Automated Repair of Resource Leaks in Android Applications

Bhargav Nagaraja Bhatt Carlo A. Furia
bhattb@usi.ch bugcounting.net
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 uses static analysis to find execution traces that may leak a resource. The information built for detection also undergirds 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 and produces correct fixes for a variety of resource leak bugs. This indicates it 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 device's user (for example, switching apps) or other changes in the environment (for example, 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 common in Android apps [33], where they often result in buggy behavior that ultimately degrades an app's responsiveness and usability.

Research in the last few years (which we summarize in Sec. 5) has developed techniques to *detect* resource leaks using dynamic analysis [10, 33], static analysis [41, 42], or a combination of both [9]. 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 Sec. 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).

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; 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.

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. We evaluated PlumbDroid's performance empirically on leaks in 17 Android apps from the curated collection DroidLeaks [21]. These experiments, described in Sec. 4, confirm that PlumbDroid is a scalable automated leak repair technique (less than 1.5 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).

The implementation of PlumbDroid and a replication package of our experiments are available as open source at `[To be released after double-blind reviewing]`.

2 AN EXAMPLE OF PLUMBDROID IN ACTION

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

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 6. 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

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

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

method `onPause`, so that the app has a chance to appropriately react to such changes in the environment. Unfortunately, the implementation of `onPause` in Fig. 1 does not *release* the media player, even though the app will be unable to use it while paused [30], and just acquires a new handle to it when it resumes. This causes a *resource leak*: the acquired resource `MediaPlayer` is not appropriately released. If the user flips the phone back and forth, this leak will result in wasting system resources and possibly in an overall performance loss.

PlumbDroid can automatically analyze the implementation of IRCCloud looking for leaks such as the one highlighted in Fig. 1. PlumbDroid generates an abstraction 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 Fig. 1's example, PlumbDroid builds a fix at line 11 consisting of 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

Fig. 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 Sec. 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. Sec. 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* [35] since there are resources—such as wait locks—that may be acquired and released multiple times.[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 Sec. 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 [6]. Each component type—such as activities, services, and content providers—has an associated *callback graph*, which constraints the order in which user-defined procedures are executed. As shown by the example of Fig. 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 Fig. 1).

Because it follows 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 depend on the event that occur). PlumbDroid deals with this *implicit* execution flow in two steps. First (Sec. 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 (Sec. 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.

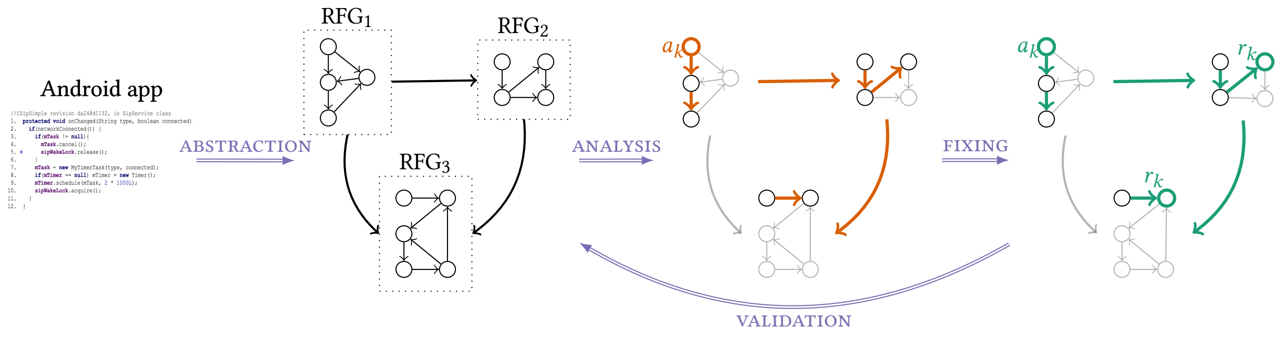
Fix generation. PlumbDroid's analysis stage extracts detailed information that is useful not only to detect leaks but also to *generate fixes* that avoid the leaks. As we describe in Sec. 3.4, PlumbDroid builds fixes by adding a release of every leaked resource as early as possible along each leaking 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 (described in Sec. 4) indicate that validation is nearly always successful.

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 $(a_1, r_1)\,(a_2, r_2)\ldots$. 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

[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.



```
//CSipSimple revision da248d1132, in SipService class
  protected void onChanged(String type, boolean connected)
    if(networkConnected()) {
      if(mTask != null){
        mTask.cancel();
+       sipWakeLock.release();
      }
      mTask = new MyTimerTask(type, connected);
      if(mTimer == null) mTimer = new Timer();
      mTimer.schedule(mTask, 2 * 1000L);
      sipWakeLock.acquire();
    }
  }
```

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.

