Repairing Leaks in Resource Wrappers

Sanjay Malakar*, Michael D. Ernst†, Martin Kellogg‡, Manu Sridharan*
*University of California, Riverside, USA †University of Washington, USA ‡New Jersey Institute of Technology, USA
Email: smala009@ucr.edu, mernst@cs.washington.edu, martin.kellogg@njit.edu, manu@cs.ucr.edu

***Abstract*—A resource leak occurs when a program fails to release a finite resource like a socket, file descriptor or database connection. While sound static analysis tools can *detect* all leaks, automatically *repairing* them remains challenging. Prior work took the output of a detection tool and attempted to repair only leaks from a hard-coded list of library resource types. That approach limits the scope of repairable leaks: real-world code uses *resource wrappers* that store a resource in a field and must themselves be closed.**

This paper makes four key contributions to improve resource leak repair in the presence of wrappers. (1) It integrates inference of resource management specifications into the repair pipeline, enabling extant fixing approaches to reason about wrappers. (2) It transforms programs into variants that are easier to analyze, making inference, detection, and fixing tools more effective; for instance, it makes detection tools report problems closer to the root cause, often in a client of a resource wrapper rather than within the wrapper class itself. (3) A novel *field containment analysis* reasons about resource lifetimes, enabling repair of more leaks involving resources stored in fields. (4) It introduces a new repair pattern and more precise reasoning to better handle resources stored in non-final fields.

Prior work fixed 41% of resource leak warnings in the NJR benchmark suite; our implementation Arodnap fixes 68%.

***Index Terms*—Program repair, resource leaks**

I. Introduction

Resource leaks, such as unreleased file handles, sockets, or database connections, are a persistent source of reliability issues. These defects often evade detection during testing and manifest only after extended run time, leading to degraded performance, outages, and even security vulnerabilities [1], [2]. Static analysis is a powerful tool for detecting such defects early in development, and many modern analyzers for Java — such as Infer [3], SpotBugs [4], and the Checker Framework's Resource Leak Checker [5] — can report potential leaks based on ownership and control-flow reasoning.

However, detection alone is insufficient. Developers complain that static analyses surface true issues but fail to provide actionable suggestions [2] and have too many false positives [6]. Automated repair tools that rely only on test-based validation [7], [8] are ineffective for leak repair, because resource management leaks typically do not manifest in test failures. Many leaks share common manifestation and repair patterns [9]; the state-of-the-art tool RLFixer [10] takes advantage of this insight. RLFixer relies on extant leak-detection tools like Infer, SpotBugs, and the Resource Leak Checker to detect leaks, which it repairs using a fixed set of repair templates. However, because RLFixer treats the leak-detection tools as a black-box warning oracle, it is limited to repairing leaks of *library resources*, like sockets or file descriptors, that the leak-detection tool tracks by default.

Our key insight is that repairing only leaks of library resources is usually insufficient, because many resources in practice are managed by *wrappers*: programmer-written classes that themselves act as resources, like the examples in figs. 1 and 2. Acting on this insight requires detecting and reasoning about such wrappers during the leak-detection stage, but extant leak-detection tools only reason about library resources (e.g., those defined in the JDK) by default. Recent work has extended one detection tool — the Resource Leak Checker (RLC) — with a specification generator ("RLC Inference" or "RLCI") that automates this manual process [11]. Our first contribution is to extend the combination of RLFixer and RLC to take advantage of RLCI specifications. This makes it possible to automatically detect and fix leaked programmer-written wrappers instead of only library resources.

However, the RLCI+RLC+RLFixer combination is only marginally better than the base RLC+RLFixer combination, repairing 50% instead of 41% of leaks in our experiments. The reason is that many resource leaks involving wrappers require reasoning about resources stored in *fields*. RLFixer marks a leak as "unfixable" whenever a resource may escape into a field. Our remaining contributions are a set of new program transformations and analyses, embodied in a tool called Arodnap. Arodnap extends RLFixer to reason about resources that are stored in fields, as well as fix leaks resulting from mishandling of those resources.

Arodnap improves on the handling of fields in prior work in three main ways. (1) It adds a code transformation stage to the detection-and-fixing pipeline (fig. 3) that runs after inference and leak detection. This code transformation stage enables precise reasoning and inference for many resource-containing fields; for example, it adds the `final` qualifier to eligible fields, converts resource-containing fields that can be scoped to a single method into local variables, and adds missing finalizer methods to classes that contain a resource field. After these transformations, inference and leak detection generate more actionable leak warnings. (2) We enhanced RLFixer to reason about fields: a new *field containment analysis* makes RLFixer's escape logic sound and less conservative in its field handling. (3) We improved handling of field overwrites. For overwrites in constructors, we improved RLC's reasoning. For overwrites in other methods, we added a new repair pattern to RLFixer, along with an analysis to determine when the new repair can be applied soundly. With these improvements, Arodnap can resolve 68% of all resource leak warnings on the same set of

benchmark programs used in RLFixer's evaluation, vs. 41% for RLFixer alone. In sum, our contributions are:

- We extend RLFixer to use RLCI to reason about wrappers.
- We introduce code transformations to ease analysis of and inference for resource fields.
- We introduce *field containment analysis*, a static analysis that identifies resource wrapper classes whose internal fields do not escape, enabling sound repair of more leaks.
- We improve analysis and repair for leaks due to non-final resource fields.
- We implemented our approach in a tool Arodnap and evaluated it on the NJR dataset [12] originally used to evaluate RLFixer, improving the fix rate from 41% for RLFixer to 68% for Arodnap.

Our artifact includes all the code, data, and scripts used in this paper [13]. Arodnap itself is also open-source [14].

II. Background

This section describes the three state-of-the-art tools on which our work builds: RLC for detecting resource leaks (§II-A), RLCI for inferring resource specifications (§II-B), and RLFixer (§II-C) for suggesting repairs. It explains how these tools operate, their limitations, and the motivation for the enhancements introduced in this work.

A. Static Leak Detection with the Resource Leak Checker

The Resource Leak Checker (RLC) is a pluggable type system built on top of the Checker Framework [15], [5]. It verifies that objects such as files, sockets, or streams are cleaned up (by an explicit call to a finalizer method) before they become unreachable. RLC is sound by design, scales to real-world codebases, and requires a manageable annotation burden.

At the core of RLC is the notion of `@MustCall` obligations. A type `T` annotated with `@MustCall("close")` indicates that `close()` must be invoked on every `T` object before its lifetime ends. The RLC analysis ensures that all such required methods are called along every path that leads to the object becoming unreachable (e.g., via scope exit or variable overwrite). Programmer-written resource specifications (expressed as RLC annotations) express ownership, obligation transfer, and aliasing relationships.

- `@Owning` references are responsible for eventually satisfying the `@MustCall` obligations of the object they refer to.
- `@NotOwning` marks a reference that is not responsible for the obligation, such as shared or borrowed values.
- `@EnsuresCalledMethods(x, y)`, written on a method m, guarantees that `x.y()` is called before m returns.

These annotations allow RLC to track resource lifecycle responsibilities across field assignments, parameter passing, method returns, and resource wrappers. Rather than relying on whole-program alias analysis, RLC uses these annotations to reason about ownership and aliasing in a modular, sound way [16]. To statically verify that obligations are fulfilled, RLC runs three cooperating analyses [5]. First, a type system computes the set of required `@MustCall` methods for each reference. Second, a type system computes which methods have definitely been invoked on a given value. Finally, a dataflow analysis verifies that all required methods have been called before a resource becomes unreachable.

```
1  +@MustCall("close")
2   class MyWriter {
3  +  @Owning PrintWriter pw;
4     MyWriter(String path) {
5       pw = new PrintWriter(path);
6     }
7  +  @EnsuresCalledMethods(value="pw", methods="close")
8     void close() {
9       pw.close();
10    }
11  }
12  void use() {
13    MyWriter writer = new MyWriter("f.txt");
14  }
```

Fig. 1: Specification inference makes leaks repairable. With no resource management specification annotations, RLC reports a leak at line 5, where repair is not possible. With added specifications (highlighted in green), RLC reports the leak at line 13 — where repair is feasible by closing the `MyWriter` object.

