Automated Repair of Resource Leaks in Android Applications

Bhargav Nagaraja Bhatt[a], Carlo A. Furia[a]

[a]*Software Institute, USI Università della Svizzera italiana, Lugano, Switzerland*

Abstract

Resource leaks—a program does not release resources it previously acquired—are a common kind of bug in Android applications. Even with the help of existing techniques to automatically detect leaks, writing a leak-free program remains tricky. One of the reasons is Android's event-driven programming model, which complicates the understanding of an application's overall control flow.

In this paper, we present PLUMBDROID: a technique to automatically *detect and fix* resource leaks in Android applications. PLUMBDROID builds a succinct abstraction of an app's control flow, and uses it to find execution traces that may leak a resource. The information built during detection also enables automatically building a fix—consisting of release operations performed at appropriate locations—that removes the leak and does not otherwise affect the application's usage of the resource.

An empirical evaluation on resource leaks from the DROIDLEAKS curated collection demonstrates that PLUMBDROID's approach is scalable, precise, and produces correct fixes for a variety of resource leak bugs: PLUMBDROID automatically found and repaired 50 leaks that affect 9 widely used resources of the Android system, including all those collected by DROIDLEAKS for those resources; on average, it took just 2 minutes to detect and repair a leak. PLUMBDROID also compares favorably to Relda2/RelFix—the only other fully automated approach to repair Android resource leaks—since it can often detect more leaks with higher precision and producing smaller fixes. These results indicate that PLUMBDROID can provide valuable support to enhance the quality of Android applications in practice.

1. Introduction

The programming model of the Android operating system makes its mobile applications ("apps") prone to bugs that are due to incorrect usage of shared resources. An app's implementation typically runs from several entry points, which are activated by callbacks of the Android system in response to events triggered by the mobile device's user (for example, switching apps) or other changes in the environment (for example,

Email address: bhattb@usi.ch (Bhargav Nagaraja Bhatt)
URL: bugcounting.net (Carlo A. Furia)

losing network connectivity). Correctly managing shared resources is tricky in such an event-driven environment, since an app's overall execution flow is not apparent from the control-flow structure of its source code. This explains why *resource leaks*—bugs that occur when a shared resource is not correctly released or released too late—are one of the most common kind of performance bugs in Android apps [44, 30, 38], where they often result in buggy behavior that ultimately degrades an app's responsiveness and usability.

Motivated by their prevalence and negative impact, research in the last few years (which we summarize in Section 5) has developed numerous techniques to *detect* resource leaks using dynamic analysis [44, 13], static analysis [53, 55, 28], or a combination of both [12]. Automated detection is very useful to help developers in debugging, but the very same characteristics of Android programming that make apps prone to having resource leaks also complicate the job of coming up with leak *repairs* that are correct in all conditions.

To address these difficulties, we present a technique to detect *and fix* resource leaks in Android apps completely automatically. Our technique, called PLUMBDROID and described in Section 3, is based on static analysis and can build fixes that are correct (they eradicate the detected leaks for a certain resource) and "safe" (they do not introduce conflicts with the rest of the app's usage of the resource, and follow Android's recommendations for resource management [4]).

PLUMBDROID's analysis is scalable because it is based on a succinct abstraction of an app's control-flow graph called *resource-flow graph*. Paths on an app's resource-flow graph correspond to all its possible usage of resources. Avoiding leaks entails matching each acquisition of a resource with a corresponding release operation. PLUMBDROID supports the most general case of reentrant resources (which can be acquired multiple times, typically implemented with reference counting in Android): absence of leaks is a context-free property [46]; and leak detection amounts to checking whether every path on the resource-flow graph belongs to the context-free language of leak-free sequences. PLUMBDROID's leak model is more general than most other leak detection techniques'—which are typically limited to non-reentrant resources (see Table 8).

The information provided by our leak detection algorithm also supports the automatic *generation of fixes* that remove leaks. PLUMBDROID builds fixes that are correct by construction; a final validation step reruns the leak detection algorithm augmented with the property that the new release operations introduced by the fix do not interfere with the existing resource usages. Fixes that pass validation are thus correct and "safe" in this sense.

We implemented our technique PLUMBDROID in a tool, also called PLUMBDROID, that works on Android bytecode. PLUMBDROID can be configured to work on any Android resource API; we equipped it with the information about acquire and release operations of 9 widely used Android resources (including `Camera` and `WifiManager`), so that it can automatically repair leaks of those resources.

The current implementation of PLUMBDROID is not equipped with any aliasing analysis technique; therefore, it may incur a large number of false positives when applied to apps that use resources under lots of different aliases. We found that different Android resources have distinct usage patterns: some, such as database cursors, are frequently used under many aliases; others, such as the camera or the Wi-Fi adapter, are not.

PLUMBDROID's current implementation is geared towards analyzing the latter kind of resources, which we call *non-aliasing* resources and are the main target of our experiments with PLUMBDROID.

We evaluated PLUMBDROID's performance empirically on leaks of 9 non-aliasing resources in 17 Android apps from the curated collection DroidLeaks [29]. These experiments, described in Section 4, confirm that PLUMBDROID is a scalable automated leak repair technique (around 2 minutes on average to find and repair a leak) that consistently produces correct and safe fixes for a variety of Android resources (including all 26 leaks in DROIDLEAKS affecting the 9 analyzed resources).

We also experimentally compared PLUMBDROID with Relda2/RelFix [55, 28]—the only other fully automated approach to repair Android resource leaks that has been developed so far—by running the former on the same apps used in the latter's evaluation. The comparison, also described in Section 4, indicates that, on non-aliasing resources, PLUMBDROID detects more true leaks (79 vs. 53) with a higher average precision (89% vs. 55%) than Relda2/RelFix, and produces fixes that are one order of magnitude smaller.

PLUMBDROID's novelty and effectiveness lie in how it combines and fine-tunes existing analysis techniques to the usual characteristics of Android resource usage. 1. PLUMBDROID's fine-grained, yet succinct, abstract model of resource usage supports a sound static analysis with high precision; 2. PLUMBDROID's fix generation process follows Android's guidelines about the program locations where each resource should be released; 3. PLUMBDROID's validation step further guarantees that the fixes that it automatically generates are suitable and correct.

In summary, this paper makes the following contributions:

- It introduces PLUMBDROID: a fully automated technique based on static analysis for the detection and repair of Android resource leaks.
- It evaluates the performance of PLUMBDROID on apps in DROIDLEAKS, showing that it achieves high precision and recall, and scalable performance.
- It experimentally compares PLUMBDROID to the other approach Relda2/RelFix on the same apps used in the latter's evaluation, showing that it generally achieves higher precision and recall.
- For reproducibility, the implementation of PLUMBDROID, as well as the details of its experimental evaluation (including the produced fixes), are publicly available in a replication package:

  `https://github.com/bhargavbh/PlumbDROID` (Cite as [25])

2. An Example of PLUMBDROID in Action

IRCCloud is a popular Android app that provides a modern IRC chat client on mobile devices. Figure 1 shows a (simplified) excerpt of class `ImageViewerActivity` in IRCCloud's implementation.

```
1  public class ImageViewerActivity extends Activity {
2
3      private MediaPlayer player;
4
5      private void onCreate(BundleSavedInstance) {
6          // acquire resource MediaPlayer
7          player = new MediaPlayer();
8          final SurfaceView v = (SufaceView) findViewById(...);
9      }
10
11     public void onPause() {
12         v.setVisibilty(View.INVISIBLE);
13         super.onPause();
14         // 'player' not released: leak!
15     }
16
17 }
```

Figure 1: An excerpt of class `ImageViewerActivity` in Android app IRCCloud, showing a resource leak that PLUMBDROID can fix automatically.

As its name suggests, this class implements the *activity*—a kind of task in Android parlance—triggered when the user wants to view an image that she downloaded from some chat room. When the activity starts (method `onCreate`), the class acquires permission to use the system's media player by creating an object of class `MediaPlayer` on line 7. Other parts of the activity's implementation (not shown here) use `player` to interact with the media player as needed.

When the user performs certain actions—for example, she flips the phone's screen—the Android system executes the activity's method `onPause`, so that the app has a chance to appropriately react to such changes in the environment. Unfortunately, the implementation of `onPause` in Figure 1 does not *release* the media player, even though the app will be unable to use it while paused [40]. Instead, it just acquires a new handle to the resource when it resumes. This causes a *resource leak*: the acquired resource `MediaPlayer` is not appropriately released. Concretely, if the user flips the phone back and forth—thus generating a long sequence of unnecessary new acquires—the leak will result in wasting system resources and possibly in an overall performance loss.

Such resource leaks can be tricky for programmers to avoid. Even in this simple example, it is not immediately obvious that `onPause` may execute after `onCreate`, since this requires a clear picture of Android's reactive control flow. Furthermore, a developer may incorrectly assume that calling the implementation of `onPause` in Android's base class `Activity` (with `super.onPause()`) takes care of releasing the held resources. However, `Activity.onPause` cannot know about the resources that have been specifically allocated by the app's implementation; in particular, it does not release `MediaPlayer()` instance `player`.

PLUMBDROID can automatically analyze the implementation of IRCCloud looking for leaks such as the one highlighted in Figure 1. PLUMBDROID generates an abstraction

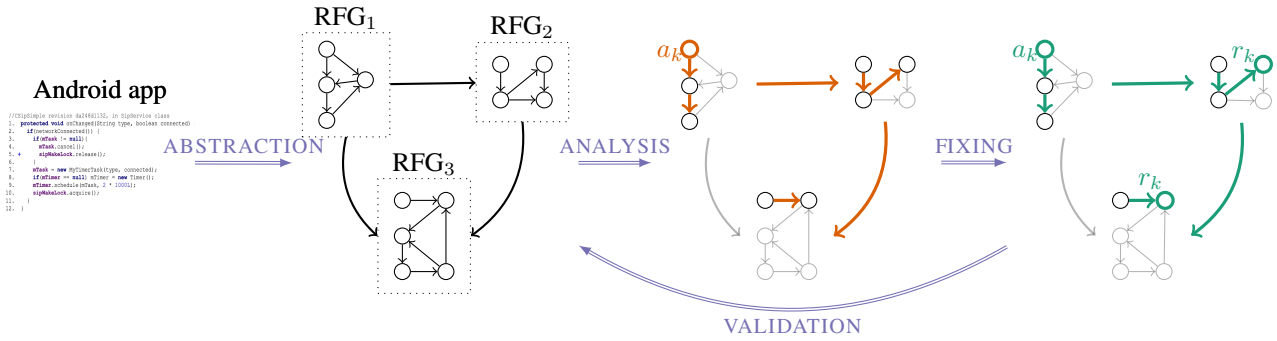


Figure 2: How PLUMBDROID works: First, PLUMBDROID builds a finite-state ABSTRACTION of the Android app under analysis, which captures *acquire* and *release* operations of an API's resources. The abstraction models each function of the application with a *resource-flow graph* (RFG)—a special kind of control-flow graph—and combines resource-flow graphs to model inter-procedural behavior. In the ANALYSIS step, PLUMBDROID searches the graph abstraction for *resource leaks*: paths where a resource k is acquired (a_k) but not eventually released. In the FIXING step, PLUMBDROID injects the missing *release operations* r_k where needed along the leaking path. In the final VALIDATION step, PLUMBDROID abstracts and analyzes the code after fixing, so as to ensure that the fix does not introduce unintended interactions that cause new resource-usage related problems.

of the whole app's control-flow that considers all possible user interactions that may result in leaks. For each detected leak, PLUMBDROID builds a *fix* by adding suitable release statements.

For Figure 1's example, PLUMBDROID builds a succinct fix at line 14 consisting of the conditional release operation `if (player != null) player.release()`. PLUMBDROID also checks that the fix is correct (it removes the leak) and "safe" (it only releases the resource after the app no longer uses it). Systematically running PLUMBDROID on Android apps can detect and fix many such resource leaks completely automatically.

3. How PLUMBDROID Works

Figure 2 gives a high-level overview of how PLUMBDROID works. Each run of PLUMBDROID analyzes an app for leaks of resources from a specific Android API—consisting of acquire and release operations—modeled as described in Section 3.1.

The key abstraction used by PLUMBDROID is the *resource-flow graph*: a kind of control-flow graph that captures the information about possible sequences of acquire and release operations. Section 3.2.1 describes how PLUMBDROID builds the resource-flow graph for each procedure individually.

A *resource leak* is an execution path where some *acquire* operation is not eventually followed by a matching *release* operation. In general, absence of leaks (leak freedom) is a *context-free property* [46] since there are resources—such as wait locks—that may

be acquired and released multiple times (they are *reentrant*).[1] Therefore, finding a resource leak is equivalent to analyzing context-free patterns on the resource-flow graph. PLUMBDROID's detection of resource leaks at the *intra-procedural* level is based on this equivalence, which Section 3.2.2 describes in detail.

Android apps architecture. An Android application consists of a collection of standard *components* that have to follow a particular programming model [9]. Each component type—such as activities, services, and content providers—has an associated *callback graph*, which constrains the order in which user-defined procedures are executed. As shown by the example of Figure 3, the *states* of a callback graph are macro-state of the app (such as *Starting*, *Running*, and *Closed*), connected by edges associated with callback *functions* (such as `onStart`, `onPause`, and `onStop`). An app's implementation defines procedures that implement the appropriate callback functions of each component (as in the excerpt of Figure 1).

Following this programming model, the overall control-flow of an Android app is not explicit from the app's implementation. Rather, the Android system triggers callbacks according to the transitions that are taken at run time (which, in turn, depend on the events that occur). PLUMBDROID deals with this *implicit* execution flow in two steps. First (Section 3.3.1), it defines an *explicit inter-procedural* analysis: it assumes that the inter-procedural execution order is known, and combines the intra-procedural analysis of different procedures to detect leaks across procedure boundaries. Second (Section 3.3.2), it unrolls the callback graph to enumerate sequences of callbacks that may occur when the app is running, and applies the explicit inter-procedural analysis to these sequences.

