abstract objects that may leak into missing code. In fact, it is even sound to use the monitoring scheme $\tilde{M}_{\text{alloc}}$, which monitors allocation statements $o$ such that $o$ may be explicitly passed to the library via a function call $x \leftarrow m(y)$, where $y \leftarrow o$:

$$\tilde{M}_{\text{alloc}} = \{o \in O \mid y \leftarrow o \in \Pi \text{ where } x \leftarrow m(y)\}.$$
This monitoring scheme is subtler than the schemes described previously, since the monitors \( \tilde{M}_{\text{alloc}} \) depend on the current points-to edges \( \Pi \). Therefore, the instrumentation may need to be updated when counterexamples are reported and \( \Pi \) is updated. In particular, if \( y \leftarrow o \) is newly added to \( \Pi \), where \( x \leftarrow m(y) \), then \( o \) is added to \( \tilde{M}_{\text{alloc}} \) so the instrumentation must be updated. We can soundly use \( \tilde{M}_{\min} = \tilde{M}_{\text{alloc}} \cup \tilde{M}_{\text{call}} \):

**Proposition 3.7.** The monitoring scheme \( \tilde{M}_{\min} \) constructed using the current points-to edges \( \Pi \) is sound and precise.

We give a proof in Appendix A.2. Since the number of possible counterexamples is still finite, at some point no further counterexamples are reported. By Proposition 3.7, no counterexamples occur in any subsequent executions, i.e., \( \Pi \) is sound for all subsequent executions.

**Minimality.** Our monitoring scheme is minimal in the following sense:

**Proposition 3.8.** Assume that the rules used to compute \( \tilde{M}_{\text{alloc}} \) do not generate any false positives, i.e., for every allocation \( o \in \tilde{M}_{\text{alloc}} \), there exists an execution during which a concrete object allocated at \( o \) is passed as an argument to a library function. Then, for any strict subset \( M \subset \tilde{M}_{\min} \), there exist implementations of the library and program executions such that \( M \) fails to report a counterexample, i.e., using \( M \) is not eventually sound.

In other words, our monitoring scheme is minimal except for potential imprecision when computing \( \tilde{M}_{\text{alloc}} \). We give a proof in Appendix A.3.

## 4 ABSTRACT OBJECTS IN THE LIBRARY

We now remove the assumption that no allocations occur inside missing code.

### 4.1 Proxy Objects

Suppose that allocations can occur inside library code. Let \( O = O_P \cup O_L \), where abstract objects in \( O_P \) are in available code and abstract objects in \( O_L \) are in missing code. Then, our analysis cannot compute points-to edges \( x \leftarrow o \), where \( o \in O_L \). As described previously, we assume that the static analysis only needs to compute relations involving program values. However, points-to edges \( x \leftarrow o \in V_P \times O_L \) (i.e., \( x \) is in the program but \( o \) is not) are often needed to compute relations between program variables, e.g., aliasing and concrete types.

For example, in Figure 1, if \( \text{mkStr} \) is missing, then \( o_{\text{str}} \) is missing, so our static analysis cannot compute \( \text{str} \leftarrow o_{\text{str}} \) (among others). Furthermore, we do not assume the ability to instrument missing code, so we cannot dynamically detect these points-to edges. However, this points-to edge is needed to determine that \( \text{str} \) may have type \( \text{String} \), and that \( \text{str} \) and \( \text{data} \) may be aliased.

We handle allocations in library code by constructing the following:

**Definition 4.1.** A proxy object mapping \( \phi \) maps \( \hat{o} \rightsquigarrow p \), where \( \hat{o} \) is a concrete object allocated in missing code, and \( p = \phi(\hat{o}) \in \mathcal{P} \) is a fresh abstract object called a *proxy object*; here, \( \mathcal{P} \) is the set of all proxy objects.

In other words, \( \phi \) is the abstract object mapping for concrete objects allocated in missing code. We describe how to construct \( \phi \) and \( \mathcal{P} \) below.

Given \( \phi \), our analysis proceeds as before. It makes optimistic assumptions, initializes \( \Pi_{\text{miss}} \leftarrow \emptyset \), and instruments the program using the monitoring scheme \( \tilde{M}_{\min} \) defined in Proposition 3.6. For any report \((x \leftarrow e, \hat{o})\), if \( \hat{o} \) is not allocated at a visible allocation, our analysis concludes that \( \hat{o} \) must have been allocated in missing code, so it adds \( \hat{o} \rightsquigarrow p = \phi(\hat{o}) \) to the abstract object mapping. Now, if a detected dynamic points-to edge \( x \leftarrow p \) is missing, it is reported as a counterexample and...
added to $\Pi_{\text{miss}} \subseteq \mathcal{V}_P \times (O_P \cup \mathcal{P})$, and $\Pi$ is recomputed using the static analysis in Figure 2. As long as $\mathcal{P}$ is finite, then this approach is eventually sound, since there can only be a finite number of counterexamples $x \leftarrow p$.

In our example, $\text{str}$ is monitored since $\text{mkStr}$ is missing. Upon execution, our instrumentation detects $\text{str} \leftarrow \delta_{\text{str}}$, and determines that $\delta_{\text{str}}$ (allocated at $\delta_{\text{str}}$) is allocated in missing code. Supposing that $p_{\text{str}} = \phi(\delta_{\text{str}}) \in \mathcal{P}$, our analysis adds $\delta_{\text{str}} \leadsto p_{\text{str}}$ to the abstract object mapping. Thus, our instrumentation reports the counterexample $\text{str} \leftarrow p_{\text{str}}$. Assuming execution continues, then our instrumentation additionally reports the counterexample data $\leftarrow p_{\text{str}}$. Both counterexamples are added to $\Pi_{\text{miss}}$, from which our static analysis computes $\text{dataCopy} \leftarrow p_{\text{str}} \in \Pi$.

We now discuss how to construct $\phi$ and $\mathcal{P}$. The relevant information characterizing a concrete object is the following:

**Definition 4.2.** The dynamic footprint of a concrete object $\delta$ is the set of all visible variables that ever point to $\delta$ during an execution.

The concrete type of $\delta$ may also be available to the static analysis, which we discuss in Section 6.3. Aside from concrete types, the dynamic footprint contains all information about $\delta$ available to the static analysis, namely, the visible variables that point to $\delta$.

Then, the proxy object mapping $\phi$ should map each concrete object $\delta$ to a proxy object $p$ so that the corresponding static footprint $\{x \in \mathcal{V}_P \mid x \leftarrow p \in \Pi\}$ soundly overapproximates the dynamic footprint of $\delta$ as precisely as possible. This way, clients of the points-to analysis can be eventually soundly and precisely computed (as long as they only depend on available information), e.g., it ensures that aliasing for program variables is eventually soundly and precisely computed (concrete types are eventually soundly and precisely computed using a simple extension; see Section 6.3).

On the other hand, $\phi$ should also avoid introducing unnecessary proxy objects, or else more executions may be required for the analysis to become sound. Two extremes highlight these opposing desirable properties:

- **Unbounded $\mathcal{P}$:** Map each concrete object to a fresh proxy object $\phi(\delta) = p_0$.
- **Singleton $\mathcal{P}$:** Map each concrete object to a single proxy object $\phi(\delta) = p$.

On the one hand, if we use a fresh proxy object for every concrete object, then there would be an unbounded number of proxy objects, which would mean our algorithm is no longer eventually sound (since there may be an unbounded number of missing points-to edges). Alternatively, using a single proxy object can be very imprecise; for example, for any pair of calls $x \leftarrow m(y)$ and $x' \leftarrow m'(y')$, our analysis concludes that $x$ and $x'$ may be aliased.

We first describe an ideal proxy object mapping, which constructs $\mathcal{P}$ as the set of possible dynamic footprints, and constructs $\phi$ to map $\delta$ to its dynamic footprint. Points-to sets computed using any static analysis together with the ideal proxy object mapping satisfy the above property, i.e., that the static footprints soundly overapproximate the dynamic footprints as precisely as possible.

Because the static analysis is flow-insensitive, the ideal proxy mapping is actually more precise than necessary. Therefore, our analysis uses a coarser proxy object mapping computed by our analysis, which essentially restricts the dynamic footprint to function return values. Finally, we show that this coarser proxy object mapping is as precise as the ideal proxy object mapping for our points-to analysis described in Figure 2.