RLC is a specify-and-verify system: the programmer writes explicit ownership and aliasing annotations, and the programmer obtains a sound guarantee of no resource leaks. Unfortunately, most Java codebases lack these specifications. RLC ships with specifications for the JDK standard library; programmers are expected to write annotations like `@Owning` and `@MustCallAlias` to specify their own resource-handling code. Writing these annotations is tedious and error-prone, motivating automatic inference of resource-management specifications.

B. Inference for Resource Management Annotations

RLC Inference (RLCI) [11] statically discovers specifications related to resource lifecycles and ownership. By analyzing how objects are allocated, passed, and used throughout a program, the inference identifies and automatically annotates patterns of resource management.

This approach is especially valuable when analyzing a user-defined resource *wrapper* — a class that internally manages a resource delegate but may not explicitly expose it. In such cases, inference can insert annotations documenting that the wrapper owns its internal resources and exposes a finalizer method like `close()` that satisfies their obligations. These inferred specifications enable RLC to verify both the implementation and the use sites of the wrapper class.

Figure 1 demonstrates how inference enables more actionable leak warnings. The class `MyWriter` internally allocates a `PrintWriter` and provides a `close()` method that properly closes it. Without inference, RLC does not recognize that `MyWriter` is a wrapper that is responsible for closing the resource. RLC therefore reports a leak at the `PrintWriter` allocation inside the constructor (because the `PrintWriter` is not closed before the constructor returns). Once inference recovers the necessary specifications (in green in fig. 1), including `@MustCall` on the class and `@EnsuresCalledMethods` on the cleanup method, RLC shifts the warning to the `use()` method, which erroneously allocates a `MyWriter` object without closing it.

```
1  class TempFileWriter {
2    private PrintStream stream;
3
4    public TempFileWriter(String path) {
5        stream = new PrintStream(path);
6    }
7
8    void resetStream(String path) {
9
10     stream = new PrintStream(path);
11   }
12
13   public void printSomething() {
14     stream.println("hello");
15   }
16
17
18
19
20 }
21
22 class Client {
23   public static void print() {
24
25
26     TempFileWriter tmp = new TempFileWriter("f.txt");
27     tmp.printSomething();
28
29
30
31   }
32 }
```

(a) Leaky version without cleanup. RLC reports leaks at lines 5 and 10.

```
1  class TempFileWriter implements AutoCloseable {
2    @Owning private PrintStream stream;
3
4    public TempFileWriter(String path) {
5        stream = new PrintStream(path);
6    }
7
8    void resetStream(String path) {
9      if (stream != null) stream.close();
10     stream = new PrintStream(path);
11   }
12
13   public void printSomething() {
14     stream.println("hello");
15   }
16
17   public void close() {
18     stream.close();
19   }
20 }
21
22 class Client {
23   public static void print() {
24     TempFileWriter tmp = null;
25     try {
26       tmp = new TempFileWriter("f.txt");
27       tmp.printSomething();
28     } finally {
29       if (tmp != null) tmp.close();
30     }
31   }
32 }
```

(b) Arodnap inserts a finalizer (in blue), letting inference infer ownership (in red), shifting RLC's leak warning from line 5 to line 26, enabling RLFixer to insert cleanup logic (in green).

Fig. 2: Simplified leak repair example from the NJR dataset, benchmark `url882f91ec97_WenboCao_Microsoft_Drone`, file `GenerateDeterminantFromMinor.java`. RLFixer cannot repair the original code (fig. 2a). In fig. 2b Arodnap inserts the missing finalizer, which enables inference to infer ownership. RLFixer then eliminates both leaks: one inserting a `try–finally` wrapper at lines 25–30 and one by inserting a *pre-close* on line 9 before the field overwrite.

By *shifting* the warning from inside the wrapper to its call site, inference makes the leak warning more actionable and amenable to automatic repair.

C. Semi-Automated Repair with RLFixer

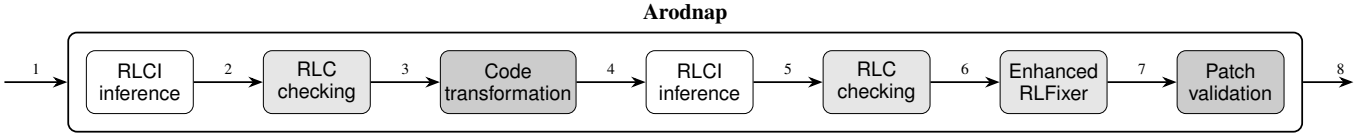
RLFixer [10] automatically generates fix hints for resource leaks based on warnings emitted by static analysis tools, such as RLC or Infer [3], which it treats as black-box warning generators. It aims to generate repairs that do not alter core program logic, using control-flow scaffolding like `try–finally` blocks to enforce cleanup. RLFixer emits its output as textual hints without applying or validating them, requiring manual developer intervention to write patches.

For each leak warning, RLFixer performs a lightweight alias analysis to group variables referring to the same resource instance — handling both direct aliases and a special case of user-defined wrapper types (those that encapsulate a resource via constructor injection and a `close()` method). It then conducts a demand-driven resource escape analysis [10, §3.3] to determine whether the resource escapes its enclosing method through fields, return values, method parameters, or data structures. If any such escape is detected, RLFixer conservatively marks the leak as unrepairable; inserting a `close()` call at the warning site in such cases risks a "use-after-close" error, because external code might still use the resource afterward.

After deeming a leak fixable, RLFixer selects a repair template from a set of hard-coded options. Each involves inserting a `close()` call in a `finally` block, optionally wrapping existing resource usage in a structured control-flow construct such as a `try–catch`.

While RLFixer repairs some leaks, it struggles with many real-world code patterns. It cannot handle any leak caused by reassignment — overwriting a field without first closing the resource — since it cannot guarantee safe cleanup when the resource may escape the current scope. Its ability to reason about wrapper classes is limited to cases where the class explicitly defines a method named `close()`. In some cases, RLFixer's alias reasoning concludes a resource may escape when in fact it does not, thereby missing fix opportunities.

About 62% of leaks that RLFixer cannot repair in the NJR benchmarks [12] involve resources stored in fields. Arodnap's techniques address exactly this scenario: ownership inference moves the leak warning from the field assignment back to the wrapper allocation, and a new pre-close insertion repair, guided by a more precise escape analysis, lets RLFixer safely close resources even when the field is reassigned.



Inputs and Outputs:
1. Input Source Code, 2. Inferred Specifications, 3. RLC Warnings, 4. Transformed Code, 5. Updated Specifications, 6. Updated Warnings, 7. Patched Code, 8. Validated Patched Code

Fig. 3: Arodnap's leak-repair pipeline. Dark gray boxes indicate new components we introduced; light gray boxes are existing components we extended; white boxes are existing components we reused.

III. Example Repair

Figure 2 is a simplified but representative real-world resource leak from the NJR dataset [12] that RLFixer cannot repair without Arodnap. Class `TempFileWriter` allocates a `PrintStream` both in its constructor and in the `resetStream` method. RLC reports leaks at both these locations. The `PrintStream` objects are written into fields and thus might escape the current scope, so RLFixer deems both warnings unrepairable and emits no fix. Further, RLC inference (RLCI) cannot infer that `TempFileWriter` is a wrapper type, because the class does not implement a finalizer method like `close`.