Fix generation. PLUMBDROID's analysis stage extracts detailed information that is useful not only to detect leaks but also to *generate fixes* that avoid the leaks. As we will describe in Section 3.4, PLUMBDROID builds fixes by adding a release of every leaked resource as early as possible along each leaking execution path.

PLUMBDROID's fixes are *correct by construction*: they release previously acquired resources in a way that guarantees that the previously detected leaks no longer occur. However, it might still happen that a fix releases a resource that is used later by the app—thus introducing a use-after-release error. In order to rule this out, PLUMBDROID also runs a final *validation step* which reruns the leak analysis on the patched program. If validation fails, it means that the fix should not be deployed as is; instead, the programmer should modify it in a way that makes it consistent with the rest of the app's behavior. Our experiments with PLUMBDROID (in particular in Section 4.2.1) indicate that validation is nearly always successful; even it fails, it is usually clear how to reconcile the fix with the app's behavior.

.

3.1. Resources

A PLUMBDROID analysis targets a specific Android API, which we model as a *resource list* L representing acquire and release operations of the API as a list of pairs

[1] For resources that do not allow nesting of acquire and release, leak freedom is a regular property—which PLUMBDROID supports as a simpler case.

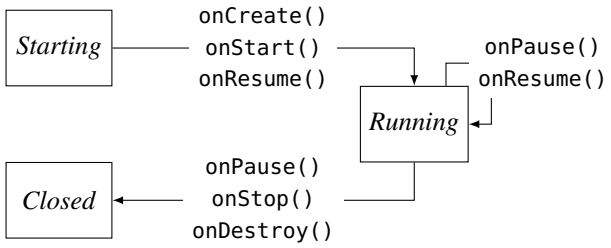


Figure 3: Simplified callback graph of an Android component.

$(a_1, r_1)\,(a_2, r_2)\ldots$. A pair (a_k, r_k) denotes an operation a_k that *acquires* a certain resource together with another operation r_k that *releases* the same resource acquired by a_k. The same operation may appear in multiple pairs, corresponding to all legal ways of acquiring and then releasing it. For simplicity, we sometimes use L to refer to the whole API that L represents. For example, resource `MediaPlayer` can be acquired and released with (**`new`**, `release`)—used in Figure 1.

3.2. Intra-Procedural Analysis

In the intra-procedural analysis, PLUMBDROID builds a resource-flow graph for every procedure in the app under analysis. In the example of Figure 1, it builds one such graph for every callback function, and for all methods called within those functions.

3.2.1. Resource-Flow Graphs

PLUMBDROID's analysis works on a modified kind of control-flow graph called *resource-flow graph* (RFG). The control-flow graphs of real-world apps are large and complex, but only a small subset of their blocks typically involve accessing resources. Resource-flow graphs abstract the control flow by only retaining information that is relevant for detecting resource leaks.[2]

A procedure's resource-flow graph R abstracts the procedure's control-flow graph C in two steps. First, it builds a *resource path* graph p for every *basic block* in C—as described by Algorithm 1. Then, it builds the resource-flow graph R by connecting the resource path graphs according to the control-flow structure—as described by Algorithm 2.

Resource path graph. A basic block corresponds to a sequence of statements without internal branching. Algorithm 1 builds the resource path graph p for any basic block b. It creates a node n in p for each statement s in b that is relevant to how L's resources are used: a resource is acquired or released, or execution terminates with a **`return`** (which may introduce a leak). Nodes in the resource path graph also keep track of when any other operation is performed, because this information is needed for inter-procedural analysis (as we detail in Section 3.3); in other words, intra-procedural

[2]Energy-flow graphs—used by other leak detection techniques for Android [12, 55]—are similar to resource-flow graphs in that they also abstract control-flow graphs by retaining the essential information for leak detection.

Input: control-flow basic block b, resource list L
Output: resource path graph p

```
1  p ← ∅ // initialize p to empty graph
2  foreach statement s in block b do
3  |  if s invokes a resource acquire operation a in L
4  |  |  n ← new AcquireNode(a)
5  |  elseif s invokes a resource release operation r in L
6  |  |  n ← new ReleaseNode(r)
7  |  elseif s invokes any other operation o
8  |  |  n ← new TransferNode(o)
9  |  elseif s is a return statement
10 |  |  n ← new ExitNode
11 |  else
12 |  |  n ← NULL
13 |  if n ≠ NULL
14 |  |  append node n to path graph p's tail
15 // if b contains no resource-relevant statements
16 if p = ∅
17 |  p ← new TrivialNode    // return a trivial node
```

Algorithm 1: Algorithm $Path$ that builds the *resource path graph* p modeling control-flow basic block b.

Input: control-flow graph C, resource list L
Output: resource-flow graph R

```
1  foreach block b in control-flow graph C do
2  |  // p(b) is the path graph corresponding to b ∈ C
3  |  p(b) ← Path(b, L)    // call to Algorithm 1
4  R ← { entry node s }
5  c0 ← the entry block of C
6  add an edge from s to p(c0)'s entry
7  foreach block b1 in control-flow graph C do
8  |  foreach block b2 in b1's successors in C do
9  |  |  add an edge from p(b1)'s exit to p(b2)'s entry
10 |  foreach ExitNode e in p(b1) do
11 |  |  add an edge from e's predecessors to f
12 |  |  (the exit node of R)
```

Algorithm 2: Algorithm RFG that builds a *resource-flow graph* R modeling control-flow graph C.

analysis is sufficient whenever a procedure doesn't have any transfer nodes. Graph p connects the nodes in the same sequential order as statements in b. When a block b does not include any operations that are relevant for resource usage, its resource path

graph p consists of a single *trivial* node, whose only role is to preserve the overall control-flow structure in the resource-flow graph. Since b is a basic block—that is, it has no branching—p is always a path graph—that is a linear sequence of nodes, each connected to its unique successor, starting from an entry node and ending in an exit node.

Resource-flow graph. Algorithm 2 builds the resource-flow graph R of control-flow graph C—corresponding to a single procedure. First, it computes a *path graph* $p(b)$ for every (basic) block b in C. Then, it connects the various path graphs following the control-flow graph's edge structure: it initializes R with an entry node s and connects it to the entry node of $p(c_0)$—the path graph of C'e entry block; for every edge $b_1 \rightarrow b_2$ connecting block b_1 to block b_2 in C, it connects the exit node of $p(b_1)$ to the entry node of $p(b_2)$. Since every executable block $b \in C$ is connected to C's entry block c_0, and c_0's path graph is connected to R's entry node s, R is a *connected* graph that includes one path subgraph for every executable block in the control-flow graph C. Also, R has a single entry node s and a single exit node f.

Given that R's structure matches C's, if there is a path in C that leaks some of L's resources, there is also a path in R that exposes the same leak, and vice versa. Thus, we use the expression "R has leaks/is free from leaks in L" to mean "the procedure modeled by C has leaks/is free from leaks of resources in the API modeled by L".

Figure 4 shows the (simplified) resource-flow graphs of methods `onCreate` and `onPause` from Figure 1's example. Since each method consists of a single basic block, the resource-flow graphs are path graphs (without branching).

3.2.2. Context-Free Emptiness: Overview

Given a resource-flow graph R—abstracting a procedure P of the app under analysis—and a resource list L, P is free from leaks of resources in L if and only if every execution trace in R consistently acquires and releases resources in L. We express this check as a formal-language inclusion problem—à la automata-based model-checking [48]—as follows (see Figure 5 for a graphical illustration):

1. We define a *resource automaton* A_L that accepts sequences of operations in L that are free from leaks. Since leak freedom is a (deterministic) context-free property, the resource automaton is a (deterministic) pushdown automaton.[3]

   (Resource automata are described in Section 3.2.3.)

2. We define the *complement* automaton $\overline{A_L}$ of A_L, which accepts precisely all sequences of operations in L that *leak*. Since A_L is a deterministic push-down automaton, $\overline{A_L}$ is too a deterministic push-down automaton.

   (Complement automata are described in Section 3.2.4.)

3. We define a *flow automaton* A_R that captures all possible paths through the nodes of resource-flow graph R. Since A_R is built directly from R, it is a (deterministic) finite-state automaton.

[3] We could equivalently use context-free grammars.

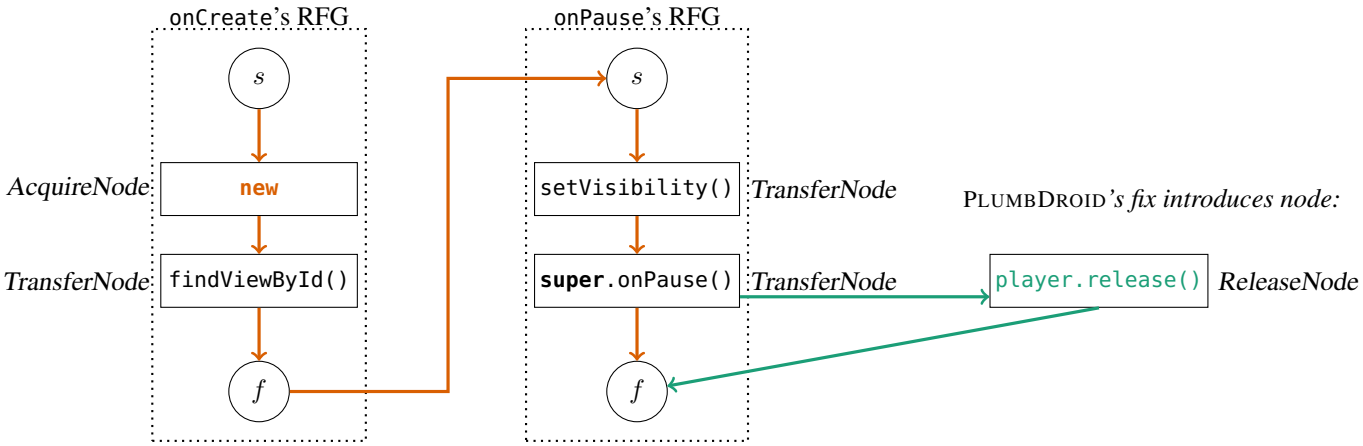


Figure 4: Some abstractions built by PLUMBDROID to analyze the example of Figure 1. The intra-procedural analysis described in Section 3.2 builds a resource-flow graph (RFG) for each procedure `onCreate` and `onPause` independently. As described in Section 3.3, the inter-procedural analysis considers, among others, the sequence of callback functions `onCreate(); onStart(); onResume(); onPause();` since `onStart()` and `onResume()` do nothing in this example, this corresponds to connecting `onCreate`'s exit to `onPause`'s entry. The inter-procedural analysis thus finds a *leaking path* (in orange), where acquire operation **`new`** is not matched by any release operation. Fixing (Section 3.4) modifies the app by adding a suitable release operation `player.release()` (in green), which completely removes the leak.

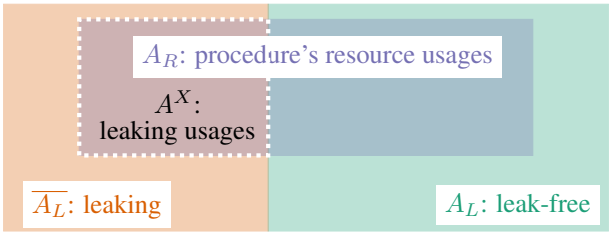


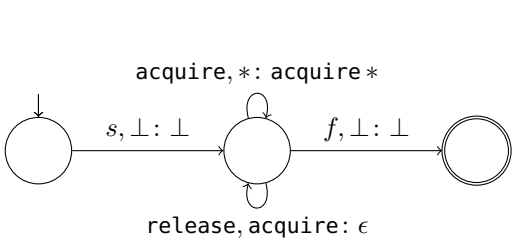
Figure 5: A graphical display of PLUMBDROID's intra-procedural leak detection. The *resource* automaton A_L captures all leak-free sequences; its *complement* $\overline{A_L}$ captures all leaking sequences. The *flow* automaton A_R captures all sequences of resource usages that may happen when the procedure under analysis runs. The *intersection* automaton A^X captures the intersection of the violet and orange areas (outlined in dotted white), which marks the procedure's resource usage sequences that are leaking. If this area is empty, we conclude that the procedure is leak free.

(Flow automata are described in Section 3.2.5.)

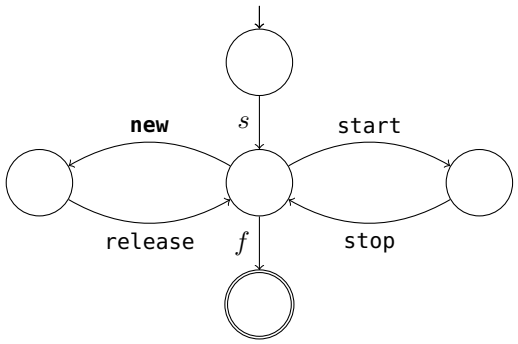
4. We define the intersection automaton $A^X = \overline{A_L} \times A_R$ that accepts all possible paths in R that introduce a leak in resource L.

   (Intersection automata are described in Section 3.2.6.)

5. We check if the intersection automaton accepts no inputs (the "empty language").



(a) Deterministic pushdown automaton A_{L_W} accepting the language of leak-free sequences of `acquire` and `release` operations of resource `WifiLock`. A transition $x, y: Z$ reads input x when y is on top of the stack, and replaces y with string Z. $*$ is a shorthand for any symbol, ϵ is the empty string, and $\perp$ is the empty stack symbol. The short arrow in the leftmost state denotes that it is an initial state; the double line in the rightmost state denotes that it is a final state.



(b) Deterministic finite-state automaton A_{L_M} accepting the language of leak-free sequences of operations of resource `MediaPlayer`. The short arrow in the top state denotes that it is an initial state; the double line in the bottom state denotes that it is a final state.

Figure 6: Resource automata for reentrant resource `WifiLock` and non-reentrant resource `MediaPlayer`.

