Eventually Sound Points-To Analysis with Specifications

Osbert Bastani
University of Pennsylvania, Philadelphia, USA
obastani@seas.upenn.edu

Rahul Sharma
Microsoft Research, Bangalore, India
rahsha@microsoft.com

Lazaro Clapp
Stanford University, USA
lazaro@stanford.edu

Saswat Anand
Stanford University, USA
saswat@cs.stanford.edu

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

Abstract

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.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases specification inference, static points-to analysis, runtime monitoring

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.11

Acknowledgements This work was supported by NSF grant CCF-1160904, and is also based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

1 Introduction

To guarantee soundness, static analyses often assume that all program source code can be analyzed. This assumption has become tenuous as programs increasingly depend on large libraries and frameworks that are prohibitively difficult to analyze [34]. For example, mobile app stores can use static analysis to improve the quality of published apps by searching for malicious behaviors [23, 21, 5, 25] or security vulnerabilities [20, 39, 17]. 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 and are often ignored by the analysis [5, 21, 25]. Furthermore, the Android framework contains deep call hierarchies, which can pose problems since static points-to analyses typically have limited context sensitivity [9]. Thus, the Android framework is often omitted from the static analysis [52, 8, 12]. We refer to such omitted code as *missing*. In any large software system, there are inevitably parts that are missing and cannot be handled soundly [34].

Two possible approaches are to sacrifice either soundness (by making optimistic assumptions about missing code) or precision (by making pessimistic assumptions about missing code). For many applications, pessimistic assumptions are so imprecise that they make the static analysis results unusable; therefore, soundness is often sacrificed instead, but doing so is a significant compromise [34]. For example, consider malware detection – a security analyst must examine every app that is potentially malicious, making false positives costly, yet unsoundness can be exploited by a knowledgeable attacker to avoid detection.

Broadly speaking, two alternative strategies have been proposed to handle missing code: (i) using runtime instrumentation, and (ii) using handwritten specifications. The first approach uses runtime instrumentation to enforce soundness – e.g., dynamic information flow control can be used to prevent information leaks [6, 14, 18]. More recent systems have focused on using dynamic analysis to fill gaps in a static analysis. These systems first statically analyze the program, keeping track of any unsound assumptions made by the static analysis, and then instrument the program to check if these assumptions are violated during runtime; if so, then execution can be terminated or continued in a safe environment such as a sandbox. This approach has been applied to type checking [22], reflective call targets [10], and determining reachable code [7]; more general systems have also been proposed [11, 15]. However, the runtime instrumentation used by these systems is typically limited – e.g., they check either lightweight type-based properties, or program invariants that can be checked locally.

In the second approach, the human user writes *specifications* (also known as *models*, *annotations*, or *stubs*) that summarize missing code [52, 8]. This approach is very flexible, since specifications can be used to model arbitrary missing code – e.g., systems have used specifications to model part [5, 25] or all [21] of the Android framework, including production systems [19]. However, specifications are costly to write, since many thousands of specifications may be needed [8]. In addition, handwritten specifications can be error prone [28], and must be updated whenever the missing code changes. Approaches have been proposed for *inferring* specifications [52, 8, 12, 28, 9], but the user must manually check each inferred specification, including ones that turn out to be wrong or irrelevant.

In this paper, we study the problem of handling unsoundness in a static points-to analysis for Android apps, where part or all of the Android framework is omitted from the analysis. We focus on points-to analysis since it lies at the core of many static analyses, and we believe that clients (e.g., static information flow analysis) can be designed around our analysis.

We propose a system that handles missing code by combining runtime monitoring and specifications. Given a program (e.g., an Android app), our system first runs the static points-to analysis that optimistically assumes missing code is empty. If no errors are found, then our system instruments the program to detect counterexamples to the optimistic assumptions – i.e., *missing points-to edges* that occur during an execution but are missing from the (optimistic) static analysis. Points-to edges are a whole-program property, so instrumenting programs to detect missing points-to edges is challenging. In particular:

- Naïvely using a dynamic points-to analysis to detecting counterexamples can incur huge overhead – e.g., as much as 20× [12] or even two orders of magnitude [38].
- It is often impossible to insert runtime checks into missing code (e.g., native code). Thus, we restrict our analysis to instrument only *available* code (in our case, the app code).

We describe how we address these challenges in more detail below. Next, the instrumented program is published (e.g., on Google Play). If our instrumentation ever detects a counterexample, then it is reported back to the publisher (e.g., Google), who can update their specifications and re-run the static analysis. Finally, we show how detected counterexamples can be used to infer specifications that summarize missing code – intuitively, these specifications transfer the knowledge gained from the counterexample to benefit the analysis of future programs. While inferred specifications need to be validated by a human, our approach focuses specification inference on addressing gaps in the static analysis that occur in actual program executions. Importantly, adding new specifications (either handwritten or inferred) can help reduce instrumentation overhead.

With an appropriate instrumentation scheme, our system satisfies two key 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.

The key property of interest is eventual soundness, which combines two guarantees. The first guarantee is a *dynamic soundness* guarantee analogous to that provided by dynamic type checking: at the cost of some runtime overhead, we guarantee that unsoundness is detected at the point when it occurs, which allows us to prevent damage (e.g., leaking of sensitive information) from occuring.

The second guarantee is that with every reported counterexample, the specifications become progressively more complete – in particular, the specifications guarantee that the static analysis is sound with respect to all executions observed so far. 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 concludes that the program is safe even with 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. Even if the static analysis itself never fully converges, the dynamic soundness guarantee ensures that the overall system is sound.