### 4.2 Ideal Proxy Object Mapping

Our “ideal” construction of proxy objects exactly captures dynamic footprints:

**Definition 4.3.** An ideal proxy object $\hat{p} \in \hat{\mathcal{P}} = \wp^{\mathcal{V}_P}$ is a set of visible variables. The ideal proxy object mapping $\hat{\phi}(\hat{\delta}) \in \hat{\mathcal{P}}$ is the dynamic footprint of $\hat{\delta}$.
For a concrete object \( \bar{o} \) allocated in missing code, we can compute \( \tilde{\phi}(\bar{o}) \) by monitoring all visible variables and identifying all visible variables that ever point to \( \bar{o} \). In our example, suppose that we continue executing main even if a counterexample is detected and reported. Furthermore, suppose that the concrete object \( \bar{o}_{str} \) is allocated at missing abstract object \( o_{str} \) in an execution where \( \text{randBool} \) returns false. Then, \( \phi \) maps \( o_{str} \) to ideal proxy object \( p_{str} = \{ \text{str}, \text{data} \} \). The reported counterexamples

\[
\Pi_{\text{miss}} = \{ \text{str} \leftarrow \bar{p}_{str}, \text{data} \leftarrow \bar{p}_{str} \}
\]

are added to our static analysis, which additionally computes \( \text{dataCopy} \leftarrow \bar{p}_{str} \).

Let \( \Pi_{\text{miss}} \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{P}) \) be the missing points to edges when using ideal proxy objects, and let \( \Pi^* \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \tilde{\mathcal{P}}) \) be the points-to edges computed using \( \Pi_{\text{miss}} = \Pi^* \). Then:

**Proposition 4.4.** If \( x \leftarrow \bar{o} \) occurs during execution and \( \bar{o} \) is allocated at abstract object \( o \), then \( x \leftarrow o \in \Pi^* \) (if \( o \in \mathcal{O}_P \)) or \( x \leftarrow \tilde{p} \in \Pi^* \) (where \( \tilde{p} = \tilde{\phi}(\bar{o}) \)).

In other words, clients of the points-to analysis that only refer to program variables are eventually sound. For example, if two program variables \( x \) and \( y \) may be aliased, then there must be some execution in which they both point to a concrete object \( \bar{o} \). Then, our analysis finds points-to edges \( x \leftarrow \tilde{p} \) and \( y \leftarrow \tilde{p} \), where \( \tilde{p} = \tilde{\phi}(\bar{o}) \), so the alias analysis determines that \( x \) and \( y \) may be aliased.

Also:

**Proposition 4.5.** Let \( \Pi \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{L}_P) \) be the points-to set computed using the static analysis in Figure 2 with all code available (and \( \Pi_{\text{miss}} = \emptyset \)). For \( o \in \mathcal{O}_P \), if \( x \leftarrow o \in \Pi^* \), then \( x \leftarrow o \in \Pi \). For \( \tilde{p} = \tilde{\phi}(\bar{o}) \in \tilde{\mathcal{P}} \), if \( x \leftarrow \bar{o} \in \Pi^* \), then \( x \leftarrow o \in \Pi \), where \( o \) is the statement where \( \bar{o} \) was allocated.

In other words, \( \Pi^* \) is at least as precise as the points-to edges \( \Pi \) computed with all code available. We prove these two propositions in Appendix B.2. In our example, with all code available, we compute \( \text{str} \leftarrow o_{str}, \text{data} \leftarrow o_{str}, \) and \( \text{dataCopy} \leftarrow o_{str} \), which is equivalent to \( \Pi^* \) (replacing \( \bar{p}_{str} \) with \( o_{str} \)).

### 4.3 Proxy Object Mapping

The ideal proxy object mapping is more precise than necessary. Continuing our example (where we assume execution continues even after counterexamples are detected and reported), consider a second execution where \( \text{randBool} \) returns true. Then, the concrete object \( \bar{o}'_{str} \) allocated at missing abstract object \( o_{str} \) is mapped to the ideal proxy object \( \tilde{p}'_{str} = \{ \text{str}, \text{data}, \text{dataCopy} \} \). However, the static footprint of \( \tilde{p}' \) equals that of \( \tilde{p} \) (from the first execution, where \( \text{randBool} \) returns false), even though \( \tilde{p} \neq \tilde{p}' \) — i.e., \( \bar{o}_{str} \) and \( \bar{o}'_{str} \) map to different ideal proxy objects, but their relevant points-to behaviors appear identical to the (flow-insensitive) static analysis. In fact, all information about a concrete object available to the static analysis can be summarized by the following:

**Definition 4.6.** The *dynamic function footprint* of a concrete object \( \bar{o} \) is the set of library functions \( m \in \mathcal{M} \) such that \( r_m \leftarrow \bar{o} \) during execution.

Now, we use the following proxy object mapping:

**Definition 4.7.** A *proxy object* \( p \in \mathcal{P} = 2^\mathcal{M} \) is a set of library functions. The *proxy object mapping* \( \phi(\bar{o}) \in \mathcal{P} \) is the dynamic function footprint of \( \bar{o} \).

To compute \( \phi \), it suffices to monitor calls \( x \leftarrow m(y) \) to missing functions. Continuing our example, \( \phi \) maps the concrete object \( \bar{o}_{str} \) allocated at missing abstract object \( o_{str} \) to \( p_{str} = \{ \text{mkStr}, \text{get} \} \) regardless of the return value of \( \text{randBool} \). If \( \text{randBool} \) returns true, then

\[
\Pi_{\text{miss}} = \{ \text{str} \leftarrow p_{str}, \text{data} \leftarrow p_{str}, \text{data} \leftarrow p_{str} \},
\]
The second step involves computing the static analysis using a shortest-path style algorithm. When \( x \rightarrow o \) returns false, then \( \Pi_{\text{miss}} = \{ \text{str} \leftarrow p_{\text{str}}, \text{data} \leftarrow p_{\text{str}} \} \), from which our static analysis also computes \( \text{dataCopy} \leftarrow p_{\text{str}} \). The static footprint of \( p_{\text{str}} \) is the same either way, and also equals those of \( \tilde{p}_{\text{str}} \) and \( \bar{p}_{\text{str}} \).

Let \( \Pi_{\text{miss}}^* \subseteq V_P \times (O_P \cup \mathcal{P}) \) be the set of all missing points-to edges using proxy objects objects, and let \( \Pi^* \subseteq V_P \times (O_P \cup \mathcal{P}) \) be the points-to edges computed using \( \Pi_{\text{miss}} = \Pi_{\text{miss}}^* \). Then:

**Proposition 4.8.** For any abstract object \( o \in O_P \), \( x \leftarrow o \in \Pi^* \Leftrightarrow x \leftarrow o \in \Pi^* \). Furthermore, for any concrete object \( \tilde{o} \) allocated in missing code, letting \( \tilde{p} = \tilde{\phi}(\tilde{o}) \) and \( p = \phi(o) \), we have \( x \leftarrow \tilde{p} \in \Pi^* \Leftrightarrow x \leftarrow p \in \Pi^* \).

In other words, the points-to edges computed using our proxy object mapping is as sound and precise as using the ideal proxy object mapping. Therefore, using proxy objects is also sound and precise in the sense of Propositions 4.4 and 4.5. We prove this proposition in Appendix B.3.

Finally, the following result says that the monitoring scheme described in Section 3.4 is still sound (it follows since we can compute \( \phi \) using only \( M_{\text{call}} \)):

**Proposition 4.9.** The monitoring scheme \( \tilde{M}_{\text{min}} \) is sound and precise.

### 5 SPECIFICATION INFERENCE

Rather than simply adding reported missing points-to edges to \( \Pi_{\text{miss}} \), we can use them to infer specifications summarizing missing code, which transfers information learned from the counterexample to other calls to the same library function. We use the specification inference algorithm in [Bastani et al. 2015b]. Given a reported missing points-to edge \( x \leftarrow o \), this algorithm infers specifications in two steps:

- **Pessimistic assumptions:** Take \( \hat{m} = m_{\text{pess}} \) for every missing function \( m \in M \), for some function \( m_{\text{pess}} \) (see below), and run the static analysis using \( \hat{m} \) in place of \( m \).
- **Minimal statements:** Compute a minimal subset of pessimistic statements (i.e., statements in the functions \( m_{\text{pess}} \)) that are needed to compute \( x \leftarrow o \) statically; these statements are the inferred specifications.

The second step involves computing the static analysis using a shortest-path style algorithm. When computing the transitive closure according to the rules in Figure 2, a priority queue is used in place of a worklist, where the priority of each points-to edge in the queue is the number of pessimistic statements needed to derive it. In particular, each time a rule is applied in conjunction with a pessimistic statement, the priority of the derived points-to edge is one more than the sum of the priorities of points-to edges in the premise.

**Pessimistic function.** A key design choice is the pessimistic function \( m_{\text{pess}} \) to use. The choice in [Bastani et al. 2015b], which we term the general function \( m_{\text{gen}} \), is shown in Figure 4 (left). Using

```java
Object m_gen(Object ob) {
    while(true) {
        ob.f = ob; ob = ob.f;
    }
    return ob;
}

Object m_res(Object ob) {
    Object r;
    this.g = ob; r = ob; r = this; r = this.g;
    return r;
}
```

---

m_{gen} is sound assuming library functions do not access global fields or allocate objects. However, it results in a huge search space of candidate specifications, so the inference algorithm produces many incorrect specifications. Instead, we use pessimistic assumptions that restrict the search space to only consider candidate specifications that are common in practice, in particular, that (i) do not accesses deep field paths, (ii) only access receiver fields, and (iii) assume the receiver has a single field \( g \). These constraints lead to the restricted function shown in Figure 4 (right).