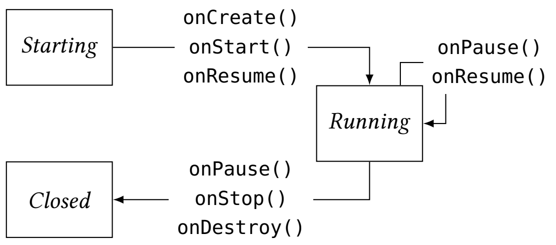


Figure 3: Simplified callback graph of Android component.

represents. For example, resource `MediaPlayer` can be acquired and released with (**`new`**, `release`)—used in Fig. 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 Fig. 1, it builds one such graph for every callback function, and for all methods called within those functions.

3.2.1 Abstraction: 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 realistic applications are large and complex, but only a small subset of their blocks typically involve accessing resources. Therefore, PlumbDroid builds resource-flow graphs, which abstract the control flow by only retaining information that is relevant for detecting resource leaks. RFGs are similar abstraction as energy-flow graphs [9, 42].

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 Alg. 1. Then, it builds the resource-flow graph R by connecting the resource path graphs according to the control-flow structure—as in Alg. 2.

Resource path graph. A basic block corresponds to a sequence of statements without internal branching. Alg. 1 builds the resource

```
Input: control-flow basic block $b$, resource list $L$
Output: resource path graph $p$
$p \leftarrow \emptyset$ // initialize $p$ to empty graph
foreach statement $s$ in block $b$ do
    if $s$ invokes a resource acquire operation $a$ in $L$ then
        $n \leftarrow$ new $AcquireNode(a)$
    else if $s$ invokes resource release operation $r$ in $L$ then
        $n \leftarrow$ new $ReleaseNode(r)$
    else if $s$ invokes any other operation $o$ then
        $n \leftarrow$ new $TransferNode(o)$
    else if $s$ is a return statement then
        $n \leftarrow$ new $ExitNode$
    else
        $n \leftarrow$ NULL
    if $n \neq$ NULL then
        append node $n$ to path graph $p$'s tail
// if $b$ contains no resource-relevant statements
if $p = \emptyset$ then
    $p \leftarrow$ new $TrivialNode$    // return a trivial node
```

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

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 Sec. 3.3); in other words, intra-procedural 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

```
**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 block $b \in C$
3      $p(b) \leftarrow Path(b, L)$    // call to Algorithm 1
4  $R \leftarrow \{$ entry node $s$ $\}$
5  $c_0 \leftarrow$ the entry block of $C$
6  add an edge connecting $s$ to $p(c_0)$'s entry
7  foreach block $b_1$ in control-flow graph $C$ do
8      foreach block $b_2$ in $b_1$'s successors in $C$ do
9          add an edge connecting $p(b_1)$'s exit to $p(b_2)$'s exit
10     foreach ExitNode $e$ in $p(b_1)$ do
11         add an edge connecting $e$'s predecessors to $f$
12         (the exit node of $C$)
```

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

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. Alg. 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 $b_1 \longrightarrow b_2$ 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. Because of this equivalence, 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".

3.2.2 Analysis: Context-Free Emptiness. 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 [37]—as follows. Since pushdown automata accept context-free languages, we encode leak-free sequences as the language accepted by a deterministic pushdown automaton.[2]

Definition 3.1 (Pushdown automaton [35]). 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 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.

Let's consider the case of a single resource $L = \{(a, r)\}$ that admits multiple acquire and release. (Generalization to multiple resources is conceptually straightforward but requires lengthier details.) The pushdown automaton A_L in Fig. 4 accepts all strings over alphabet $\Sigma^L = \{s, f, a, r\}$ of the form $s\,B\,f$ where $B \in \{a, r\}^*$ is any balanced sequence of a's and r's—that is, a leak-free sequence. (For the simpler example of resource `MediaPlayer` in Fig. 1, leak-free sequences are a regular language—because the resource is not reentrant—matching the regex $s\,($`new release`$)^*\,f$.) Since A_L is a *deterministic* pushdown automaton,[3] it is closed under complement. Therefore, there exists a deterministic pushdown automaton $\overline{A_L}$ that accepts the complement language $\overline{\mathcal{L}(L)}$ of all leaking sequences. Furthermore, testing whether any pushdown automaton accepts the empty language is decidable in polynomial time [1].