As described above, our key contribution is an instrumentation scheme for detecting counterexamples that ensures eventual soundness. To address the challenge of high runtime overhead, we leverage the fact that to be eventually sound, we do not need the program instrumentation to report *every* counterexample that occurs during an execution. Instead, it is sufficient that for any execution, the instrumentation detects the *first* counterexample to occur. For example, let $x \hookrightarrow o$ and $y \hookrightarrow o$ be two *potentially missing* points-to edges; if we can guarantee that for any execution, $x \hookrightarrow o$ can only occur after $y \hookrightarrow o$ has already occured (and we are furthermore guaranteed to detect that $y \hookrightarrow o$ has occured before $x \hookrightarrow o$ can occur), then we only need to monitor whether $y \hookrightarrow o$ 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 challenge of being unable to instrument missing code, note that because we use specifications, we are already unable to discover relationships about the missing code. Indeed, 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 [21]. 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,[1] 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**[2], which analyzes Android apps treating the entire Android framework as missing. We show that our instrumentation typically incurs low overhead – the median overhead is 4.6%, the overhead is less than 20% for more than 90% of 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). Also, we show that the instrumentation can be used to detect missing points-to specifications in the information flow client from [21], which computes explicit information flows [42]. This tool uses specifications to model missing code (i.e., the Android framework). We empirically show that 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 explicit information flow client from [21]. 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* $x \hookrightarrow o$ if variable x may point to a concrete object $\bar{o}$ allocated at allocation statement $o \in \mathcal{O}$ (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` $\hookrightarrow o_{\text{list}}$, `str` $\hookrightarrow o_{\text{str}}$, and `dataCopy` $\hookrightarrow o_{\text{str}}$, so the points-to analysis concludes that `str` and `dataCopy` may be aliased.

[1] The term *proxy object* is ours, but the concept has occurred in prior work [8].
[2] **Optix** stands for Optimistic Points-To Information from eXecutions.

```
void main() { // program
  String str = mkStr();
  List list = new List(); // o_list
  list.add(str);
  Object data = list.get(0);
  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; } }
```

Figure 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.

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 [52, 5, 8], and because it uses deep call hierarchies, which can cause significant imprecision, since static points-to analyses often have limited context sensitivity [9]. 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 $x \in \mathcal{V}_P$ 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 `data` $\hookrightarrow o_{\text{list}}$ in case the implementation of `get` is `return this`. Alternatively, optimistically assuming that missing code is empty can be unsound, for example, failing to compute `data` $\hookrightarrow o_{\text{str}}$ and `dataCopy` $\hookrightarrow o_{\text{str}}$. 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; see Figure 1 for examples. For instance, because our static points-to analysis collapses arrays into a single field, we can overapproximate the array of elements stored by the `List` class as a single field `f`.

Suppose that the analyst has provided specifications for frequently used library functions such as `mkStr` and `sendHttp`, but a long tail of specifications remain missing, including those for `add` and `get`. Therefore, the (optimistic) static information flow analysis incorrectly concludes that `dataCopy` cannot point to `str`, and that `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 `main` is instrumented to record the concrete objects pointed to by `libStr` and `data`. When the program is run: (i) `libStr` points to concrete object $\bar{o}_{\text{str}}$, so our analysis concludes that $\bar{o}_{\text{str}}$ is allocated at o_{str}, and (ii) `data` points to $\bar{o}_{\text{str}}$, so our analysis concludes that `data` $\hookrightarrow o_{\text{str}}$ and reports this counterexample. Upon receiving this report, we add `data` $\hookrightarrow o_{\text{str}}$ to the known counterexamples.

Given a new counterexample $x \hookrightarrow o$, the static analysis at the very least learns that $x \hookrightarrow o$ 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 `data` $\hookrightarrow o_{\text{str}}$, our static analysis additionally computes its consequence `dataCopy` $\hookrightarrow o_{\text{str}}$, and determines that `str` and `dataCopy` may be aliased. Thus, the security analyst learns that the return value of `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 `randBool` returns false and the data is not leaked in that specific execution.

Second, the static analysis can use *specification inference* to try to 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 specification inference to automatically infer candidate specifications that "explain" the counterexample. For example, given counterexample `data` $\hookrightarrow o_{\text{str}}$, the specification inference algorithm would infer the specifications for `add` and `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 that 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` $\hookrightarrow o_{\text{str}}$ can only occur during execution where the missing points-to edge `data` $\hookrightarrow o_{\text{str}}$ has already occured. Furthermore, once `data` $\hookrightarrow o_{\text{str}}$ has occured, it is added to the static analysis, which computes `dataCopy` $\hookrightarrow o_{\text{str}}$ as a consequence. Therefore, we never need to detect or report `dataCopy` $\hookrightarrow o_{\text{str}}$.

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 $\bar{o}_{\text{str}}$ was allocated at o_{str}. 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 $\bar{o}_{\text{str}}$. Since $\bar{o}_{\text{str}}$ was not allocated at o_{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 $r_{\texttt{mkStr}}$ of `mkStr`. We discuss proxy objects in Section 4.

Finally, we propose an eventually sound points-to analysis. Future work is needed to design an eventually sound information flow analysis; we describe a candidate in Section 9, but implementing and 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 ($x \leftarrow X()$, where $X \in \mathcal{C}$ is a class), assignments ($x \leftarrow y$, where $x, y \in \mathcal{V}_P$ are program variables), loads ($x \leftarrow y.f$, where $f \in \mathcal{F}$ is a field), stores ($x.f \leftarrow y$), and calls to library functions

1. (allocation) $\dfrac{x \leftarrow X(),\ o = (x \leftarrow X())}{x \hookrightarrow o \in \Pi}$

2. (assignment) $\dfrac{x \leftarrow y,\ y \hookrightarrow o \in \Pi}{x \hookrightarrow o \in \Pi}$

3. store $\dfrac{x.f \leftarrow y,\ x \hookrightarrow o' \in \Pi,\ y \hookrightarrow o \in \Pi}{o'.f \hookrightarrow o \in \Pi_O}$

4. load $\dfrac{x \leftarrow y.f,\ y \hookrightarrow o' \in \Pi, o'.f \hookrightarrow o \in \Pi_O}{x \hookrightarrow o \in \Pi}$

5. (missing) $\dfrac{x \hookrightarrow o \in \Pi_{\text{miss}}}{x \hookrightarrow o \in \Pi}$

Figure 2 Rules to compute sound points-to sets. Rules 1-4 are standard. Rule 5 adds reported counterexamples to the analysis.

