From Leaks to Fixes: Automated Repairs for Resource Leak Warnings

Akshay Utture
University of California, Los Angeles
USA
akshayutture@ucla.edu

Jens Palsberg
University of California, Los Angeles
USA
palsberg@ucla.edu

ABSTRACT

Resource leaks are a common and elusive source of bugs that can result in crashes and security vulnerabilities. The most effective technique to identify such leaks during development is static analysis. However, empirical studies show that in addition to leak warnings, developers often need help in the form of automated fix suggestions to correctly repair such leaks. The only existing tool that can suggest resource-leak fixes is the general-purpose tool Footpatch. Footpatch, however, performs poorly at this task; it generates fixes for only 6% of the leaks, out of which only 27% are correct.

In this paper, we introduce RLFixer, a specialized repair tool that generates high-quality fixes for resource leaks identified by any resource-leak detector. A major challenge for RLFixer is that the most general version of the resource-leak repair problem is at least as hard as compile-time object deallocation, a well-known hard problem for compilers. RLFixer tackles this issue by separating the resource-leaks that are infeasible for a compile-time tool to fix from those that are feasible to fix. RLFixer achieves this separation by using a new data-flow analysis of resource objects to classify how they escape the context of their methods. The same analysis also enables RLFixer to generate correct repairs for the feasible-to-fix leaks. RLFixer is demand-driven and hence only analyzes statements relevant to the leak, thereby keeping overhead low.

We evaluated RLFixer by applying it to warnings generated by five popular Java resource-leak detectors. We show that, on average, RLFixer generates repairs for 66% of their warnings, out of which 95% are correct. It has an average repair time of 14 seconds.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Software maintenance tools**.

KEYWORDS

Static Analysis, Resource Leaks, Automated Repair

ACM Reference Format:
Akshay Utture and Jens Palsberg. 2023. From Leaks to Fixes: Automated Repairs for Resource Leak Warnings. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616267

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616267

1 INTRODUCTION

Motivation. Most programs use resources such as files, sockets and database connections. Resource leaks are a common bug introduced unintentionally by programmers, which can result in security vulnerabilities [6] and severe failures [24]. Resource leaks are elusive because they only cause crashes when many resources leak and the OS runs out of that resource-type; this typically does not happen during testing. An effective approach for identifying these resource leaks during development is static analysis [32]. Today, developers can choose from several open-source static-analysis tools that perform resource-leak detection [1, 3, 5, 14, 32], many of which provide accurate warnings.

While static analyzers can detect resource-leaks, users also need tool-support to fix these errors. For example, Christakis and Bird's empirical study [19] shows that a lack of suggested fixes is one of the top pain points reported by static analysis users. Other developer studies [20, 31, 53] also report very similar findings. Hence, what we need is a tool to fix resource-leaks.

Existing repair tools. Since there are currently no specialized tools for resource-leak fixing, one could try using *general-purpose repair tools* [8, 15, 28, 29, 33–35, 39, 40, 45, 46, 49, 51, 62, 65, 67], which work on a wide variety of errors. These tools generate candidate patches using a variety of techniques, but they all validate a patch by checking if it passes the previously failing test case. Resource leaks, however, do not show up during tests, and hence cannot be fixed by such tools. Footpatch [61], one of the only general-purpose tools that does not rely on tests, is the current best tool for fixing resource leaks. However, it suffers from low-quality fixes for Java resource-leaks; it suggests fixes for only 6% of the leaks, out of which only 27% are correct.

Achieving a perfect *fixable-rate* (percentage of warnings for which a fix was suggested) and *fix-correctness* (percentage of correct fixes out of the suggested fixes) for Java resource-leaks is a lofty goal. The problem is at least as hard as compile-time object deallocation [17, 26] (i.e. replacing Java's runtime garbage collector with static deallocation), a known hard problem for compilers. Furthermore, in this repair problem, some corner cases involving loops or aliasing also reduce to undecidable problems. Hence, there will always be some resource-leaks that are infeasible to fix for a compile time-tool. However, we show that by separating the leaks that are infeasible to fix from those that are feasible to fix, it is possible to have better repairability than Footpatch in both fixable-rate and fix-correctness.

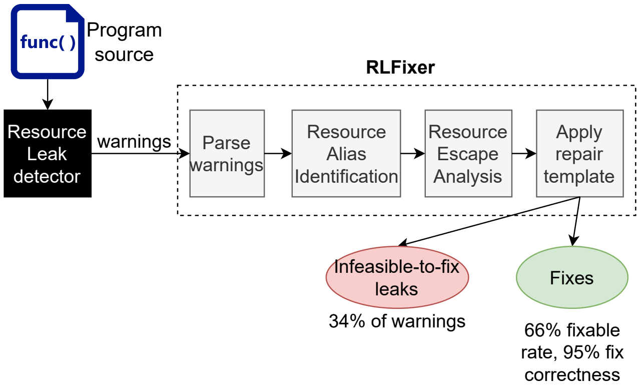

Figure 1: Overview of the RLFixer workflow

Our Approach. In this paper, we introduce *RLFixer*, a specialized repair tool for resource leaks that generates high-quality fixes. Fig. 1 gives an overview of its workflow. The warnings computed by an existing black-box resource-leak detector are first parsed to extract the location where the resource was created. Next, the *resource alias identification* step identifies pairs of resource objects that use the same underlying system resource. The third step tracks the data-flow of the resource object using a new demand-driven static analysis called a *resource escape analysis*. This analysis serves two purposes: it identifies leaks that are infeasible to fix, and it helps pick the correct repair-template for the feasible-to-fix ones. Finally, using repairs template, the last stage generates the correct fix.

In addition to generating correct fixes, we designed *RLFixer* to be fast because it will typically accompany static analysis warnings in IDEs, which are time sensitive environments. *RLFixer's* demand-driven design enables it to analyze only those statements relevant to the resource leak. It takes, on average, only 1 seconds per program, excluding the 13 seconds for setting up the call-graph, class-hierarchy, etc.