**Proxy object specifications.** We separately infer proxy object specifications of the form \( \langle X, \{ m \} \rangle \), where \( X \in C \) and \( m \in M \) is a library function. This specification says that a new object of type \( X \) is allocated onto the return value of a function. We infer a proxy object specification for any proxy object \( p \in P \) we observe dynamically such that the function footprint of \( p \) consists of a single function \( m \).

### 6 EXTENSIONS

#### 6.1 Shared Fields

In Section 3.1, we made the assumption that no shared fields \( f \in F_P \cap F_L \) exist. Our analysis handles a shared field \( f \) by converting stores \( x.f \leftarrow y \) and loads \( x \leftarrow y.f \) in the program into calls to setter and getter functions, respectively. To do so, we have to know which fields may be accessed by the library. We make the weaker assumption that the library does not access fields defined in the program—then, our analysis performs this conversion for every field \( f \) defined in the library that is accessed by the program.

#### 6.2 Callbacks

Android apps can register callbacks to be invoked by Android when certain events occur, e.g., the program can implement the callback \( \text{onLocationChanged} \), which is invoked when the user location changes. If callbacks are not specified, then the static analysis may unsoundly mark them as unreachable. We use the approach in [Bastani et al. 2015a] to eventually soundly compute reachable program functions. In particular, a potential callback, is a program function that overrides a framework function. Intuitively, potential callbacks are the functions “known” to the framework. For each potential callback \( m \) that is marked as unreachable by the static analysis, we instrument \( m \) to record whether \( m \) is ever reached. This algorithm is eventually sound since there are only finitely many potential callbacks. Also, the instrumentation eventually incurs no overhead—once no more counterexamples are reported, the instrumentation is never triggered.

In addition, some callbacks are passed parameters from the Android framework. For example, consider the code on the left:

```c
void onLocationChanged(Location loc) {
    Location copy = loc;
}
```

```c
void onLocationChanged() {
    Location loc = Location.getLocation();
    Location copy = loc;
}
```

Here, \( loc \) points to an abstract object \( o_{loc} \). In this case, \( o_{loc} \) is allocated in the framework, but it may also be allocated in program code. We must specify the abstract objects that \( loc \) may point to, or else our points-to analysis is unsound. The code on the right replaces the parameter with a call that retrieves \( loc \) from the framework, which is semantically equivalent to the code on the left. Thus, we can think of \( loc \) as a “return value” passed to \( \text{onLocationChanged} \); by Proposition 3.6, it suffices to monitor all callback parameters.

#### 6.3 Concrete Types

Some client analyses additionally need the concrete type \( X \in C \) of abstract objects \( o = (x \leftarrow X()) \), for example, virtual call resolution. To compute concrete types for proxy objects, each monitor

---

We instrument apps using the Smali assembler and disassembler [Gruver 2016]. To monitor a
server in batches (by default, once every 500ms), which post-processes it to compute missing
points-to edges and infer specifications. To obtain traces, we execute apps in the Android emulator
and use Monkey [Google 2016] to inject touch events. We measure overhead using the Android
profiler.

6.4 Context- and Object-Sensitivity
Our analysis extends to $k$-context-sensitive points-to analyses with two changes. First, the abstract
objects considered are typically pairs $o = (c, h)$, where $c$ is a calling context and $h$ is an allocation
statement, so monitors on allocation statements $x \leftarrow X()$ also record the top $k$ elements of the
current callstack. Second, the points-to edge typically keeps track of the calling context $d$ in which
a variable $v$ may point to abstract object $o$. Therefore, monitors on calls to missing functions
$x \leftarrow m(y)$ also record the top $k$ elements of the current callstack.

In particular, our analysis may (i) detect that $(d, v) \to \bar{o}$ (i.e., $d$ is the callstack when $v$ pointed to
$\bar{o}$), and (ii) $\bar{o} \sim (c, h)$ (i.e., $\bar{o}$ was allocated at statement $h$, and $c$ is the callstack when $\bar{o}$ was allocated).
Then, our analysis reports missing points-to edge $(d, v) \to (c, h)$. We use a 1-CFA points-to analysis
in our evaluation; in this case, the calling context is simply the function in which the allocation or
call to a missing function occurs. Our approach can be extended to handle object-sensitive analyses,
by including instrumentation that records the calling context (which now includes the value of the
receiver).

Finally, we can also handle on-the-fly callgraph construction—if a missing points-to edge $x \leftarrow o$
is reported, and there is a virtual function call $x.m()$ in the program, then the possible targets of
$x.m()$ are updated to take into account the concrete type of $o$. The instrumentation may need to be
updated based on this new information. Assuming the number of possible call targets is finite, this
approach is eventually sound.

7 IMPLEMENTATION
We have implemented our eventually sound points-to analysis, including all extensions described
in Section 6 (using a 1-CFA points-to analysis), for Android apps in a tool called Optix. The missing
code consists of Android framework methods, which we assume cannot be statically analyzed
(since the Android framework heavily uses native code and reflection) or instrumented (which
requires a custom Android installation). The static analysis framework we use predates Optix,
and uses hand-written specifications to model missing code. Specifications have only been written
for methods deemed relevant to a static information flow client—of the more than 4,000 Android
framework classes, only 175 classes have specifications. Framework methods without specifications
appear as missing code to our static analysis.

Optix instruments Android apps using our optimized monitoring scheme $\tilde{M}_{\text{min}}$. It computes
eventually sound points-to sets and infers specifications based on reported missing points-to edges.
We instrument apps using the Smali assembler and disassembler [Gruver 2016]. To monitor a
statement $x = \ldots$, we record (i) the value System.identityHashCode(x), which uniquely identifies
the concrete object pointed to by $x$, (ii) the concrete type $x.getClass()$ of $x$, and (iii) the method
containing the statement and the offset of that statement in the method. This data is uploaded to
a server in batches (by default, once every 500ms), which post-processes it to compute missing
points-to edges and infer specifications. To obtain traces, we execute apps in the Android emulator
and use Monkey [Google 2016] to inject touch events. We measure overhead using the Android
profiler.

We have implemented the points-to analysis, the monitoring optimization, and the specification
inference algorithm in a version of the Chord program analysis framework [Naik et al. 2006]
Fig. 5. The runtime overhead from recording data and the (compressed) size of the data generated in one hour. Each is divided into initial and worst-case. The "updated" overhead is obtained by adding specifications to the system to reduce monitoring, and "# specs" is the number of specifications added to do so. For each column, the table shows the largest six values and the median value across our benchmark.

Table: Recording Overhead (%)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Recording Overhead (%)</th>
<th>Data (MB/hr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initial</td>
<td>updated</td>
</tr>
<tr>
<td>1</td>
<td>50.0</td>
<td>31.0</td>
</tr>
<tr>
<td>2</td>
<td>46.8</td>
<td>17.6</td>
</tr>
<tr>
<td>3</td>
<td>39.2</td>
<td>6.7</td>
</tr>
<tr>
<td>4</td>
<td>30.6</td>
<td>6.9</td>
</tr>
<tr>
<td>5</td>
<td>28.3</td>
<td>19.7</td>
</tr>
<tr>
<td>6</td>
<td>19.9</td>
<td>-</td>
</tr>
<tr>
<td>median</td>
<td>4.3</td>
<td>-</td>
</tr>
</tbody>
</table>

modified to use Soot as a front end [Vallée-Rai et al. 1999]. The specification inference algorithm is based on shortest-path context-free reachability, described in [Bastani et al. 2015b]. We use a 1-CFA points-to analysis. As we discuss Section 6.4, using our more precise points-to analysis is eventually sound.

Finally, we use the information flow client proposed in [Feng et al. 2014], which uses specifications to model missing code. This tool performs a static information flow analysis based on a static points-to analysis. The information flow analysis is standard—it looks for paths from annotated sources (e.g., location, contacts, etc.) to annotated sinks (e.g., SMS messages, Internet, etc.) in the Android framework [Arzt et al. 2014; Fuchs et al. 2009]. All analyses are computed using BDDBDDB [Whaley and Lam 2004].

8 EVALUATION

We evaluate Optix on a benchmark of 73 Android apps, including battery monitors, games, wallpaper apps, and contact managers. These apps were obtained from a variety of sources, including malware from a major security company (primarily apps that leaked sensitive information such as location, contacts, SMS messages, etc.) and benign apps from Google Play Store. We omit 11 apps that fail to run on the standard Android emulator, leaving 62 apps. First, we use Optix to instrument each Android app and study the instrumentation overhead. Second, we show how Optix computes points-to edges over time, and show that the number of computed edges does not explode. Third, we show how our analysis can be used to improve an information flow client.