Injecting Finalizer to Enable Ownership Inference. Arodnap injects a `close()` finalizer and the `AutoCloseable` interface on `TempFileWriter` (shown in blue in fig. 2b) to aid RLCI. These changes enable RLCI to add an `@Owning` annotation on the `stream` field, explicitly marking ownership. Because the class now owns the resource, RLC now reports a leak at the `TempFileWriter`'s allocation inside the `print` method instead.

The new warning inside `print` is a *shifted leak warning*. A shifted leak warning occurs when RLC reports the same defect at a different program location under a different analysis configuration (e.g., after adding ownership information). The underlying defect is unchanged; only the report location shifts. Here, the added `close` method does not fix a leak, but is needed for repair. The `PrintStream` created in the `TempFileWriter` constructor *cannot* be closed there, as it may be used later by the `printSomething` method. The resource cleanup *must* instead occur at the end of the enclosing `TempFileWriter`'s lifecycle.

Fix Generation. Given these transformations, RLFixer can generate repairs. As shown in green in fig. 2b, it encloses the allocation in `print` within `try–finally`. Further, our new *pre-close insertion* repair (section IV-C2) inserts a pre-close that closes `stream` before reassignment (line 9).

This example reflects a common pattern in real-world Java code: user-defined wrapper classes that manage resources internally. Two types of mistakes can lead to resource leaks. Resources may leak because the wrapper class has mistakes (like the missing `close()` method in this example, although mistakes in real code that Arodnap can fix are often much subtler!). Resources may also leak because the wrapper class is misused (as in the `print()` method in class `Client`). Our system makes both cases analyzable and repairable with minimal structural edits, so that automated tools can handle patterns that required manual annotation and refactoring before.

IV. Detection and Repair Architecture

The Arodnap repair pipeline (fig. 3) integrates static leak analysis (RLC), inference (RLCI), code transformation, and repair (RLFixer). It begins by combining RLCI and RLC to surface both inferred `@MustCall` obligations on user-defined classes and resource leak warnings. Leveraging these outputs, the pipeline applies a suite of lightweight *Code Transformations* — marking eligible fields `final`, converting fields to local variables, and injecting missing `close()` finalizers into wrapper classes (section IV-A). It then re-invokes inference and RLC to produce updated, more actionable leak warnings.

An *enhanced RLFixer* consumes the improved warnings and specifications from the transformed code to synthesize concrete code patches (section IV-B3). (The original RLFixer requires a developer to *manually* apply its fix hints.)

Using RLCI to infer ownership annotations leads to a new class of warnings (about possible overwrites of fields containing resources) that the original RLFixer did not need to consider. We enhanced both RLC itself and RLFixer to allow Arodnap to handle these warnings; as this problem is logically distinct, section IV-C describes the necessary modifications separately.

Finally, Arodnap validates the patches with both static and dynamic checks (section IV-D): it recompiles, re-runs RLCI+RLC to ensure the warning is eliminated, and runs the project's JUnit tests to confirm test outcomes are preserved.

A. Code Transformations

Static code transformations play a central role in our pipeline by reshaping code to improve analyzability and fixability. These transformations preserve program semantics while clarifying ownership structure and exposing resource lifecycles. Each transformation is designed to be sound and grounded in software engineering best practices.

IV-A1 Field Transformations for Immutability Mutable resource-holding fields introduce risks in resource management. If a resource field is reassigned without closing the previous value, the original resource may become unreachable, causing a leak. By marking some such fields as `final` and making other fields local, Arodnap eliminates the possibility of reassignment and makes lifecycle tracking simpler and safer. Section IV-C describes further enhancements to RLC and RLFixer for cases where these transformations do not apply.

Preventing Reassignment Arodnap adds the `final` modifier to a private field if the field is only assigned once. For instance fields, the assignment must occur at object construction time [17, §8.3.1.2]. This requires that all assignments to the

```
-private ServerSocket serverSocket;
+private final ServerSocket serverSocket;

 public MyClass(int port) {
+  ServerSocket tempSocket = null;
   try {
-    serverSocket = new ServerSocket(port);
+    tempSocket = new ServerSocket(port);
   } catch (IOException e) {
     e.printStackTrace();
+  } finally {
+    serverSocket = tempSocket;
   }
 }
```

Fig. 4: Using a temporary variable for final assignment

field occur at its declaration (via an initializer), or within constructors, or within one instance initializer block (exactly one of the possibilities), and only once along any path. This is closely related to the notion of *effectively final* [17, §4.12.4], though that is defined only for local variables. Static fields are similar. When the field assignment occurs inside `try–catch`, Arodnap introduces a temporary variable (fig. 4) to satisfy Java's final-field assignment rules. Use of `final` aligns with standard principles to prefer immutability and improves lifecycle tracking in tools like RLC and RLFixer.

Validation: This transformation is semantics-preserving: the Java compiler rejects subsequent writes to `final` fields. Arodnap applies this transformation even without an RLC warning.

Reducing Scope Arodnap converts a *private* resource field into a method-local variable when a simple syntactic check finds an unconditional assignment to the field that precedes any reads in that method (including inside `try–catch`), and there is no externally callable setter that assigns to the field. This shortens the variable's lifetime and improves precision in static ownership reasoning. This transformation follows the principle of minimal variable scope [18].

Validation: This transformation is sound except when the field is reflectively accessed. Because reflective accesses do occur in practice, Arodnap validates these changes using the static and dynamic protocol in §IV-D. Arodnap applies this transformation even when there are no RLC warnings.

IV-A2 Adding Finalizers to Wrapper Classes for Ownership Visibility Wrapper classes that hold resources in fields may erroneously fail to define a public cleanup method like `close()`, e.g., `TempFileWriter` in fig. 2a. The absence of a cleanup method prevents RLC inference from inferring the field is owning, as its rule for inferring ownership requires an attempt to dispose of the resource field [11]. Arodnap injects a `close()` method into each such class `C` and adds `implements AutoCloseable` (fig. 2b). RLFixer can later insert calls to the method to fix leak warnings on `C` objects.

This transformation is only applied when a resource is allocated within a constructor for class `C` and then assigned to an instance field within the same constructor. We found that in practice, such code strongly suggests that `C` owns the resource, despite `C` lacking a cleanup method. Arodnap discovers such

```
public class FileEventProxy {
  private Scanner scanner;
  public FileEventProxy(InputStream in) {
    this.scanner = new Scanner(in);
  }
  public boolean hasNextEvent() {
    return scanner.hasNextLine();
  }
  // other uses
}
```

Fig. 5: A resource accessor class uses, but does not own or close, a resource.

constructors by parsing leak warnings from RLC. With the transformation, RLC inference determines the field is `@Owning`, shifting responsibility for closing the resource to clients of `C`.

Validation: Since the `close()` method is new, no clients of `C` will have invoked it previously. Declaring `implements AutoCloseable` merely exposes a standard cleanup interface (enabling optional `try-with-resources`).

B. Enhancements to RLFixer

We extended RLFixer in two ways to improve its ability to repair leaks involving resource-holding fields (sections IV-B1 and IV-B2). We also added support for concrete patch materialization (section IV-B3).

IV-B1 Support for Inferred Finalizers RLC inference sometimes determines that an existing class method, which may have any name, is a finalizer and annotates the code accordingly. However, RLFixer only supports adding calls to methods named `close`. To address these cases, we extended RLFixer with the ability to insert calls to any finalizer method. The enhanced RLFixer parses inferred `@MustCall` annotations on user-defined classes to determine which methods should be treated as finalizers. If a class `C` is annotated with `@MustCall("shutdown")`, our extended RLFixer treats `shutdown` as the finalizer method for `C` objects. This change broadens RLFixer's resource model: any class with an inferred finalizer is now treated as a resource wrapper. This allows RLFixer to produce repairs for classes it would previously ignore.