We evaluated *RLFixer* by applying it to the warnings from five popular Java resource-leak detectors: Infer [14], PMD [1], Checker-Framework [32] , Codeguru [5], and Spotbugs [3], each of which is run on programs from the NJR-1 dataset [58].

Our Contributions. We begin with an example of RLFixer fixing a resource leak (Section 2), and then we detail our contributions:

- We introduce a new static analysis, *resource escape analysis*, which helps identify leaks that are infeasible to fix, as well as pick the repair template for feasible ones (Section 3).
- We design and implement *RLFixer*, a specialized repair tool for resource-leaks that is based on the *resource escape analysis*, and can repair leaks from multiple leak detectors (Section 4).
- We show, experimentally, that *RLFixer* generates high-quality fixes with low overhead for five popular Java resource-leak detectors. Out of 2205 resource leaks detected in NJR-1, it generates, on average, fixes for 66% of the leaks, out of which an estimated 95% are correct (Section 6). *RLFixer* outperforms the Footpatch baseline, which generates fixes for only 6% of the leaks, out of which only 27% are correct.

```
1   void foo(File a) throws IOException{
2     FileReader fr = null;
    + try{
3     fr = new FileReader(a);
4     bar(fr);
5     int data = fr.read();
    + }finally {
    +    try{
    +      fr.close();
    +    }catch(Exception e){
    +      e.printStackTrace();
    +    }
    + }
6   }
7   void bar(FileReader f) {
8     BufferedReader r = null;
9     try {
10      r = new BufferedReader(f);
11      System.out.println(r.readline());
12    } catch (IOException e) {}
13  }
```

Figure 2: Example of a resource leak fixed by RLFixer

We end with a discussion of related work (Section 7) and our conclusion (Section 8).

2 EXAMPLES

This section shows two simplified examples of how the five resource leak detectors report leaks, and how *RLFixer* goes about repairing them. It highlights the need for suggesting fixes for resource-leaks, as well as some of the challenges in generating a correct fix.

Fig. 2 shows a simplified Java code snippet from one of the NJR benchmarks. It has two methods, each with one resource object. First, let us look at the method foo. foo creates a FileReader resource (line 3), which gets passed in to the bar method (line 4). Note that foo continues using the FileReader on line 5 after the bar function returns. The foo method also declares that it potentially throws an *IOException*. This declaration is required by the Java-compiler's type and effect system when a resource's potential exception is not handled in a *try-catch* block. The lines highlighted in green constitute the fix suggested by *RLFixer*; they have not been added to the code yet.

Next, let's focus on the method bar. It creates a BufferedReader resource object with the FileReader parameter f as an argument (line 10). Here, the BufferedReader is a wrapper resource that provides buffering functionality for the FileReader f. Hence we say that the resource variables f and r are *resource aliases*. This means that even though they point to different resource objects (f points to a FileReader and r points to a BufferedReader), the underlying system resource pointed to by those objects is the same. This implies that closing one resource object closes all its resource aliases. In this case, neither the BufferedReader nor the FileReader have been closed, and hence we get a resource leak.

We now run five resource-leak detectors (Infer, PMD, Checker-Framework, Codeguru, and Spotbugs) on this code, and Fig. 3 shows the output given by each. PMD and Infer identify a resource leak for the BufferedReader, whereas Checker-Framework identifies a resource-leak for the FileReader. Codeguru and Spotbugs do not

Tool	Output
Infer	Resource of type BufferedReader at line 10 is not released after line 11
PMD	Ensure that resources like this BufferedReader object are closed after use (line 8)
Checker-Framework	@MustCall method close may not have been invoked on ‘fr’ or any of its aliases (line 3)
Codeguru	N/A (Resource leak missed)
Spotbugs	N/A (Resource leak missed)

Figure 3: Outputs for the five resource-leak detectors, when given the code snippet from Fig. 2

report any resource leak. Even after getting one of these warning messages, a developer is still several steps away from a correct fix.

We also run the baseline repair tool, Footpatch, on this file-handle leak. Footpatch is tightly integrated with Infer, and relies on Infer's warning output for identifying fix locations. Footpatch first generates candidate patches by searching the code-base for program fragments that close a file, and then validates the patches by checking if Infer stops reporting the leak. In this case, Footpatch is unable to generate any patch candidates for the warning. Furthermore, even if Footpatch did hypothetically find a patch, it would apply the patch at the location in Infer's warning (after line 11). Closing the `BufferedReader` after line 11, or anywhere in function *bar*, will mean that the file pointed to by its resource-alias *FileReader* will be closed before it is read on line 5. This fix is dangerous since it introduces a new use-after-close error.

Finally, let us examine how *RLFixer* deals with the resource leak, assuming the warning came from Infer (i.e. for line 10). *RLFixer* starts off by performing a *resource alias identification* for the new `BufferedReader` object (line 10). This analysis reveals that `f` is a resource-alias. Next, *RLFixer* performs a *resource escape analysis*, a static analysis that computes how the `BufferedReader` and any of its aliases escapes the method. The two ways the resource escapes the `bar` method are via the `readline` method call (line 11) and via the parameter `f`. When a resource escapes via a parameter, we cannot close the resource in the current method, since the resource is still accessible after the method returns. Instead, we examine the caller instruction on line 4, which is in the method `foo`. Carrying out the *resource escape analysis* for *fr* in `foo` shows that it only escapes via method calls, and hence can be closed in the method `foo` itself. *RLFixer* then picks the correct repair-template, and it suggests the fix highlighted in green in Fig. 2. The repair-code correctly fixes the leak without introducing new errors or modifying the semantics of the original program. *RLFixer* computes the same fix for the warnings given by PMD and Checker-Framework.