8.1 Instrumentation Overhead

We evaluate the runtime overhead of our monitoring scheme $\tilde{M}_{\text{min}}$ described in Section 3.4. Recall from Section 3.4 that our optimized instrumentation scheme may add instrumentation over time. We consider two settings:

- **Initial**: This configuration represents the instrumentation overhead for a new app using the current program analysis. In particular, we use the initial instrumentation scheme where $\tilde{M}_{\text{alloc}}$ is constructed with no known counterexamples (i.e., $\Pi_{\text{miss}} = \emptyset$). Also, we use all existing handwritten points-to specifications, representing the realistic scenario where some manually provided information is used in addition to automatic inference.

- **Worst**: This configuration represents the absolute upper bound on the overhead. In particular, we monitor apps using the worst-case instrumentation scheme where $\tilde{M}_{\text{alloc}}$ contains all
abstract objects that may leak into missing code. Furthermore, we remove all handwritten
points-to specifications.

We executed instrumented apps in a standard emulator using Monkey for one hour, and then used
our algorithm to compute points-to sets.

**Results.** We show the highest runtime overheads in Figure 5 (left), including the runtime
overhead from recording data and the amount of data generated in an hour, for both the initial
setting and the worst-case setting. Columns “updated” and “# specs” are discussed below. We plot
the runtime overhead of our recording instrumentation in Figure 6 (a), where the apps along the
x-axis are sorted according to the overhead in the worst-case setting.

**Discussion.** The overhead incurred by recording data is less than 5% for more than half of
the apps, showing that in most cases the automatically instrumented programs have acceptable
performance. Even in the worst case, more than half the apps have less than 10% overhead. Still,
there are outliers, with 5 apps incurring more than 20% overhead with initial instrumentation, and
in the worst-case, 9 apps incurred more than 20% overhead. Unsurprisingly, the high-overhead
outliers have instrumentation in inner loops of the app; in such cases the overhead can be reduced
(see below). Finally, the amount of data generated is very small. Even in the worst case, for all but
one of the apps, less than 1.0 MB of (compressed) data was generated in one hour. The median
amount of data generated is about 2.0 KB, which is negligible. Data can therefore be stored and
transmitted when the app is idle, so the overhead due to uploading data does not affect the user
experience.

**Reducing runtime overhead.** Any program where instrumentation is required in a tight inner
loop is particularly challenging for dynamic analysis. Standard sampling techniques can be used
to reduce overhead in these cases [Liblit et al. 2003]. Additionally, both $M_{\text{alloc}}$ and $M_{\text{call}}$
are smaller in size as specifications are added and reach zero when there are no missing specifications. For a
given program, we can test the program to determine which monitors are frequently triggered,
and compute which missing functions require specifications for these monitors to be removed.
Providing or inferring specifications for these functions would allow us to remove the expensive
monitors. We do so for the five apps with initial overhead greater than 20%. In Figure 5 (left), we
show both the number of specifications we added for that app (“# spec”) and the resulting overhead
(“updated”). For all but the top app, we were able to reduce the overhead below 20% by adding
specifications for at most 10 Android framework methods; again, the overhead can be reduced to
any desired level by adding more specifications.

**8.2 Reported Counterexamples**

Next, we evaluate how the computed points-to edges vary over time. In particular, we show that
the number of reported counterexamples does not explode over time—otherwise, the number of
counterexamples discovered in production may be unacceptably high. Furthermore, we show that
a tail of reported counterexamples continues to occur for some apps, which shows that running
instrumented apps in production is necessary. This experiment uses the worst-case setting where
all handwritten specifications have been removed.

**Counterexamples over time.** Figure 6 (b) shows the cumulative number of reported missing
points-to edges as execution progresses. More precisely, for each point in the execution trace
(x-axis), it shows what fraction of reported missing points-to edges were discovered before that

---

3We ran a small subset of apps on a real device and consistently measured smaller overhead; the emulator gives a coarser
measure of execution time that we round up.
As can be seen, a large fraction of reports are made early on, with about 65% of reports made within 20% of the execution trace. We expect the number of reported counterexamples to continue to converge over time, and should not grow substantially larger. However, the curve is still increasing by the end of the execution trace, which indicates that more missing points-to edges are still being reported. Therefore, it is important to continue monitoring these apps in production to detect additional counterexamples.

**Last discovered counterexample.** Figure 6 (c) shows the point in the execution during which the final reported missing points-to edge occurs. More precisely, for each point in the execution trace (x-axis), it shows the fraction of the apps for which the final reported missing points-to edge was reported before that point. This curve goes to $y = 1.0$ at $x = 1.0$, but we cut off apps that have reported missing points-to edges in the last 1% of execution.

A large fraction of apps (about 45%) report no counterexamples. About 10% of apps report no further counterexamples after the first 5% of the trace. At the opposite end of the trace, about 20% of apps have the final reported counterexample in the last 1% to 10% of the trace, and 20% have the final reported counterexample in the final 1% of the trace, so more counterexamples likely remain. Again, this shows that we must continue to monitor apps in production.
Eventually Sound Points-To Analysis with Missing Code

<table>
<thead>
<tr>
<th>App</th>
<th>Jimple LOC</th>
<th>Time (min.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C2B78</td>
<td>322K</td>
<td>3.38</td>
</tr>
<tr>
<td>b9ac05</td>
<td>268K</td>
<td>1.06</td>
</tr>
<tr>
<td>highrail</td>
<td>247K</td>
<td>1.49</td>
</tr>
<tr>
<td>game</td>
<td>174K</td>
<td>0.08</td>
</tr>
<tr>
<td>androng</td>
<td>170K</td>
<td>0.48</td>
</tr>
<tr>
<td>median</td>
<td>19K</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Fig. 7. Left: The number of specifications inferred using each $m_{res}$ and $m_{gen}$, and the number of proxy object specifications inferred. Right: Statistics for five largest apps used in our evaluation, including the number of Jimple lines of code (i.e., the intermediate representation used by Soot), and the running time of specification inference.

Proxy object sizes. Since there are an exponential number of possible proxy objects (in the number of missing functions), we could hypothetically continue to discover many new proxy objects over time. In Figure 6 (d), we show the sizes of the dynamic function footprints of the reported proxy objects. More precisely, we show the number of reported proxy objects (y-axis) for different dynamic function footprint sizes. As can be seen, the vast majority (85%) of reported proxy objects have four or fewer functions in their dynamic function footprint. While there is a long tail of proxy objects with large function footprint sizes, there is no exponential blowup in the number of proxy objects discovered, ensuring that the analysis does not diverge due to proxy objects.

8.3 Specification Inference and a Static Information Flow Client

Finally, we evaluate whether OPTIX benefits an information flow client. We first infer specifications using the the algorithm in Section 5, and then run the information flow client on various sets of specifications. The information flow analysis is standard—we look for paths from a set of annotated sources (e.g., location) to a set of annotated sinks (e.g., Internet) in the Android framework [Arzt et al. 2014; Bastani et al. 2015b; Feng et al. 2014; Fuchs et al. 2009]. We demonstrate that the inferred specifications enables client to discover more information flows. However, many of the information flows remain undiscovered because the dynamic analysis is an underapproximation, which again motivates the need to run instrumented apps in production.

Specification inference. We remove all points-to specifications from OPTIX, and then infer specifications from reported counterexamples. Figure 7 (left) summarizes the inferred specifications—a specification is correct if it exactly equals the existing specification (or the one we would have written). Using $m_{res}$ is substantially more accurate than using $m_{gen}$, which does not infer a single additional specification. Compared to existing specifications, we inferred 174 new points-to specifications, of which 160 were proxy object specifications.

Furthermore, in Figure 7 (right), we show statistics for the five largest apps in our benchmark along with the running time of the inference algorithm (the information flow analysis runs much faster than the inference algorithm). As can be seen, inference scales even to very large apps.

Static information flow client. In Figure 8, we report the number of information flows and the number of malicious apps detected with varying sets of specifications (one malicious app can exhibit multiple flows). If we assume that all points-to specifications are missing (“Empty”), then the information flow client does not identify any information flows, whereas using inferred specifications (“Inf.”) computes a small number of flows.

A more representative use case is where the analysis has an incomplete baseline consisting of the most commonly used specifications (“Base”). Our baseline contains specifications for the essential Android framework classes Bundle and Intent, for the commonly used data serialization...
Fig. 8. Comparison of different sets of specifications: “Base” includes the most frequently used specifications, “Inf.” includes the inferred specifications, and “Ex.” includes all handwritten specifications. For each set of specifications, we show the number of specifications in that set (“specs.”), the number of information flows computed using those specifications (“flows”), and the number of malicious apps identified, i.e., some malicious information flow was discovered (“malware”).

classes JSONArray, JSONObject, and BasicNameValuePair, and for a few methods in java.util. As can be seen from Figure 8, when using the baseline in conjunction with the inferred specifications, the analysis computes a considerable number of additional flows compared to using the baseline alone (39 vs. 3). The reason the inferred specifications are more beneficial in this setting is that an information flow usually depends on multiple specifications—if a single one of these specifications is missing, then the flow is missing.