IV-B2 Field Containment Analysis RLFixer avoids introducing use-after-close errors by checking if resources *escape* into fields or data structures before generating a fix [10]. However, RLFixer allows a resource to be written into a field of an object it determines to be a *resource alias*. An object is a possible resource alias if its class contains a resource field, assigns to that field in its constructor, and defines a cleanup method. Identified resource alias objects are then considered during RLFixer's escape analysis to ensure that they themselves do not escape into other fields.

We identified two issues with RLFixer's escape reasoning. First, RLFixer is *too conservative* in the presence of *resource accessor* objects that use a resource but do not take ownership of it. A resource accessor object takes a resource as a constructor argument and uses it during its lifetime, but does not close the resource. For example, in fig. 5, a `FileEventProxy` object uses the `InputStream` passed to its constructor but does

not close it. RLFixer does not treat resource accessors as aliases of the underlying resource, since they cannot be used to close the resource. So, RLFixer treats passing a resource into a resource accessor as a field escape, preventing repair. However, as long as a resource accessor does not outlive the underlying resource, it should not preclude a repair.

Second, RLFixer's checking is *unsound*, as it does not check whether objects may escape further via reads of resource alias fields. For example, consider a class `Wrapper` with a field `InputStream s` and a method `getStream()` that returns `s`. Even if `Wrapper` meets RLFixer's resource alias conditions outlined above, it may not be safe to close the resource passed to `Wrapper` in the original scope, since the resource may leak further via `getStream()`; RLFixer does not check this condition. We did not observe this unsoundness in the NJR dataset [12], but addressing this issue remains important for overall safety.

To handle these issues, we introduced sound support for resource accessors via *field containment analysis*, a lightweight extension to RLFixer's escape analysis. Field containment analysis checks that resources stored in a field do not further escape to some long-lived data structure. Field containment analysis is used both to identify resource accessors and to soundly check for resource aliases. Field containment analysis is again used when introducing fixes for non-final field overwrites; see Section IV-C2.

Definition IV-B2.1 (Field containment)**.** *An instance field f of a class C is* contained *iff for every C object c, there is no data flow from $c.f$ into any field, array, or data structure whose lifetime may exceed that of c.*

Our analysis conservatively checks for field containment. We require the field f is private, to limit the initial scope of analysis to C. Then, for each read of f within C, we check if the value read from f may flow into some field, array, or data structure, re-using the extant RLFixer escape analysis [10].

Given our field containment analysis, we enhanced RLFixer's identification of resource aliases as follows. During resource alias identification, the enhanced RLFixer also runs field containment analysis to ensure that further escapes from the field of the resource alias are not possible. Further, if a class meets all requirements for being a resource alias but lacks a finalizer method (which can be `close` or any method inferred via `@MustCall`), our enhanced RLFixer categorizes it as a resource accessor. We enhanced RLFixer's escape analysis to treat resource accessor objects identically to resource aliases, except that a resource accessor cannot be used to close a resource.

As an example, suppose a leaking resource is stored in a `FileEventProxy` object (fig. 5). With RLFixer's original logic, this would prevent a leak repair due to a field escape. With our enhancement, RLFixer can soundly show that the object is a resource accessor, as field containment analysis on the field `scanner` proves the resource does not leak further via the field. Then, as long as the `FileEventProxy` object does not escape, the repair can be applied, as seen in fig. 6.

An alternative to introducing resource accessors would have been to insert a `close` method into `FileEventProxy`

```
  InputStream s = new FileInputStream("file.txt");
  FileEventProxy proxy = new FileEventProxy(s);
+ try {
+   // use proxy
+ } finally {
+   s.close();
+ }
```

Fig. 6: Fix for a client of fig. 5, enabled by handling of resource accessors.

(section IV-A2), making it a resource alias. However, such a change is unnecessary to repair the leak: the resource is passed into the constructor, so the client can close it (contrast with fig. 2b where the resource is allocated in the constructor). Adding support for resource accessors leads to less intrusive changes, avoiding insertion of unnecessary `close` methods into types that do not require them.

IV-B3 Patch Materialization While RLFixer produces structured, parameterized repair hints, they are purely textual, e.g.,

> Add following code below line `22` (WriterFile.java):
> `finally{ try{ <NEW_VARIABLE>.close(); } }`
> // where variable `<NEW_VARIABLE>` points to the resource from line `17`.

Arodnap extends RLFixer with a deterministic *patch materializer* that takes (i) the RLC warning, (ii) the corresponding RLFixer hint, and (iii) the relevant source, and then (a) parses the compilation unit into an AST, (b) locates the edit sites indicated by the hint, and (c) applies template-guided AST rewrites to realize the repair (e.g., wrapping with `try–finally` or try-with-resources when legal, or inserting a `close()` call). The modified AST is pretty-printed and diffed against the original file to produce a concrete patch.

C. Field Overwrite Handling

RLC issues an *owning field overwrite* warning when a resource field is reassigned without first being closed. The assignment potentially causes a leak by making the original resource unreachable. Arodnap addresses this issue in two ways: by eliminating false positives in safe constructor-based assignments, and by safely inserting closure logic before actual reassignments. The original RLFixer — which runs RLC without any annotations — did not need to consider this case. RLC only issues these warnings if at least one field is annotated as `@Owning` (in Arodnap, by inference).

IV-C1 Filtering False Positives on Constructor Assignments Previously, RLC would report an overwrite warning for any write to a non-final resource field in a constructor, even when it was the first write to the field. We improved RLC to not issue a warning if 6 conditions apply. The field (1) is private, (2) has no initializer at its declaration, and (3) is not written in any instance initializer block. (4) The assignment occurs directly in the constructor body. The constructor (5) writes the field exactly once and (6) neither delegates via `this(...)` nor performs any method calls before the assignment. These constraints conservatively guarantee, based on Java's initialization order, that the assignment is the first write to the field, not an overwrite.

```
+  if (socket != null) {
+    try {
+      socket.close();
+    } catch (IOException e) {
+      e.printStackTrace();
+    }
+  }
   socket = new Socket();
```

Fig. 7: Example repair for field overwrite.

The first write to a field cannot cause a leak since the field has no previous value.

IV-C2 Safe Reassignment Fixing via Pre-Close Insertion For cases where a resource field f is legitimately reassigned outside a constructor, we devised a new repair that inserts a conditional close of the field's current value just before the reassignment, thereby preventing a leak. Care must be taken to ensure that this repair does not introduce a use-after-close error due to some other outstanding pointer alias for the resource. Arodnap only applies this transformation when the following conditions hold:

1) f is a `private` field of the enclosing class C.
2) All writes to f assign it a newly-allocated resource (e.g., `new Socket(...)`).
3) Field containment (Def. IV-B2.1) holds for f, i.e., the resource never escapes the class from f.

Condition 1 limits the scope of analysis, while conditions 2 and 3 ensure there cannot be other references to the resource that outlive the method re-assigning the field. When these conditions are met, enhanced RLFixer safely inserts the repair shown in fig. 7. This fix closes the previously held resource before overwriting it, preventing a leak. It is possible that the resource was already closed, and hence the inserted call is a duplicate. But, this does not cause problems in practice, because Java close methods are typically specified to be idempotent and hence safe to repeat (see, e.g., the `java.io.Closeable#close` documentation [19]). This repair template simply prints the stack trace of any exception thrown by `close()`, but the behavior in the catch block could easily be customized (e.g., to re-throw the exception or perform logging).