A Resource Leak that is Infeasible to fix. Fig. 4 shows an example of a resource leak that may be infeasible to fix at compile time. *RLFixer*, during its *resource escape analysis*, tracks the `FileWriter` resource (line 4) through the call to the method `store`, and identifies that it is assigned the field `fw` on line 9. Since this field is accessible as long as its parent *A* object is alive, we can only safely close this resource when the *A* object is getting deallocated. This makes the problem at least as hard as compile-time object deallocation [17, 26], a known

```
1  class A{
2    FileWriter fw;
3    void create(File b) throws IOException{
4      FileWriter f = new FileWriter(b);
5      store(f);
6    }
7
8    void store(FileWriter a) {
9      fw = a; /* Resource escapes to a field */
10   }
11 }
```

Figure 4: Example of a resource leak that is infeasible to fix

hard problem for compilers. Prior research has only managed to statically deallocate some objects in the program [26], and the hardness of this problem is the reason why Java uses a runtime garbage collector. This is just one of the infeasible cases for resource-leak repair; we discuss the full list of cases in Section 3.

There will always be resource-leaks that are too hard to fix statically. *RLFixer* aims to identify and separate out the hard-to-fix leaks like the one in Fig. 4, while correctly fixing the rest of the resource-leaks, like the one in Fig. 2.

3 APPROACH

This section gives an overview of *RLFixer's* approach to fixing resource-leaks. Fig. 1 shows the four main components of *RLFixer*: the warning parser, the resource alias identification, the *resource escape analysis*, and the application of repair templates; we now discuss each of these in detail.

3.1 Warning Parser

The first component parses the resource-leak detector's warning and extracts the source file and line number where the leaked resource was created. Each resource-leak detector needs a separate parser because each tool uses a different output format, but this component is simple and small. On average, it takes only 15 lines of Python code per new tool.

3.2 Resource Alias Identification

The second step for *RLFixer* is identifying resource aliases for the leaked resource objects. This is an important step because a majority of resource usage in Java involves resource aliasing. We have already seen an example of resource aliasing in Fig. 2, where the *FileReader* and *BufferedReader* objects pointed to the same OS resource. Prior research [32, 57] has studied resource-aliasing from the point of view of pruning false-positive resource-leak warnings. Here, we study resource-aliasing from the point of view of generating correct repairs. Below is the resource-aliasing definition that *RLFixer* uses.

(1) Variables x and y are resource-aliases if x is a wrapper for y, or y is a wrapper for x.
(2) Resource W constructed with the constructor C_W is a wrapper for resource R if:
   (a) R is passed as a parameter to C_W, and
   (b) R is a must-alias of a field of W at the end of C_W, and
   (c) The must-alias field always gets closed in the `close()` function of W

```
1  class WrapperType{
2    private ResourceType out;
3    public WrapperType(ResourceType w){
4      out = w;
5    }
6    public close(){
7      out.close();
8    }
9  }
10 ..
11   {
12     /* Resource-leak reported here*/
13     x = new ResourceType("a.txt");
14     y = new WrapperType(x);
15   }
```

Figure 5: Resource Alias Identification: checking if the WrapperType object is a wrapper for the ResourceType object

(3) All pointer aliases are treated as resource-aliases.

This definition also serves as a specification for a static analysis, which *RLFixer* implements to identify resource alias pairs. Let us use the example from Fig. 5 to check if the variables x and y are resource-aliases. The resource leak warning is reported for the `ResourceType` object on line 13. The `ResourceType` cannot be a wrapper for any other object because its constructor only takes a string input, and this will never satisfy condition (2a). So let us check the 3 conditions for the `WrapperType` to be a wrapper for the `ResourceType`. We first perform a def-use analysis [52] of x, which identifies all uses of x. Since x is used as a parameter in the constructor for the `WrapperType` (line 14), condition (2a) is satisfied. Next, for condition (2b), we check the `WrapperType` constructor and its callees for an assignment of its parameter w (or one of its aliases) to a field of the `WrapperType`. In this case we have such an assignment (line 4) for the field, *out*; the condition is satisfied. Analyzing the `close` function reveals that the resource from the field *out* gets closed in it (line 7), and this satisfies condition (2c). Thus, all three conditions are satisfied; the `WrapperType` and `ResourceType` are resource-aliases. The final part of the definition says that all pointer-aliases are resource-aliases. Pointer-aliases can be found using a typical demand-driven pointer analysis [56].

We now know how to identify pairs of resource-aliases, but we also need to consider resource-objects that are linked by multiple layers of resource-wrapping. This is quite common in Java programs; a resource can be wrapped in up to four or five layers of resource wrappers. We identify this linking by computing a transitive closure over the resource-aliasing relationship.

3.3 Resource Escape Analysis

The third component, the *resource escape analysis*, computes all the types of program constructs that the resource can escape to. This analysis is used by *RLFixer* for two purposes: it helps separate out the infeasible-to-fix leaks, and it helps compute repairs for the feasible-to-fix leaks.

The *resource escape analysis* is carried out on the WALA IR [2] because it is easier to write a data-flow analysis on WALA IR than on Java source code. WALA is the static analysis framework used by *RLFixer*, and the WALA IR is very close to Java bytecode.

```
Program :: (C, C, ... C)
C :: <cname> Ext ImpDS {fields:{f;..f;} methods:{M,..,M}}
M :: <mname>($V_1$,..,$V_n$) {instructions:{I;..I;}}
I :: ArrayStore | FieldWrite | Assgn | PhiStmt |
     ReturnStmt | Invoke | FieldRead | ConditionalBranch
     | NewStmt | ArrayLoad
ArrayStore :: $V_{arr}$[$V_{idx}$] = $V_{rhs}$
FieldWrite :: $V_{lhs}$.f = $V_{rhs}$
Assgn :: $V_{lhs}$ = $V_{rhs}$
PhiStmt :: $V_{lhs}$ = phi($V_1$,..,$V_n$)
ReturnStmt :: return V
Invoke :: $V_{lhs}$ = $V_{rhs}$.<mname>($V_1$,..,$V_n$)
FieldRead :: $V_{lhs}$ = $V_{rhs}$.f
ConditionalBranch :: if $V_{bool}$ goto <instr-index>
NewStmt :: $V_{lhs}$ = new <cname>
ArrayLoad :: $V_{lhs}$ = $V_{arr}$[$V_{idx}$]
V :: <var-name>
Ext :: extends <cname> | ''
ImpDS :: implements <data-structure-interface> | ''
```

Figure 6: Simplified grammar for the WALA IR

Escape type	Resource has this escape type if it:
Field-Escape	aliases with an instance field or static field
Data-Structure-Escape	aliases with an element in an array or data-structure.
Return-Escape	gets returned by the method it is created in.
Parameter-Escape	aliases with a formal parameter of the method it is created in
Invoke-Escape	is passed as an argument to an invoke statement (i.e. method call)

Figure 7: The five escape mechanisms for a resource object

Fig. 6 gives a simplified grammar of the WALA IR. Most of the grammar terms are common to most intermediate representations. The `PhiStmt` is a special instruction in all SSA-based IRs to merge values from a source-variable that appears on two different control-flow paths. The `ImpDS` non-terminal specifies if the class implements any of the data-structure interfaces (such as Map, Collections, etc.).