Compared to the existing, handwritten specifications (“Ex.”), using inferred specifications (together with the baseline specifications) identifies almost a third of the information flows. However, random testing cannot reveal all malicious behavior, since malware developers often try to hide malicious behaviors by triggering them only in response to very specific events, for example, at a certain time [Mishra and Jha 2007]. Therefore, our instrumentation is necessary to ensure that we identify additional malicious behaviors as soon as or before they occur, thereby limiting potential damage. Note that we do not recover any new flows when combining inferred specifications with existing specifications—prior to our evaluation, we have already identified all specifications needed to recover flows in these apps.

Finally, an alternative way to evaluate the value of the inferred specifications, we consider omitting the inferred specifications from the set of inferred specifications. Doing so limits the information flow client to identify only 34 flows, which demonstrates that the inferred specifications are crucial for finding many of the information flows present in these apps.

9 DISCUSSION

Dynamically loaded code. Our approach can be used for dynamically loaded code—the dynamically loaded code is taken to be the missing code, and the code that loads the dynamically loaded code is the available code. We guarantee eventual soundness for points-to edges in the available code. If points-to edges for dynamically loaded code must be computed, then the loaded code can be reported to the static analysis, but the analysis is no longer eventually sound—infinitely many reports may be issued since infinitely many different code fragments may be loaded.

Eventual soundness for clients. Our approach is automatically eventually sound for client analyses that depend only on aliasing information and concrete types for visible program variables (e.g., callgraph resolution). In general, missing code can introduce unsoundness into the static information flow analysis beyond missing points-to edges. For example, consider the code

```java
void main() { // program
    int val = source();
    int valDup = add(val, 1);
    sink(valDup);
}
```

```java
int add(int x, int y) { // library
    return x+y;
}
```
which calls the missing function add. Even with a sound points-to analysis, the static analysis would not recover the taint flow from source to sink. Sources and sinks in missing code must be specified, since there is no way to detect whether calling missing code leaks information out of the system or introduces sensitive information into the system.

In general, we can perform eventually sound analysis for clients that are abstract interpretations with finite abstract domain (at least, satisfying the ascending chain condition) [Cousot and Cousot 1977], if the abstraction function $\alpha$ can be computed for values in the available code based on observations in the available code alone. In particular, for a call $y \leftarrow m(x)$, the concrete values of $x$ and $y$ are recorded. Then, we can construct a transfer function $f_m$ to be analyzed in place of $m$. Initially, $\bot = f_m(\alpha(x))$ for all $x$; whenever a previously unobserved relation $\alpha(y) = f_m(\alpha(x))$ is detected during execution, a report is issued and $f_m$ updated. Since the abstract domain is finite, only finitely many reports can be issued, so the analysis is eventually sound. Finally, we use our points-to analysis to handle aliasing.

The challenge with information flow is that the abstraction function cannot be computed from observations in the available code alone, since information flow is a property of the computation, not just the input-output values. It may be possible to use techniques such as multi-execution [Devriese and Piessens 2010], which keep pair of values $\langle x_{\text{private}}, x_{\text{public}} \rangle$ for each (visible) program variable $x$, where $x_{\text{private}}$ may depend on sensitive data whereas $x_{\text{public}}$ does not. For example, the value for program variable $val$ may be $\langle 14, 0 \rangle$, where $14$ is a sensitive value and $0$ is a public value. Then, we can execute add using both $x = 14$ and $x = 0$, and obtain return value $r_{\text{add}} = \langle 15, 1 \rangle$. Since these two values differ, we conclude that $r_{\text{add}}$ depends on the sensitive input $14$, and report that add transfers information from its argument $x$ to its return value $r_{\text{add}}$. Essentially, this approach transforms the program so the abstraction function becomes computable. Alternatively, existing techniques for specification inference such as [Bastani et al. 2015b] may be used to infer specifications describing how information flows through missing code.

### 10 RELATED WORK

**Program monitoring.** There has been work using runtime checks to complement static analysis. For instance, [Bodden et al. 2011] proposes to use dynamic information to resolve reflective call targets, and then instruments the program to report additional counterexamples. Similarly, [Bastani et al. 2015a] proposes to compute reachable code by inserting runtime checks to report counterexamples to optimistic assumptions, and [Flanagan 2006] uses a combination of static type checking and runtime checks to enforce type safety. Points-to analysis with missing code is far more challenging, because dynamic points-to analysis incurs unreasonable overhead [Clapp et al. 2015; Mock et al. 2001], and also requires instrumenting missing code.

Additionally, [Hirzel et al. 2007] uses dynamic information to complement static points-to analysis. However, their analysis is unreasonably imprecise for programs that make substantial use of native code, since they pessimistically assume returns from native code can point to arbitrary abstract objects. For demanding, whole-program clients such as static taint analysis, such imprecision generates a huge number of false positives, since every abstract object that leaks into missing code becomes aliased with every return value from missing code. Even with such coarse assumptions, their runtime overhead can be higher than 300%, which is not suitable for use in production code.

In contrast, our analysis is both completely precise and incurs reasonable overhead.

There has also been work identifying bugs [Jin and Orso 2012, 2013; Liblit et al. 2003] and information leaks [Austin and Flanagan 2012; Devriese and Piessens 2010; Enck et al. 2014] by monitoring production executions. Our work similarly monitors production code to identify unsoundness that can be used to find bugs, information flows, and so forth, but our approach differs in that we aim to
use the reported counterexamples to compute static points-to sets that are eventually sound; these points-to sets can be used with any client.

**Specification inference.** There has been recent work on inferring specifications, e.g., purely static approaches that interact with a human analyst [Albarghouthi et al. 2016; Bastani et al. 2015b; Zhu et al. 2013], and approaches that rely on dynamic traces [Bastani et al. 2018; Clapp et al. 2015; Heule et al. 2015]. Purely static approaches can give certain soundness guarantees, but suffer from imprecision and rely heavily on interaction. In contrast, dynamic approaches are fully automatic, but necessarily incomplete since dynamic analysis is an underapproximation. Our goal is to develop a fully automatic approach where runtime checks are used to detect when specifications are missing. Furthermore, [Ali and Lhoták 2013] enables sound callgraph analysis using only information available in the library interface by using the separate compilation assumption, which says that the library can be compiled separately from the program. This assumption is similar to our disjoint fields assumption (with extensions to shared fields and callbacks), in particular, we assume that the only information about the program “known” to the library are fields and methods that appear in the library interface. While the callgraph can be computed with reasonable precision using pessimistic assumptions, the same is not true of points-to edges.

**Static analysis with specifications.** A large number of static analyses rely on specifications to model missing code, including a number specifically designed to detect Android malware using information flow analysis [Clapp et al. 2015; Feng et al. 2014; Gordon et al. 2015], as well as production systems designed to find bugs in Android apps [Facebook 2017]. In all of these systems, specifications are implemented as needed for the most frequently used library functions; thus, specifications relevant to the client may be missing. Thus, OPTIX can be used in conjunction with these tools to detect potential unsoundness due to missing specifications.

**Static points-to analysis.** There is a large literature on static points-to analysis [Andersen 1994; Milanova et al. 2002; Shivers 1991; Sridharan and Bodík 2006; Whaley and Lam 2004; Wilson and Lam 1995]. Our focus is on the new problem of automatic inference of precise points-to information when some of the code is missing.

**Static information flow analysis.** Static information flow analysis has been applied previously to the verification of security policies [Arzt et al. 2014; Feng et al. 2014; Fuchs et al. 2009; Gordon et al. 2015; Livshits and Lam 2005; Tripp et al. 2009; Xie and Aiken 2006]. All of these approaches depend on alias analysis, and many use specifications to improve precision and scalability. Our techniques for automatically synthesizing points-to specifications can make implementing any static analysis for large software systems, including information flow analysis, more practical.

**Synthesis.** Program synthesis has also been applied to inferring specifications from dynamic traces [Heule et al. 2015]. This approach requires fine-grained instrumentation (specifically, leveraging features of the Javascript language to obtain alias traces), but they recover all method functionality. They accomplish this using MCMC on a restricted space of potential specifications. Our approach requires significantly less instrumentation, but our goal is only to recover aliasing behaviors, and our specifications are furthermore flow insensitive. There have been other approaches to synthesizing programs from traces [Gulwani 2011; Lau et al. 2003]. See [Heule et al. 2015] for a detailed discussion.

**11 CONCLUSION**

We have described an approach to points-to analysis when code is missing. Our approach is completely precise and fully automatic, and while it forgoes ahead-of-time soundness, it achieves
eventual soundness by using runtime checks in production code. We implement our approach in a tool called OPTIX to compute points-to sets for Android apps, where the Android framework is missing, and show that our approach achieves low runtime overhead and data usage on almost all apps in a large benchmark suite.

REFERENCES

Karim Ali and Ondrej Lhotáč. 2013. Averroes: Whole-program analysis without the whole program. In ECOOP.
Thomas H Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In POPL.
Osbert Bastani, Saswat Anand, and Alex Aiken. 2015b. Specification inference using context-free language reachability. In POPL.
Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In ICSE.
Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: mining explicit information flow specifications from concrete executions. In ISSTA.
Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In CCS.
Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In FSE.
Cormac Flanagan. 2006. Hybrid type checking. In POPL.
Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In POPL.
Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing models for opaque code. In FSE.