$m \in \mathcal{M}$ library $(x \leftarrow m(y))$. 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 may points-to analysis, shown in Figure 2, is a standard flow- and context-insensitive analysis for computing points-to edges $\Pi \subseteq \mathcal{V}_P \times \mathcal{O}$ [3, 46, 8]; we describe how our results can be extended to context- and object-sensitive analyses with on-the-fly callgraph construction in Section 6.4. Rule 1 handles object allocations $x \leftarrow X()$. In particular, recall that an abstract object is an allocation statement $o = (x \leftarrow X())$, representing all concrete objects allocated at that statement. Then, the rule says that x may point to o. Rule 2 handles assignments $x \leftarrow y$ – i.e., if y may point to o, then x may point to o as well. Rules 3-4 handle program loads and stores. They introduce a new relationship $\Pi_O \subseteq \mathcal{O} \times \mathcal{F} \times \mathcal{O}$, which denotes points-to relationships between abstract objects. Intuitively, the relationship $o'.f \hookrightarrow o \in \Pi_O$ (written $o'.f$ *may point to* o) means that a field f of abstract object o' may reference an abstract object o. Rule 3 handles stores – given a statement $x.f \leftarrow y$, it says that if y may point to o and x may point to o', then $o'.f$ may point to o. Rule 4 handles loads – given a statement $x \leftarrow y.f$, it says that if y may point to o' and $o'.f$ may point to o, then x may point to o. Together, rules 3 and 4 are equivalent to the single rule

$$\frac{x \leftarrow y.f,\ z.f \leftarrow w,\ y \hookrightarrow o' \in \Pi,\ z \hookrightarrow o' \in \Pi,\ w \hookrightarrow o \in \Pi}{x \hookrightarrow o \in \Pi}.$$

Rule 5 handles known counterexamples $\Pi_{\text{miss}} \subseteq \mathcal{V}_P \times \mathcal{O}$. A function call $x \leftarrow m(y)$ is treated as an assignment of y to the parameter p_m and an assignment of the return value r_m to x.

We initially make three simplifying assumptions. First, we assume 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 load and store statements in the library (resp., program), i.e., there are no *shared fields* $f \in \mathcal{F}_L \cap \mathcal{F}_P$. [3] 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. Finally, **Optix** assumes that library functions do not access global variables; we describe how we could remove this assumption in Section 9.

[3] In practice, a shared field is typically a public field in the library code that is accessed by the program.

3.2 Eventual Soundness

We first define soundness relative to an execution:

▶ **Definition 1.** Let Π be a points-to set. We say an execution e *exhibits a counterexample* with respect to Π if during the execution, there is a dynamic points-to edge $x \hookrightarrow o \notin \Pi$. We say Π is *sound* with respect to e if e does not exhibit any counterexamples.

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, Π_i is computed as a function of the previous points-to set Π_{i-1} and the counterexamples from e_i (if any). Note that the instrumentation for e_{i+1} can be chosen adaptively based on Π_i and that $\Pi_i \subseteq \Pi_j$ if $i \leq j$.

▶ **Definition 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 (if any) that occurs during the execution, and (ii) there are only finitely many $i \in \mathbb{N}$ such that the execution e_i exhibits a counterexample with respect to the previous points-to set Π_{i-1}.

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). The second property implies that the points-to sets eventually become sound with respect to all subsequent executions – more precisely, there exists $n \in \mathbb{N}$ such that the points-to set Π_n is sound with respect to all executions e_i such that $n \leq i < \infty$.[4]

We remark that we do *not* require that the different executions use the same inputs, random seeds, or have the same execution time. We can also handle programs that run continuously – we simply split the execution into intervals (e.g., 1st hour, 2nd hour, ...) and treat each interval as a different execution. However, one of our optimizations may need to be disabled, since it requires that the instrumentation be modified over time, which may not be possible for a continuously running program.

▶ **Definition 3.** The points-to analysis is *precise* if for every $i \in \mathbb{N}$, the points-to set Π_i is a subset of the points-to set computed by analyzing the implementation of the missing code.

Note that while progress towards static soundness is guaranteed, it is not possible to report how many sources of static unsoundness remain at any point in time. Even if all program paths are executed, there may be missing points-to edges – e.g., in the following, suppose that `foo` is missing; then, if `randInt` never evaluates to `0`, `y` $\hookrightarrow o$ remains missing:

```
void main() { // program
  Object x = new Object(); // o
  Object y = foo(x); }
```

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

Despite the inability to quantify progress, the property of eventual soundness is useful, since (in addition to guaranteeing dynamic soundness) 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 $x \leftarrow m()$

[4] Note the upper bound $i < \infty$; this property only holds for executions indexed by the natural numbers $\mathbb{N}$.

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` $\hookrightarrow o$ does not occur in any execution. Eventual soundness is still satisfied, since it is defined relative to the sequence of observed executions – if a counterexample exists but is never observed, then the analysis is still statically sound for all observed executions.

3.3 Naïve Algorithm

We first describe a naïve eventually sound points-to analysis.

Optimistic analysis. We use the static analysis in Figure 2 to compute static points-to edges Π, 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 $x \leftarrow *$. Note that upon executing, a monitor only records a single integer – i.e., the memory address pointed to by x; it does not traverse the heap more deeply. Thus, a monitor is very lightweight instrumentation. A *monitoring scheme* M is a set of program statements to be monitored. Our goal is to design monitoring schemes that satisfy the following:

▶ **Definition 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 $x \in \mathcal{V}_P$. Then, we can map each concrete object $\bar{o}$ to its allocation:

▶ **Definition 5.** An *abstract object mapping* for an execution is a mapping $\bar{o} \rightsquigarrow o$, where $\bar{o}$ is a concrete object allocated at abstract object o.

For every report $(x \leftarrow X(), \bar{o})$, we add $\bar{o} \rightsquigarrow o$ to the abstract object mapping, where $o = (x \leftarrow X())$. Then, for every report $(x \leftarrow *, \bar{o})$ and $\bar{o} \rightsquigarrow o$, we conclude that $x \hookrightarrow o$ occurred dynamically; if missing, we report it as a counterexample. In our example, we monitor `libStr`, detect that $\bar{o}_{\text{str}} \rightsquigarrow o_{\text{str}}$, and report counterexample `data` $\hookrightarrow \bar{o}_{\text{str}}$.

Updating the static analysis. We add every reported counterexample to Π_{miss}. Our static analysis in Figure 2 adds Π_{miss} to Π and computes the consequences of these added edges. Continuing our example, our static analysis adds `data` $\hookrightarrow o_{\text{str}}$ to Π (rule 5), and computes its consequence `dataCopy` $\hookrightarrow o_{\text{str}}$ (rule 2).