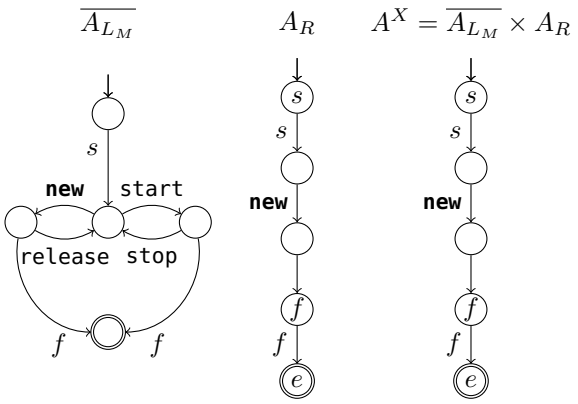


Figure 7: The intersection automaton A^X (right) combines the complement automaton $\overline{A_{L_M}}$ (left) of the `MediaPlayer`'s resource automaton A_{L_M} from Figure 6b and the flow automaton A_R (middle) of method `onCreate()` from Figure 1.

If this is the case, it means that R cannot leak; otherwise, we found a leaking trace.

(Emptiness checking is described in Section 3.2.7.)

The following subsections present these steps in detail.

3.2.3. Resource Automata

Given a resource list $L = \{(a_1, r_1)\,(a_2, r_2)\,\ldots\,(a_n, r_n)\}$, the *resource automaton* A_L is a deterministic pushdown automaton[4] that accepts all strings that begin with a start character s, end with a final character f, and include all sequences of the characters $a_1, r_1, a_2, r_2, \ldots, a_n, r_n$ that encode all leak-free sequences of acquire and release operations.

More precisely, A_L is a deterministic pushdown automaton if the resource modeled by L is a *reentrant*, and hence it can be acquired multiple times. If the resource modeled by L is not reentrant, A_L is an ordinary finite state automaton (a simpler subclass of pushdown automata). In the remainder of this section, we recall the definitions of deterministic pushdown automata and finite-state automata, and we show the example of resource automaton of a reentrant resource, as well as one of a non-reentrant resource.

Pushdown automata. Pushdown automata [46] are finite-state automata equipped with an unbounded memory that is manipulated as a stack. For leak detection, we only need *deterministic* pushdown automata, where the input symbol uniquely determines the transition to be taken in each state.

Definition 1 (Deterministic pushdown automaton)**.** A *deterministic pushdown automaton* A is a tuple $\langle \Sigma, Q, I, \Gamma, \delta, F \rangle$, where: 1. Σ is the input alphabet; 2. Q is the set of control states; 3. $I \subseteq Q$ and $F \subseteq Q$ are the sets of initial and final states; 4. Γ is the stack alphabet, which includes a special "empty stack" symbol $\perp$; 5. and $\delta\colon Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$ is the transition function. An automaton's computation starts in an initial state with an empty stack $\perp$. When the automaton is in state q_1 with stack top symbol γ and input σ, if $\delta(q_1, \sigma, \gamma) = (q_2, G)$ is defined, it moves to state q_2 and replaces symbol γ on the stack with string G. $\mathcal{L}(A) \subseteq \Sigma^*$ denotes the set of all input strings s accepted by A, that is such that A can go from one of its initial states to one of its final states by inputting s.

Reentrant resources. Consider a *reentrant* resource with resource list $L = \{(a_1, r_1)\,(a_2, r_2)\,\ldots\,(a_n, r_n)\}$. The resource automaton A_L is a deterministic pushdown automaton that operates as follows. When A^L inputs an acquire operation a_k, it pushes it on to the stack; when it inputs a release operation r_k, if the symbol on top of the stack is some a_k such that $(a_k, r_k) \in L$, it pops a_k—meaning that the release matches the most recent acquire: since all operations in L correspond to the same resource, any release has to refer to the latest acquire. Finally, A^L accepts the input if it ends up with an empty stack.

Example 1 (Resource automaton for `WifiLock`)**.** Let's illustrate this construction in detail for the case of resource `WifiLock` with $L_W = \{(\texttt{acquire}, \texttt{release})\}$ that is reentrant (see Table 1). Pushdown automaton A_{L_W} in Figure 6a accepts all strings over alphabet $\Sigma^{L_W} = \{s, f, \texttt{acquire}, \texttt{release}\}$ of the form $s\,B\,f$ where B is a string $B \in \{\texttt{acquire}, \texttt{release}\}^*$ is any balanced sequence of `acquire` and `release`—that is, a leak-free sequence.

[4]Precisely, a visibly pushdown automaton (a subclass of deterministic pushdown automata [1]) would be sufficient.

Finite-state automata. Finite-state automata can be seen as a special case of pushdown automata without stack.

Definition 2 (Finite-state automaton)**.** A *finite-state automaton* A is a five-element tuple $\langle \Sigma, Q, I, \delta, F \rangle$, where: 1. Σ is the input alphabet; 2. Q is the set of control states; 3. $I \subseteq Q$ and $F \subseteq Q$ are the sets of initial and final states; 4. and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. An automaton's computation starts in an initial state. When the automaton is in state q_1 with input σ, if $q_2 \in \delta(q_1, \sigma)$, it may move to state q_2. $\mathcal{L}(A) \subseteq \Sigma^*$ denotes the set of all input strings s accepted by A, that is such that A can go from one of its initial states to one of its final states by inputting s.

A finite-state automaton is *deterministic* when its transition relation δ is actually a function: the input uniquely determines the next state.

Non-reentrant resources. Consider a *non*-reentrant resource with resource list $L = \{(a_1, r_1)\,(a_2, r_2)\,\ldots\,(a_n, r_n)\}$. The resource automaton A_L is a finite-state automaton that operates as follows. When A^L inputs an acquire operation a_k, it moves to a new fresh state that is not final; the only valid transitions out of this state correspond to release operations r_k such that $(a_k, r_k) \in L$. Thus, A^L accepts the input only if every acquire operation is immediately followed by a matching release operation.

Example 2 (Resource automaton for `MediaPlayer`)**.** Resource `MediaPlayer` (used in the example of Section 2) is instead not reentrant (see Table 1), and offers operations $L_M = \{(\texttt{new}, \texttt{release})\,(\texttt{start}, \texttt{stop})\}$. The finite-state automaton A_{L_M} in Figure 6b accepts all strings over alphabet $\Sigma^{L_M} = \{s, f, \texttt{new}, \texttt{release}, \texttt{start}, \texttt{stop}\}$ of the form $s\,(\texttt{new}\,\texttt{release} \mid \texttt{start}\,\texttt{stop})^*\,f$, where each acquire operation is immediately followed by the matching release operation—that is, all leak-free sequences.

3.2.4. Complement Automata

Deterministic pushdown automata are a strict subclass of (nondeterministic) pushdown automata [1]. Unlike general pushdown automata, deterministic pushdown automata are closed under *complement*. That is, given any deterministic pushdown automaton A, we can always build the complement $\overline{A}$: a deterministic pushdown automaton that accepts precisely the inputs that A rejects and vice versa. For brevity, we do not repeat the classic construction of complement automata [46] The key idea is to switch final and non-final states, so that every computation that ends in a final state in A will be rejected in $\overline{A}$ and vice versa. Since finite-state automata are a subclass of deterministic pushdown automata, they are closed under complement too.

For our purposes, we need a slightly different complement automaton: one that accepts all sequences that begin with s, end with f, and include any leaking sequence of acquire and release in between these markers. For example, the automaton on the left in Figure 7 is the complement of `MediaPlayer`'s resource automaton in Figure 6b that PLUMBDROID builds: it only accepts sequences that terminate with an f when there is a pending acquired resource before it's released.

3.2.5. Flow Automata

Given a resource-flow graph $R = \langle V, E \rangle$, the *flow automaton* A_R is a deterministic finite-state automaton that accepts precisely the language $\mathcal{L}(R)$ of all paths π through R such that π starts in R's entry node s and ends in R's exit node f.

More precisely, the flow automaton A_R is a tuple $\langle \Sigma^R, Q^R, I^R, \delta^R, F^R \rangle$, where: 1. $\Sigma^R = \Sigma^L$ are all acquire and release operations, plus symbols s and f; 2. $Q^R = V \cup \{e\}$ are all nodes of R plus a fresh exit node e; 3. $I^R = \{s\}$ is the unique entry node of R, and $F^R = \{e\}$ is the new unique exit node; 4. the transition relation δ^R derives from R's edges: for every edge $m \to n$ in R, $n \in \delta^R(m, \mathit{type}(m))$ is a transition from state m to state n that reads input symbol $\mathit{type}(m)$ corresponding to the type of node m (start, acquire, or release); plus a transition from R's exit node to the new exit node e reading f.

Without loss of generality, we can assume that A_R is *deterministic*, since finite-state automata are closed under determinization [46]. That is, even if the construction above gives a nondeterministic finite-state automaton A_R, we can always build an equivalent deterministic variant of it that accepts precisely the same inputs.

The middle automaton in Figure 7 is the flow automaton A_R of `onCreate` in Figure 1's running example: it is isomorphic to `onCreate`'s resource-flow graph (left in Figure 4) except for an additional final node e that follows f.

3.2.6. Intersection Automata

Given (the complement of) a resource automaton $\overline{A_L}$ and flow automaton A_R, the *intersection* automaton $A^X = \overline{A_L} \times A_R$ is a deterministic pushdown automaton that accepts precisely the intersection language $\overline{\mathcal{L}(A_L)} \cap \mathcal{L}(A_R)$, that is the inputs accepted by both the complement automaton $\overline{A_L}$ and the flow automaton A_R. Therefore, A^X precisely captures all sequences of acquire and release operations that may occur in R (that is, they are in $\mathcal{L}(A_R)$) and that leak some resource in L (that is, they are in $\overline{\mathcal{L}(A_L)} = \mathcal{L}(\overline{A_L})$ and thus are rejected by A_L).

More precisely, the intersection automaton A^X is a deterministic pushdown automaton $\langle \Sigma^X, Q^X, I^X, \Gamma^X, \delta^X, F^X \rangle$, where: 1. $\Sigma^X = \Sigma^L$ is the usual alphabet of all acquire and release operations, plus symbols s and f; 2. $Q^X = Q^R \times Q^L$ is the Cartesian product of A_R's and $\overline{A_L}$'s states; 3. $I^X = (i_1, i_2)$, where $i_1 \in I^R$ is an initial state of A_R and $i_2 \in I^L$ is an initial state of $\overline{A_L}$; 4. $F^X = (f_1, f_2)$, where $f_1 \in F^R$ is a final state of A_R and $f_2 \in F^L$ is a final state of $\overline{A_L}$; 5. $\Gamma^X = \Gamma^L$ is the stack alphabet of $\overline{A_L}$; 6. for every transition $p_2 \in \delta^R(p_1, \sigma)$ in A_R and every transition $(q_2, G) = \delta^L(q_1, \sigma, \gamma)$ in $\overline{A_L}$ that input the same symbol σ, A^X includes transition $((p_2, q_2), G) = \delta^X((p_1, q_1), \sigma, \gamma)$ that manipulates the stack as in $\overline{A_L}$'s transition. Since both A_R and $\overline{A_L}$ are deterministic, so is A^X.

The rightmost automaton in Figure 7 is the intersection automaton of Figure 1's running example. Since the flow automaton A_R is just a single path, it is identical to its intersection with $\overline{A_{L_M}}$, which accepts the single leaking path.

3.2.7. Emptiness Checking

In the last step of its intra-procedural leak detection, PLUMBDROID has built the intersection automaton A^X: a deterministic pushdown automaton that accepts all sequences of operations on resources L that may occur in the piece of code P corresponding to R and that *leak* some resource. In other words, R is free from leaks in L if and only if the intersection automaton A^X accepts the *empty* language—that is, no inputs at all.

It is another classic result of automata theory that checking whether any pushdown automaton accepts the empty language (that is, it accepts no inputs) is decidable in polynomial time [1].

PLUMBDROID applies this classic decision procedure to determine whether A^X accepts the empty language. If it does, then procedure P is leak free; otherwise, we found a *trace* of P that leaks.

3.3. Inter-Procedural Analysis

PLUMBDROID lifts the intra-procedural analysis to a whole app by analyzing all possible calls between procedures. The analysis of a given sequence of procedure calls combines the results of intra-procedural analysis as described in Section 3.3.1. Since in Android system callbacks determine the overall execution order of an app, Section 3.3.2 explains how PLUMBDROID unrolls the callback graph to enumerate possible sequences of procedure calls—which are analyzed as if they were an explicit call sequence.

Input: call graph $C = \langle V, E \rangle$, automaton $\overline{A^L}$
Output: $H = \{H_p \mid V \ni p \text{ is not called by any procedure}\}$

```
1  N ← topological sort of C
2  foreach n ∈ N do
3  |  // for each procedure n
4  |  if n is not calling any other procedure
5  |  |  // leaking paths in
6  |  |  // intra-procedural analysis of n
7  |  |  // R_n is the RFG of procedure n
8  |  |  H_n ← LeakingPaths(R_n, Ā^L, L)
9  |  elseif n calls procedures m_1, m_2, ...
10 |  |  R'_n ← R_n
11 |  |  foreach m ∈ {m_1, m_2, ...} do
12 |  |  |  // R'_n is R_n with call-to-m nodes
13 |  |  |  // replaced by H_m
14 |  |  |  R'_n ← R'_n[TransferNode(m) ↦ H_m]
15 |  |  // leaking paths in
16 |  |  // intra-procedural analysis of R'_n
17 |  |  H_n ← LeakingPaths(R'_n, Ā^L, L)
```

Algorithm 3: Algorithm $AllCalls$ which computes inter-procedural resource-flow paths accepted by "leaking" pushdown automaton $\overline{A^L}$. Function $LeakingPaths$ performs the intra-procedural detection of leaking paths described in Section 3.2.2.