The list of instructions in the grammar show that there are five program constructs to which a resource can escape from its method: a field, a data-structure (an array or a data-structure class), a return variable, a parameter, or an invoke (i.e. method-call).

We define five escape types corresponding to these five constructs: *Field-Escape, Data-Structure-Escape, Return-Escape, Parameter-Escape, Invoke-Escape*. Fig. 7 describes what each escape type means. A resource can have multiple escape types. By enumerating through the instruction types in Fig. 6, we can see that the five escape types exhaustively cover all the ways a resource can escape the method it is created in.

Fig. 8 gives pseudo-code for the analysis. It is designed to be demand-driven, and its output is the set of possible escape types for the resource. The analysis procedure, whose name is shortened to `rea`, takes two arguments: the resource variable whose escape mechanisms need to be analyzed, and the method it is declared in. *escTypes*, the set of escape types, is initialized to the empty set in

```
 1: procedure REA($resourceVariable$, $method$)
 2:     $escTypes = \emptyset$
 3:     for I in getUses($resourceVariable$) do
 4:         switch I.instructionType do
 5:             case ArrayStore:
 6:                 $escTypes \ni$ Data-Structure
 7:             case FieldWrite:
 8:                 $escTypes \ni$ Field
 9:             case Assgn:
10:                 $escTypes \supseteq$ rea(Assgn.$V_{lhs}$, $method$)
11:             case PhiStmt:
12:                 $escTypes \supseteq$ rea(PhiStmt.$V_{lhs}$, $method$)
13:             case ReturnStmt:
14:                 if $method$ == originalWarningMethod then
15:                     $escTypes \ni$ Return
16:                 end if
17:                 for $caller$ in callers($method$) do
18:                     $escTypes \supseteq$ rea($caller.V_{lhs}$, $caller$.method)
19:                 end for
20:             case Invoke:
21:                 for M in invoke.targets do
22:                     if M.isDataStructureMethod() then
23:                         $escTypes \ni$ Data-Structure
24:                     else
25:                         $escTypes \ni$ Invoke
26:                         $p$ =M.matchingParam($resourceVariable$)
27:                         $escTypes \supseteq$ rea(p, M)
28:                     end if
29:                 end for
30:             case FieldRead: do nothing
31:             case ConditionalBranch: do nothing
32:             case NewStmt: do nothing
33:             case ArrayLoad: do nothing
34:     end for
35:     if isParameter($resourceVariable$) then
36:         if $method$ == originalWarningMethod then
37:             $escTypes \ni$ Parameter
38:         end if
39:         for $caller$ in callers($method$) do
40:             $p$ = $caller$.matchingArgument($resourceVariable$)
41:             $escTypes \supseteq$ rea(p, $caller$.method)
42:         end for
43:     end if
44:     return $escTypes$
45: end procedure
```

Figure 8: Resource Escape Analysis (Name shortened to *rea*)

the beginning. The bulk of the method is a for loop over the uses of the resource variable.

For each use, *RLFixer* performs a case analysis based on the type of the use instruction (line 4). The possible use instructions come from the grammar in Fig. 6. If the use-instruction is an *ArrayStore*, it follows that the resource object aliases with an array element, and according to Fig. 7, this implies a *Data-Structure Escape*; we add this to $escTypes$. Similarly, a *FieldWrite* implies a *Field Escape*. An *Assgn* or *PhiStmt* requires us to recursively track the assigned variable; hence we call `rea` on it. Being used in a *ReturnStmt* in the warning's original method implies that *Return* should be in $escTypes$. Additionally, since we need to track the returned variable in all callers, we add the escape types of the callers' call-sites to $escTypes$. If the use-instruction is an *Invoke* (i.e. method call), we split it into two sub-cases. If the method belongs to a data-structure class, we add *Data-Structure* to $escTypes$. If not, we add *Invoke* to $escTypes$, and track the escape types in the method call by recursively calling `rea` on the matching argument in the invoke targets. We do not need to do anything for the last four instruction types. A *FieldRead* does not propagate any escape information from the resource variable because only the field is read. *ConditionalBranch*, *NewStmt*, and *ArrayLoad* do not even support the use of a resource variable.

In addition to checking for the uses of the resource variable, we also need to check if it escapes to a parameter (line 35). If so, we add *Parameter* to $escTypes$. Additionally, we recursively track the escape types in the caller methods, by calling `rea` on the resource variable's matching argument in the caller methods (line 41).

Finally, the analysis returns $escTypes$, the aggregate set of escape types for $resourceVariable$. Since resource aliases point to the same underlying resource, an escape type for one alias applies to all other aliases. Hence, the resource escape analysis must be repeated for all resource-aliases of $resourceVariable$, and their escape types added to $escTypes$.

3.4 Applying Repair Templates

The final step for *RLFixer* is generating repair code. The repair code has the following specification: it should close the leak after the last use of the resource, without introducing new errors (such as a new leak, a use-after-close error, or a null pointer exception) or modifying the semantics of the original program.