Guarantees. Let Π^* be the points-to edges computed in the case that $\Pi_{\text{miss}} = \Pi^*_{\text{miss}}$ holds, where Π^*_{miss} is the set of all missing points-to edges. Then, our analysis is:

Data Structure	Rules for Construction
monitors	(allocation) $M_{\text{alloc}} = \mathcal{O}_P$ (function call) $\dfrac{x \leftarrow m(y)}{(x \leftarrow m(y)) \in M_{\text{call}}}$
reports	(allocation) $\dfrac{x \leftarrow X(),\ x \hookrightarrow \bar{o}}{(x \leftarrow X(), \bar{o}) \in R_{\text{alloc}}}$ (function call) $\dfrac{x \leftarrow m(y),\ x \hookrightarrow \bar{o}}{(x \leftarrow m(y), \bar{o}) \in R_{\text{call}}}$
abstract object mapping	(program abstract objects) $\dfrac{(x \leftarrow X(), \bar{o}) \in R_{\text{alloc}}}{\bar{o} \rightsquigarrow o = (x \leftarrow X())}$ (proxy objects) $\dfrac{(*, \bar{o}) \notin R_{\text{alloc}},\ \forall i \in \{1, ..., k\}\ (x_i \leftarrow m_i(y_i), \bar{o}) \in R_{\text{call}}}{\bar{o} \rightsquigarrow p = \{m_1, ..., m_k\}}$
missing points-to edges	$\dfrac{(x \leftarrow m(y), \bar{o}) \in R_{\text{call}},\ \bar{o} \rightsquigarrow o,\ x \hookrightarrow o \notin \Pi}{x \hookrightarrow o \in \Pi_{\text{miss}}}$
optimistic static points-to edges	$\Pi =$ (apply Figure 2 with the constructed Π_{miss})

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

- **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 Π^*_{miss} is finite, only finitely many counterexamples are ever reported. Thus, there exists some execution $i_0 \in \mathbb{N}$ of the program during which the last counterexample is reported. Then, the points-to set Π_i computed by our algorithm after the ith execution is sound for all executions $i \geq i_0$.
- **Precise:** Any sound set of points-to edges Π' must contain the missing points-to edges Π^*_{miss}. Therefore, $\Pi_{\text{miss}} \subseteq \Pi^*_{\text{miss}} \subseteq \Pi'$. Since computing a transitive closure is monotone, it follows that $\Pi \subseteq \Pi^* \subseteq \Pi'$.
- **Automatic:** Our static analysis requires no human input.

In our example, the static points-to set Π is sound after the counterexample $\texttt{data} \hookrightarrow o_{\text{str}}$ is reported, since the static analysis then computes the remaining missing points-to edge $\texttt{dataCopy} \hookrightarrow o_{\text{str}}$.

3.4 Optimized Monitoring

We now describe how to reduce monitoring.

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

▶ **Proposition 6.** The monitoring scheme $M_{\text{min}} = M_{\text{alloc}} \cup M_{\text{call}}$ is sound and precise, where $M_{\text{alloc}} = \mathcal{O}$ and $M_{\text{call}} = \{x \leftarrow m(y) \mid m \in \mathcal{M}\}$.

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

Restricting to leaked abstract objects. We can further reduce the size of M_{alloc} – 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, so it suffices to restrict M_{alloc} to include abstract objects that may leak into missing code. It is even sound to use the monitoring scheme $\tilde{M}_{\text{alloc}}$, which monitors allocations o such that o may be explicitly passed to the library via a function call $x \leftarrow m(y)$, where $y \hookrightarrow o$:

$$\tilde{M}_{\text{alloc}} = \{o \in \mathcal{O} \mid y \hookrightarrow o \in \Pi \text{ where } x \leftarrow m(y)\}.$$

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

▶ **Proposition 7.** The monitoring scheme $\tilde{M}_{\text{min}}$ constructed using the current points-to edges Π 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 7, no counterexamples occur in any subsequent executions, i.e., Π is sound for all subsequent executions.

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

▶ **Proposition 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 $M \subsetneq \tilde{M}_{\text{min}}$, 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 $\mathcal{O} = \mathcal{O}_P \cup \mathcal{O}_L$, where abstract objects in $\mathcal{O}_P$ are in available code and abstract objects in $\mathcal{O}_L$ are in missing code. Then, our analysis cannot compute points-to edges $x \hookrightarrow o$, where $o \in \mathcal{O}_L$. As described previously, we assume that the static analysis only needs to compute relations involving program values. However, points-to edges $x \hookrightarrow o \in \mathcal{V}_P \times \mathcal{O}_L$ (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 `mkStr` is missing, then o_{str} is missing, so our static analysis cannot compute $\texttt{str} \hookrightarrow o_{\text{str}}$ (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 `str` may have type `String`, and that `str` and `data` may be aliased.

We handle allocations in library code by constructing the following:

▶ **Definition 9.** A *proxy object mapping* ϕ maps $\bar{o} \rightsquigarrow p$, where $\bar{o}$ is a concrete object allocated in missing code, and $p = \phi(\bar{o}) \in \mathcal{P}$ is a fresh abstract object called a *proxy object*; here, $\mathcal{P}$ is the set of all proxy objects.

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

Given ϕ, our analysis proceeds as before. It makes optimistic assumptions, initializes $\Pi_{\text{miss}} \leftarrow \emptyset$, and instruments the program using the monitoring scheme $\tilde{M}_{\text{min}}$ defined in Proposition 6. For any report $(x \leftarrow *, \bar{o})$, if $\bar{o}$ is not allocated at a visible allocation, our analysis concludes that $\bar{o}$ must have been allocated in missing code, so it adds $\bar{o} \rightsquigarrow p = \phi(\bar{o})$ to the abstract object mapping. Now, if a detected dynamic points-to edge $x \hookrightarrow p$ is missing, it is reported as a counterexample and added to $\Pi_{\text{miss}} \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{P})$, and Π 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 $x \hookrightarrow p$.

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

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

▶ **Definition 10.** The *dynamic footprint* of a concrete object $\bar{o}$ is the set of all visible variables that ever point to $\bar{o}$ during an execution.

The concrete type of $\bar{o}$ 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 $\bar{o}$ available to the static analysis, namely, the visible variables that point to $\bar{o}$.

Then, the proxy object mapping ϕ should map each concrete object $\bar{o}$ to a proxy object p so that the corresponding *static footprint* $\{x \in \mathcal{V}_P \mid x \hookrightarrow p \in \Pi^*\}$ soundly overapproximates the dynamic footprint of $\bar{o}$ 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, ϕ 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 $\phi(\bar{o}) = p_{\bar{o}}$.
- **Singleton $\mathcal{P}$:** Map each concrete object to a single proxy object $\phi(\bar{o}) = p$.