3.3.1. Explicit Call Sequences

As it is customary, PLUMBDROID models calls between procedures with a *call graph* C: every node v in C is one of the procedures that make the app under analysis; and an

edge $u \to v$ in C means that u calls v directly. In our analysis, a call graph may have multiple entry nodes, since Android applications have multiple entry points.

PLUMBDROID follows Algorithm 3 to perform inter-procedural analysis based on the call graph. First of all, we use topological sort (line 1) to rank C's nodes in an order that is consistent with the call order encoded by C's edges: if a node P has lower rank than a node Q it means that P does not call Q. Topological sort is applicable only if C is acyclic, that is there are no circular calls between procedures. If it detects a cycle, PLUMBDROID's implementation issues a warning and then breaks the cycle somewhere. As we discuss in Section 3.6 and Section 4, the restriction to acyclic call graphs seems minor in practice since all apps we analyzed had acyclic call graphs.

Once nodes in C are ranked according to their call dependencies, Algorithm 3 processes each of them starting from those corresponding to procedures that do not call any other procedures (line 4). The resource-flow graph of such procedures doesn't have any *transfer* nodes, and hence it can be completely analyzed using intra-procedural analysis.

Function $LeakingPaths$ performs the intra-procedural leak detection technique of Section 3.2.2 and returns any *leaking paths* in the procedure. The leaking path, if it exists, is used as a *summary* of the procedure.

Procedures that are free from leaks have an empty path as summary; therefore, they are neutral for inter-procedural analysis. In contrast, procedures that may leak have some non-empty path as summary, which can be combined with the summary of other procedures they call to find out whether the combination of caller and callee is free from leaks. This is done in lines 9–17 of Algorithm 3: the resource-flow graph of a procedure n that calls another procedure m includes some transfer nodes to m; we replace those nodes with the summary of m (which was computed before thanks to the topological sorting), and perform an analysis of the call-free resource-flow graph with summaries. The output of Algorithm 3 are complete summaries for the whole app starting from the entry points.

3.3.2. Implicit Call Sequences

Callbacks in every component used by an Android app have to follow an execution order given by the component's callback graph: a finite-state diagram with callback functions defined on the edges (see Figure 3 for a simplified example). Apps provide implementations of such callback functions, which PLUMBDROID can analyze for leaks. When an edge's transition is taken, *all* callback functions defined on the edge are called in the order in which they appear.

The documentation of every resource defines callback functions where the resource ought to be released. PLUMBDROID enumerates all paths that first go from the callback graph's entry to the nodes from where the release callback functions can be called, and then continue looping until states are traversed up to D times—where D is a configurable parameter of PLUMBDROID called "unrolling depth" (see Section 4.2.5 to see its impact in practice). Each unrolled path determines a sequence of procedures $P_1; P_2; \ldots$ used in the callback functions in that order. PLUMBDROID looks for leaks in these call sequences by analyzing them as if they were explicit calls in that sequence—using the approach of Section 3.3.1.

For example, Figure 1's resource `MediaPlayer` should be released in callback function `onPause`. For a component with the callback graph of Figure 3, and unrolling depth $D = 2$, PLUMBDROID enumerates the path $\mathit{Starting} \rightarrow \mathit{Running} \rightarrow \mathit{Running} \rightarrow \mathit{Closed}$, corresponding to callback sequence `onCreate(); onStart(); onResume(); onPause(); onResume(); onPause(); ...`. If the media player is acquired and not later released in these call's implementations, PLUMBDROID will detect a leak. Since callback functions `onStart` and `onResume` are not implemented in Figure 1's simple example, Figure 4 displays the initial part of this sequence of callbacks by connecting in a sequence the resource-flow graphs of `onCreate` and `onPause`.

3.4. Fix Generation

Fix templates. Once PLUMBDROID detects a resource leak, fixing it amounts to injecting missing release operations at suitable locations in the app's implementation. PLUMBDROID builds fixes using the conditional template:
`if (resource != null &&` held`) resource.`r`()`, where `resource` is a reference to the resource object, r is the release operation (defined in the resource's API), and held is a condition that holds if and only if the resource is actually not yet released. Calls to release operations must be conditional because PLUMBDROID's analysis is an over-approximation (see Section 3.6) and, in particular, a *may leak* analysis [42]: it is possible that a leak occurs only in certain conditions, but the fix must be correct in all conditions. Condition held depends on the resource's API: for example, wake locks have a method `isHeld()` that perfectly serves this purpose; in other cases, the null check is enough (and hence held is just `true`). Therefore, PLUMBDROID includes a definition of held for every resource type, which it uses to instantiate the template.

Another complication in building a fix arises when a reference to the resource to be released is not visible in the callback where the fix should be added. In these cases, PLUMBDROID's fix will also introduce a fresh variable in the same component where the leaked resource is *acquired*, and make it point to the resource object. This ensures that a reference to the resource to be released is visible at the fix location.

Fix injection. A fix's resource release statement may be injected into the application at different locations. A simple, conservative choice would be the component's final callback function (`onDestroy` for activity components). Such a choice would be simple and functionally correct but very inefficient, since the app would hold the resource for much longer than it actually needs it.

Instead, PLUMBDROID uses the information computed during leak analysis to find a suitable release location. As we discussed in Section 3.3.2, the overall output of PLUMBDROID's leak analysis is an execution path that is leaking a certain resource. The path traverses a sequence $C_1; C_2; \ldots; C_n$ of callback functions determined by the component's callback graph, and is constructed by PLUMBDROID in a way that it ends with a call C_n to the callback function where the resource may be released (according to the resource's API documentation). Therefore, PLUMBDROID adds the fix statement in callback C_n just after the last usage of the resource in the callback (if there is any).

In the running example of Figure 1, PLUMBDROID inserts the call to `release` in callback `onPause`, which is as early as possible in the sequence of callbacks—as shown in the rightmost node of Figure 4.

3.5. Validation

Since leak analysis is sound (see Section 3.6), PLUMBDROID's fixes are correct by construction in the sense that they will remove the leak that is being repaired. However, since the resource release statement that fixes the leak is inserted in the first suitable callback (as described in Section 3.4), it is possible that it interferes in unintended ways with other *usages* of the resource.

In order to determine whether its fixes may have introduced inconsistencies of this kind, PLUMBDROID performs a final *validation* step, which runs a modified analysis that checks absence of new leaks as well as absence of use-after-release errors. This analysis reuses the techniques of Section 3.2 and Section 3.3 with the only twist that the pushdown automaton characterizing the property to be checked is now extended to also capture absence of use-after-release errors.

PLUMBDROID's fixes release resources in the recommended callback function, but the app's developer may have ignored this recommendation and written code that still uses the resource in callbacks that occur later in the component's lifecycle. Such scenarios are the usual origin of failed validation: PLUMBDROID would fix the leak but it would also introduce a use-after-release error by releasing the resource too early.

Continuing Figure 1's example of resource `MediaPlayer`, suppose that the implementation does call the `release` operation but only in activity `onStop`. PLUMBDROID would still report a leak, since it does not find a `release` in `onPause`—which is where `MediaPlayer` should be released according to its API documentation. Accordingly, PLUMBDROID's fix for this leak would add `release` in `onPause` as described in Section 2. Validation of the leak would, however, fail because the programmer-written `release` in `onStop` introduces a double-release of the same resource, which validation detects.

If validation fails, PLUMBDROID still outputs the invalid fix, which can be useful as a *suggestion* to the developer—who remains responsible for modifying it in a way that doesn't conflict with the rest of the app's behavior. In particular, the experiments of Section 4.2.1 confirm the intuition that validation fails when PLUMBDROID releases resources in the callback function recommended by the official resource documentation but the rest of the app's behavior conflicts with this recommendation. Therefore, the developer has the options of adjusting the fix or refactoring the app to follow the recommended guidelines.

In the running example of `MediaPlayer`, the programmer may either ignore PLUMBDROID's suggestion and keep releasing the resource in `onStop` (against the resource's recommendations), or accept it and remove the late call of `release` that becomes no longer necessary.

Validation is an *optional* step in PLUMBDROID. This is because it is not needed if we can assume that the app under repair follows Android's recommendation for when (in which callbacks) a resource should be used and released. As we will empirically demonstrate in Section 4, validation is indeed usually not needed—but it remains available as an option in all cases where an additional level of assurance is required.

3.6. Features and Limitations

Let us summarize PLUMBDROID's design features and the limitations of its current implementation. Section 4 will empirically confirm these features and assess the prac-

tical impact of the limitations.

3.6.1. Soundness

A leak detection technique is *sound* [42] if, whenever it finds no leaks for a certain resource, it really means that no such leaks are possible in the app under analysis.

PLUMBDROID's intra-procedural analysis is sound: it performs an exhaustive search of all possible paths, and thus it will report a leak if there is one. The inter-procedural analysis, however, has two possible sources of unsoundness. (a) Since it performs a fixed-depth unrolling of paths in the callback graph (Section 3.3.2), it may miss leaks that only occur along longer paths. (b) Since it ranks procedures according to their call order (Section 3.3.1), and such an order is not uniquely defined if the call graph has cycles, it may miss leaks that only occur in other procedure execution orders.

Both sources of unsoundness are unlikely to be a significant limitation in practice [32]. A leak usually does not depend on the absolute number of times a resource is acquired or released, but only on whether acquires and releases are balanced. As long as we unroll each loop a sufficient number of times, unsoundness source (a) should not affect the analysis in practice. Furthermore, leak detection is monotonic with respect to the unrolling depth D: as D increases, PLUMBDROID may find more leaks by exploring more paths, but a leak detected with some D will also always be detected with a larger $D' > D$.

As for the second source of unsoundness, the Android development model, where the overall control flow is determined by implicit callbacks, makes it unlikely that user-defined procedures have circular dependencies. More precisely, PLUMBDROID's soundness is only affected by cycles in paths with acquire and release—not in plain application logic—and hence unsoundness source (b) is also unlikely to occur.

The experiments of Section 4 will confirm that PLUMBDROID is sound in practice by demonstrating that a wide range of Android applications trigger neither source of unsoundness.

3.6.2. Precision

A leak detection technique is *precise* [42] if it never reports false alarms (also called false positives): whenever it detects a leak, that leak really occurs in some executions of the app under analysis. In the context of leak repair, many false alarms would generate many spurious fixes, which do not introduce bugs (since the analysis is sound) but are useless and possibly slow down the app.

PLUMBDROID's analysis is, as is common for dataflow analyses, flow-sensitive but path-insensitive. This means that it over-approximates the paths that an app *may* take without taking into account the feasibility of those paths. As a very simple example, consider a program that only consists of statement `if (false) res.a()`, where `res` is a reference to a resource and `a` is an acquire operation. This program is leak free, since the lone acquire will never execute. However, PLUMBDROID would report a leak because it conservatively assumes that every branch is feasible.

Aliasing occurs when different references to the same resource may be available in the same app. Since PLUMBDROID does not perform alias analysis, this is another source of precision loss: a resource with two aliases `x` and `y` that is acquired using `x`

and released using `y` will be considered leaking by PLUMBDROID, which thinks `x` and `y` are different resources.

In practice, these two sources of imprecision are limitations to PLUMBDROID's applicability. When aliasing is not present, the experiments of Section 4 indicate that the path-insensitive over-approximation built by PLUMBDROID is very precise in practice. Whether aliasing is present mostly depends on the kind of resource that is analyzed. As we discuss in Section 4.1.1, it is easy to identify resources whose usage is unlikely to introduce aliasing. PLUMBDROID is currently geared towards detecting and repairing leaks of such resources.

A comprehensive empirical study of performance bugs in mobile apps [38] found that about 36% of the analyzed Android resource leaks were related to non-aliasing resources, such as `WiFi` and `Camera`, whose operations are energy intensive. The same study also reported that the leaks that affect these resources tend to have a much longer life than the memory-related leaks, as it takes developers longer, on average, to detect and fix the former over the latter. These data indicate that non-aliasing resource leaks are a significant fraction of the most common resource-related issues in Android apps, can have a negative impact on performance, and can be challenging to detect and fix. These observations motivate PLUMBDROID's focus on non-aliasing resources.

3.7. Implementation

We implemented PLUMBDROID in Python on top of ANDROGUARD [8] and APKTOOL [10].

PLUMBDROID uses ANDROGUARD—a framework to analyze Android apps—mainly to build the control-flow graphs of methods (which are the basis of our resource-flow graphs) and to process manifest files (extracting information about the components that make up an app). Using ANDROGUARD ensures that the control-flow graphs capture all behavior allowed by Java/Android (including, for example, exceptional control flow) in a simple form in terms of a few basic operations.

APKTOOL—a tool for reverse engineering of Android apps—supports patch generation: PLUMBDROID uses it to decompile an app, modify it with the missing release operations, and recompile the patched app back to executable format. PLUMBDROID's analysis and patching work on *Smali* code—a human-readable format for the binary bytecode format *DEX*, which is obtained by decompiling from and compiling to the *APK* format.

4. Experimental Evaluation

The overall goal of our experimental evaluation is to investigate whether PLUMBDROID is a practically viable approach for detecting and repairing resource leaks in Android applications. We consider the following research questions.

RQ1: Does PLUMBDROID generate fixes that are *correct* and "safe"?

RQ2: How do PLUMBDROID's fixes compare to those written by *developers*?

RQ3: Is PLUMBDROID *scalable* to real-world Android apps?

RESOURCE	OPERATIONS a_k	OPERATIONS r_k	RELEASED CALLBACK	REENTRANT?
AudioRecorder	**new**	release	onPause, onStop	N
BluetoothAdapter	enable startDiscovery	disable cancelDiscovery	onStop onPause	N
Camera	lock open startPreview	unlock release stopPreview	onPause	N
LocationListener	requestUpdates	removeUpdates	onPause	N
MediaPlayer	**new** start	release stop	onPause, onStop	N
Vibrator	vibrate	cancel	onDestroy	N
WakeLock	acquire	release	onPause	Y
WifiLock	acquire	release	onPause	Y
WifiManager	enable	disable	onDestroy	N

Table 1: Android resources analyzed with PLUMBDROID. For each RESOURCE, the table reports the acquire a_k and release r_k OPERATIONS it supports (according to the resource's API documentation [3]), the CALLBACK function `on...` where the resource should be RELEASED (according to the *Android developer guides* [4]), and whether the resource is REENTRANT (YES implies that absence of leaks is a context-free property and NO implies that it is a regular property).