D. Patch Validation: Static and Dynamic

Arodnap validates each materialized patch with two gates:

IV-D1 Static Validation. The project must recompile cleanly, and a re-run of RLCI+RLC on the patched code must confirm that the originally reported leak at the patched site is eliminated.

IV-D2 Dynamic Validation. Arodnap executes the project's JUnit test suite and checks that no previously passing test fails post-patch (i.e., no new regressions relative to the pre-patch baseline).

V. Implementation

Arodnap builds on the Checker Framework's Resource Leak Checker (RLC) [5]. The Checker Framework distribution includes RLCI [11]. We wrote Error Prone [20] plugins for field transformations: converting fields to be final and converting resource fields to locals. Arodnap injects wrapper-finalizers using JavaParser [21]. For static analysis, field containment and escape analyses are run on WALA's SSA intermediate representation [22]. We also fixed RLFixer's source-to-IR matching logic to correctly map warnings within nested and anonymous classes, enabling more repairs. The full pipeline targets Java 11.

VI. Experimental Setup

Our evaluation of Arodnap in detecting and repairing resource leaks aimed to answer the following research questions:

RQ1 How effective is Arodnap in reducing and repairing leak warnings compared to RLC+RLFixer?

RQ2 How effective is Arodnap compared to RLCI+RLC+RLFixer (adding inference [11] to the RLC+RLFixer combination, without our other improvements)?

RQ3 How much do the different components of Arodnap contribute to its effectiveness?

RQ4 For leaks that cannot be repaired by Arodnap, what is the root cause?

A. Dataset

The evaluation uses 285 of the 293 Java 8 projects in the NJR-1 dataset [12], with each project averaging 6,028 non-blank, non-comment lines of Java code. 8 projects are excluded due to timeouts during RLC inference. This same benchmark was used to evaluate RLFixer [10], though they did not exclude any projects because they did not run inference. The projects cover a wide range of domains.

The NJR-1 benchmark is an unlabeled snapshot of open-source Java projects; it does not contain ground-truth defects annotations. Our evaluation depends on resource-leak warnings reported by RLC, which is a sound tool. Many files have no resource leaks; our evaluation shows that Arodnap not only repairs leaks but does not break files that have no leaks.

B. Configurations

We evaluated three configurations to answer RQ1 and RQ2:

1) **RLC+RLFixer**: RLC without resource specification inference and the original RLFixer without our enhancements (for RQ1).
2) **RLCI+RLC+RLFixer**: RLC with specification inference but unmodified RLFixer (for RQ2).
3) **Arodnap**: As in fig. 3.

All configurations are evaluated after patch materialization (section IV-B3) and validation (section IV-D), for fairness.

C. Weighted Fix Count

RLC inference and Arodnap's code transformations often cause leak warnings to shift from uses of library resources to uses of corresponding wrapper types, as discussed in section II-B (see the discussion of fig. 1). While this shifting makes the leak warnings more actionable and useful, from an experimental point of view it makes comparisons across our configurations difficult, since a single warning on a library

Configuration	Leak warnings			Fixed warnings		Repair rate
	CL	XE	XR	F_{CL}	F_{XE}	
RLC+RLFixer	1909			783		41%
	1909	0	0	783	0	
RLCI+RLC+RLFixer	2213			760		50%
	1537	320	356	755	5	
Arodnap	2136			1014		68%
	1446	243	447	952	62	

Fig. 8: Leak resolution breakdown across configurations. "Fixed warnings" values are *weighted fix counts* (section VI-C).

resource in one configuration could be shifted to multiple warnings about wrapper type objects in another. Consistent with Shadab et al. [11], we see an *increase* in the number of total leaks when inference is enabled (see fig. 8).

To conservatively account for these differences, we define a *weighted fix count*. For each leak warning (see section III) on a library resource, we determine how many of its shifted warnings were successfully repaired. If k out of n shifted sites are fixed, we assign a weighted fix score of k/n to that leak. *Non-shifted* leak warnings (i.e., leaks reported on library resources) are assigned a score of 1 if fixed or 0 if not. The *weighted fix count* ensures that each root library leak contributes in proportion to the fraction of its associated warnings that are fixed, regardless of shifting due to inference and transformations.

In the general case, mapping a shifted leak warning back to a library leak warning can be quite challenging and require inter-procedural data flow analysis. And for leak warnings on non-final field overwrites (section IV-C), there may be multiple library resources possibly leaked by the overwrite. In our experiments, we used a combination of automatic and manual analysis to compute the shifted leak warning mapping, and we separately categorized non-final field overwrites to avoid the complications of mapping those warnings.

VII. Evaluation

A. Results

To compare different configurations (section VI-B), we partition leak warnings into three categories. Let W_{orig} be the warnings produced by RLC alone on the original code. Let W_{xform} be the warnings produced by RLC immediately before the enhanced RLFixer is run (at point 6 in fig. 3), after mapping any shifted leak warnings (section III) back to their corresponding library leak warning. Line-number changes do not affect whether two warnings are considered the same.

- **Core Leaks (CL)** are $W_{orig} \cap W_{xform}$. These are warnings produced by RLC directly indicating a library resource or wrapper object is leaking.
- **Transformation-Exposed (XE) warnings** are $W_{xform} \setminus W_{orig}$. These new warnings appear as a result of inference and transformation. The dominant warning type in this category is overwrites of non-final `@Owning` fields (section IV-C), where `@Owning` was added by RLCI inference.
- **Transformation-Resolved (XR) warnings** are $W_{orig} \setminus W_{xform}$. These warnings do not need repair: they were false positives that were fixed by semantics-preserving transformations or by adding resource specifications. To be treated as resolved, a warning must no longer appear *and* no warnings on wrappers can be mapped to it (section VI-C). It is possible that the resource from a resolved warning could reappear as a non-final field overwrite warning, but we manually inspected a sample of 50 **XR** warnings and never observed this to occur.

The warning universe is:

$$T = CL + XE + XR$$

and the *resolution rate* is

$$R = \frac{F_{CL} + F_{XE} + XR}{T}$$

where F_{CL} and F_{XE} denote the leaks actually fixed in each category.

Figure 8 presents our main results. For RLC+RLFixer, Utture et al. [10] reported a 51% average fixable rate for RLC warnings. However, this calculation excluded RLC warnings that RLFixer could not map to a WALA IR instruction, preventing repair. Counting all reported leaks, we found the actual fix rate for RLFixer for RLC warnings was 41%.

For **RLCI+RLC+RLFixer**, which just adds RLC inference to RLC+RLFixer, 320 new warnings are reported due to inference (the **XE** category), nearly all due to overwrites of non-final fields marked as `@Owning` by inference. At the same time, inference leads to 357 of the original warnings being resolved, raising the resolution rate from 41% to 50%. The resolution was significantly due to inference discovering 352 wrapper types, with 498 `@Owning` fields total.

In **Arodnap** we observe a significantly larger 1014 leaks repaired (952 core and 62 inference exposed) and 447 warnings resolved, pushing the resolution rate to 68%. Arodnap discovers 443 wrapper types and 627 `@Owning` fields. This is a significant increase over RLCI+RLC+RLFixer, due to our injection of `close` methods. At the same time, the raw **XE** count *drops* from 320 in RLCI+RLC+RLFixer to 243, despite the increase in the number of `@Owning` fields. This decrease is due to the false-positive filtering of section IV-C1, which significantly reduces the number of **XE** warnings. Overall, by exposing wrapper types and significantly enhancing repair capabilities related to wrappers, Arodnap resolves over two-thirds of all reported leak warnings, significantly improving on other configurations.