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 ϕ to map $\bar{o}$ 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 11.** An *ideal proxy object* $\tilde{p} \in \tilde{\mathcal{P}} = 2^{\mathcal{V}_P}$ is a set of visible variables. The *ideal proxy object mapping* $\tilde{\phi}(\bar{o}) \in \tilde{\mathcal{P}}$ is the dynamic footprint of $\bar{o}$.

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 $\bar{o}_{\text{str}}$ is allocated at missing abstract object o_{str} in an execution where `randBool` returns `false`. Then, $\tilde{\phi}$ maps $\bar{o}_{\text{str}}$ to ideal proxy object $\tilde{p}_{\text{str}} = \{\texttt{str}, \texttt{data}\}$. The reported counterexamples

$$\tilde{\Pi}_{\text{miss}} = \{\texttt{str} \hookrightarrow \tilde{p}_{\text{str}}, \texttt{data} \hookrightarrow \tilde{p}_{\text{str}}\}$$

are added to our static analysis, which additionally computes $\texttt{dataCopy} \hookrightarrow \tilde{p}_{\text{str}}$.

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

▶ **Proposition 12.** If $x \hookrightarrow \bar{o}$ occurs during execution and $\bar{o}$ is allocated at abstract object o, then $x \hookrightarrow o \in \tilde{\Pi}^*$ (if $o \in \mathcal{O}_P$) or $x \hookrightarrow \tilde{p} \in \tilde{\Pi}^*$ (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 $x \hookrightarrow \tilde{p}$ and $y \hookrightarrow \tilde{p}$, where $\tilde{p} = \tilde{\phi}(\bar{o})$, so the alias analysis determines that x and y may be aliased. Also:

▶ **Proposition 13.** Let $\Pi \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{O}_L)$ 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 $x \hookrightarrow o \in \tilde{\Pi}^*$, then $x \hookrightarrow o \in \Pi$. For $\tilde{p} = \tilde{\phi}(\bar{o}) \in \tilde{\mathcal{P}}$, if $x \hookrightarrow \tilde{o} \in \Pi^*$, then $x \hookrightarrow o \in \Pi$, where o is the statement where $\bar{o}$ was allocated.

In other words, $\tilde{\Pi}^*$ is at least as precise as the points-to edges Π computed with all code available. We prove these two propositions in Appendix B.1 & B.2. In our example, with all code available, we compute $\texttt{str} \hookrightarrow o_{\text{str}}$, $\texttt{data} \hookrightarrow o_{\text{str}}$, and $\texttt{dataCopy} \hookrightarrow o_{\text{str}}$, which is equivalent to $\tilde{\Pi}^*$ (replacing $\tilde{p}_{\text{str}}$ 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 `randBool` returns true. Then, the concrete object $\bar{o}'_{\text{str}}$ allocated at missing abstract object o_{str} is mapped to the ideal proxy object $\tilde{p}'_{\text{str}} = \{\texttt{str}, \texttt{data}, \texttt{dataCopy}\}$. However, the static footprint of $\tilde{p}'$ equals that of $\tilde{p}$ (from the first execution, where `randBool` returns false), even though $\tilde{p} \neq \tilde{p}'$ – i.e., $\bar{o}_{\text{str}}$ and $\bar{o}'_{\text{str}}$ 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 14.** The *dynamic function footprint* of a concrete object $\bar{o}$ is the set of library functions $m \in \mathcal{M}$ such that $r_m \hookrightarrow \bar{o}$ during execution.

Now, we use the following proxy object mapping:

▶ **Definition 15.** 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 ϕ, it suffices to monitor calls $x \leftarrow m(y)$ to missing functions. Continuing our example, ϕ maps the concrete object $\bar{o}_{\text{str}}$ allocated at missing abstract object o_{str} to $p_{\text{str}} = \{\texttt{mkStr}, \texttt{get}\}$ regardless of the return value of `randBool`. If `randBool` returns true, then the reported counterexamples are

$$\Pi_{\text{miss}} = \{\texttt{str} \hookrightarrow p_{\text{str}},\ \texttt{data} \hookrightarrow p_{\text{str}},\ \texttt{data} \hookrightarrow p_{\text{str}}\},$$

in which case our static points-to analysis does not compute any additional edges. If `randBool` returns false, then the reported counterexamples are

$$\Pi_{\text{miss}} = \{\texttt{str} \hookrightarrow p_{\text{str}},\ \texttt{data} \hookrightarrow p_{\text{str}}\},$$

from which our static analysis also computes $\texttt{dataCopy} \hookrightarrow p_{\text{str}}$. The static footprint of p_{str} is the same either way, and also equals those of $\tilde{p}_{\text{str}}$ and $\tilde{p}'_{\text{str}}$.

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

▶ **Proposition 16.** For any abstract object $o \in \mathcal{O}_P$, $x \hookrightarrow o \in \tilde{\Pi}^* \Leftrightarrow x \hookrightarrow o \in \Pi^*$. Furthermore, for any concrete object $\bar{o}$ allocated in missing code, letting $\tilde{p} = \tilde{\phi}(\bar{o})$ and $p = \phi(\bar{o})$, we have $x \hookrightarrow \tilde{p} \in \tilde{\Pi}^* \Leftrightarrow x \hookrightarrow p \in \Pi^*$.

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 12 and 13. 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 ϕ using only M_{call}):

▶ **Proposition 17.** The monitoring scheme $\tilde{M}_{\text{min}}$ is sound and precise.

5 Specification Inference

Rather than simply adding reported missing points-to edges to Π_{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 [8]. Given a reported missing points-to edge $x \hookrightarrow o$, this algorithm infers specifications in two steps:

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

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

Figure 4 Pessimistic functions used for specification inference; m_{gen} (left) is fully general (assuming functions do not access global state), whereas m_{res} (right) is restricted in various ways. For simplicity, we omit the receiver in m_{gen}.

- **Pessimistic assumptions:** Take $\hat{m} = m_{\text{pess}}$ for every missing function $m \in \mathcal{M}$, for some function m_{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_{pess}) that are needed to compute $x \hookrightarrow o$ statically; these statements are the inferred specifications.