RQ4: How does PLUMBDROID *compare* with other automated repair tools for Android resource leaks?

RQ5: How does PLUMBDROID's behavior depend on the *unrolling depth* parameter, which controls its analysis's level of detail?

4.1. Experimental Setup

This section describes how we selected the apps used in the experimental evaluation of PLUMBDROID, how we ran the experiments, and how we collected and assessed the experiments' results to answer the research questions.

4.1.1. Subjects: RQ1, RQ2, RQ3, RQ5

Our experiments to assess correctness and scalability target apps in DROIDLEAKS [29]—a curated collection of resource leak bugs in real-world Android applications. DROIDLEAKS collects a total of 292 leaks from 32 widely used open-source Android apps. For each leak, DROIDLEAKS includes both the buggy (leaking) version of an app and a leak-free version obtained by manually fixing the leak.

Leaks in DROIDLEAKS affect 22 resources. The majority of them (13) are Android-specific resources (such as `Camera` or `WifiLock`), while the others are standard Java

APP	KLOC	RESOURCE	LEAKS
APG	42.0	`MediaPlayer`	1
BarcodeScanner	10.6	`Camera`	1
CallMeter	13.5	`WakeLock`	3
ChatSecure	37.2	`BluetoothAdapter`	0
		`Vibrator`	1
ConnectBot	17.6	`WakeLock`	0
CSipSimple	49.0	`WakeLock`	2
IRCCloud	35.3	`MediaPlayer`	0
		`WifiLock`	1
K-9 Mail	78.5	`WakeLock`	2
OpenGPSTracker	12.3	`LocationListener`	1
OsmDroid	18.4	`LocationListener`	2
ownCloud	31.6	`WifiLock`	2
QuranForAndroid	21.7	`MediaPlayer`	1
SipDroid	24.5	`Camera`	4
SureSpot	41.0	`MediaPLayer`	2
Ushahidi	35.7	`LocationListener`	1
VLC	18.1	`WakeLock`	2
Xabber	38.2	`AudioRecorder`	2
AVERAGE	30.9		
TOTAL	525.2		26

Table 2: DROIDLEAKS apps analyzed with PLUMBDROID. For each APP, the table reports its size KLOC in thousands of lines of code. For each RESOURCE used by the app, the table then reports the number of LEAKS of that resource and app included in DROIDLEAKS. The two bottom rows report the AVERAGE (mean) and TOTAL for all apps.

APIs (such as `InputStream` or `BufferReader`). PLUMBDROID's analysis is based on the Android programming model, and every Android-specific resource expresses its usage policy in terms of the callback functions where a resource can be acquired or released—an information that is not available for standard Java API's resources. Therefore, our evaluation only targets leaks affecting Android-specific resources.

As we discussed in Section 3.6, PLUMBDROID is oblivious of possible aliases between references to the same resource object. If such aliasing happens within the same app's implementation, it may significantly decrease PLUMBDROID's precision. We found that each Android resource can naturally be classified into *aliasing* and *non-aliasing* according to whether typical usage of that resource in an app may introduce multiple references that alias one another.[5] Usually, a non-aliasing resource is one that is accessed in strict mutual exclusion, and hence such that obtaining a handle is a relatively expensive operation; `Camera`, `MediaPlayer`, and `AudioRecorder` are examples of non-aliasing resources. In contrast, aliasing resources tend to support a high degree of concurrent access, and hence it is common to instantiate fresh handles for each usage; a database `Cursor` is a typical example of such resources, as creating a new cursor is inexpensive, and database systems support fine-grained concurrent access. Out of all 13 Android resources involved in leaks in DROIDLEAKS, 9 are non-aliasing; our experiments ran PLUMBDROID on all apps in DROIDLEAKS that use these resources.[6]

Table 1 summarizes the characteristics of the 9 resources we selected for our experiments according to the above criteria. Then, Table 2 lists all apps in DROIDLEAKS that use some of these resources, their size, and how many leaks of each resource DROIDLEAKS includes with fixes. Thus, the first part of our experiments will target 16 Android apps with a total size of about half a million lines of code; DROIDLEAKS collects 26 leaks in these apps affecting the 9 non-aliasing resources we consider.

4.1.2. Subjects: RQ4, RQ5

At the time of writing, RelFix [28] is the only fully automated tool for the detection and repair of Android resource leaks, which can be quantitatively compared to PLUMBDROID. Since RelFix uses Relda2 [55] to *detect* leaks, and Relda2's experimental evaluation is broader than RelFix's, we also compare PLUMBDROID's leak detection capabilities to Relda2's.

As we discuss in Section 5.3, other tools exist that detect other kinds of leaks; since they are not directly applicable to the same kinds of resources that PLUMBDROID analyzes, we only compare them to PLUMBDROID in a qualitative way in Section 5.

The experimental evaluations of RelFix and Relda2, as reported in their publications [28, 55], targeted 27 Android apps that are not part of DROIDLEAKS. Unfortunately, neither detailed experimental results (such as the app versions that were targeted or the actual produced fixes) nor the RelFix and Relda2 tools are publicly available.[7]

[5] Note that the classifications of resources into aliasing/non-aliasing and reentrant/non-reentrant are orthogonal: PLUMBDROID fully supports reentrant resources, but achieves a high precision only when analyzing non-aliasing resources.

[6] We also tried PLUMBDROID on the 4 ($= 13 - 9$) aliasing resources in DROIDLEAKS; Section 4.3 discusses the outcome of these secondary experiments..

[7] The authors of [55, 28] could not follow up on our requests to share the tools or details of their experi-

Therefore, a detailed, direct experimental comparison is not possible. However, we could still run PLUMBDROID on the same apps analyzed with RelFix and Relda2, and compare our results to those reported by [28, 55] in terms of number of fixes and precision.

Out of the 27 apps used in RelFix and Relda2's experimental evaluations, 22 use some of the 9 non-aliasing resources that PLUMBDROID targets (see Table 1). More precisely, Relda2's evaluation in [55] *only* targets resources that are non-aliasing, thus we consider all of their experiments in our comparison. RelFix's evaluation in [28] targets 4 non-aliasing and 4 aliasing resources; we only consider the former for our comparison with PLUMBDROID. In order to include apps as close as possible to those actually analyzed by [28, 55], we downloaded the `apk` release of each app that was closest in time to the publication time of [28, 55]. This excluded 2 apps whose older releases we could not retrieve. In all, this process identified 20 apps: 16 used in the evaluation of Relda2 (listed in Table 4) and 4 used in the evaluation of RelFix (listed in Table 5), for a total of 260 000 lines of code and using 7 of the 9 non-aliasing resources.

Relda2 supports both a flow-insensitive and a flow-sensitive detection algorithm. According to [55], the flow-insensitive approach is faster but much less precise. Since PLUMBDROID's analysis is also flow-sensitive, we only compare it to Relda2's flow-sensitive analysis (option op_2 in [55]), which is also the one used by RelFix.

4.1.3. Experimental Protocol

In our experiments, each run of PLUMBDROID targets one app and repairs leaks of a specific resource.[8] The run's output is a number of leaks and, for each of them, a fix.

After each run, we manually inspected the fixes produced by PLUMBDROID, confirmed that they are syntactically limited to a small number of release operations, and checked that the app with the fixes still runs normally. Unfortunately, the apps do not include tests that we could have used as additional evidence that the fixes did not introduce any regression. However, PLUMBDROID's soundness guarantees that the fixes are correct by construction; and its validation phase further ascertains that the fixes do not introduce use-after-release errors.

In all experiments with DROIDLEAKS, we also tried to match, as much as possible, the leaks detected and repaired by PLUMBDROID to those reported in DROIDLEAKS. This was not always possible: some apps' are only available in obfuscated form, which limits what one can conclusively determine by inspecting the bytecode. In addition, DROIDLEAKS's collection is not meant to be exhaustive: therefore, it is to be expected that PLUMBDROID finds leaks that are not included in DROIDLEAKS. In the experiments with the apps analyzed by RelFix and Relda2, we did not have any "ground truth" to compare them to, but we still performed manual checks and testing.

More precisely, we followed these steps to manually inspect and validate all leak detected by PLUMBDROID: 1. We consider the leaking path on the resource-flow graph reported by PLUMBDROID and determine whether it is feasible. If this is not the case

mental evaluation.

[8] PLUMBDROID can analyze leaks for multiple resources in the same run, but we do not use this features in the experiments in order to have a fine-grained breakdown of PLUMBDROID's performance.

(for example, two elements of the path condition are contradictory), then we classify the leak as a false positive. 2. If the leaking path is feasible, we consider the path's matching sequence of callbacks, and use that as a guide to write a test that tries to cover the path on the real program—and thus confirms that the leak is a true positive. For short paths, the path condition usually contains enough information to come up with a test after some trial-and-error. 3. In more complex cases (especially paths involving UI interactions), we use Android Studio's *Monkey Testing* framework [5] to generate several random input sequences that thoroughly exercise the app; then, we monitor the app running on those inputs, and select any execution paths that matched the leaking path on the resource-flow graph. 4. Finally, we re-compile the app after injecting the fix produced by PLUMBDROID, and run the patched app on the tests generated as described above, as well as on a few other random inputs and also trying the app interactively, checking that the leak is no longer triggered and there are no other changes in behavior (in particular, no negative impact on performance). In all cases, we were conservative in assessing which leaks and fixes are correct, marking as "confirmed" only cases where the collected evidence that a leak may occur and its fix safely repairs it is conclusive.

The main parameter regulating PLUMBDROID's behavior is the unrolling depth D. We ran experiments with D ranging from 1 to 6, to demonstrate empirically that the default value $D = 3$ is necessary and sufficient to achieve soundness (i.e., no leaks are missed).

Hardware/software setup. All the experiments ran on a MacBook Pro equipped with a 6-core Intel Core i9 processor and 16 GB of RAM, running macOS 10.15.3, Android 8.0.2 with API level 26, Python 3.6, ANDROGUARD 3.3.5, APKTOOL 2.4.0.

4.2. Experimental Results

This section summarizes the results of our experiments with PLUMBDROID, and discusses how the results answer the research questions.

4.2.1. RQ1: Correctness

Column FIXED in Table 3 reports the number of DROIDLEAKS leaks that PLUMBDROID detected and fixed with a correct fix (i.e., a fix that prevents leaking); column INVALID reports how many of these fixes failed validation (were "unsafe"). PLUMBDROID was very effective at detecting leaks in non-aliasing resources. In particular, it detected and fixed *all* 26 leaks reported by DROIDLEAKS and included in our experiments (see Table 2), building a correct fix for each of them. In addition, it detected and fixed another 24 leaks in the same apps.

Precision. Empirically evaluating *precision* is tricky because we lack a complete baseline. By design, DROIDLEAKS is not an *exhaustive* collection of leaks. Therefore, when PLUMBDROID reports and fixes a leak it could be: (a) a real leak included in DROIDLEAKS; (b) a real leak *not* included in DROIDLEAKS; (c) a spurious leak. By inspecting the leak reports and the apps (as discussed in Section 4.1.3) we managed to confirm that 44 leaks (88%) reported by PLUMBDROID are in categories (a) (26 leaks or 52%, matching all leaks included in DROIDLEAKS) or (b) (18 leaks or 36%) above—and thus are real leaks (true positives). Unfortunately, the remaining 6 leaks (12%) reported by PLUMBDROID were found in apps whose bytecode is only available

	RFG		CC		TIME (s)							
APP	$\|V\|$	$\|E\|$	M/M'	RESOURCE	ABSTRACTION	ANALYSIS	FIXING	VALIDATION	TOTAL	FIXED	?	INVALID
APG	4 968	7 442	0.47	`MediaPlayer`	32.7	273.4	0.2	55.5	361.8	1	0	0
BarcodeScanner	1 189	2 462	0.35	`Camera`	6.8	69.2	0.3	21.8	98.1	3	0	0
CallMeter	1 840	3 216	0.35	`WakeLock`	11.2	100.7	0.6	35.1	147.6	4	0	0
ChatSecure	5 430	8 686	0.48	`BluetoothAdapter`	23.0	316.6	0.1	114.6	454.3	2	0	0
				`Vibrator`		273.7	0.5	93.3	390.2	2	0	0
ConnectBot	1 956	3 814	0.23	`WakeLock`	12.1	107.0	0.2	32.7	152.0	2	0	0
CSipSimple	5 712	9 154	0.42	`WakeLock`	38.0	433.8	0.1	111.4	583.3	4	2	0
IRCCloud	4 782	9 755	0.43	`MediaPlayer`	25.1	239.8	0.5	80.5	345.9	3	0	0
				`WifiLock`		295.3	0.3	58.0	380.8	2	0	0
K-9 Mail	8 831	16 390	0.29	`WakeLock`	64.1	475.7	0.4	165.8	706.0	2	0	0
OpenGPSTracker	1 418	2 791	0.29	`LocationListener`	7.1	107.4	0.4	33.7	148.6	2	0	0
OsmDroid	2 222	3 545	0.36	`LocationListener`	12.9	161.9	0.4	32.9	208.1	4	2	0
ownCloud	4 444	8 980	0.6	`WifiLock`	20.7	238.6	0.5	47.8	307.6	4	2	0
QuranForAndroid	2 898	4 545	0.43	`MediaPlayer`	14.9	177.5	0.2	63.8	256.4	2	0	0
SipDroid	3 178	4 583	0.38	`Camera`	14.1	176.6	0.5	39.7	230.9	4	0	0
SureSpot	3 575	7 240	0.37	`MediaPLayer`	33.0	246.4	0.4	54.1	333.9	3	0	3
Ushahidi	5 073	10 417	0.43	`LocationListener`	24.4	201.9	0.3	58.5	285.1	2	0	2
VLC	2 689	4 199	0.55	`WakeLock`	14.4	119.5	0.5	34.4	168.8	2	0	0
Xabber	4 194	8 478	0.31	`AudioRecorder`	25.4	256.9	0.3	76.6	359.2	2	0	0
AVERAGE	3 788	6 805	0.4		23.4	231.2	0.3	65.2	320.2			
TOTAL	64 399	115 697	6.74		444.6	4392.8	5.9	1238.8	6084.8	50	6	5