Note that the weighted fix count metric (section VI-C) used for fig. 8 is intentionally conservative: that is, it *understates* Arodnap's effectiveness at repairing leaks of wrappers. In particular, consider a case where a warning about single library resource in a wrapper class is mapped to ten uses of that wrapper class. In this case, if Arodnap fixed 5 of the 10 warnings, it would only get credit for fixing $0.5 = 5/10$ of a warning. To illustrate Arodnap's effectiveness for wrappers more directly, in RLCI+RLC+RLFixer, RLFixer could repair only 28 of 543 core leaks on wrappers (5%); Arodnap

Configuration	Leak warnings			Fixed warnings		Repair rate
	CL	XE	XR	F_{CL}	F_{XE}	
Arodnap	2136			1014		68%
	1446	243	447	952	62	
– Code Transformations	2151			1005		64%
	1537	253	361	951	54	
– RLFixer Enhancements	2148			799		57%
	1477	253	418	755	44	
– Field Overwrite Handling	2216			972		63%
	1478	324	414	967	5	

Fig. 9: Ablation study: impact of disabling components of Arodnap. "Fixed warnings" values are *weighted fix counts* (section VI-C).

repairs 412 of 838 warnings (49%), an order-of-magnitude improvement in repair effectiveness.

B. Ablation Study

To measure the contribution of individual components of Arodnap (RQ3), we conduct an ablation study using Arodnap as a base and selectively disabling key modules:

1) **Code Transformations** (section IV-A),
2) **RLFixer Enhancements** (section IV-B),
3) **Field Overwrite Handling** (section IV-C).

The results are shown in Figure 9. The numbers are non-trivial to interpret, as disabling certain features may change the total number and location of warnings reported (see section VI-C), which has downstream impacts on repair effectiveness. We explain the results for each configuration below.

When code transformations were disabled, we saw the same number of wrapper types discovered as in RLCI+RLC+RLFixer, as finalizer methods are not inserted. This led to 81 fewer resolved warnings on library resources (they instead appear as core leaks), decreasing the resolution rate to 64%. The raw number of core wrapper leak warnings decreases from 838 in Arodnap to 543 with code transformations disabled, with only 152 of them being repaired instead of 412. The magnitude of this improvement is understated in Figure 9 due to use of weighted fix counts (previously discussed in section VII-A).

When disabling RLFixer enhancements, repair effectiveness is significantly decreased (from 1025 fixed leaks to 814), reducing the resolution rate to 57%. Finally, with field overwrite handling disabled, we see an increase of 81 in **XE** leaks, due to the lack of filtering of false positives from constructors. And, there is a decrease in repaired **XE** leaks (from 62 to 5), due to absence of our new repair pattern for field overwrites, leading to an overall reduction of the resolution rate to 63%.

Overall, we see that all components of Arodnap contribute significantly to its effectiveness.

C. Run time

We run our experiments on an Ubuntu 20.04.6 LTS cloud VM with 16 vCPUs and 60 GB RAM. Across our evaluation set, the end-to-end Arodnap pipeline averages **582** seconds per project (fig. 10). On average, the second RLCI+RLC pass takes less time than the initial/final RLCI+RLC run because we execute this second pass only on projects where a code transformation was applied.

Stage	Mean time (s)
RLCI+RLC (first pass)	211
Code transformations	15
RLCI+RLC (second pass)	78
Enhanced RLFixer	52
Patch validation	226
Total (per project)	**582**

Fig. 10: Breakdown of per-project average run time.

D. Patch Validation

As discussed in section IV-B3, Arodnap automatically generates code patches for each repair and validates them using the protocol in section IV-D. In contrast, prior work [10] validated only a subset of RLFixer-generated hints. In 59 cases with especially complex code structures (49 deeply nested control flow; 10 patch conflicts), automatic materialization did not yield a compilable patch; for those, we applied the RLFixer hints manually, following the hint template. After automatic materialization, we ran static and dynamic validation for all patches generated by Arodnap with all functionality enabled, and found that over 99% of the patches were validated.

Static validation. For Arodnap, the generated patches eliminate the reported leak in **99.1%** of cases ($1016/1025$). Patch materialization failed in $9/1025$ attempts, chiefly due to finalizer visibility (private `close()` methods) and cases where the repair template structure was insufficient.

Dynamic validation. Across 285 projects, 11,929 previously passing JUnit tests were re-run; 7 newly failed post-repair (an $\approx 0.06\%$ failure rate). Five failures were due to the field-to-local conversion, as those fields were accessed elsewhere via reflection. The other two cases stemmed from control-flow-dependent resource acquisition and finalizer injection closing resources in the wrong order. As a caveat, the overall code coverage of these tests is low (12.9% statement coverage), and hence only 11% of Arodnap's patches (107/1,014) were executed by the tests.

In short, we successfully validated nearly all repairs generated by Arodnap; static / dynamic validation and code review should be performed before such repairs are merged.

E. Example Fixed Leaks

As in the motivating example (fig. 2), Arodnap's transformations make ownership explicit and can *shift* warnings to more actionable sites (section VI-C). We highlight two additional patterns taken from the NJR dataset that prior logic did not repair but Arodnap now fixes.

VII-E1 Safe Pre-close Before Field Overwrite Reassigning a resource field can leak the prior value. When the field is private, every write stores a freshly allocated resource, and no aliases can escape, Arodnap inserts a conditional cleanup *before* the write (fig. 11).

Why this was hard before. Without containment and ownership

```
 class AbstractParserTables {
   private Writer f = null;

   String toSourceFile(String fileName) {
     File file = new File(fileName);
+    if (f != null) {
+        f.close();
+    }
     f = new BufferedWriter(new FileWriter(file));
     // ...
   }
 }
```

Fig. 11: Inserting a pre-close before overwriting a resource field. Adapted from benchmark `url270fc4f5ee_ykcilborw_Joust_tgz`, file `AbstractParserTables.java`.

```
 class Task extends TimerTask {
   private final Puppeteer m_Puppeteer;
   public Task(Puppeteer puppeteer) { m_Puppeteer =
     puppeteer; }
   public void run() { // uses m_Puppeteer};
 }

 class ActorsTest {
   void startPuppeteer() {
-    Puppeteer puppeteer = new Puppeteer("localhost");
-    (new java.util.Timer("Puppeteer")).schedule(new
     Task(puppeteer), 0, 1000);
+    Puppeteer puppeteer = null;
+    try {
+      puppeteer = new Puppeteer("localhost");
+      (new java.util.Timer("Puppeteer")).schedule(new
      Task(puppeteer), 0, 1000);
+    } finally {
+      if (puppeteer != null) puppeteer.finish();
+    }
   }
 }
```

Fig. 12: Client-side try/finally for a resource accessor that does not own the resource. Adapted from benchmark `urlc98c3b97d2_Trimax_venta_tgz`, file `ActorsTest.java`.

information, a pre-close risks use-after-close via surviving aliases, so prior repair avoids inserting it.

VII-E2 Containment Proves an Accessor, Enabling Client–Side Repair As shown in fig. 12, the user-defined class `Task` caches an incoming resource in a private field, never lets it escape, and exposes no finalizer. Treating such a class as an *owner* blocks repair; with containment, we classify it as an *accessor* and fix at the client `startPuppeteer`.

Why this was hard before. Without containment, the wrapper is conservatively treated as a potential owner or rejected for lacking a finalizer, so no safe fix is emitted. Containment shows the field does not escape; the client-side `try/finally` is then sound under our ownership model.

F. Remaining Unfixed Leaks: Case Study

To address RQ4 and better understand Arodnap's limitations, we manually inspected 100 randomly chosen locations of resource leak warnings that remained unfixed by Arodnap.