The second step involves computing the static analysis using a shortest-path style algorithm, where pessimistic statements are assigned weight 1 and all other statements are assigned weight 0. 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, the priority of the derived points-to edge is the sum of the weights of the statements in the premise as well as the priorities of points-to edges in the premise. Thus, if a computed points-to edge has positive weight, we know that it can only be derived using a statement in m_{pess}.

Pessimistic function. A key design choice is the pessimistic function m_{pess} to use. The choice in [8], which we term the *general* function m_{gen}, is shown in Figure 4 (left). The code shown in this figure is compilable Java code, except we have added a new field `f` to the `Object` class. In particular, this code can be analyzed by our static analysis. As described above, during specification inference, we run our static analysis using either m_{gen} or m_{res} in place of m for every missing function $m \in \mathcal{M}$.

Using m_{gen} is sound, assuming library functions do not access global fields; see [8] for a formal proof. Intuitively, the argument ob of m_{gen} can be assigned to its return value $r_{m_{\text{gen}}}$; furthermore, the while loop ensures that arbitrary fields in the argument ob can be stored into fields in the return value $r_{m_{\text{gen}}}$ or into other fields of ob. In principle, we could even modify m_{gen} to handle library functions that access global fields by adding statements `x = X.g` and `X.g = x`, for every global field `X.g`. However, none of the specifications we have written so far access global fields, so this restriction improves running time without unsoundness in practice.

Unfortunately, even with this restriction, using m_{gen} results in a huge search space of candidate specifications. As a consequence, when using m_{gen}, the specification inference algorithm infers many incorrect specifications – in particular, there may be many specifications that yield a missing points-to edge, so the algorithm may infer the wrong ones.

Instead, we use pessimistic assumptions that restrict the search space to only consider candidate specifications that are common in practice, thus reducing the possibility of inferring an incorrect specification (at the cost of being unable to infer more complex specifications). In particular, we only consider candidate specifications that (i) do not accesses deep field paths, (ii) only access receiver fields, and (iii) does not allocate objects. These constraints lead to the *restricted* function m_{res} shown in Figure 4 (right).

Proxy object specifications. One of the restrictions on m_{res} is to assume that the candidate specifications do not allocate objects. To alleviate the consequences of this restriction, we separately infer specifications that allocate objects. In particular, we infer *proxy object specifications* of the form $(X, \{m\})$, where $X \in \mathcal{C}$ and $m \in \mathcal{M}$ 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 $p \in \mathcal{P}$ 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 $f \in \mathcal{F}_P \cap \mathcal{F}_L$ 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 `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 [7] 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:

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

```
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 `onLocationChanged`; by Proposition 6, it suffices to monitor all callback parameters.

6.3 Concrete Types

Some client analyses additionally need the concrete type $X \in \mathcal{C}$ of abstract objects $o = (x \leftarrow X())$, for example, virtual call resolution. To compute concrete types for proxy objects, each monitor $x \leftarrow *$ additionally records the concrete type of the concrete object pointed to by x

after executing the statement. Then, the proxy objects are extended to $\mathcal{P} = \mathcal{C} \times 2^{\mathcal{M}}$, and the proxy object mapping ϕ maps $\bar{o} \rightsquigarrow p = (X, F)$, where $F \subseteq 2^{\mathcal{M}}$ is the dynamic function footprint of $\bar{o}$, and $X \in \mathcal{C}$ is the recorded concrete type of $\bar{o}$.

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 $(d, v) \hookrightarrow \bar{o}$ (i.e., d is the callstack when v pointed to $\bar{o}$), and (ii) $\bar{o} \rightsquigarrow (c, h)$ (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 $(d, v) \hookrightarrow (c, h)$. 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 $x \hookrightarrow o$ 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 and precise.

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 [26]. To monitor a statement `x=...`, we record (i) the value `System.identityHashCode(x)`, which 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. [5] This data is uploaded to a server in batches (by default, once every 500ms), which post-processes

[5] Even though `System.identityHashCode` is not guaranteed to return a unique hash, it is a sound overapproximation.

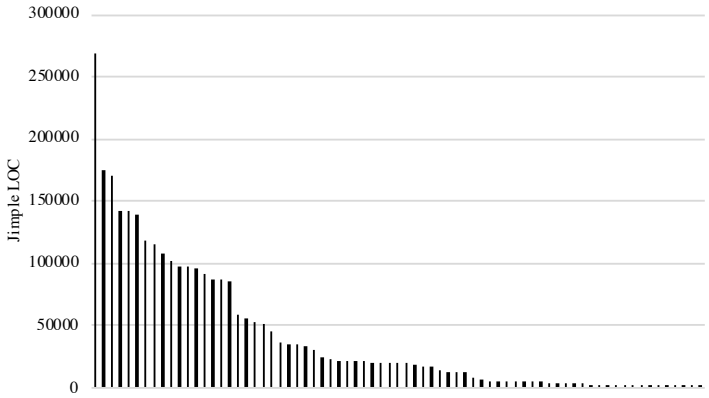

Figure 5 Sizes of the 73 apps in our benchmark in terms of Jimple list of code (LOC).

it to compute missing points-to edges and infer specifications. To obtain traces, we execute apps in the Android emulator and use Monkey [24] 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 [40] modified to use Soot as a front end [48]. The specification inference algorithm is based on shortest-path context-free reachability, described in [8]. We use a 1-CFA points-to analysis. As we discuss Section 6.4, using our more precise points-to analysis is eventually sound.

We use the information flow client from [21], which uses specifications to model missing code. It performs a static explicit information flow analysis [42] 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 [23, 5]. All analyses are computed using BDDBDDB [49].

8 Evaluation

We evaluate **Optix** on a benchmark of 73 Android apps, including battery monitors, games, wallpaper apps, and contact managers. The benchmarks are from two sources: the majority (40) are provided by a major security corporation (the "industry benchmark"), and the remaining (33) are provided as part of the DARPA APAC project on Android malware (the "DARPA benchmark"). The industry benchmark includes apps collected from the Google Play Store; the apps are primarily malware that leak sensitive information such as location, contacts, SMS messages, etc. The DARPA apps were challenging malware instances developed by a third party contractor as part of the program. Examples of apps from the industry benchmark include an app for monitoring your battery consumption, side scrolling action game, an implementation of the game of Mahjong, a maze game, and apps that let you set animated wallpapers. Examples of apps from the DARPA benchmark include a note taking app, an app that lets you use SMS messages to command your phone to perform various tasks, an app that keeps track of your jogging routes, a news collator, an app for organizing your podcasts, and an app for sharing your location. We provide the industry benchmark; [6] we cannot release the DARPA benchmark, but provide brief descriptions of

[6] `https://drive.google.com/open?id=1LzRhwtPisWWwKTd7lnHy7CE5Gf9iRkH4`

Rank	Recording Overhead (%)				Data (MB/hr)	
	initial	updated	# specs	worst	initial	worst
1	52.0 (0.38)	31.0	15	90.8 (0.02)	0.80 (0.59)	1.59 (0.16)
2	50.0 (0.02)	17.6	10	77.6 (0.01)	0.39 (0.01)	0.57 (0.39)
3	38.7 (0.10)	6.7	5	72.4 (0.04)	0.29 (0.38)	0.46 (0.30)
4	33.5 (0.65)	6.9	1	70.6 (0.15)	0.24 (0.15)	0.33 (0.78)
5	31.7 (0.10)	19.7	5	61.6 (0.23)	0.24 (0.55)	0.33 (0.64)
6	26.7 (0.25)	–	–	52.5 (0.03)	0.23 (0.80)	0.25 (0.04)
median	4.6 (0.10)	–	–	8.3 (0.10)	0.01 (0.35)	0.02 (0.30)

Figure 6 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 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. The coefficient of variation (across three runs) is shown in parentheses.

the apps in this benchmark in Appendix C. Finally, in Figure 5, we show a plot of the sizes of the apps in Jimple lines of code (LOC). 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 measure instrumentation overhead in each setting, on a 3.30GHz Intel Xeon E5-2667 CPU with 256 GB of memory. The server for collecting and analyzing reports was run concurrently on a different CPU core of the same machine. Our results are averaged over three runs.

Results. We show the highest runtime overheads in Figure 6, 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.[7] Columns "updated" and "# specs" are discussed below. We plot the runtime overhead of our recording instrumentation in Figure 7 (a), where the apps along the x-axis are sorted according to the overhead in the worst-case setting.

[7] We 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.

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 [33], though eventual soundness may no longer hold. Alternatively, both $\tilde{M}_{\text{alloc}}$ and M_{call} decrease 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 6 (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. We use our algorithm to compute points-to sets based on the reported counterexamples from Section 8.1. 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. Note that the worst-case instrumentation detects all counterexamples since it never needs to be updated, so this approach is sound. It may actually overapproximate the number of counterexamples – e.g., if a points-to edge $x \hookrightarrow o$ can be computed from another counterexample $y \hookrightarrow o$ that was already detected, our worst-case instrumentation reports $x \hookrightarrow o$ even though it is no longer a counterexample.

Counterexamples over time. Figure 7 (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 point. The values are averaged over all apps. By definition, at the end of the trace ($x = 1.0$), the fraction of reported missing points-to edges also goes to $y = 1.0$.

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

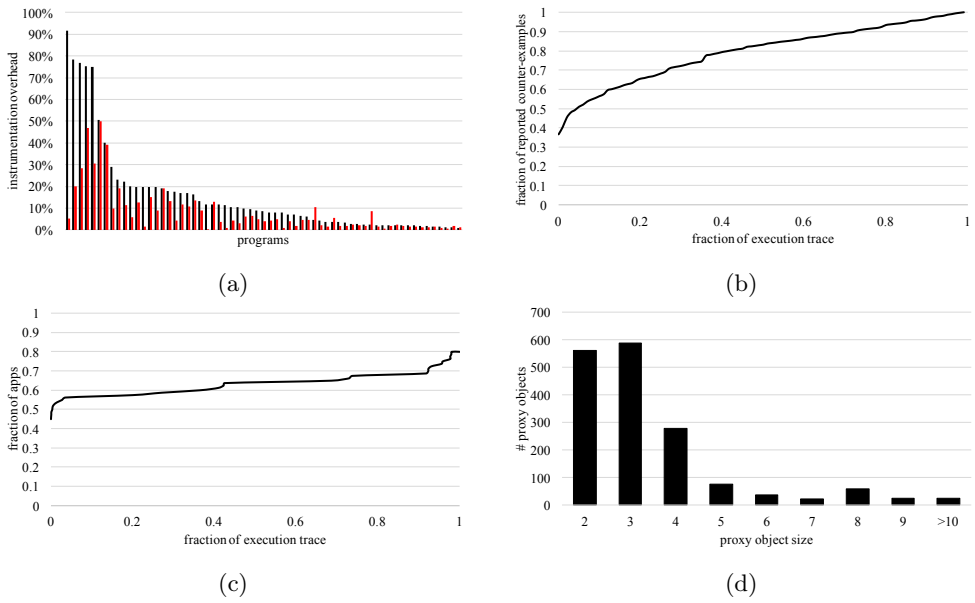

Figure 7 (a) Runtime overhead of our recording instrumentation in the worst-case setting (black) and the initial setting (red). The overheads are sorted by the overhead for the worst-case setting. (b) The x-axis is a fraction of the execution trace, and the curve shows the fraction of discovered missing points-to edges that discovered up to that point in the execution trace (averaged over all apps). (c) The x-axis is again a fraction of the execution trace, and the curve shows the number of apps for which no further missing points-to edges are reported after that point in the execution trace. (c) The distribution of the sizes of the proxy objects (i.e., the size of its dynamic function footprint), omitting footprints of size one.

curve is not yet flat at 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.[8]

Last discovered counterexample. Figure 7 (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.

[8] Since this curve is an overapproximation of the number of counterexamples, the true curve may actually be decreasing faster than this one. Nevertheless, we believe these results emphasize the importance of continued monitoring, since there are inevitably previously untested code paths that may contain bugs.

	Existing	Inferred	Correct	Accuracy
m_{res}	299	58	49	0.84
m_{gen}	299	159	33	0.22
proxy object	330	422	383	0.91

Figure 8 The number of specifications inferred using each m_{res} and m_{gen}, and the number of proxy object specifications inferred.

App	Jimple LOC	Time (min.)
0C2B78	322K	3.38
b9ac05	268K	1.06
highrail	247K	1.49
game	174K	0.08
androng	170K	0.48
median	19K	0.08

Figure 9 Statistics for the 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 the specification inference algorithm.

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 7 (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 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 [23, 5, 21, 8]. We demonstrate that the inferred specifications enable clients 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 8 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.

	Empty	Inf.	Base	Base ∪ Inf.	Ex. \ Inf.	Ex.	Ex. ∪ Inf.
specs.	0	432	189	621	371	629	803
flows	0	4	3	39	34	125	125
malware	0	2	3	22	12	49	49

Figure 10 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").

In Figure 9, 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 10, 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 classes `JSONArray`, `JSONObject`, and `BasicNameValuePair`, and for a few methods in `java.util`. As can be seen from Figure 10, 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 [37]. 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, as 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 in these apps.

9 Discussion

Global variables. We have assumed that library code does not access global variables; so far, none of the specifications we have written so far access global variables. We can extend our framework to handle global variables by instrumenting every load and store to global variables in the library, at the cost of additional runtime overhead. With this modification,

our theoretical results continue to hold using the same proofs – in particular, we can think of loads and stores to global variables in the library as calls to setter and getter functions in the library that load and store data from those variables.

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

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