Table 3: Results of running PLUMBDROID on apps in DROIDLEAKS. For every APP, the table reports the number of nodes $|V|$ and edges $|E|$ of its resource-flow graph RFG, and the ratio M/M' between the RFG's cyclomatic complexity M and the cyclomatic complexity M' of the whole app's control-flow graph. For every RESOURCE used by the app, the table then reports PLUMBDROID's running time to perform each of the steps of Figure 2 (ABSTRACTION, ANALYSIS, FIXING, and VALIDATION); as well as the TOTAL running time; since the abstraction is built once per app, time ABSTRACTION is the same for all resources used by an app. Finally, the table reports the number of leaks of each resource detected and FIXED by PLUMBDROID; how many of these fixed leaks we could not conclusively classify as real leaks (**?**); and the number of the fixes that PLUMBDROID classified as INVALID (that is, they failed validation). The two bottom rows report the AVERAGE (mean, per app or per app-resource) and TOTAL in all experiments.

in *obfuscated* form, which means we cannot be certain they are not spurious; these unconfirmed cases are counted in column "**?**" in Table 3. Even in the worst case in which all of these are spurious, PLUMBDROID's precision would remain high (88%). The actual precision is likely higher: in all cases where we could analyze the code,

we found a real leak; unconfirmed cases probably just require more evidence such as access to unobfuscated bytecode. Anyway, note that any spurious fixes would still be safe to apply—albeit unnecessary—because they do not introduce bugs: since all release operations added by PlumbDroid are conditional (Section 3.4), a fix "repairing" a spuriously detected leak simply introduces release operations that is neve executed in actual program executions.

Correctness and safety. All fixes built by PLUMBDROID are correct in the sense that they release resources so as to avoid a leak; manual inspection, carried out as described in Section 4.1.3, confirmed this in all cases—with some remaining uncertainty only for obfuscated apps.

PLUMBDROID's validation step assesses "safety": whether a fix does not introduce a use-after-release error. All but 5 fixes built by PLUMBDROID for non-aliasing resources are safe. The 5 unsafe fixes are:

(i) Three identical fixes (releasing the same resource in the same location) repairing three distinct leaks of resource `MediaPlayer` in app SureSpot.

According to the Android reference manual [6], this resource can be released either in the `onPause` or in the `onStop` callback. PLUMBDROID releases resources as early as possible by default, and hence it built a fix releasing the `MediaPlayer` in `onPause`. The developers of SureSpot, however, assumed that the resource is only released later (in `onStop`), and hence PLUMBDROID's fix introduced a use-after-release error that failed validation.

To deal with such situations—resources that may be released in different callbacks—we then could introduce a configuration option to decide whether to release resources *early* or *late*. An app developer could therefore configure our analyzer in a way that suits their design decisions. In particular, configuring PLUMBDROID with option *late* in these cases would generate fixes that pass validation.

(ii) Two identical fixes (releasing the same resource in the same location) repairing two distinct leaks of resource `LocationListener` in app Ushahidi.

The fix generated by PLUMBDROID failed validation because the app's developers assumed that the resource is only released in callback `onDestroy`. This assumption conflicts with Android's recommendations to release the resources in earlier callbacks.

In this case, the best course of action would be amending the app's usage policy of the resource so as to comply with Android's guidelines. PLUMBDROID's fix would pass validation after this modification.

Note that PLUMBDROID does not output fixes that do not pass the validation step; therefore, there is no risk that such unsafe fixes are accidentally deployed.

PLUMBDROID *detected and fixed 50 leaks in* DROIDLEAKS *producing correct-by-construction fixes.* PLUMBDROID*'s detection is very precise on the resources it supports.*

```
1  public class NetworkConnection {
2
3      public void disconnect() {
4          if (client != null) {
5              state = STATE_DISCONNECTING;
6              client.disconnect();
7          } else {
8              state = STATE_DISCONNECTED;
9          }
10         if (idleTimer != null) {
11             idleTimer.cancel();
12             idleTimer = null;
13         }
14         if (wifiLock.isHeld()) wifiLock.release();
15     }
16
17 }
18
```

Figure 8: An excerpt of class `NetworkConnection` in Android app IRCCloud, showing the `disconnect()` method with a resource leak (code in black, executed in callback `onPause()`), and how the leak was patched by the app developers in commit 113555e9ae (code in **orange**). PLUMBDROID generates a patch for this leak that exactly matches the developer-written one shown here.

4.2.2. RQ2: Comparison with Developer Fixes

The manual inspection and validation protocol (Section 4.1.3) that we followed to confirm the correctness of all leaks and fixes reported by PLUMBDROID already strongly suggests that the results of PLUMBDROID's analysis are usually of high quality. To better understand whether PLUMBDROID's fixes are comparable to those written by developers, we further scrutinized the fixes produced by PLUMBDROID for leaks in apps IRCCloud and Ushahidi. We selected these two apps as they are among the largest in the DROIDLEAKS collection, they are open source, and their public code repositories are well organized and feature a long-running development history. Furthermore, as shown in Table 3, PLUMBDROID detected and fixed 5 leaks in IRCCloud; and detected 2 leaks in Ushahidi, for which it could only produce fixes that fail the validation step. Thus, there is a mix of cases where PLUMBDROID is completely successful and cases where PLUMBDROID's validation fail—which mitigates the risk of biasing the analysis in PLUMBDROID's favor.

Based on these results, we went through the bug reports and commit history of the two apps, looking for developer-written fixes of the 7 leaks that PLUMBDROID detected and fixed. Indeed, we found that developers eventually found and fixed all these leaks,[9]

[9]See IRCCloud's commits: 113555e9ae, 35e0a587e3, 0cd91bc5ca, d7a441e3a6, 113555e9ae; and Ushahidi's commits: 9d0aa75b84, 337b48f5f2.

```
1  public class NetworkConnection  {
2
3    public void removeHandler(Handler handler) {
4      handlers.remove(handler);
5      if (handlers.isEmpty()) {
6        if (shutdownTimer == null) {
7          // ...
8          disconnect();
9        }
10       if (idleTimer != null
11           && state != STATE_CONNECTED) {
12         idleTimer.cancel();
13         idleTimer = null;
14         failCount = 0;
15         if (wifiLock.isHeld()) wifiLock.release();
16         reconnect_timestamp = 0;
17         state = STATE_DISCONNECTED;
18       }
19     }
20   }
21
22 }
23
```

(a) The app developers fixed the leak in commit 35e0a587e3 by adding a conditional release at line 15 (code in **orange**).

```
1  public class NetworkConnection  {
2
3    public void removeHandler(Handler handler) {
4      handlers.remove(handler);
5      if (handlers.isEmpty()) {
6        if (shutdownTimer == null) {
7          // ...
8          disconnect();
9        }
10       if (idleTimer != null
11           && state != STATE_CONNECTED) {
12         idleTimer.cancel();
13         idleTimer = null;
14         failCount = 0;
15
16         reconnect_timestamp = 0;
17         state = STATE_DISCONNECTED;
18         if (wifiLock.isHeld()) wifiLock.release();
19       }
20     }
21   }
22
23 }
24
```

(b) PLUMBDROID fixed the leak by adding a conditional release at line 18 (code in **green**). PLUMBDROID only detects this leak with unrolling depth $D \geq 2$, as it affects re-entrant resource `WifiLock`.

Figure 9: An excerpt of class `NetworkConnection` in Android app IRCCloud, showing the `removeHandler()` method with a resource leak (code in black, executed in callback `onPause()`), and how it was fixed by the app developers (Figure 9a) and by PLUMBDROID (Figure 9b).

which confirms that they are considered serious enough faults.

In 2 out of the 5 leaks in IRCCloud, the fix produced by PLUMBDROID is identical to the developer-written one, as they both release the same resource in the same location. Figure 8 shows one of these leaks and its fix.[10] In the other 3 leaks in IRCCloud, the fix produced by PLUMBDROID released the same resource as the developer-written one but in a different location of the same basic block. Figure 9 shows one of these leaks and its fix by developer (Figure 9a) and by PLUMBDROID (Figure 9b). In all these cases, the difference of release location is immaterial, as the statements that are between the release point in the developer-written fix and the release point in PLUMBDROID's fix are simple assignments do not affect any shared resources and execute quickly.

In both leaks in Ushahidi, the difference between the (invalid) fixes produced by PLUMBDROID and the developer-written ones is also the location of the release. Sticking to the Android developer guidelines [4], PLUMBDROID only releases the resource

[10] In these examples, we express the fix as Java source code, even though PLUMBDROID works at the level of the human-readable bytecode Smali.

`LocationListener` in the callback `onPause()` for one of these leaks; doing so fails the validation phase, since the Ushaidi app still uses location resources when it is running in the background. Therefore, the developers fixed the leak by releasing the resource in the callback `onDestroy()`, that is only when the app is shut down. The other leak has a similar discrepancy between the Android guidelines followed by PLUMBDROID and how the Ushahidi app is written.

In all, PLUMBDROID's fixes are often very similar and functionally equivalent to those written by programmers for the same leaks. PLUMBDROID's validation phase is useful to detect when an app deviates from Android's guidelines on resource management; in these cases, the user of PLUMBDROID can decide whether to refactor the app to follow the guidelines, or modify PLUMBDROID so that it generates a fix at a different location.

The similarity between PLUMBDROID and developer-written fixes, as well their usual syntactic simplicity, is also evidence that these fixes are unlikely to negatively alter the running-time performance of an app (and hence the user experience). This is also consistent with our manual analysis (Section 4.1.3), which never found an app's responsiveness to worsen after applying a resource-leak fix.

> *According to the analysis of a sample, we found that the fixes produced by* PLUMBDROID *often are functionally equivalent to those written by the app developers.*

4.2.3. RQ3: Performance

Columns TIME in Table 3 report the running time of PLUMBDROID in each step. As we can expect from a tool based on static analysis, PLUMBDROID is generally fast and scalable on apps of significant size. Its average running time is around 5 minutes *per app-resource* and around 2 minutes *per repaired leak* ($121\,\mathrm{s} \simeq 6084.8/50$).

The ANALYSIS step dominates the running time, since it performs an exhaustive search. In contrast, the ABSTRACTION step is fairly fast (as it amounts to simplifying control-flow graphs); and the FIXING step takes negligible time (as it directly builds on the results of the analysis step).

PLUMBDROID's abstractions are key to its performance, as we can see from Table 3's data about the size of the resource-flow graphs. Even for apps of significant size, the resource-flow graph remains manageable; more important, its *cyclomatic complexity* M—a measure of the number of paths in a graph [39]—is usually much lower than the cyclomatic complexity M' of the full control-flow graph, which makes the exhaustive analysis of a resource-flow graph scalable.

> PLUMBDROID *is scalable: it takes about 2 minutes on average to detect and fix a resource leak.*

4.2.4. RQ4: Comparison with Other Tools

Comparison with Relda2. Table 4 compares the leak detection capabilities of PLUMBDROID and Relda2 on the 17 apps used in the latter's experiments [55]—which only target non-aliasing resources. Relda2 reports more leaks than PLUMBDROID, but PLUMBDROID's precision ($90\% = 70/78$) is much higher than Relda2's ($54\% = 46/81$),

			PLUMBDROID			Relda2	
APP	KLOC	RESOURCE	TIME (s)	LEAKS	✔	LEAKS	✔
Andless	0.5	`WakeLock`	13.1	1	1	1	1
Apollo	24.5	`Camera`	176.4	2	2	4	2
CheckCheck	4.5	`WakeLock`	35.4	3	3	6	4
Impeller	4.6	`WiFiLock`	25.9	4	3	1	1
Jane	42.7	`LocationListener`	280.5	2	2	0	0
MiguMusic	13.0	`MediaPlayer`	121.6	8	8	12	5
PicsArt	17.6	`WakeLock`	92.7	2	2	2	2
QRScan	24.5	`Camera`	144.1	4	4	6	3
Runnerup	37.2	`LocationListener`	232.9	13	11	8	5
Shopsavvy	5.3	`LocationListener`	41.3	8	7	5	3
SimSimi	3.9	`WakeLock`	35.6	7	6	11	4
SuperTorch	3.8	`Camera`	24.5	3	3	1	1
TigerMap	3.7	`LocationListener`	36.7	2	2	6	2
Utorch	1.0	`Camera`	20.5	2	2	1	1
Vplayer	6.6	`MediaPlayer`	49.8	7	5	7	5
WeatherPro	5.0	`LocationListener`	55.8	8	7	12	5
Yelp	14.3	`LocationListener`	156.4	2	2	2	2
AVERAGE	12.5		90.8				
TOTAL	212.7		1543.2	78	70	85	46

Table 4: Comparison of PLUMBDROID's and Relda2's leak detection capabilities. For every APP used in the comparison, the table reports its size KLOC in thousands of lines of code, and the resource analyzed for leaks. Then, it reports PLUMBDROID's running TIME in seconds, the number of LEAKS PLUMBDROID detected, and how many of these we definitely confirmed as true positives by manual analysis (✔). These results are compared to the number of LEAKS detected by Relda2, and how many of these were true positives (✔) according to the experiments reported in [55]. The two bottom rows report the AVERAGE (mean) and TOTAL for all apps/resources.