63% of the unrepaired cases would require more global analysis and/or transformation to repair. In 49 cases, we found that the resource truly escaped the local scope of the warning to some longer-lived object or data structure. Repairing such cases could require significant changes across the codebase and advances in verification reasoning to prove safety. In 14 other cases, there was an overwrite of an owning field where the checks of section IV-C could not prove it was safe to close the field before the overwrite. In most of these cases, the field was not private, or a resource stored in the field was passed in from outside the enclosing class, so more global analysis would be required to prove safety of the repair.

The remaining 37% of unrepaired cases could be addressed with further engineering improvements to RLFixer that are orthogonal to our contribution. In 28 cases, the repair could not be performed due to the need for more complex repair templates in RLFixer, e.g., to handle resources allocated in loops or exceptions thrown from a constructor after a field assignment. Finally, the remaining 9 cases could not be repaired due to remaining limitations in RLFixer's logic to match leak warnings to WALA IR instructions.

VIII. Limitations and Threats to Validity

VIII-1 Limitations Currently, Arodnap cannot repair leak warnings related to creation of fresh obligations on wrappers with non-final `@Owning` fields. (RLC expresses this with a `@CreatesMustCallFor` annotation [5].) An overwrite of such a field could "reset" the obligation on a wrapper after its finalizer method has already been called, necessitating another finalizer call. RLC is currently very imprecise in reasoning about such cases, and hence the reports are not amenable to repair; Shadab et al.'s work on inference also ignored such warnings [11].

Arodnap's transformations are sound and conservative, and therefore they miss some opportunities. Invoking the injected finalizer can interact with framework lifecycles; we therefore inject it only when there is a relevant RLC warning. For pre-close on reassigned fields, we apply the edit only when the field is *private*, every write stores a freshly allocated resource, and field containment (section VII-E2) holds; otherwise we skip to avoid alias-related use-after-close risks.

VIII-2 Threats to Validity Regarding external validity, our evaluation is conducted on 285 Java 8 benchmarks from the NJR-1 dataset, which was used in prior resource leak repair research [10]. While the dataset is diverse, our results may not fully generalize to programs targeting more modern Java versions, Android applications, or other programming ecosystems with different resource management idioms.

Regarding internal validity, to compare leak warnings across configurations, we use an automated mapping process that matches leaks on wrapper types to the corresponding leaking library resources (section VI-C). When the automation failed to establish a match, we manually completed the mapping. There may still be minor inconsistencies in our mapping, due to imperfect alias resolution or complex control flow.

Beyond the correctness of Arodnap's implementation, our results rely on the correctness of RLC, RLCI, RLFixer, and supporting tools like JavaParser and WALA. Limitations or bugs in any of these components may affect the accuracy of detection or the applicability of repairs. We have validated all

of our generated repairs against RLC (section VII-D), a strong consistency check for the full Arodnap pipeline.

IX. Related Work

The research most related to our work spans *static analysis for defect detection*, *specification inference*, and *automated program repair (APR)*. We discuss these areas in turn.

Static Analysis for Detection. Static analysis techniques have long been employed to detect resource management errors. RLFixer [10] was specifically tested with the Infer [3], SpotBugs [4], CodeGuru [23], PMD [24], and RLC [5] leak detectors. These techniques were built based on earlier research on static leak detection for Java like that of Torlak and Chandra [25]. Other languages have their own leak detection tools, e.g., the Clang Static Analyzer [26] for C/C++. Our work focuses on repair of leaks reported by RLC, as its corresponding inference technique [11] uniquely and automatically exposes key information needed to repair leaks on wrapper types.

Specification Inference. The inference technique for RLC [11] is built on the whole-program inference framework of Kellogg et al. [27]. Other recent approaches to annotation inference have been based on black-box search [28], information retrieval [29], and machine-learning [30], [31], [32]. Arodnap uses the extant RLC inference without modification, and improved inference techniques could be easily incorporated. Recent work on mining API-level resource patterns — e.g. MiROK's large-scale extraction of acquisition-release pairs [33], MAPO's protocol mining [34] — provides a complementary source of cleanup specifications that could be used to extend the applicability of Arodnap in the future.

Automated Repair Techniques. Automated repair has been studied across multiple paradigms, and we categorize the most relevant work into static template-based repair and machine learning-driven repair.

Static Analysis Template-Based Repair. Template-driven *static* APR has evolved along two complementary lines. First, *general-purpose* systems such as GenProg [7], SemFix [35], and CoCoNuT [36] explore very large patch spaces via mutation or semantic search guided by test outcomes. Because resource leaks seldom manifest as failing tests, these techniques are largely ineffective in our setting [10]. Van Tonder and Le Goues showed that separation-logic proofs can facilitate synthesis of verified patches for heap anomalies, including leaks, in C programs [8]. Second, a broad ecosystem of *domain-specific static APR tools* shows that carefully crafted templates plus static reasoning can repair defects even when no failing tests exist. MemFix formulates C memory-deallocation faults (leaks, double-frees, use-after-frees) as an exact-cover problem over allocation-free pairs and solves it with a SAT solver [37]. ARC applies a genetic search that mutates Java synchronization constructs to eliminate deadlocks and data races, then prunes excess locks for performance [38]. NPEFix dynamically guards or substitutes risky dereferences to avert null-pointer crashes in Java [39]. Our tool Arodnap is a domain-specific approach targeted at extending repair of resource leaks to wrapper types.

LLM-Based Repair Techniques. Recent work couples static analysis with large-language-model (LLM) patch generation. InferFix augments Infer warnings with retrieval-based context before feeding them to a fine-tuned LLM, achieving strong results on test-oriented benchmarks [40], while FixrLeak deploys a prompt-engineering workflow at Uber that turns static leak warnings into try-with-resources rewrites for Java services [1]. Neither of these approaches is applicable to the wrapper type leaks targeted by Arodnap. Transformer-based systems such as TFix [41], CURE [42], and ReCoder [43] learn edit patterns from historical commits. These models excel at syntactic and localized edits but often lack deep inter-procedural reasoning, limiting their potential effectiveness for wrapper type leaks. Our approach applies code transformations, inference, and targeted static analysis to discover and repair leaks involving wrapper types across a large program scope.

Android Resource Leaks. For Android, datasets such as DroidLeaks curate real resource-leak defects and are widely used for evaluation [44]. Analyses like PlumbDroid detect and automatically repair Android resource leaks by reasoning over event-driven control flow and lifecycle callbacks [45]. While not leak-specific, FixDroid is an Android Studio assistant that flags security/privacy pitfalls and offers quick fixes [46]. Arodnap, by contrast, targets general Java projects and wrapper-based ownership patterns. Its analyses and transformations operate at the language level and are agnostic to Android-specific lifecycles, making these lines of work complementary rather than overlapping.

X. Conclusion

Arodnap is a technique and tool that extends resource leak repair for Java to apply to resource wrappers. Java programs often store resources in fields of wrappers, and repairing leaks of such resources requires techniques targeted at the wrapper types and their fields. Arodnap demonstrates that new static analysis techniques, not just better integration, are needed to close the gap on resource leak repair for wrappers. Through a combination of code transformations, new repair templates, and enhanced reasoning about fields during both leak detection and repair, Arodnap achieved a leak resolution rate of 68%, improving over the 41% rate of the prior state of the art.

Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590132, the National Science Foundation under grants CCF-2223826, CCF-2312262, CCF-2312263, and CNS-2120070, a gift from Oracle Labs, and a Google Research Award.

References

[1] Z. Zhang, A. Utture, M. Sridharan, and J. Palsberg, "FixrLeak: GenAI-based resource leak fix for real-world Java programs," in *Machine Learning for Systems Workshop at 38th NeurIPS*, 2024.