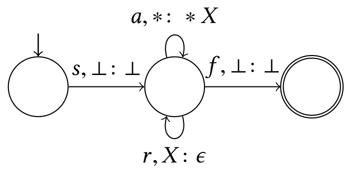


Figure 4: Deterministic pushdown automaton A_L accepting the language of leak-free sequences over nested acquire a and release r. A transition $x, y\colon Z$ reads input x when y is on top of the stack, and replaces it with string Z. $*$ is a shorthand for any symbol, ϵ is the empty string, and $\perp$ is the empty stack symbol.

Given a resource-flow graph $R = \langle V, E \rangle$, the language $\mathcal{L}(R)$ accepted by R is the set of all paths π through R such that π starts in R's entry node s and ends in R's exit node f. Equivalently, we can define a finite-state automaton $A_R = \langle \Sigma^R, Q^R, I^R, \delta^R, F^R \rangle$ that accepts precisely the language $\mathcal{L}(R)$, defines as follows: (1) $\Sigma^R = \Sigma^L$ defined above; (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 function δ^R derives from R's edges: for every edge $m \to n$ in R, $n \in \delta^R(m, type(m))$ is a transition from state m to state n that reads input symbol $type(m)$ 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 also assume that A_R is *deterministic*: even though the definition above may introduce nondeterminism, every finite-state automaton can be converted to an equivalent one that is deterministic.

Intersection automaton. Given that L encodes leak-free sequences, R is free from leaks in L iff $\mathcal{L}(R) \subseteq \mathcal{L}(L)$; that is, every

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

[3]Precisely, a visibly pushdown automaton [1] would be sufficient.

entry-to-exit path in R belongs to the language $\mathcal{L}(L)$ of well-formed acquire and release operations. Equivalently, we can check that the intersection $\mathcal{L}(R) \cap \overline{\mathcal{L}(L)} = \emptyset$ is empty, where $\overline{\mathcal{L}(L)}$ is the *complement* of $\mathcal{L}(L)$—the set of all sequences over Σ_L that are *not* leak-free.

In order to check whether the intersection of those two languages is empty, let us construct a deterministic pushdown automaton $A^X = \langle \Sigma^X, Q^X, I^X, \Gamma^X, \delta^X, F^X \rangle$ that accepts precisely $\mathcal{L}(R) \cap \overline{\mathcal{L}(L)}$. We can build X from A_R and $\overline{A_L}$ as follows: (1) $\Sigma^X = \Sigma^L$ is the usual alphabet; (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 A_L that input the same symbol σ, A^X has a transition $((p_2, q_2), G) = \delta^X((p_1, q_1), \sigma, \gamma)$ that manipulates the stack as in A_L's transition. Since both A_R and $\overline{A_L}$ are deterministic, so is A^X.

Intra-procedural leak detection. To sum up, PlumbDroid detects leaks of resources L in a procedure P as follows:

(1) Build resource-flow graph R modeling P, and its equivalent finite-state automaton A_R
(2) Build pushdown automaton $\overline{A_L}$ modeling *leaking traces*
(3) From A_R and $\overline{A_L}$, build pushdown automaton A^X modeling leaking traces of R
(4) If A^X accepts the empty language, then 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 Sec. 3.3.1. Since in Android system callbacks determine the overall execution order of an app, Sec. 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.

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 $u \longrightarrow v$ 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 Alg. 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 Sec. 3.6 and Sec. 4, the limitation 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, Alg. 3 processes each of them starting from those corresponding to procedures that do not call any other procedures (line 4). The

```
Input: call graph $C = \langle V, E \rangle$, pushdown automaton $\overline{A^L}$
Output: $H = \{H_p \mid V \ni p$ is not called by any procedure$\}$
1  $N \leftarrow$ topological sort of $C$
2  foreach $n \in N$ do
3      // for each procedure $n$
4      if $n$ is not calling any other procedure then
5          // leaking paths in $n$'s intra-procedural analysis
6          $H_n \leftarrow LeakingPaths(R_n, \overline{A^L}, L)$
7      else if $n$ calls procedures $m_1, m_2, \dots$ then
8          $R'_n \leftarrow R_n$
9          foreach $m \in \{m_1, m_2, \dots\}$ do
10             // $R'_n$ is $R_n$ with call-to-$m$ nodes replaced by $H_m$
11             $R'_n \leftarrow R'_n[TransferNode(m) \mapsto H_m]$
12         // leaking paths in intra-procedural analysis of $R'_n$
13         $H_n \leftarrow LeakingPaths(R'_n, \overline{A^L}, L)$
```
Algorithm 3: Algorithm *AllCalls* which computes inter-procedural resource-flow paths accepted by "leaking" pushdown automaton $\overline{A^L}$.