A SOUNDNESS AND MINIMALITY OF OPTIMIZED MONITORING

We prove the propositions in Section 3.4 that show that our optimized monitoring schemes are sound. First, Proposition 3.6 says that we only need to monitor function call returns (in addition to abstract objects). Second, Proposition 3.7 says we only need to monitor abstract objects that leak to missing code (in addition to function call returns).

Finally, we prove Proposition 3.8, which describes the sense in which our proposed instrumentation scheme is minimal.

A.1 Proof of Proposition 3.6

Precision is straightforward; we show that the monitoring scheme $M_{\text{min}}$ is sound. In particular, we need to show that if $x \mapsto o$ occurs during execution, then we either report $x \leftarrow o$ as a counterexample, or $x \leftarrow o$ cannot be the first counterexample that occurred during that execution. Suppose $x \mapsto o$ occurs because $x \leftarrow \bar{o}$ during execution, where $\bar{o} \sim_o o$. Because we instrument every allocation (and we have assumed that there are no allocations in missing code), we detect $\bar{o} \sim_o o$ before $x \mapsto o$ occurs. We prove by induction on the execution trace:

- Allocation: If $\bar{\bar{o}}$ is assigned to $x$ at statement $x \leftarrow X()$, then $\bar{o} \sim_o o = (x \leftarrow X())$, so $x \leftarrow o$ cannot be missing (since it is derived by rule 1 in Figure 2).
- Assignment: If $\bar{o}$ is assigned to $x$ at statement $x \leftarrow y$, then at this point in the execution, $y \leftarrow \bar{o}$. By induction, we derive $y \leftarrow o$ statically, so by rule 2 in Figure 2, we statically derive $x \leftarrow o$.
- Load: Suppose that $\bar{o}$ is assigned to $x$ at statement $x \leftarrow y.f$, where $y \leftarrow \bar{o}'$ at this point in the execution. Furthermore, at a prior point in the execution, $\bar{\bar{o}}$ must have been stored into the field $f$ of $\bar{o}'$ via a statement $z.f \leftarrow w$, where $z \leftarrow \bar{o}'$ and $w \leftarrow \bar{o}$ at this point in the execution. By our disjoint fields assumption, $z.f \leftarrow w$ must be in available code, so $z$ and $w$ are visible. Therefore, by induction, we statically derive $y \leftarrow o'$, $z \leftarrow o'$, and $w \leftarrow o$, where $\bar{o}' \sim_o o'$ (note that we observe $\bar{o}' \sim_o o'$ for the same reason we observe $\bar{o} \sim_o o$). Therefore, by rule 3 in Figure 2, we statically derive $x \leftarrow o$.
- Store: A store statement cannot assign $\bar{o}$ to $x$.
- Function call: If $\bar{\bar{o}}$ is assigned to $x$ at statement $x \leftarrow m(y)$, then we record $x \leftarrow \bar{o}$, since we monitor all such statements.

Therefore, $M_{\text{min}}$ is sound. □

A.2 Proof of Proposition 3.7

As before, precision is straightforward; we show that the monitoring scheme $\tilde{M}_{\text{min}}$ is sound. In particular, let $x \mapsto o$ be the first missing points-to edge that to occur in the execution trace. We claim that our instrumentation $\tilde{M}_{\text{min}}$ reports $x \mapsto o$. Our proof of this claim proceeds in two steps:

- First, suppose that $x \mapsto o$ occurs because $x \leftarrow \bar{o}$, where $\bar{o} \sim_o o$. Then, we show that $\bar{o}$ must have been assigned to $x$ via a call to a missing function $x \leftarrow m(y)$.
- Second, we show that for any $\bar{o}$ assigned to $x$ via a call to a missing function $x \leftarrow m(y)$, we must have $y' \leftarrow \bar{o}$, where $y'$ is an argument passed to missing code via a call $x' \leftarrow m'(y')$ to a missing function $m'$.

Intuitively, the first claim says that a missing points-to edge $x \leftarrow \bar{o}$ can only “introduced” into the program as a result of a call to a missing function, and the second claim says that a concrete object $\bar{o}$ can only be returned by a call to a missing function if it was previously passed as the argument to a (possibly different) missing function.

We show the first claim; i.e., that $\bar{o}$ must have been assigned to $x$ via a call $x \leftarrow m(y)$ to a missing function $m \in M$. We proceed by induction on the execution trace:
• Allocation: If $x \leftarrow X()$ assigns $\ddot{o}$ to $x$, then $\ddot{o} \sim o = (x \leftarrow X())$, so $x \leftarrow o$ cannot be missing.

• Assignment: If $x \leftarrow y$ assigns $\ddot{o}$ to $x$, then $y \leftarrow \ddot{o}$, so $y \leftarrow o$ must also have been missing (or we would have derived $x \leftarrow o$ statically).

• Load: Suppose $x \leftarrow y.f$ assigns $\ddot{o}$ to $x$. Then, assuming $y \leftarrow \ddot{o}'$ at this point in the execution, $\ddot{o}$ must have been stored into field $f$ of $\ddot{o}'$ by a statement $z.f \leftarrow w$ (which is also in available code by our disjoint fields assumption), where $z \leftarrow \ddot{o}'$ and $w \leftarrow \ddot{o}$. One of the edges $y \rightarrow \ddot{o}'$, $z \leftarrow \ddot{o}'$, and $w \rightarrow o$ must have been missing (or we would have derived $x \leftarrow o$ statically).

• Store: Such a statement cannot assign $\ddot{o}$ to $x$.

Now, we show the second claim; i.e., that $\ddot{o}$ must have been assigned to some argument $y'$ passed to missing code via a function call $x' \leftarrow m'(y')$. Our proof uses the points-to set $\tilde{I}$ computed by including all missing code in the static analysis in Figure 2 (with $\Pi_{\text{miss}} = \emptyset$). We cannot compute $\tilde{I}$ since we do not have the missing code, but $\tilde{I}$ is sound, so in particular, $x \leftarrow o \in \tilde{I}$ reported by our instrumentation, we have $x \leftarrow o \in \tilde{I}$.

We prove the following stronger claim. Suppose that the points-to edge $x' \leftarrow \ddot{o}$ occurs during execution, where $x'$ is either in missing code or equal to the parameter $p_m$ or return value $r_m$ of a missing function $m$, and $\ddot{o}$ is allocated at allocation statement $o$ in available code. Then, at a prior point in the execution, $y' \leftarrow \ddot{o}$ for some argument $y'$ (in visible code) passed to missing code via a call $x' \leftarrow m'(y')$ to missing function $m'$. We prove by induction on the execution trace:

• Allocation: Note that $\ddot{o}$ cannot be assigned to $x'$ via an allocation statement $x' \leftarrow X()$, since we have assumed that $\ddot{o}$ is allocated in available code.

• Assignment: If $x' \leftarrow y'$ assigns $\ddot{o}$ to $x'$, then either $y'$ is also missing, in which case our claim follows by induction, or $x' = p_{m'}$ is a parameter and the (visible) variable $y'$ is an argument passed to missing code via a call $x'' \leftarrow m'(y')$ to missing function $m'$, so again our claim follows.

• Load: Suppose that $x' \leftarrow y'.f$ assigns $\ddot{o}$ is assigned to $x'$. Furthermore, suppose that $y' \leftarrow \ddot{o}'$ at this point in the execution; then, $\ddot{o}$ must have been stored into field $f$ of $\ddot{o}'$ by statement $z.f \leftarrow w$ at a prior point in the execution, where $z \leftarrow \ddot{o}'$ and $w \leftarrow \ddot{o}$. By our separation of fields assumption, $z.f \leftarrow w$ must be in missing code. Since $w$ is in missing code, our claim follows by induction.

• Store: Such a statement cannot assign $\ddot{o}$ to $x$.

By our first claim, the first missing points-to edge that occurs during execution is $x \leftarrow \ddot{o}$ (where $\ddot{o} \sim o$ is allocated in available code), where $\ddot{o}$ is assigned to $x$ via a call $x \leftarrow m(y)$ via a missing function $m$. Letting $x' = r_m$ be the return value $m$, we can apply our second claim, which says that $y' \leftarrow \ddot{o}$ for some visible variable $y'$ passed to missing code via a call $x' \leftarrow m'(y')$ to a missing function $m'$. By our assumption that $x \leftarrow \ddot{o}$ is the first missing points-to edge, the points-to edge $y \leftarrow \ddot{o}$ cannot be missing. In other words, we compute $y \leftarrow o \in \Pi$ using our static analysis. Therefore, $o \in M_{\text{alloc}}$ is monitored, from which our result follows. □

A.3 Proof of Proposition 3.8

Our proof depends on the extension to handling allocations in library code described in Section 4. Assume $\Pi_{\text{miss}} = \emptyset$. Consider any library function return value $r_m$. The library function $m$ could allocate a fresh abstract object to its return value, which would be expressed by the proxy object $p = \{r_m\} \in \mathcal{P}$. In this case, the points-to edge $r_m \leftarrow p$ is missing, since it occurs during an execution but is not computed statically. Therefore, if $r_m$ is not monitored, then we would not detect the missing points-to edge $r_m \leftarrow p$.