RLFixer uses the decision tree from Fig. 9 to pick the correct repair strategy. If a resource escapes to a field or data-structure, *RLFixer* marks it as infeasible to fix. If the resource does not escape to a field or data-structure, but does escape to a return or parameter, *RLFixer* creates dummy leak warnings at the caller methods, and closes the leak there. If it does not have any of these four escape types, we can close the resource in the same method as it was created. Based on how the resource is used, we then apply one of three repair templates. Since the decision tree covers all the five escape types, it exhaustively covers all the ways a single resource can leak. Let us now examine each decision-tree node in detail.

Field-Escape. Resources with a *Field Escape* are infeasible to fix. Closing a resource that escapes to an instance field, like in the example from Fig. 4, is at least as hard as compile-time object deallocation. Closing a resource that escapes to a static field is not possible becuase static fields are alive throughout the program's life. Furthermore, in cases where we don't have access to all the code at compile time (such as when designing a library), it is impossible to statically even identify all uses of a field; in this case, a *Field-Escape* will never be safe to close.

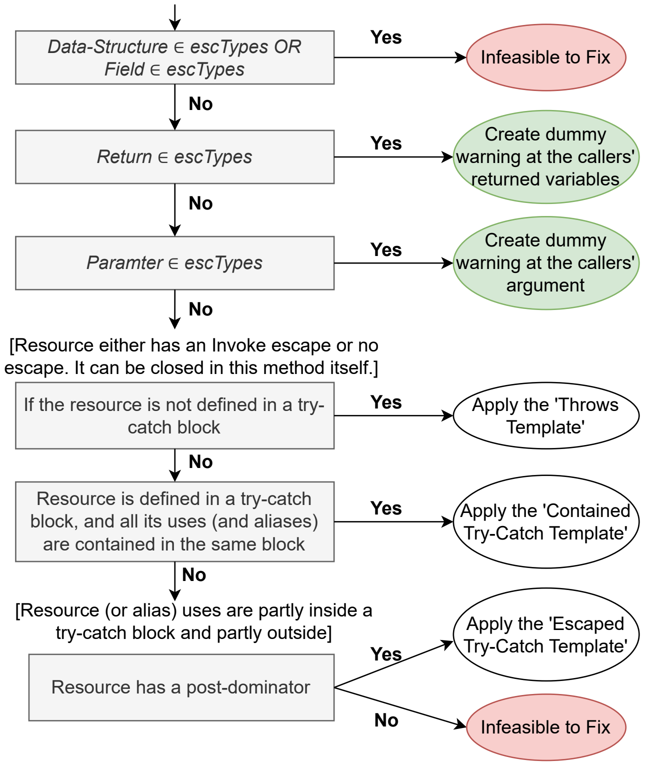
Data-Structure ∈ escTypes OR Field ∈ escTypes
Yes
Infeasible to Fix
No
Return ∈ escTypes
Yes
Create dummy warning at the callers' returned variables
No
Paramter ∈ escTypes
Yes
Create dummy warning at the callers' argument
No
[Resource either has an Invoke escape or no escape. It can be closed in this method itself.]
If the resource is not defined in a try-catch block
Yes
Apply the 'Throws Template'
No
Resource is defined in a try-catch block, and all its uses (and aliases) are contained in the same block
Yes
Apply the 'Contained Try-Catch Template'
No
[Resource (or alias) uses are partly inside a try-catch block and partly outside]
Resource has a post-dominator
Yes
Apply the 'Escaped Try-Catch Template'
No
Infeasible to Fix

Figure 9: Decision-tree depicting how RLFixer decides which leaks are infeasible to fix, and picks the correct repair template to apply.

Data-Structure-Escape. Fixing *Data-Structure-Escapes* is hard because it is well-known that unbounded data structures such as arrays are hard to accurately model using static analysis [25]. Hence, static analysis tools model data structures using *over-approximation*. In the case of an array, the over-approximation is to assume that a read or write to the array could affect any possible index. Such an over-approximation is safe for resource-leak detection because it will never miss out on a leak that occurs in a possible execution. However, it is unsafe for our repair problem because closing a resource from an array requires us to know the exact index that the resource is at. A similar argument applies to other data-structures. Hence, *RLFixer* does not generate any repairs for this case.

Return Escape. At the *Return ∈ escTypes* node of the decision tree, we already know that the resource does not escape to a field or data-structure. If the resource does escape via a *Return*, we create one dummy warning for each caller at the returned variable. For example, in the snippet (`Resource r = getRes()`), a resource object gets returned by the `getRes` method and hence is still alive after the `getRes` method returns. Consequently, we cannot close the resource inside the `getRes` method. Instead, we create dummy leak warning at the returned variable (`r` in this case). We then recursively apply *RLFixer* to the dummy warning(s), and suggest their repairs as a fix for the original warning.

Parameter Escape. The *Parameter Escape* case is similar to the *Return Escape* case, where we create a dummy warning at the caller methods. The only difference is that the dummy warning is created at the corresponding argument of the paramter in the caller. We already saw this strategy being applied to the example in Fig. 2. The resource leak was reported at the new Buffered Reader in the `bar` method of Fig. 2 (line 10). The Buffered Reader resource escapes to the parameter via its resource alias (*FileReader f*). Hence, we create a dummy warning for the argument `fr` at the method call to `bar` in the caller method `foo` (line 4). We then recursively apply *RLFixer* on the dummy warning(s), and suggest their repairs as a fix for the original warning.

Invoke Escape and Non-escape. At the decision tree node where we have neither a *Data-Structure*, *Field*, *Return* or *Parameter* escape, we are left with resource-leaks that either have an *Invoke Escape* or no escape types. In these two cases, the resource is not used after the method completes, and hence should closed in the same method. Based on whether the resource is defined inside a try-catch block or not, we define three repair templates: the *Throws Template*, the *Contained Try-Catch Template*, and the *Escaped Try-Catch Template*. Fig. 10 illustrates these templates. The lines highlighted in green give the fix suggested by *RLFixer*.