			PLUMBDROID					RelFix		
APP	KLOC	RESOURCE	TIME (s)	LEAKS	✔	FIXED	INVALID	LEAKS	✔	FIXED
BlueChat	13.1	`MediaPlayer`	93.9	3	2	3	0	1	0	1
		`WakeLock`	111.7	2	2	2	0	2	1	2
FooCam	14.7	`Camera`	152.7	1	1	1	0	3	1	3
		`MediaPlayer`	124.5	0	–	0	0	1	1	1
GetBackGPS	21.4	`LocationListener`	153.4	2	2	2	1	3	3	3
SuperTorch	3.8	`Camera`	24.5	3	2	3	1	1	1	1
AVERAGE	50.2		110.1							
TOTAL	200.6		660.7	11	9	11	2	11	7	11

Table 5: Comparison of PLUMBDROID's and RelFix's leak repair capabilities (on non-aliasing resources). For every APP used in the comparison, the table reports its size KLOC in thousands of lines of code, and the resources whose leaks are repaired. Then, it reports PLUMBDROID's running TIME in seconds, the number of leaks DETECTED and FIXED by PLUMBDROID, how many of these leaks we could conclusively classify as real leaks (✔, true positives), and the number of fixes that PLUMBDROID classified as INVALID (that is, they failed validation). These results are compared to the number of leaks DETECTED and FIXED by Relfix, and how many of these are considered real leaks (✔) according to the experiments reported in [28]. The two bottom rows report the AVERAGE (mean) and TOTAL for all apps/resources.

and hence PLUMBDROID reports several more *true* leaks (70 vs. 46). The difference between the two tools varies considerably with the app. On 4 apps (Andless, PicsArt, Vplayer, and Yelp) both tools detect the same number of leaks with the same (high) precision; even though we cannot verify this conjecture, it is quite possible that exactly the same leaks are detected by both tools in these cases. On 2 apps (Impeller and CheckCheck), neither tool is strictly better: when PLUMBDROID outperforms Relda2 in number of confirmed detected leaks (app Impeller) it also achieves a lower precision; when PLUMBDROID outperforms Relda2 in precision (app CheckCheck) it also finds one less correct leak. On the remaining 11 apps (Apollo, Jane, MiguMusic, QRScab, Runnerup, Shopsavvy, SimSimi, SuperTorch, TigerMap, Utorch, and WeatherPro) PLUMBDROID is at least as good as Relda2 in both number of confirmed detected leaks and precision, and strictly better in detected leaks, precision, or both. We cannot directly compare PLUMBDROID's and Relda2's running times, since we could not run them on the same hardware, but we notice that the running times reported in [55] are in the ballpark of PLUMBDROID's. In all, PLUMBDROID's leak detection capabilities often outperforms Relda2's on the non-aliasing resources that we currently focus on.

Comparison with RelFix. Table 5 compares the fixing capabilities of PLUMBDROID and RelFix (which uses Relda2 for leak detection) on the 4 non-aliasing resources used in the latter's experiments [28]. Here too PLUMBDROID generally appears more effective than RelFix: conservatively assuming that all fixes reported by [28] are

	DROIDLEAKS			Relda2/RelFix	
D	TIME (s)	FIXED	MISSED	FIXED	MISSED
1	116.7	40	10	75	14
2	184.9	50	0	84	5
3	320.2	50	0	89	0
4	501.3	50	0	89	0
5	907.8	50	0	89	0
6	1894.3	50	0	89	0

Table 6: Results of running PLUMBDROID with different unrolling depths on the DROIDLEAKS benchmark and on the apps used in the comparison with Relda2 and RelFix. For each unrolling depth D, the table reports the AVERAGE (mean, per app-resource) running TIME of PLUMBDROID on all leaks of non-aliasing resources; and the total number of FIXED LEAKS and MISSED LEAKS (not detected, and hence not fixed). The row with $D = 3$ corresponds to the DROIDLEAKS data in Table 3, and the Relda2/RelFix data in Tables 4–5.

genuine and "safe",[11] PLUMBDROID has higher precision (82%$= 9/11$ vs. 64%$= 7/11$) and fixes at least as many confirmed leaks (even after discarding those that fail validation).

PLUMBDROID generates small patches, each consisting of just 11 bytecode instructions on average, which corresponds to an average 0.017% increase in size of a patched app. This approach leads to much smaller patches than RelFix's, whose patches also include instrumentation to detect the leaks on which the fixes depend to function correctly. As a result, the average increase in size introduced by a RelFix patch is 0.3% in terms of bytecode instructions [28], which is one order of magnitude larger than PLUMBDROID's.

As explained in Section 4.1.2, all results in this section are subject to the limitation that a direct comparison with Relda2 and RelFix was not possible. Despite this limitation, the comparison collected enough evidence to indicate that PLUMBDROID's analysis is often more thorough and more precise than the other tools'.

On non-aliasing resources, PLUMBDROID *is usually more effective and precise than other techniques for the automated detection and repair of Android resource leaks.*

4.2.5. RQ5: Unrolling

In the experiments reported so far, PLUMBDROID ran with the unrolling depth parameter $D = 3$, which is the default. Table 6 summarizes the key data about experiments on the same apps but using different values of D. These results indicate that a value of $D \geq 3$ is required for soundness: PLUMBDROID running with $D = 1$ missed 24 leaks (10 in DROIDLEAKS, and 14 in the apps used by Relda2/RelFix); with $D = 2$ it missed 5 leaks (all in the apps used by Relda2/RelFix). All missed leaks

[11] RelFix's paper [28] does not explicitly discuss possible fix validation errors.

only occur with resources that are acquired multiple times—that is they affect *reentrant* resources: `WakeLock` and `WifiLock` in apps CallMeter, CSipSimple, IRCCloud (from DROIDLEAKS), CheckCheck, Impeller, PicsArt, SimSimi (from the comparison with Relda2), and BlueChat (from the comparison with RelFix).

Is $D = 3$ also sufficient for soundness? While we cannot formally prove it, the experiments suggest this is the case: increasing D to larger values does not find new leaks but only increases the running time. Unsurprisingly, the running time grows conspicuously with the value of D, since a larger unrolling depth determines a combinatorial increase in the number of possible paths. Thus, for the analyzed apps, the default $D = 3$ is the empirically optimal value: it achieves soundness without unnecessarily increasing the running time. It is possible this result does not generalize to apps with more complex reentrant resource management; however, PLUMBDROID always offers the possibility of increasing D until its analysis is sufficiently exhaustive.

> PLUMBDROID*'s analysis is sound provided it unrolls callback loops a sufficient number of times (thrice in the experiments).*

4.3. Aliasing Resources

We repeatedly remarked that PLUMBDROID's current implementation is effective only on non-aliasing resources; it remains applicable to aliasing resources, but it is bound to generate a large number of false positives. In order to get a clearer picture of PLUMBDROID's limitations in the presence of aliasing, this section reports some additional experiments on aliasing resources. It remains that our contributions focus on non-aliasing resources; an adequate support of aliasing belongs to future work.

TOOL	SUBJECTS	RESOURCES	LEAKS	✔	PRECISION
Relda2	[28]	4	108	24	22%
PLUMBDROID	DROIDLEAKS	6	273	86	32%

Table 7: Detection of leaks of *aliasing* resources by Relda2 (according to the experiments reported in [28]) and PLUMBDROID (on the programs in DROIDLEAKS). The table reports the number of aliasing RESOURCES involved in leaks, the reported LEAKS, how many of them are confirmed true leaks ✔, and the corresponding precision ✔/LEAKS.

Table 7 shows the behavior of PLUMBDROID on leaks of the 6 aliasing resources in DROIDLEAKS apps. PLUMBDROID reported a total of 273 leaks, but we could only confirm about one third of them as real leaks (true positives). This is a lower bound on PLUMBDROID's precision on aliasing resources, as it's possible that more reported leaks are real but we could not conclusively confirm them as such because they affect obfuscated apps. Nevertheless, it's clear the precision is much lower than on non-aliasing resources—as we expected.

For comparison, Table 7 also reports the performance of Relda2 according to the experiments reported in [28] on 4 aliasing resources (Section 4.2.4 discusses the data from the same source but involving non-aliasing resources). Relda2 reported a total of 108 leaks, but only 22% of them are real leaks according to [28]. Even though the

experiments with PLUMBDROID and with Relda2 are not directly comparable, it's clear both tools achieve a low precision on aliasing resources—arguably low to the point that practical usefulness is severely reduced.

4.4. Threats to Validity

The main threats to the validity of our empirical evaluation come from the fact that we analyzed Android apps in *bytecode* format; furthermore, some of these apps' bytecode was only available in *obfuscated* form. In these cases, we were not able to inspect in detail how the fixes modified the original programs; we could not always match with absolute certainty the leaks and fixes listed in DROIDLEAKS with the fixes produced by PLUMBDROID; and we could not run systematic testing of the automatically fixed apps. This threat was significantly mitigated by other sources of evidence that PLUMBDROID indeed produces fixes that are correct and do not alter program behavior except for removing the source of leaks: first, the manual inspections we could carry out on the apps that are not obfuscated confirmed in all cases our expectations; second, PLUMBDROID's analysis is generally *sound*, and hence it should detect all leaks (modulo bugs in our implementation of PLUMBDROID); third, running the fixed apps for significant periods of time did not show any apparent change in their behavior.

As remarked in Section 4.1.2, we could only perform an indirect comparison with Relda2/RelFix (the only other fully automated approach for the repair of Android resource leaks that is currently available) since neither the tools nor details of their experiments other than those summarized in their publications [55, 28] are available. To mitigate the ensuing threats, we analyzed app versions that were available around the time when the Relda2/RelFix experiments were conducted, and we excluded from the comparison with PLUMBDROID measures that require experimental repetition (such as running time). While it is still possible that measures such as precision were assessed differently than how we did, these should be minor differences that do not invalidate the high-level results of the comparison.

Our evaluation did not assess the acceptability of fixes from a programmer's perspective. Since PLUMBDROID works on bytecode, its fixes may not be easily accessible by developers familiar only with the source code. Nonetheless, fixes produced by PLUMBDROID are succinct and correct by construction, which is usually conducive to readability and acceptability. As future work, one could implement PLUMBDROID's approach at the level of source code, so as to provide immediate feedback to programmers as they develop an Android app. PLUMBDROID in its current form could instead be easily integrated in an automated checking system for Android apps—for example, within app stores.

We didn't formally prove the *soundness* or *precision* of PLUMBDROID's analysis, nor that our implementation is free from bugs. Since PLUMBDROID is implemented on top of ANDROGUARD [8] and APKTOOL [10] (Section 3.7), any bugs or limitations of these tools may affect PLUMBDROID's analysis. In particular, ANDROGUARD cannot currently analyze native code[12]—a common limitation of static analysis. By and large, however, these tools' support of Android is quite extensive, and hence any current

[12] `https://github.com/androguard/androguard/issues/566#issuecomment-431090708`

limitations are unlikely to significantly impact the soundness of the results obtained in PLUMBDROID's experimental evaluation.

Nonetheless, we analyzed in detail the features of PLUMBDROID's analysis both theoretically (Section 3.6) and empirically (Section 4). The empirical evaluation corroborates the evidence that PLUMBDROID is indeed sound (for sufficiently large unrolling depth D), and that *aliasing* is the primary source of imprecision. In future work, we plan to equip PLUMBDROID with alias analysis [23], in order to boost its precision on the aliasing resources that currently lead to many false positives.

DROIDLEAKS offers a diverse collection of widely-used apps and leaked resources, which we further extended with apps used in Relda2/RelFix's evaluations. In our experiments, we used all apps and resources in DROIDLEAKS and in Relda2/RelFix's evaluations that PLUMBDROID can analyze with precision. This helps to generalize the results of our evaluation, and led to finding numerous leaks not included in DROIDLEAKS nor found by Relda2/RelFix. Further experiments in this area would greatly benefit from extending curated collections of leaks and repairs like DROIDLEAKS. Our replication package is a contribution in this direction.

5. Related Work

TOOL	ANALYSIS	LEAKS	AUTOMATED	REENTRANT	SOUNDNESS	REPAIR
FindBugs [17]	static	Java resources	full	No	No	No
EnergyTest [13]	dynamic	energy	partial	No	No	No
LeakCanary [24]	dynamic	memory	full	No	No	No
Android Studio [7]	dynamic	memory	full	No	No	No
FunesDroid [2]	dynamic	memory	partial	No	No	No
Sentinel [54, 52]	static+dynamic	sensor	partial	Yes	No	No
Relda2/RelFix [55, 28]	static	Android resources	full	No	No	(Yes)
EnergyPatch [12, 14]	static+dynamic	energy	full	No	No	(Yes)
PLUMBDROID	static	Android resources	full	Yes	Yes	Yes

Table 8: Comparison of tools that detect and repair leaks in Android apps. For each tool, the table report the kinds of code ANALYSIS it performs (static or dynamic), the kinds of LEAKS it primarily targets, whether it is fully or partially AUTOMATED, whether it models REENTRANT behavior, whether its detection is SOUND, and whether it can also generate REPAIRs of the detected leaks. Entries (Yes) in column REPAIR denote fixes that release any leaking resource at the very end of an app's lifecycle, without following the resource-specific Android guidelines.

5.1. Automated Program Repair

PLUMBDROID is a form of automated program repair (APR) targeting a specific kind of bugs (resource leaks) and programs (Android apps). The bulk of "classic" APR research [19, 41, 51, 37] usually targets general-purpose techniques, which are applicable in principle to any kinds of program and behavioral bugs. The majority of these techniques are based on dynamic analysis—that is, they rely on *tests* to define

expected behavior, to detect and localize errors [21, 15], and to validate the generated fixes [35, 22, 45]. General-purpose APR completely based on static analysis is less common [33, 34, 18], primarily because tests are more widely available in general-purpose applications, whereas achieving a high precision with static analysis is challenging for the same kind of applications and properties.

5.2. Leak Analysis

Whereas PLUMBDROID is one of only two fully automated approaches for *fixing Android resource* leaks (the other is RelFix, discussed below and in Section 4.2.4), *detection* of leaks and other defects is more widely studied and has used a broad range of techniques—from static analysis to testing. Table 8 outlines the features of the main related approaches, which we discuss in the rest of this section.