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 analysis of Sec. 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 7–13 of Alg. 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 Alg. 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 Fig. 3 for a simplified example). Apps provide implementations of such callback functions, which PlumbDroid can analyze for leaks. When a 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 should be released. PlumbDroid enumerates all paths that first go from the graph's entry to the states 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". Each path determines a sequence of procedures $P_1; P_2; \dots$ 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 Sec. 3.3.1.

For example, Fig. 1's resource `MediaPlayer` should be released in callback function `onPause`. For a component with the callback graph of Fig. 3, and $D = 2$, PlumbDroid enumerates the path *Starting* $\rightarrow$ *Running* $\rightarrow$ *Running*, corresponding to the sequence of callbacks `onCreate(); onStart(); onResume(); onPause(); onResume(); onPause()`. If the media recorded is acquired and not later released in these call's implementations, PlumbDroid will detect a leak.

3.4 Fix Generation

Fix templates. Once PlumbDroid detects a resource leak, fixing it amounts to injecting suitable release operations at suitable locations in the app's implementation. PlumbDroid builds fixes using the 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 Sec. 3.6): 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 Sec. 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 Fig. 1, PlumbDroid inserts the call to `release` in callback `onPause`, which is as early as possible in the sequence of callbacks.

3.5 Validation

Since leak analysis is sound (see Sec. 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 Sec. 3.4), it is possible that it interferes in unintended ways with other *usages* of the resource. In particular, 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. In such cases, PlumbDroid would fix the leak but it would also introduce a use-after-release error by releasing the resource too early.

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 leaks as well as absence of use-after-release errors. This analysis reuses the techniques of Sec. 3.2 and Sec. 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. If validation fails, PlumbDroid's fix can still be used 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.

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 Sec. 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

3.6.1 Soundness. A leak detection technique is *sound* [32] 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. (1) Since it performs a fixed-depth unrolling of paths in the callback graph (Sec. 3.3.2), it may miss leaks that only occur along longer paths. (2) Since it ranks procedures according to their call order (Sec. 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 [23]. 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 at least once (i.e., $N > 1$), unsoundness source 1) should not affect the analysis of resources of the usual types. 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 2) is also unlikely to occur. The experiments of

Sec. 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* [32] if it never reports false alarms: 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 `r` and `s` that is acquired using `r` and released using `s` will be considered leaking by PlumbDroid, which thinks `r` and `s` are two different resources.

In practice, these two sources of imprecision are limitations to PlumbDroid's applicability. When aliasing is not present, the experiments of Sec. 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.

3.7 Implementation

We implemented PlumbDroid in Python on top of AndroGuard [5] and Apktool [7]. 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). 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: Is PlumbDroid *scalable* to real-world Android apps?
RQ3: How does PlumbDroid's behavior depend on the *unrolling depth* parameter, which controls its analysis's level of detail?

4.1 Experimental Setup

4.1.1 Subjects. Our experiments target apps in DroidLeaks [21]—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 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 acquire 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 Sec. 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. 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.[4]

Tab. 1 summarizes the characteristics of the 9 resources we selected for our experiments according to the above criteria. The 17 apps in DroidLeaks that we analyzed are listed in (the lefthand side of) Tab. 2.

4.1.2 Experimental Protocol. In our experiments, each run of PlumbDroid targets one app and repairs leaks of a specific resource,[5] and reports a number of leaks and, for each of them, a fix.

After each experiment we did a *sanity check*: we manually inspected the fixes, 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; its validation phase further ascertains that the fixes do not introduce use-after-release errors.

The main parameter regulating PlumbDroid's behavior is the unrolling depth D. We ran experiments with $D = 1$, $D = 2$, $D = 3$,

[4] We also tried PlumbDroid on a sample of aliasing resources, which confirmed it remains effective but is also prone to generating a significant number of false positives ("fixes" that are harmless but not really necessary).
[5] PlumbDroid can analyze leaks for multiple resources in the same run, but we do not use this features in the experiments to have a fine-grained breakdown of PlumbDroid's performance.

and $D = 5$. Our goal is to demonstrate that the default value $D = 2$ 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.