[2] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *ASE 2016: Proceedings of the 31st Annual International Conference on Automated Software Engineering*, Singapore, Singapore, Sep. 2016, pp. 332–343.

[3] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NFM 2015: 7th NASA Formal Methods Symposium*, Pasadena, CA, USA, Apr. 2015, pp. 3–11.

[4] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA Companion: Companion to Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, BC, Canada, Oct. 2004, pp. 132–136.

[5] M. Kellogg, N. Shadab, M. Sridharan, and M. 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, Aug. 2021, pp. 181–192.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE 2013, Proceedings of the 35th International Conference on Software Engineering*, San Francisco, CA, USA, May 2013, pp. 672–681.

[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE TSE*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[8] R. van Tonder and C. Le Goues, "Static automated program repair for heap properties," in *ICSE 2018, Proceedings of the 40th International Conference on Software Engineering*, Gothenburg, Sweden, May 2018, pp. 151–162.

[9] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak, "Memory and resource leak defects and their repairs in Java projects," *Empirical Softw. Engg.*, vol. 25, no. 1, pp. 678–718, Jan. 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09731-8

[10] A. Utture and J. Palsberg, "From leaks to fixes: Automated repairs for resource leak warnings," in *OOPSLA 2023, Object-Oriented Programming Systems, Languages, and Applications*, Cascais, Portugal, Oct. 2023, pp. 159–171.

[11] N. Shadab, P. Gharat, S. Tiwari, M. D. Ernst, M. Kellogg, S. Lahiri, A. Lal, and M. Sridharan, "Inference of resource management specifications," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, article #282, pp. 1705–1728, Oct. 2023.

[12] J. Palsberg and C. V. Lopes, "NJR: A normalized Java resource," in *Proceedings of the 2018 ACM SIGPLAN International Conference on Software Engineering Companion (ISSTA Companion/ECOOP)*, 2018, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3236454.3236501

[13] "Arodnap: Repairing leaks in resource wrappers," https://doi.org/10.5281/zenodo.15542576.

[14] "Arodnap," https://github.com/typetools/arodnap.

[15] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 2008, pp. 201–212.

[16] M. Kellogg, N. Shadab, M. Sridharan, and M. D. Ernst, "Accumulation analysis," in *ECOOP 2022 — Object-Oriented Programming, 33rd European Conference*, Berlin, Germany, June 2022, pp. 10:1–10:31.

[17] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification*, Java SE 8 ed. Boston, MA: Addison Wesley, 2014.

[18] J. Bloch, *Effective Java Programming Language Guide*. Boston, MA: Addison Wesley, 2001.

[19] "Javadoc for close method of java.io.Closeable," https://docs.oracle.com/javase/8/docs/api/java/io/Closeable.html#close--, accessed: 2025-05-26.

[20] "Error prone," https://errorprone.info, 2012–2025, google compiler plug-in that flags common Java mistakes at compile time.

[21] JavaParser Project, "JavaParser," 2019, accessed: 2025-05-27. [Online]. Available: https://javaparser.org

[22] IBM, "T.J. Watson Libraries for Analysis (WALA)," 2006, accessed: 2025-05-27. [Online]. Available: http://wala.sourceforge.net

[23] "Amazon CodeGuru Security," 2025, accessed: 2025-05-28. [Online]. Available: https://aws.amazon.com/codeguru/

[24] The PMD Development Team, "PMD: A multilanguage static code analyzer," https://pmd.github.io/, 2024, accessed: 2025-05-21.

[25] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," in *ICSE 2010, Proceedings of the 32nd International Conference on Software Engineering*, Cape Town, South Africa, May 2010, pp. 535–544.

[26] "Clang static analyzer," https://clang-analyzer.llvm.org/, accessed: 2024-05-18.

[27] M. Kellogg, D. Daskiewicz, L. N. D. Nguyen, M. Ahmed, and M. D. Ernst, "Pluggable type inference for free," in *ASE 2023: Proceedings of the 38th Annual International Conference on Automated Software Engineering*, Luxembourg, Sep. 2023, pp. 1542–1554.

[28] N. Karimipour, J. Pham, L. Clapp, and M. Sridharan, "Practical inference of nullability types," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds., 2023, pp. 1395–1406. [Online]. Available: https://doi.org/10.1145/3611643.3616326

[29] J. Wu and C. Lemieux, "QuAC: Quick attribute-centric type inference for Python," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 2040–2069, 2024. [Online]. Available: https://doi.org/10.1145/3689783

[30] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *ESEC/FSE 2018: The ACM 26th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lake Buena Vista, FL, USA, Nov. 2018, p. 152–162.

[31] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, and M. Lyu, "Static inference meets deep learning: a hybrid type inference approach for Python," in *ICSE 2022, Proceedings of the 43rd International Conference on Software Engineering*, Pittsburgh, PA, USA, May 2022, pp. 2019–2030.

[32] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "TypeWriter: neural type prediction with search-based validation," in *ESEC/FSE 2020: The ACM 28th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Sacramento, CA, USA, Nov. 2020, pp. 209–220.

[33] C. Wang, Y. Lou, X. Peng, J. Liu, and B. Zou, "Mining resource-operation knowledge to support resource leak detection," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, 2023, pp. 986–998. [Online]. Available: https://doi.org/10.1145/3611643.3616315

[34] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending api usage patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa, 2009, pp. 318–343. [Online]. Available: https://doi.org/10.1007/978-3-642-03013-0_15

[35] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.

[36] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, 2020, pp. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[37] J. Lee, S. Hong, and H. Oh, "MemFix: static analysis-based repair of memory deallocation errors for C," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, 2018, pp. 95–106. [Online]. Available: https://doi.org/10.1145/3236024.3236079

[38] D. Kelk, K. Jalbert, and J. S. Bradbury, "Automatically repairing concurrency bugs with ARC," in *Proceedings of the International Conference on Multicore Software Engineering, Performance, and Tools - Volume 8063*, ser. MUSEPAT 2013, 2013, pp. 73–84. [Online]. Available: https://doi.org/10.1007/978-3-642-39955-8_7

[39] J. Lee, S. Hong, and H. Oh, "NPEX: repairing Java null pointer exceptions without tests," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, pp. 1532–1544. [Online]. Available: https://doi.org/10.1145/3510003.3510186

[40] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "InferFix: End-to-end program repair with LLMs," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, 2023, pp. 1646–1656. [Online]. Available: https://doi.org/10.1145/3611643.3613892

[41] B. Berabi, J. He, V. Raychev, and M. Vechev, "TFix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol.

139. PMLR, 18–24 Jul 2021, pp. 780–791. [Online]. Available: https://proceedings.mlr.press/v139/berabi21a.html

[42] W. Zhong, C. Li, J. Ge, and B. Luo, "Neural program repair: Systems, challenges and solutions," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, ser. Internetware '22, 2022, pp. 96–106. [Online]. Available: https://doi.org/10.1145/3545258.3545268

[43] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, 2021, pp. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[44] Y. Liu, J. Wang, L. Wei, C. Xu, S.-C. Cheung, T. Wu, J. Yan, and J. Zhang, "DroidLeaks: a comprehensive database of resource leaks in Android apps," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3435–3483, 2019.

[45] B. N. Bhatt and C. A. Furia, "Automated repair of resource leaks in Android applications," *J. Syst. Softw.*, vol. 192, no. C, Oct. 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111417

[46] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A stitch in time: Supporting Android developers in writing secure code," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1065–1077. [Online]. Available: https://doi.org/10.1145/3133956.3133977