The *Throws Template* (see Fig. 10a) applies when the resource is not created or used within a *try-block*. The repaired code places all the resource (and alias) uses within a *try-finally* block. The *try* block starts at the first line where the resource is used. The *finally* block starts after the last line where the resource is used, but with adjustments to match the scope of the newly added try-block. Note that the new close statement is placed within its own try-catch block to handle any exception (related to resource access, null-pointers, etc.) it may throw, thereby avoiding modifying the control flow of the original program. Modifying the control flow of the original program modifies its semantics, and this goes against our repair specification. We will see the same pattern with the next two templates.

The *Contained Try-Catch Template* (see Fig. 10b) is applied when the resource creation and all its uses (and resource aliases) are contained within a *try* block. In this case, the correct repair is to attach a corresponding *finally* block that closes the resource. If a *finally* block is already present, *RLFixer* adds the close statement to the existing finally block. The *finally* block in Java always executes after the *try-catch* block, even if the *try* block has a return statement or an exception. Hence, with this fix, the resource is closed on all program paths, including ones involving an exception.

The *Escaped Try-Catch Template* (see Fig. 10c) applies when resource creation and use statements (and resource aliases) are partly inside a *try* block and partly outside. For example, the use on line 5 is inside, whereas the one on line 8 is outside. Here, *RLFixer* first places all statements that are outside a *try* block (e.g. line 8) in a new *try-catch* block; this prevents control from escaping the method *bar* before the resource can be closed. Note that *RLFixer* re-throws the exception in the fix code to preserve control-flow to any exception handler in the caller method of *bar*. Finally, the resource needs to be closed at its earliest *post-dominator*. A *post-dominator* for a resource is an instruction in the method that appears on every control-flow path from a resource use to the end of the

```
1   void foo() throws Exception{
2     Resource r = null;
    + try{
3     r = new Resource(..);
4     r.useResource();
    + } finally {
    +     try{ r.close(); }
    +     catch(Exception e) {
    +       e.printStackTrace();
    +     }
    + }
5   }
```

(a) Throws Template

```
1   Resource r = null;
2   try {
3     r = new Resource(..);
4     r.useResource();
5   } catch (IOException e) {
6     e.printStackTrace();
7   }
    + finally{
    +   try{ r.close(); }
    +   catch(Exception e) {
    +     e.printStackTrace();
    +   }
    + }
8   /* No Resource use after this */
```

(b) Contained Try-catch Template

```
1   void bar() throws Exception{
2     Resource r = null;
3     try {
4       r = new Resource(..);
5       r.useResource();
6     } catch (Exception e) {}
7     /* Some code in between */
    + try{
8       r.useResource();
    + } catch (Exception e) {
    +     try{ r.close(); }
    +     catch (Exception f) {}
    +     throw e;
    + }
9     /* Some code in between */
10    /* No Resource use after this */
    + try{
    +   r.close();
    + } catch (Exception E) {
    +   e.printStackTrace();
    + }
11  }
```

(c) Escaped Try-Catch Template

Figure 10: *RLFixer's* Repair Templates

method. In Fig. 10c, assuming the earliest post-dominator for the resource is on line 10, *RLFixer* closes the resource at this point. If the earliest post-dominator is inside a try-catch block, the close statement goes inside a finally block. Choosing the earliest post-dominator for closing the resource is always safe, but in corner cases with a method having multiple exit points, a resource may have no post-dominator; *RLFixer* does not suggest any fix in this corner case.

We avoid Java's try-with-resources statement because it only applies to resources that implement the `AutoCloseable` interface. Additionally, it supresses exceptions in the try-with statement in some cases, thereby modifying the control flow. Futhermore, it cannot handle the resource usage pattern from the Escaped Try-Catch Template.

Loops and Existing Close Statements

There are two more details we need to handle in the repair code: loops and existing close statements.

Dealing with Loops. We can divide the resource-leaks in loops into two sub-cases. The first, more common sub-case, occurs when the resource is created during a loop iteration and is never used after the end of that iteration. *RLFixer* deals with this sub-case by extracting the loop-body and computing the fix on this loop-body as it would for any loop-free resource leak. For the very rare sub-case where a resource created in a loop stays alive beyond the end of a loop iteration, *RLFixer* does not suggest a fix; this sub-case gives an undecidable problem. [1]

Deleting Existing Close Statements. In addition to adding repair code, *RLFixer* also needs to remove unnecessary `close` statements added by the programmer to avoid a double close. We design each of our repair-templates to require a single close statement; hence, *RLFixer* deletes any existing close statements that were added by the programmer. For example, in Fig. 10b, if the programmer had inserted a close statement inside the `try` block after line 4, *RLFixer* would need to delete it (in addition to generating new repair code).

4 IMPLEMENTATION

This section discusses the implementation details for *RLFixer. RLFixer* is primarily implemented in the WALA static analysis framework for Java bytecode [2]. We wrote *RLFixer's* analyses on WALA's IR instead of Java's source AST because the IR has simpler control flow, fewer instruction types, and is already in SSA form. Furthermore, WALA automatically sets up the core information needed by any static analysis, such as computing the class-hierarchy, call-graph (using the 0-CFA algorithm) and basic-blocks. The Repair-Template stage of *RLFixer* additionally uses JavaParser [4] to scan the Java source ASTs for scoping and line number information.

The resource-escape analysis, call-graph, and resource-alias analysis all use a context-insensitive analysis. Context-sensitivity is not needed because we know of no way to represent context in repair code. Our analyses automatically get partial flow-sensitivity because of the WALA IR's SSA form. Field-sensitivity is redundant because all resources aliasing fields become *Field Escapes* and do not get fixed. Reflection support can trivially be added by turning on WALA's reflection analysis, but we skip this option; it only benefits a tiny fraction of repairs, while increasing call-graph computation time by many fold. In our experiments, out of the 150 resource-leaks that were manually examined, none were affected by reflection.

The output format of the tool is much like that in Fig. 2, and can easily be incorporated into an IDE or existing static analysis tool. Note that *RLFixer* does not automatically adjust variable scopes in its generated fix; it is up to the programmer to correct this.

[1] Fixing the leak in this rare sub-case requires us to identify the last loop iteration (to close the leak), which is a known hard problem for compilers.