	OPERATIONS		RELEASED	
RESOURCE	a_k	r_k	`on`	REENTR
`AudioRecorder`	`new`	`release`	`Pause,Stop`	N
`BluetoothAdapter`	`enable` `startDiscovery`	`disable` `cancelDiscovery`	`Stop` `Pause`	

`Camera`	`lock`
	`open`
	`startPreview`
		`unlock`
		`release`
		`stopPreview`
			`Pause`	N

`LocationListener`	`requestUpdates`	`removeUpdates`	`Pause`	N
`MediaPlayer`	`new` `start`	`release` `stop`	`Pause,Stop`	N
`Vibrator`	`vibrate`	`cancel`	`Destroy`	N
`WakeLock`	`acquire`	`release`	`Pause`	Y
`WifiLock`	`acquire`	`release`	`Pause`	Y
`WifiManager`	`enable`	`disable`	`Destroy`	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, the callback function `on...` where the resource should be RELEASED according to documentation, 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).

4.2 Experimental Results

4.2.1 RQ1: Correctness. Column FIXED in Tab. 2 reports the number of leaks that PlumbDroid detected and fixed with a correct fix; column INVALID how many of these fixes failed validation (were "unsafe"). PlumbDroid was very effective at detecting leaks in non-aliasing resources. It detected and fixed *all* 26 leaks reported by DroidLeaks and included in our experiments, building a correct fix for each of them.

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: (1) a real leak included in DroidLeaks; (2) a real leak *not* included in DroidLeaks; (3) a spurious leak. By inspecting the leak reports and the apps we managed to confirm that 44 leaks (88%) reported by PlumbDroid are in categories 1) (26 leaks or 52%) or 2) (18 leaks or 36%) above—and thus are real leaks. Unfortunately, the remaining 6 leaks (12%) reported by PlumbDroid were found in apps whose bytecode is only available in *obfuscated* form, which means we cannot be certain they are not spurious; these unconfirmed cases are counted in column "?" in Tab. 2. Even in the worst case in which all of these are spurious, PlumbDroid's precision would remain high (88%). The actual precision is likely much higher: in all cases where we could analyze the code, we found a real leak; unconfirmed cases probably just require more evidence.

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 confirmed this. 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: (1) Three identical fixes (releasing the same resource in the same spot) repairing three distinct leaks of resource `MediaPlayer` in app SureSpot. According to the Android reference manual [3], 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 introduced a configuration option to tell PlumbDroid whether it should release resources *early* or *late*. An app developer can therefore configure our analyzer in a way that suits their design decisions. In particular, configuring PlumbDroid with option *late* in this case generates a fix that passes validation. (2) Two identical fixes (releasing the same resource in the same spot) 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.

PlumbDroid detected and fixed 50 leaks in DroidLeaks producing correct-by-construction fixes. PlumbDroid's detection is very precise on non-aliasing resources.

4.2.2 RQ2: Performance. Columns TIME in Tab. 2 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 under 5 minutes *per app-resource* and under 80 seconds *per repaired leak.*

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 Tab. 2'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—is usually much lower than the cyclomatic complexity [29] M' of the full control-flow graph, which makes the exhaustive analysis of a resource-flow graph scalable.

PlumbDroid is scalable: it takes less than 80 seconds on average to detect and fix a resource leak.

4.2.3 RQ3: Unrolling. In the experiments reported so far, PlumbDroid ran with the unrolling depth parameter $D = 2$, which is the default. Tab. 3 summarizes the key data about experiments using different values of D. A value of $D \geq 2$ is required for soundness:

		RFG		CC		DroidLeaks	TIME (s)							
APP	KLOC	$\lvert V\rvert$	$\lvert E\rvert$	M/M'	RESOURCE	LEAKS	ABSTRACTION	ANALYSIS	FIXING	VALIDATION	TOTAL	FIXED	?	INVALID
APG	42.0	4 968	7 442	0.47	`MediaPlayer`	1	32.5	212.5	0.2	75.1	320.4	1	0	0
BarcodeScanner	10.6	1 189	2 462	0.35	`Camera`	1	8.0	43.8	0.2	14.5	66.6	3	0	0
CallMeter	13.5	1 840	3 216	0.35	`WakeLock`	3	10.2	66.0	0.2	18.9	95.2	4	0	0
ChatSecure	37.2	5 430	8 686	0.48	`BluetoothAdapter`	0	22.6	167.5	0.4	42.4	232.9	2	0	0
					`Vibrator`	1		140.6	0.4	46.2	209.8	2	0	0
ConnectBot	17.6	1 956	3 814	0.23	`WakeLock`	0	10.4	62.8	0.3	19.2	92.7	2	0	0
CSipSimple	49.0	5 712	9 154	0.42	`WakeLock`	2	40.3	222.2	0.5	71.3	334.4	4	2	0
IRCCloud	35.3	4 782	9 755	0.43	`MediaPlayer`	0	21.2	165.3	0.4	46.5	233.5	3	0	0
					`WifiLock`	1		163.9	0.2	38.3	223.7	2	0	0
K-9 Mail	78.5	8 831	16 390	0.29	`WakeLock`	2	50.2	422.9	0.1	123.3	596.5	2	0	0
OpenGPSTracker	12.3	1 418	2 791	0.29	`LocationListener`	1	9.7	68.4	0.4	20.9	99.4	2	0	0
OsmDroid	18.4	2 222	3 545	0.36	`LocationListener`	2	14.3	63.2	0.2	14.8	92.5	4	2	0
ownCloud	31.6	4 444	8 980	0.6	`WifiLock`	2	17.7	137.7	0.4	46.1	201.9	4	2	0
QuranForAndroid	21.7	2 898	4 545	0.43	`MediaPlayer`	1	16.9	105.8	0.5	30.2	153.4	2	0	0
SipDroid	24.5	3 178	4 583	0.38	`Camera`	4	15.8	104.2	0.4	23.7	144.1	4	0	0
SureSpot	41.0	3 575	7 240	0.37	`MediaPLayer`	2	24.6	177.6	0.2	48.7	251.1	3	0	3
Ushahidi	35.7	5 073	10 417	0.43	`LocationListener`	1	26.4	175.6	0.5	42.0	244.5	2	0	2
VLC	18.1	2 689	4 199	0.55	`WakeLock`	2	10.7	79.9	0.3	24.4	115.4	2	0	0
Xabber	38.2	4 194	8 478	0.31	`AudioRecorder`	2	29.7	173.2	0.3	38.7	241.9	2	0	0
AVERAGE	30.9	3 788	6 805	0.4			21.3	144.9	0.3	41.3	207.9			
TOTAL	525.2	64 399	115 697	6.74		26	405.2	2753.0	6.2	785.4	3949.9	50	6	5

Table 2: Results of running PlumbDroid on apps in DroidLeaks. For every APP, the table reports its size KLOC in thousands of lines of code, 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 the number of LEAKS of that resource and app included in DroidLeaks; PlumbDroid's running time to perform each of the steps of Fig. 2 (ABSTRACTION, ANALYSIS, FIXING, and VALIDATION); as well as the TOTAL running time; since the abstraction is built once per app, the corresponding time is the same for all resources used by the 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 across all experiments.

	AVERAGE	FIXED	MISSED	
D	TIME (s)	LEAKS	LEAKS	INVALID
1	125.4	40	10	5
2	207.9	50	0	5
3	342.4	50	0	5
5	1612.3	50	0	5

Table 3: Results of running PlumbDroid on non-aliasing resources in DroidLeaks with different unrolling depths. 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; the total number of FIXED LEAKS, of MISSED LEAKS (not detected, and hence not fixed); and the number of INVALID fixes. The row with $D = 2$ corresponds to the data in Tab. 2.

PlumbDroid running with $D = 1$ missed 10 leaks affecting reentrant resources—leaks that only occur if the resource is acquired multiple times. Specifically, it missed leaks of resources `WakeLock` and `WifiLock` in apps CallMeter, CSipSimple, and IRCCloud. On the other hand, the running time grows conspicuously with the value of D. Therefore, the default $D = 2$ is the empirically optimal value: it achieves soundness without unnecessarily increasing the running time.

PlumbDroid's analysis is sound provided it unrolls each callback loop at least twice.

4.3 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. Therefore, we could not match with absolute certainty the leaks and fixes listed in DroidLeaks with the fixes produced by PlumbDroid, nor could we run systematic testing of the automatically fixed apps. This threat is significantly mitigated by other sources of evidence that PlumbDroid indeed produced fixes matching those in DroidLeaks: first, the manual inspection we could carry out on the apps that are not obfuscated confirmed in all cases our expectations; second, PlumbDroid's analysis is *sound*, and hence it should detect all leaks (except possibly for bugs in its implementation); third, running the fixed apps did not show any apparent change in their behavior.

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, at the level of app stores.

We didn't formally prove the *soundness* or *precision* of PlumbDroid's analysis, nor that our implementation is free from bugs. Nonetheless, the empirical evaluation provides convincing evidence that PlumbDroid is indeed sound (for $D \geq 2$), and that *aliasing* is the primary source of imprecision. In future work, we plan to equip PlumbDroid with alias analysis, 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 helps to generalize our evaluation. We used all apps and resource in DroidLeaks that PlumbDroid can analyze with precision, which led to finding numerous leaks not included in DroidLeaks. In the future, we would like to run PlumbDroid on other apps too; the main obstacle to doing this is the difficulty of obtaining the ground truth for apps that are only available in bytecode format, often in obfuscated form—which makes curated collections like DroidLeaks particularly valuable.