Next, consider any allocation $o$ such that a concrete object $\ddot{o}$ is passed as an argument to a library function $m$, i.e., $p_m \leftarrow o$. The library function $m$ could assign assign its parameter to its return
value. In this case, the points-to edge \( r_m \leadsto o \) is missing, since it occurs during an execution but is not computed statically. Furthermore, all other library functions could be no-ops, in which case no other points-to edge is missing. If \( o \) is not monitored (but \( r_m \) is monitored), then we would not observe \( o \) in the program, so we would incorrectly conclude that \( r_m \) points to an object allocated in library code. Therefore, \( M_{\text{alloc}} \) is minimal as claimed. □

### B GUARANTEES FOR PROXY OBJECTS

We prove the propositions in Section 4 that show that the points-to sets we compute with proxy objects are sound and precise. Proposition 4.4 says that proxy objects are sound, Proposition 4.5 says that ideal proxy objects are precise, and Proposition 4.8 says that from the perspective of the static analysis, our proxy objects are sound and as precise as ideal proxy objects.

#### B.1 Soundness for Ideal Proxy Objects

First, we prove our soundness result for ideal proxy objects. Suppose that \( x \leadsto o \) during an execution.

We prove that \( x \leadsto o \in \vec{\Pi}^* \) (if \( o \) is allocated at \( o \), in which case \( o \leadsto o \) and \( x \leadsto \vec{p} \in \vec{\Pi}^* \) (if \( o \) is allocated in missing code and \( \phi(o) = \vec{p} \), in which case \( o \leadsto \vec{p} \)). We prove by induction on the execution trace:

- **Allocation**: If \( x \leadsto X() \) assigns \( o \) to \( x \), then \( o \) is allocated at \( o = (x \leadsto X()) \), so we derive \( x \leadsto o \) using rule 1 in Figure 2.
- **Assignment**: If \( x \leadsto y \) assigns \( o \) to \( x \), then by induction, either \( y \leadsto o \in \vec{\Pi}^* \) (if \( o \leadsto o \) or \( y \leadsto \vec{p} \in \vec{\Pi}^* \) (if \( o \leadsto \vec{p} \)). We derive \( x \leadsto o \) (in the former case) or \( x \leadsto \vec{p} \) (in the latter case) by rule 2 in Figure 2.
- **Load**: Suppose \( x \leadsto y.f \) assigns \( o \) to \( x \), and \( y \leadsto o' \) at this point. Then, \( o \) must have been stored in field \( f \) of \( o' \) by some statement \( z.f \leftarrow w \), where \( z \leadsto o' \) and \( w \leftarrow o \). By our disjoint fields assumption, \( z.f \leftarrow w \) is in available code, so by induction, either \( w \leadsto o \in \vec{\Pi}^* \) (if \( o \leadsto o \) or \( w \leadsto \vec{p} \in \vec{\Pi}^* \) (of \( o \leadsto \vec{p} \)). Furthermore, by induction, either \( y \leadsto o' \in \vec{\Pi}^* \) and \( z \leadsto o' \in \vec{\Pi}^* \) (if \( o' \leadsto o' \in O_P \)) or \( y \leadsto \vec{p}' \in \vec{\Pi}^* \) and \( z \leadsto \vec{p}' \in \vec{\Pi}^* \) (if \( o' \leadsto \vec{p}' \)). Therefore, we derive \( x \leadsto o \in \vec{\Pi}^* \) (if \( o \leadsto o \)) or \( x \leadsto \vec{p} \in \vec{\Pi}^* \) (if \( o \leadsto \vec{p} \) using rule 3 in Figure 2).
- **Store**: Note that \( o \) cannot be assigned to \( x \) using this statement.
- **Function call**: Suppose \( x \leadsto m(y) \) assigns \( o \) to \( x \). If \( o \leadsto o \), then \( x \leadsto o \in \vec{\Pi}_{\text{miss}}^+ \) is missing, so \( x \leadsto o \in \vec{\Pi}^+ \) by rule 4. If \( o \leadsto \vec{p} \), then \( x \leadsto \phi(o) \in \vec{\Pi}^+ \) is missing, where \( \vec{p} = \phi(o) \). Again, \( x \leadsto \vec{p} \in \vec{\Pi}^+ \) by rule 4.

The result follows. □

#### B.2 Precision for Ideal Proxy Objects

Now, we prove our precision result for ideal proxy objects. Recall that our goal is to prove that (i) if \( x \leadsto o \in \vec{\Pi}^+ \), where \( o \in O_P \), then also \( x \leadsto o \in \vec{\Pi} \), and (ii) if \( x \leadsto \vec{p} \in \vec{\Pi}^+ \), where \( \vec{p} = \phi(o) \in P \), then \( x \leadsto o \in \vec{\Pi} \), where \( o \) is the (missing) allocation statement where \( o \) was allocated. We proceed by structural induction, first for \( x \leadsto o \in \vec{\Pi}^+ \):

- **Allocation**: If \( x \leadsto o \) is derived using rule 1 in Figure 2, then \( o = (x \leadsto X()) \), so \( x \leadsto o \in \vec{\Pi} \) by rule 1 as well.
- **Assignment**: If \( x \leadsto o \) is derived using rule 2 in Figure 2, then \( x \leadsto y \) and \( y \leadsto o \in \vec{\Pi}^+ \). By induction, \( y \leadsto o \in \vec{\Pi} \), so \( x \leadsto o \in \vec{\Pi} \) is derived using rule 2 in Figure 2.
- **Load/store**: Suppose that \( x \leadsto o \) is derived using rule 3 in Figure 2—i.e., from premise

\[
x \leftarrow y.f, \quad z.f \leftarrow w, \quad w \leftarrow o, \quad y \leftarrow o', \quad z \leftarrow o',
\]

where \( o' \in O_P \), or premise
\[
x \leftarrow y . f , \quad z . f \leftarrow w , \quad w \leftarrow o , \quad y \leftarrow \tilde{p}' , \quad z \leftarrow \tilde{p}' ,
\]
where \( \tilde{p}' \in \tilde{P} \). In either case, by induction, \( w \leftarrow o \in \Pi \). In the former case, \( y \leftarrow o' \in \Pi \) and \( z \leftarrow o' \in \Pi \) by induction, so we derive \( x \leftarrow o \in \Pi \) using rule 3 in Figure 2. In the latter case, suppose that \( \tilde{p}' = \hat{\phi}(\tilde{o}') \), and let \( o' \) be the statement at which \( \tilde{o}' \) was allocated. By induction, \( y \leftarrow o' \in \Pi \) and \( z \leftarrow o' \in \Pi \), so we again derive \( x \leftarrow o \in \Pi \) using rule 3 in Figure 2.

- Missing: If \( x \leftarrow o \in \Pi_{miss} \), then \( x \leftarrow \tilde{o} \) during a concrete execution, where \( \tilde{o} \) was allocated at \( o \). Therefore, any sound points-to analysis must compute \( x \leftarrow o \).

Second, we prove the claim for \( x \leftarrow \tilde{p} \in \tilde{P}^* \):

- Allocation: We cannot derive \( x \leftarrow \tilde{p} \) using rule 1 in Figure 2.
- Assignment: If \( x \leftarrow \tilde{p} \) is derived using rule 2 in Figure 2, then \( x \leftarrow y \) and \( y \leftarrow \tilde{p} \in \tilde{P}^* \). Suppose that \( \tilde{p} = \hat{\phi}(\tilde{o}) \), and \( \tilde{o} \) is allocated at statement \( o \). By induction, \( y \leftarrow o \in \Pi \), so we derive \( x \leftarrow o \in \Pi \) using rule 2 in Figure 2.
- Load/store: Suppose that \( x \leftarrow \tilde{p} \) is derived using rule 3 in Figure 2—i.e., from premise
\[
x \leftarrow y . f , \quad z . f \leftarrow w , \quad w \leftarrow \tilde{p} , \quad y \leftarrow o' , \quad z \leftarrow o' ,
\]
where \( o' \in O_P \), or premise
\[
x \leftarrow y . f , \quad z . f \leftarrow w , \quad w \leftarrow \tilde{p} , \quad y \leftarrow \tilde{p}' , \quad z \leftarrow \tilde{p}' ,
\]
where \( \tilde{p}' \in \tilde{P} \). In either case, by induction, \( w \leftarrow o \in \Pi \), where \( \tilde{p} = \hat{\phi}(\tilde{o}) \) and \( \tilde{o} \) is the statement at which \( \tilde{o} \) is allocated. In the former case, by induction, \( y \leftarrow o' \in \Pi \), and \( z \leftarrow o' \in \Pi \), so we derive \( x \leftarrow o \in \Pi \) using rule 3 in Figure 2. In the latter case, let \( o' \) be the allocation statement of \( \tilde{o}' \), where \( \tilde{p}' = \hat{\phi}(\tilde{o}') \). By induction, \( y \leftarrow o' \in \Pi \), and \( z \leftarrow o' \in \Pi \), so we again derive \( x \leftarrow o \in \Pi \) using rule 3 in Figure 2.