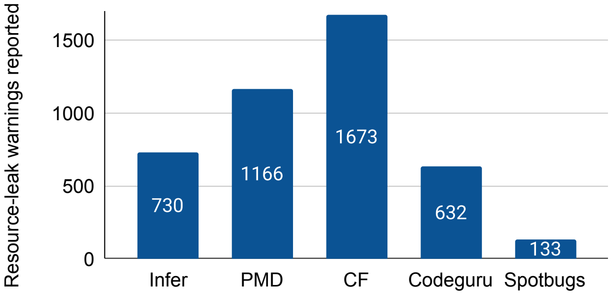

Figure 11: Warnings reported by the five resource-leak analyzers when applied to the NJR dataset.

5 DATASET

We use the NJR-1 dataset (available here [58]), as our benchmark-set. It consists of a diverse set of 293 Java-8 programs from GitHub, and has been used in several recent static analysis papers such as [36, 59, 60]. It is well-suited to evaluate *RLFixer* because it has several resource leaks, and runs off-the-shelf with several existing resource-leak detectors and Java analysis tools like WALA and JavaParser. We leave out 6 of the 293 NJR-1 benchmarks; Footpatch runs out of memory for three benchmarks, and three are missing a library class. This leaves us with 287 benchmarks for our experiments.

We run five popular Java resource-leak detectors on this dataset: Infer [14], PMD [1], Checker-Framework [32] (shortened to CF), Codeguru [5], and Spotbugs [3]. The warnings given by these tools are then fed to *RLFixer*. We ran all the tools with their default options, and after post-processing the warnings to filter out duplicate warnings, etc., we got a total of 2205 unique resource-leaks. During the duplicate filtering, if there are two warnings for a resource-alias pair (i.e. same root cause), one gets removed. Fig. 11 lists the number of warnings given by each tool after duplicate filtering. CF gets the most resource-leaks, probably because of its commitment to soundness (i.e. catching as many possible leaks). Spotbugs gets the fewest resource-leaks, probably because soundness was traded off for speed. Similarly, the other tools differ in their set of reported leaks because of different design decisions.

Fig. 12 reports some statistics about the frequency of resource leaks in the NJR dataset. 207 out of 287 programs have at least one leak. By taking a union of the resource leaks by the five tools, we get an estimated total of 2205 unique resource leak warnings. This averages to 8 resource leaks per program. Given that the average number of lines of application code in the NJR dataset is almost 10k, we can expect one resource leak in every 1300 lines of code. Thus, resource leaks are prevalent in the dataset; developers need better tool-support for fixing these leaks.

6 EXPERIMENTAL RESULTS

In this section, we discuss our experimental results which answer the following four research questions:

(1) **RQ1:** How many warnings can RLFixer suggest fixes for?
(2) **RQ2:** How many of RLFixer's suggested fixes are correct?
(3) **RQ3:** How does RLFixer compare to Footpatch?
(4) **RQ4:** How long does RLFixer take to generate repairs?

Tool	Output
Total number of programs:	287
Programs with at least one resource leak:	207
Estimated number of unique resource leaks (across the five tools)	2205
Lines of application code per benchmark	9911

Figure 12: Statistics about the frequency of resource leaks in the NJR dataset

The four questions are answered by the following four claims, which are in turn validated in the next four subsections (all numbers are averages across the five resource leak detectors).

(1) RLFixer suggests fixes for 66% of the resource-leak warnings.
(2) 95% of the fixes suggested by RLFixer are correct.
(3) RLfixer produces higher quality fixes than Footpatch.
(4) RLFixer takes, on average, 1 seconds per program, excluding the 13-second WALA setup time.

The experiments were carried out on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 GB RAM. For the JVM, the default heap size of 32GB, and default stack size of 1MB, was used.

6.1 RQ1: Fixable Rate

Fixable Rate is the percentage of warnings for which a fix was suggested. It is defined as:

$$\text{Fixable Rate} = \frac{\text{\# warnings for which a fix was suggested}}{\text{total \# warnings}}$$

Fig. 13 gives a split up of the fixable and unfixable resource-leaks for *RLFixer* on each of the five tools. On average, *RLFixer* gets a 66% fixable rate, with PMD getting the highest fixable-rate (75%). The unfixable resource-leaks are further split based on the reason they are not fixed: From the graph, we see that the main reason for unfixed leaks are *Field Escapes* (20%). CF gets a lower fixable-rate than the other tools because of a large percentage of its leaks being *Field Escapes*. A smaller contributor to unfixed leaks are *Data-Structure Escapes* (9%). Some 1% of resource leaks escape to both, a data-structure and a field. We report these as data-structure escapes to simplify the graph. The last 5% of leaks are not fixed (in red color) because, as discussed in Section 3.4, there are corner cases for some templates that result in undecidable problems.

6.2 RQ2: Fix Correctness

Another important metric is *Fix Correctness*, the percentage of correct fixes out of the suggested fixes. It is defined as:

$$\text{Fix Correctness} = \frac{\text{\# warnings with a correct fix suggestion}}{\text{\# warnings with a fix suggestion}}$$

We picked a sample of 150 fixes (30 per resource-leak detector) suggested by *RLFixer* to estimate the fix-correctness. We re-ran the resource-leak detector on the fixed code to ensure that the old leaks disappeared. For 2 fixes, the old leaks remained, and

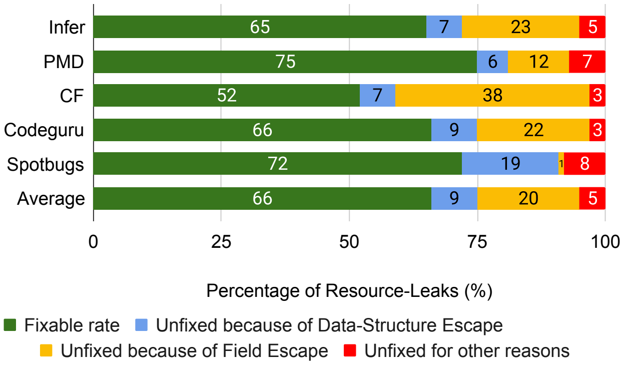

Figure 13: Fixable-rate for RLFixer for each resource-leak detector, along with reasons for the unfixed leaks.