5 RELATED WORK

Automated program repair (APR). PlumbDroid is a form of APR targeting a specific kind of bugs (resource leaks) and programs (Android apps). The bulk of "classic" APR research [14, 28, 31, 39] 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 detect and localize errors [11, 16], and to validate the generated fixes [17, 26, 34]. General-purpose APR completely based on static analysis is less common [13, 24, 25], 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.

Static analysis for Android. Since Android apps run on mobile devices, they are prone to defects such as privacy leak [15], permission misuse [20], and other security vulnerabilities [44] 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 [27] and other kinds of control-flow based analyses [8, 19]. There has been plenty of work that applied static analysis to analyze resource management in Java [12, 36, 38], but these techniques are not directly applicable to mobile apps written in Java due to the peculiarities of the Android programming model.

Resource leaks in Android. The amount of work on *detecting* resource leaks [9, 10, 22, 33, 42] and the recent publication of the DroidLeaks curated collection of leaks [21] indicate that leak detection is a practically important problem in Android programming. Whereas PlumbDroid is the first fully automated approach for *fixing* resource leaks, leak detection has used a broad range of techniques—from static analysis to testing.

Among the approaches using testing, [10] proposes a test-generation framework capable of building inputs exposing resource leaks that lead to energy inefficiencies. Since it targets energy efficiency, [10]'s framework consists of a hybrid setup that includes hardware to physically measure energy consumption; its measurements are combined with more traditional software metrics to generate testing oracles for energy leak detection. The framework's generated tests consist of sequences of UI events that trigger energy leaks or other inefficiencies exposed by the oracles. As it is usual for test-case generation, [10]'s technique is based on heuristics and statistical assumptions about energy consumption patterns, and hence it is not exhaustive. Other tools [2, 4, 18, 40, 43] exist that use tests to detect resource leaks—such as memory leaks. Key idea underlying these approaches is to combine precise resource profiling and search-based test-case generation looking for inputs that expose leaks.

Approaches based on static analysis build an *abstraction* of an app's behavior, which can be searched exhaustively for leaks. For example, Relda2 [42] is a techniques that 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 unrolling them a finite number of times, and does not accurately track nested resource acquisitions, its analysis is generally unsound (leaks may go undetected) and imprecise (spurious errors may be reported).

Energy-patch [9] is another approach for leak detection based on static techniques. It 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). For each executable leaking behavior, symbolic execution can also generate a concrete program input that triggers the energy-leak bug. Energy-patch targets a different kind of resources leak (energy consumption related) than PlumbDroid, and it focuses on leak detection. Even though [9] presents a simple technique for generating fixes, it follows the simple approach of releasing all resources in the very last callback of an activity's lifecycle, its approach to fixing leaks may be impractical because it would not follow Android programming's best practices (as we discuss in Sec. 3.4).

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

This work was partially supported by the Swiss National Science Foundation (SNF) under grant Hi-Fi 200021-182060.

REFERENCES

[1] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, László Babai (Ed.). ACM, 202–211. https://doi.org/10.1145/1007352.1007390

[2] Domenico Amalfitano, Vincenzo Riccio, Porfirio Tramontana, and Anna Rita Fasolino. 2020. Do Memories Haunt You? An Automated Black Box Testing Approach for Detecting Memory Leaks in Android Apps. *IEEE Access* 8 (2020), 12217–12231. https://doi.org/10.1109/ACCESS.2020.2966522

[3] Android Reference Manual [n.d.]. Android Reference Manual. https://developer.android.com/reference. Accessed: 2020-03-03.

[4] Android Studio Monitor [n.d.]. Android Studio Monitor. https://developer.android.com/studio/profile/monitor. Accessed: 2020-03-04.

[5] AndroidGuard [n.d.]. AndroGuard. https://github.com/androguard/androguard. Accessed: 2020-03-03.

[6] AndroidPlatform [n.d.]. Android Platform Architecture. https://developer.android.com/guide/platform/. Accessed: 2020-03-04.