```
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) [13], if the abstraction function α 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. Thus, 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 [16], which keep a pair of values $\langle x_{\text{private}}, x_{\text{public}} \rangle$ for each (visible) program variable x, where x_{private} may depend on sensitive data whereas x_{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 $r_{\texttt{add}} = \langle 15, 1 \rangle$. Since these two values differ, we conclude that $r_{\texttt{add}}$ depends on the sensitive input 14, and report that `add` transfers information from its argument `x` to its return value $r_{\texttt{add}}$. Essentially, this approach transforms the program so the abstraction function becomes computable. Alternatively, existing techniques for specification inference such as [8] 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, [10] proposes to use dynamic information to resolve reflective call targets, and then instruments the program to report additional counterexamples. Similarly, [7] proposes to compute reachable code by inserting runtime checks to report counterexamples to optimistic assumptions, and [22] uses a combination of static type checking and runtime checks to enforce type safety. General systems using dynamic analysis to detect gaps in the static analysis, typically local invariants assumed true by the static analysis, have also been proposed [11, 15]. Our setting is far more challenging, because (i) naïve dynamic points-to analyses incur unreasonable overhead [38, 12], and (ii) we cannot instrument missing code.

Additionally, [29] 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 [30, 31, 33] and information leaks [6, 16, 18] 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. Finally, there has been work on sound gradual typing [44, 41], which uses runtime monitoring to enforce given type specifications. The key difference between our approach and sound gradual typing is that the results of our static analysis improve as more counterexamples are detected.

Specification inference. There has been work on inferring specifications, e.g., purely static approaches that interact with a human analyst [52, 8, 1], and ones that rely on dynamic traces [12, 28, 4, 9]. 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, [2] 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) – 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.

Program synthesis has been used to infer specifications from dynamic traces [28]. This approach requires fine-grained instrumentation (specifically, leveraging features of the Javascript language to obtain alias traces), but they recover all method functionality. Their algorithm for doing so uses 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 [32, 27]. See [28] for a detailed discussion.

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 [21, 12, 25], as well as production systems designed to find bugs in Android apps [19]. 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 [43, 3, 50, 36, 49, 45]. 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 used to verify of security policies [35, 51, 23, 47, 5, 21, 25]. These approaches all 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.

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. In particular, our approach is dynamically sound in the sense that unsoundness (if any) is detected at the point when it occurs, thus enabling the user to terminate execution to prevent any damage from happening. We implement our approach in a tool called **Optix** to compute points-to sets for Android apps, where the Android framework is missing. For efficiency, **Optix** assumes that library functions do not access global variables; we have empirically found that this assumption holds. With this assumption, **Optix** achieves low runtime overhead and data usage on almost all apps in a large benchmark suite.

References

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

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

3 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

4 Steven Arzt and Eric Bodden. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 725–735. IEEE, 2016.

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

6 Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.

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

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

9 Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Active Learning of Points-To Specifications. In *PLDI*, 2018.

10 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

11 Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *International Symposium on Formal Methods*, pages 132–146. Springer, 2012.

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

13 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

14 Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, 2012.

15 David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 348–362. ACM, 2018.

16 Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. In *IEEE Symposium on Security and Privacy*, 2010.

17 Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *CCS*, 2013.

18 William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *TOCS*, 2014.

19 Facebook. Adding models, 2017. URL: `http://fbinfer.com/docs/adding-models.html`.

20 Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *CCS*, 2012.

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

22 Cormac Flanagan. Hybrid type checking. In *POPL*, 2006.

23 Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android, 2009.

24 Google. UI/Application Exerciser Monkey, 2016. URL: `https://developer.android.com/studio/test/monkey.html`.

25 Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS*. Citeseer, 2015.

26 Ben Gruver. Smali project homepage, 2016. URL: `https://github.com/JesusFreke/smali`.

27 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

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

29 Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *TOPLAS*, 2007.

30 Wei Jin and Alessandro Orso. BugRedux: reproducing field failures for in-house debugging. In *ICSE*, 2012.

31 Wei Jin and Alessandro Orso. F3: fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 213–223. ACM, 2013.

32 Tessa Lau, Pedro Domingos, and Daniel S Weld. Learning programs from traces using version space algebra. In *International Conference on Knowledge Capture*, 2003.

33 Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

34 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *CACM*, 2015.

35 V Benjamin Livshits and Monica S Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Usenix Security*, 2005.

36 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Software Engineering Notes*, 2002.

37 Bimal Kumar Mishra and Navnit Jha. Fixed period of temporary immunity after run of anti-malicious software on computer nodes. *Applied Mathematics and Computation*, 2007.

38 Markus Mock, Manuvir Das, Craig Chambers, and Susan J Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *PASTE*, 2001.

39 Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John C. Mitchell. Target Fragmentation in Android Apps. In *IEEE Security and Privacy Workshops*, 2016.

40 Mayur Naik, Alex Aiken, and John Whaley. *Effective static race detection for Java.* ACM, 2006.

41 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *POPL*, volume 50, pages 167–180, 2015.

42 Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 2003.

43 Olin Shivers. *Control-flow analysis of higher-order languages.* PhD thesis, Carnegie Mellon University, 1991.

44 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

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

46 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *ACM SIGPLAN Notices*, volume 40 (10), pages 59–76. ACM, 2005.

47 Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

48 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

49 John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

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

51 Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security*, 2006.

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