5.2.1. Static Analysis

Approaches based on static analysis build an *abstraction* of a program's behavior, which can be searched exhaustively for leaks. Since Android apps run on mobile devices, they are prone to defects such as privacy leaks [20], permission misuses [27], and other security vulnerabilities [57, 43] that are less prominent (or have less impact) in traditional "desktop" applications. In such specialized domains, where *soundness* of analysis is paramount, static analysis is widely applied—for example to perform taint analysis [36] and other kinds of control-flow based analyses [26, 11]. Indeed, there has been plenty of work that applied static analysis to analyze resource management in Java [50, 16], including detecting resource leaks [47, 23].

Whereas some of these contributions could be useful also to analyze mobile apps written in Java, in order to do so the techniques should first be extended to support the peculiarities of the Android programming model—in particular, its event-driven control flow.

General-purposes static analyzers for Java like FindBugs [17], are also capable of detecting a variety of issues in common Java resources (e.g., files); however, FindBugs is neither sound nor very precise [49], as it is based on heuristics and pattern-matching that are primarily geared towards detecting stylistic issues and code smells.

5.2.2. Dynamic Analysis

Many approaches use dynamic analysis (testing), and hence are not sound (they may miss leaks).

One example is a test-generation framework capable of building inputs exposing resource leaks that lead to energy inefficiencies [13]. Since it targets energy efficiency, the framework consists of a hybrid setup that includes hardware to physically measure energy consumption, which makes it less practical to deploy (hence, Table 8 classifies it as "partial" automation). Its measurements are then combined with more traditional software metrics to generate testing oracles for energy leak detection. The framework's generated tests are sequences of UI events that trigger energy leaks or other inefficiencies exposed by the oracles. As it is usual for test-case generation, [13]'s framework is based on heuristics and statistical assumptions about energy consumption patterns, and hence its detection capabilities are not exhaustive (i.e., not sound).

Other tools [54, 52, 24, 7, 2, 56] exist that use tests to detect resource leaks—such as sensor leaks [54, 52, 56] and memory leaks [24, 2]. A key idea underlying these approaches is to combine runtime resource profiling [7] and search-based test-case generation looking for inputs that expose leaks.

LeakCanary [24] and the Android Studio Monitor [7] are two of the most popular tools used by developer to detect memory leaks as they have high *precision* and are fully automated. Like all approaches based on testing, they are unsound in general, and their capabilities strongly depend on the quality of the tests that are provided.

FunesDroid [2] generates inputs that correspond to pre-defined user interactions—like rotating the screen—which can trigger memory leaks when executed in certain activities. Besides being unsound like all testing techniques, FunesDroid's testing capabilities are also limited by the kinds of interactions that it supports. Furthermore, FunesDroid's test generation capabilities are not fully automated for leak detection: while the tool provides user interactions as inputs, one still needs to provide tests that run the app where leaks are being detected.

Sentinel [54, 52], an approach based on generating GUI events for testing, models resource usage as context-free languages, and hence it is one of the few leak analysis techniques that is capable of fully modeling reentrant resources. The tool first performs a static analysis of app code to build a model that maps GUI events to callback methods that affect sensor behavior. Then, it traverses the model to enumerate paths, which in turn are used to generate test cases. The last step (from paths to actual test inputs) requires users to manually come up with suitable inputs that match the abstract paths; therefore, the Sentinel approach is not fully automated.

5.3. Leak Repair

The amount of work on *detecting* various kinds of leaks [12, 13, 55, 31, 44] and the recent publication of the DROIDLEAKS curated collection of leaks [29] indicate that leak detection is considered a practically important problem in Android programming. In this section, we discuss in greater detail two approaches that are also capable, like PLUMBDROID, of *fixing* the Android leaks they detect.

5.3.1. Relda2 and RelFix

Relda2 [55] combines flow-insensitive and flow-sensitive static analyses to compute resource *summaries*: abstract representations of each method's resource usage, which can be combined to perform leak detection across different procedures and callbacks. Since Relda2 approximates loops in an activity's lifecycle by abstracting away some of the flow information (even in its "flow-sensitive" analysis), and does not accurately track nested resource acquisitions, its analysis is generally unsound (leaks may go undetected) and imprecise (spurious errors may be reported). In contrast, PLUMBDROID performs a more thorough and precise inter-procedural analysis by considering all possible callback sequences. PLUMBDROID even allows users to set the *unrolling depth D* (affecting the maximum length of analyzed callback sequences), which is a means of trading off soundness (i.e., how thorough the analysis is) with running time (i.e., how long/how many computational resources the analysis takes). PLUMBDROID's modeling of resources is more detailed than Relda2's also because it supports

a validation step (Section 3.5), which can detect inconsistencies between a resource's recommended usage guidelines and how it is actually used by an app. After building a control-flow model of resource usages, Relda2 uses an off-the-shelf model checker to analyze it. In contrast, PLUMBDROID uses a custom automata-theoretic analysis algorithm to search for leaks on resource-flow graphs, which may contribute to more precise and scalable results.

RelFix [28] can patch resource leaks in Android apps that have been detected by Relda2. It applies patches at the level of Dalvik bytecode, whereas PLUMBDROID does so at the level Smali (a human-readable format of Dalvik), which makes PLUMBDROID's output more accessible to human programmers. Section 4.2.4 discussed a detailed comparison between PLUMBDROID and Relda2/RelFix in terms of precision and patch sizes. Another significant difference is how they build fixes: RelFix follows the simple approach of releasing resources in the very last callback of an activity's lifecycle, whereas PLUMBDROID builds fixes that adhere to Android's recommended guidelines.

5.3.2. EnergyPatch

EnergyPatch [12, 14] is another approach for leak detection based on static techniques where resource usage are modeled as regular expressions. EnergyPatch uses abstract interpretation to compute an over-approximation of an app's energy-relevant behavior; then, it performs symbolic execution to detect which abstract leaking behaviors are false positives and which are executable (i.e., correspond to a real resource leak); therefore, its analysis is hybrid as it combines static and dynamic techniques. For each executable leaking behavior, symbolic execution can also generate a concrete program input that triggers the energy-leak bug. EnergyPatch targets a different kind of resource leaks (energy-consumption related, which lead to a wasteful usage of a mobile device's energy resources) than PLUMBDROID. Since EnergyPatch only analyzes simple paths (i.e., without loops) in the callback graph, its analysis may be unsound (especially for reentrant resources).

While EnergyPatch focuses on leak detection, it also offers a simple technique for generating fixes, which simply releases all resources in the very last callback of an activity's lifecycle. As we discussed in Section 3.4, this approach is sometimes impractical because it may conflict with some of Android programming's best practices. In contrast, PLUMBDROID releases the resource aggressively, in the earliest callback, since our validation step later can filter out patches that result in *use-after-release* issues.

6. Conclusions and Future Work

This paper presented PLUMBDROID: a technique and tool to detect and automatically fix resource leaks in Android apps. PLUMBDROID is based on succinct static abstractions of an app's control-flow; therefore, its analysis is sound and its fixes are correct by construction. Its main limitation is that its analysis tends to generate false positives on resources that are frequently *aliased* within the same app. In practice, this means that PLUMBDROID's is currently primarily designed for the numerous Android resources that are not subject to aliasing. On these resources, we demonstrated

PLUMBDROID's effectiveness and scalability. Extending PLUMBDROID's approach with aliasing information is an interesting and natural direction for future work.

Acknowledgments

Work partially supported by SNF grant 200021-182060 (Hi-Fi).

References

[1] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 202–211. ACM, 2004.

[2] Domenico Amalfitano, Vincenzo Riccio, Porfirio Tramontana, and Anna Rita Fasolino. Do memories haunt you? An automated black box testing approach for detecting memory leaks in Android apps. *IEEE Access*, 8:12217–12231, 2020.

[3] The Android API reference. `https://developer.android.com/reference/packages`. Accessed: 2022-04-28.

[4] The activity lifecycle (from the *Android Developer Guides*). `https://developer.android.com/guide/components/activities/activity-lifecycle`. Accessed: 2022-02-10.

[5] UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/other-testing-tools/monkey`. Accessed: 2022-02-10.

[6] Android Reference Manual. `https://developer.android.com/reference`. Accessed: 2021-02-16.

[7] Android Studio Monitor. `https://developer.android.com/studio/profile/monitor`. Accessed: 2021-02-16.

[8] AndroGuard. `https://github.com/androguard/androguard`. Accessed: 2021-02-16.

[9] Android Platform Architecture. `https://developer.android.com/guide/platform/`. Accessed: 2021-02-16.

[10] Apktool. `https://ibotpeaches.github.io/Apktool/`. Accessed: 2021-02-16.

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[12] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. EnergyPatch: Repairing resource leaks to improve energy-efficiency of Android apps. *IEEE Trans. Software Eng.*, 44(5):470–490, 2018.

[13] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 588–598, 2014.

[14] Abhijeet Banerjee and Abhik Roychoudhury. Automated re-factoring of Android apps to enhance energy-efficiency. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, page 139–150, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Liushan Chen, Yu Pei, and Carlo A. Furia. Contract-based program repair without the contracts. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 637–647, Piscataway, NJ, USA, 2017. IEEE Press.

[16] Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. The CLOSER: automating resource management in Java. In Richard E. Jones and Stephen M. Blackburn, editors, *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*, pages 1–10. ACM, 2008.

[17] FindBus. `http://findbugs.sourceforge.net/`. Accessed: 2021-02-16.

[18] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for C programs. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 459–470. IEEE Computer Society, 2015.

[19] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1219–1219, New York, NY, USA, 2018. ACM.

[20] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In Stefan Katzenbeisser, Edgar R. Weippl, L. Jean Camp, Melanie Volkamer, Mike K. Reiter, and Xinwen Zhang, editors, *Trust and Trustworthy Computing - 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings*, volume 7344 of *Lecture Notes in Computer Science*, pages 291–307. Springer, 2012.

[21] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 12–23, 2018.

[22] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 298–309, 2018.

[23] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Lightweight and modular resource leak verification. In *ESEC/FSE 2021: The ACM 29th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Athens, Greece, August 2021.

[24] Leak Canary Tool. `https://square.github.io/leakcanary/`. Accessed: 2021-02-16.

[25] PlumbDroid: Automated Repair of Resource Leaks in Android Applications. `https://doi.org/10.5281/zenodo.6759123`. Accessed: 2022-06-27.

[26] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of Android apps: A systematic literature review. *Inf. Softw. Technol.*, 88:67–95, 2017.

[27] Shuying Liang, Matthew Might, and David Van Horn. Anadroid: Malware analysis of Android with user-supplied predicates. *Electron. Notes Theor. Comput. Sci.*, 311:3–14, 2015.

[28] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. Fixing resource leaks in Android apps with light-weight static analysis and low-overhead instrumentation. In *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, pages 342–352. IEEE Computer Society, 2016.

[29] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. Droidleaks: a comprehensive database of resource leaks in Android apps. *Empirical Software Engineering*, 24(6):3435–3483, 2019.

[30] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. ICSE 2014, New York, NY, USA, 2014. Association for Computing Machinery.

[31] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Jian Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Trans. Software Eng.*, 40(9):911–940, 2014.

[32] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.

[33] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 133–146. ACM, 2012.

[34] Francesco Logozzo and Matthieu Martel. Automatic repair of overflowing expressions with abstract interpretation. In Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff, editors, *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013*, volume 129 of *EPTCS*, pages 341–357, 2013.

[35] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178. ACM, 2015.

[36] Linghui Luo, Eric Bodden, and Johannes Späth. A qualitative analysis of Android taint-analysis results. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 102–114. IEEE, 2019.

[37] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering*, 2016.

[38] Alejandro Mazuera-Rozo, Catia Trubiani, Mario Linares-Vásquez, and Gabriele Bavota. Investigating types and survivability of performance bugs in mobile apps. *Empirical Softw. Engg.*, 25(3):1644–1686, may 2020.

[39] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[40] MediaPlayer Overview. `https://developer.android.com/guide/topics/media/mediaplayer`. Accessed: 2021-02-16.

[41] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018.

[42] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[43] Y. Pan, X. Ge, C. Fang, and Y. Fan. A systematic literature review of android malware detection using static analysis. *IEEE Access*, 8:116363–116379, 2020.

[44] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, New York, NY, USA, 2011. Association for Computing Machinery.

[45] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 648–659, 2017.

[46] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[47] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 535–544. ACM, 2010.

[48] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 332–344. IEEE Computer Society, 1986.

[49] Antonio Vetrò, Marco Torchiano, and Maurizio Morisio. Assessing the precision of FindBugs by mining Java projects developed at a university. In Jim Whitehead and Thomas Zimmermann, editors, *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, pages 110–113. IEEE Computer Society, 2010.

[50] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 419–431. ACM, 2004.

[51] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering (ICSE)*, pages 364–374. IEEE, 2009.

[52] Haowei Wu, Yan Wang, and Atanas Rountev. Sentinel: generating GUI tests for Android sensor leaks. In Xiaoying Bai, J. Jenny Li, and Andreas Ulrich, editors, *Proceedings of the 13th International Workshop on Automation of Software Test, AST@ICSE 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 27–33. ACM, 2018.

[53] Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in Android applications. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 185–195, New York, NY, USA, 2016. Association for Computing Machinery.

[54] Haowei Wu, Hailong Zhang, Yan Wang, and Atanas Rountev. Sentinel: Generating GUI tests for sensor leaks in Android and Android Wear apps. *Software Quality Journal*, 28(1):335–367, mar 2020.

[55] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Trans. Software Eng.*, 42(11):1054–1076, 2016.

[56] Dacong Yan, Shengqian Yang, and Atanas Rountev. Systematic testing for resource leaks in Android applications. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 411–420. IEEE Computer Society, 2013.

[57] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in Android applications. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.