[7] Apktool [n.d.]. Apktool. https://ibotpeaches.github.io/Apktool/. Accessed: 2020-03-03.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. 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)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[9] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2018. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Trans. Software Eng.* 44, 5 (2018), 470–490. https://doi.org/10.1109/TSE.2017.2689012

[10] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. 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*. 588–598. https://doi.org/10.1145/2635868.2635871

[11] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based Program Repair Without the Contracts. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 637–647. http://dl.acm.org/citation.cfm?id=3155562.3155642

[12] Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. 2008. The CLOSER: automating resource management in Java. In *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*, Richard E. Jones and Stephen M. Blackburn (Eds.). ACM, 1–10. https://doi.org/10.1145/1375634.1375636

[13] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 459–470. https://doi.org/10.1109/ICSE.2015.64

[14] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2018. Automatic Software Repair: A Survey. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1219–1219. https://doi.org/10.1145/3180155.3182526

[15] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Trust and Trustworthy Computing - 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings (Lecture Notes in Computer Science)*, Stefan Katzenbeisser, Edgar R. Weippl, L. Jean Camp, Melanie Volkamer, Mike K. Reiter, and Xinwen Zhang (Eds.), Vol. 7344. Springer, 291–307. https://doi.org/10.1007/978-3-642-30921-2_17

[16] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. 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*. 12–23.

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. 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*. 298–309.

[18] leakcanary [n.d.]. Leak Canary Tool. https://square.github.io/leakcanary/. Accessed: 2020-03-03.

[19] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of Android apps: A systematic literature review. *Inf. Softw. Technol.* 88 (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

[20] Shuying Liang, Matthew Might, and David Van Horn. 2015. AnaDroid: Malware Analysis of Android with User-supplied Predicates. *Electron. Notes Theor. Comput. Sci.* 311 (2015), 3–14. https://doi.org/10.1016/j.entcs.2015.02.002

[21] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. 2019. DroidLeaks: a comprehensive database of resource leaks in Android apps. *Empirical Software Engineering* 24, 6 (2019), 3435–3483. https://doi.org/10.1007/s10664-019-09715-8

[22] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Jian Lu. 2014. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Trans. Software Eng.* 40, 9 (2014), 911–940. https://doi.org/10.1109/TSE.2014.2323982

[23] 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. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[24] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In *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*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 133–146. https://doi.org/10.1145/2384616.2384626

[25] Francesco Logozzo and Matthieu Martel. 2013. Automatic Repair of Overflowing Expressions with Abstract Interpretation. In *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 (EPTCS)*, Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff (Eds.), Vol. 129. 341–357. https://doi.org/10.4204/EPTCS.129.21

[26] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 166–178. https://doi.org/10.1145/2786805.2786811

[27] Linghui Luo, Eric Bodden, and Johannes Späth. 2019. 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*. IEEE, 102–114. https://doi.org/10.1109/ASE.2019.00020

[28] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2016. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset. *Empirical Software Engineering* (2016). https://doi.org/10.1007/s10664-016-9470-4

[29] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Software Eng.* 2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[30] MediaPlayerOverview [n.d.]. MediaPlayer Overview. https://developer.android.com/guide/topics/media/mediaplayer. Accessed: 2020-03-03.

[31] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[32] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer. https://doi.org/10.1007/978-3-662-03811-6

[33] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 2011. 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)*. Association for Computing Machinery, New York, NY, USA, Article Article 5, 6 pages. https://doi.org/10.1145/2070562.2070567

[34] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. 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*. 648–659.

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

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

(Eds.). ACM, 535–544. https://doi.org/10.1145/1806799.1806876

[37] Moshe Y. Vardi and Pierre Wolper. 1986. 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*. IEEE Computer Society, 332–344.

[38] Westley Weimer and George C. Necula. 2004. Finding and preventing run-time error handling mistakes. In *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*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 419–431. https://doi.org/10.1145/1028976.1029011

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

[40] Haowei Wu, Yan Wang, and Atanas Rountev. 2018. Sentinel: generating GUI tests for Android sensor leaks. In *Proceedings of the 13th International Workshop on Automation of Software Test, AST@ICSE 2018, Gothenburg, Sweden, May 28-29, 2018*, Xiaoying Bai, J. Jenny Li, and Andreas Ulrich (Eds.). ACM, 27–33. https://doi.org/10.1145/3194733.3194734

[41] Haowei Wu, Shengqian Yang, and Atanas Rountev. 2016. Static Detection of Energy Defect Patterns in Android Applications. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. Association for Computing Machinery, New York, NY, USA, 185–195. https://doi.org/10.1145/2892208.2892218

[42] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. *IEEE Trans. Software Eng.* 42, 11 (2016), 1054–1076. https://doi.org/10.1109/TSE.2016.2547385

[43] Dacong Yan, Shengqian Yang, and Atanas Rountev. 2013. 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*. IEEE Computer Society, 411–420. https://doi.org/10.1109/ISSRE.2013.6698894

[44] Yajin Zhou and Xuxian Jiang. 2013. 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. https://www.ndss-symposium.org/ndss2013/detecting-passive-content-leaks-and-pollution-android-applications