The result follows. \( \square \)

### B.3 Soundness and Precision for Proxy Objects

In this section, we prove Proposition 4.8, which essentially says that our proxy object \( \phi \) mapping is as precise as the ideal proxy object mapping \( \hat{\phi} \). Suppose that \( \tilde{p} = \hat{\phi}(\tilde{o}) \) and \( p = \phi(\tilde{o}) \) for concrete object \( \tilde{o} \). We prove that \( x \leftarrow \tilde{p} \in \tilde{P}^* \) if and only if \( x \leftarrow p \in \Pi^* \).

First, we prove that if \( x \leftarrow \tilde{p} \in \tilde{P}^* \) (resp., \( x \leftarrow o \in \Pi^* \)), then \( x \leftarrow p \in \Pi^* \) (resp., \( x \leftarrow o \in \Pi^* \)). We proceed by structural induction on the derivation of \( x \leftarrow \tilde{p} \) (resp., \( x \leftarrow o \)) in \( \tilde{P}^* \):

- Allocation: Note that \( x \leftarrow \tilde{p} \) cannot be derived using rule 1 in Figure 2 since it only applies to allocations \( o \in O_P \). For \( x \leftarrow o \), we must have \( o = (x \leftarrow X()) \), in which case we derive \( x \leftarrow o \in \Pi^* \) as well using rule 1.
- Assignment: If \( x \leftarrow \tilde{p} \) (resp., \( x \leftarrow o \)) is derived from \( x \leftarrow y \), then \( y \leftarrow \tilde{p} \in \tilde{P}^* \) (resp., \( y \leftarrow o \in \Pi^* \)), so by induction \( y \leftarrow p \in \Pi^* \) (resp., \( y \leftarrow o \in \Pi^* \)). Therefore, we derive \( x \leftarrow p \in \Pi^* \) (resp., \( x \leftarrow o \in \Pi^* \)) using rule 2 in Figure 2.
- Load/store: Suppose \( x \leftarrow \tilde{p} \) is derived using rule 3 in Figure 2 from premise
\[
x \leftarrow y . f , \quad z . f \leftarrow w , \quad y \leftarrow \tilde{p}' \in \tilde{P}^* \), \quad z \leftarrow \tilde{p}' \in \tilde{P}^* \), \quad w \leftarrow \tilde{p},
\]
where \( \tilde{p}' = \hat{\phi}(\tilde{o}') \in \tilde{P} \). Let \( p' = \phi(\tilde{o}') \). Then, by induction, we derive \( y \leftarrow p' \in \Pi^* \), \( z \leftarrow p' \in \Pi^* \), and \( w \leftarrow p \in \Pi^* \), so we derive \( x \leftarrow p \in \Pi^* \) using rule 3 in Figure 2. The cases where \( \tilde{p} \) is instead \( o \in O_P \) and/or \( \tilde{p}' \) is instead \( o' \in O_P \) follow similarly.
• Missing: Suppose \( x \leftarrow \tilde{p} \) is derived using rule 4 in Figure 2, so \( x \leftarrow \tilde{p} \in \Pi^*_{\text{miss}} \). Then, \( x \) is in the dynamic footprint of \( \tilde{o} \) (i.e., \( x \leftarrow \tilde{o} \) during execution). We show below that if \( x \) is in the dynamic footprint of \( \tilde{o} \), then we derive \( x \leftarrow p \in \Pi^* \). Note that \( x \leftarrow o \) cannot be derived using this rule.

We prove the claim in the last case by induction on the execution trace:

• Allocation: Note that \( \tilde{o} \) cannot be assigned to \( x \) using an allocation \( x \leftarrow X() \), since we have assumed that \( \tilde{o} \) is allocated in missing code.

• Assignment: If \( x \leftarrow y \) assigns \( \tilde{o} \) to \( x \), then \( y \leftarrow \tilde{o} \) at that point in the execution. By induction, \( y \leftarrow p \in \Pi^* \), so we derive \( x \leftarrow p \in \Pi^* \) using rule 2 in Figure 2.

• Load: Suppose that \( x \leftarrow y.f \) assigns \( \tilde{o} \) to \( x \), and \( y \leftarrow \tilde{o}' \) at that point in the execution. Then, at a previous point in the execution, \( \tilde{o} \) must have been assigned to field \( f \) of \( \tilde{o}' \) by a statement \( z.f \leftarrow w \), where \( z \leftarrow \tilde{o}' \) and \( w \leftarrow \tilde{o} \). By induction on the execution trace, we have \( w \leftarrow p \in \Pi^* \).

If \( \tilde{o}' \) is allocated in missing code, then by Proposition 4.4, \( y \leftarrow \tilde{p}' \in \Pi^* \) and \( z \leftarrow \tilde{p}' \in \Pi^* \), where \( \tilde{p}' = \tilde{p}(\tilde{o}') \). Then, by structural induction (on the derivation of \( x \leftarrow \tilde{p} \)), we have \( y \leftarrow p' \in \Pi^* \) and \( z \leftarrow \tilde{p}' \in \Pi^* \), where \( p' = \phi(\tilde{o}') \). Otherwise, if \( \tilde{o}' \) is allocated at \( o' \in \mathcal{O}_p \), then by Proposition 4.4, \( y \leftarrow o' \in \Pi^* \) and \( z \leftarrow o' \in \Pi^* \). Then, by structural induction (on the derivation of \( x \leftarrow \tilde{p} \)), we have \( y \leftarrow o' \in \Pi^* \) and \( z \leftarrow o' \in \Pi^* \). Either way, we derive \( x \leftarrow p \in \Pi^* \) using rule 3 in Figure 2.

• Store: Note that \( \tilde{o} \) cannot be assigned to \( x \) using this statement.

• Function call: If \( x \leftarrow m(y) \) assigns \( \tilde{o} \) to \( x \), then \( m \) is in the function footprint of \( \tilde{o} \), so \( x \leftarrow p \in \Pi^*_{\text{miss}} \subseteq \Pi^* \).

Therefore, we have shown the forward implication. Next, we show if \( x \leftarrow p \in \Pi^* \) (resp., \( x \leftarrow o \in \Pi^* \)), then \( x \leftarrow \tilde{p} \in \Pi^* \) (resp., \( x \leftarrow o \in \Pi^* \)). We prove by structural induction on the derivation of \( x \leftarrow p \):

• Allocation: As before, \( x \leftarrow p \) cannot be derived using rule 1 in Figure 2, and for \( x \leftarrow o \), we must have \( o = (x \leftarrow X()) \), in which case we derive \( x \leftarrow o \in \Pi^* \) as well using rule 1.

• Assignment: If \( x \leftarrow p \) (resp., \( x \leftarrow o \)) is derived from \( x \leftarrow y \), then \( y \leftarrow o \in \Pi^* \) (resp., \( y \leftarrow o \in \Pi^* \)), so by induction, \( y \leftarrow \tilde{p} \in \Pi^* \) (resp., \( y \leftarrow o \in \Pi^* \)), in which case we derive \( x \leftarrow \tilde{p} \in \Pi^* \) using rule 1 in Figure 2.

• Load/store: Suppose \( x \leftarrow p \) (resp., \( x \leftarrow o \)) is derived from premise

\[
\begin{align*}
  x &\leftarrow y.f, \quad z.f \leftarrow w, \quad y \leftarrow p' \in \Pi^*, \\
  z &\leftarrow p' \in \Pi^*, \quad w \leftarrow p \in \Pi^*,
\end{align*}
\]

where \( p' = \phi(\tilde{o}') \). Let \( \tilde{p}' = \tilde{p}(\tilde{o}') \). By induction, \( y \leftarrow \tilde{p}' \in \Pi^* \), \( z \leftarrow \tilde{p}' \in \Pi^* \), and \( w \leftarrow \tilde{p} \in \Pi^* \), so we derive \( x \leftarrow \tilde{p} \in \Pi^* \) using rule 3 in Figure 2.

• Missing: Suppose \( x \leftarrow p \in \Pi^*_{\text{miss}} \), so \( x \leftarrow m(y) \), where \( m \) is in the function footprint of \( \tilde{o} \). Then, \( x \) is in the dynamic footprint of \( \tilde{o} \), so \( x \leftarrow \tilde{p} \in \Pi^*_{\text{miss}} \) so we derive \( x \leftarrow \tilde{p} \in \Pi^* \) using rule 4 in Figure 2.

Therefore, the backwards implication follows, so we are done. □