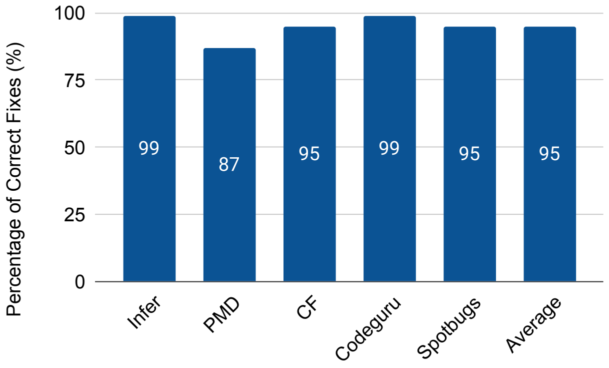

Figure 14: Percentage of correct fixes by *RLFixer* (i.e. fix-correctness) for the five resource-leak detectors

these were marked as incorrect. For the remaining fixes, we had 5 volunteer programmers classify the fixes as correct or incorrect. The volunteers, none of whom are authors, are computer-science graduate students who are familiar with Java and resource leaks. The volunteers classify different subsets of the fixes, but each fix is classified by at least 3 volunteers. Each volunteer uses the following criteria to evaluate correctness, and a fix is considered incorrect even if one of these criteria is not met.

1. The fix repairs the leak.
2. The fix does not introduce a null-pointer error.
3. The fix does not introduce a use-after-close error (e.g. file written after being closed).
4. The fix does not introduce a double close.
5. The fix does not modify the behavior of the program.

Finally, we computed the fix-correctness by taking an average over the scores of the volunteers. The inter-rater agreement, calculated using Krippendorff's Alpha, is 0.86. The scores for each tool are shown in Fig. 14. On average, *RLFixer's* fix-correctness is 95%, with Infer and Codeguru getting near perfect fixes. Given that less than one in twenty fixes by *RLFixer* are incorrect, we can put high confidence in its generated repairs.

Examining the small fraction of incorrect fixes shows that there are two major roadblocks to *RLFixer* reaching perfect fix-correctness. The first is that the definition for *resource-alias analysis* works well in most cases, but it does not exhaustively capture all the ways that two Java objects can share a resource. Missed resource-aliases in turn give incorrect fixes. Designing a perfect resource-alias analysis is hard. The second roadblock is that *RLFixer's* templates are designed to fix individual resource-leaks, and hence do not work perfectly when multiple resource-leaks occur in the same code block.

Another correctness issue that most repair-tools need to deal with is false-positive warnings, and whether one suggests repairs for these false warnings; this, however, does not seem to be an issue in practice for *RLFixer*. To get a measure of false-positive warnings, we asked the volunteers to also examine the same 150 repairs and decide whether the original leak-detector warning was a false positive. All five resource-leak detectors gave zero false-positive warnings for the leaks fixed by *RLFixer*. This low false-positive rate is expected, since these are all mature tools that have been heavily engineered to weed out false-positive warnings. Note that there could still be false-positive warnings among the infeasible-to-fix leaks, but this doesn't affect *RLFixer*.

6.3 RQ3: Comparison with Footpatch

Fig. 15 summarizes the comparison between Footpatch and *RLFixer*. We only use the Infer warnings for the comparison because Footpatch is tightly integrated with Infer; it cannot be used with other resource-leak detectors. We split the results into two parts: the first part (columns 2 and 3) shows the results on the warnings from the NJR benchmarks, and the second part (columns 4 and 5) gives the results on the *apktool* benchmark from the Footpatch paper [61].

For the NJR benchmarks, Infer generates 730 warnings, for which Footpatch generates 46 fixes, giving us a 6% fixable-rate. The fixable-rate for *RLFixer* (65%) on NJR is the same as the Infer entry in Fig. 13. For the NJR fix-correctness, we chose a random sample of 30 fixes for each tool, and evaluated for correctness using the criteria from Section 6.2. Out of the sample of 30 Footpatch fixes, 8 were correct, giving a 27% fix-correctness. *RLFixer's* fix-correctness (99%) is the same as the entry for Infer in Fig. 14. Thus, on the NJR benchmarks RLFixer performs significantly better on both, fixable-rate and fix-correctness.

For *apktool*, the only benchmark from the Footpatch paper [61] with Java resource leaks, Infer gives 19 warnings. Out of these 6 are duplicates and we remove them. For the remaining 13 warnings, Footpatch attempts a fix for 1 warning (fixable-rate 8%), and RLFixer attempts a fix for 12 warnings (fixable-rate of 92%). Both tools produce only correct fixes for this benchmark (fix-correctness 100%).

The large gap in fix-quality between Footpatch and *RLFixer* is expected; Footpatch is a more general purpose tool that works with multiple kinds of errors, as well as on both C and Java. *RLFixer*, on the other hand, is specialized for resource-leaks in Java, and hence is able to vastly outperform Footpatch on this task.

Tool	NJR benchmarks		apktool	
	Fixable Rate	Fix Correctness	Fixable Rate	Fix Correctness
Footpatch	6%	27%	8%	100%
RLFixer	65%	99%	92%	100%

Figure 15: Comparing the repair quality of RLFixer and the Footpatch baseline when fixing the Infer warnings. We show the results separately for the NJR benchmarks and *apktool.*

Tool	Leak detector time (s)	Overall fix time	
		WALA setup time (s)	RLFixer time (s)
Infer	42	13	1
PMD	6	12	1
Spotbugs	13	13	1
CF	63	12	1
Average	31	13	1

Figure 16: Split up of the time taken per program by RLFixer and the resource-leak detectors

6.4 RQ4: Repair Time

Fig. 16 shows the time taken per program by *RLFixer* and four of the resource-leak detectors. We do not report the time for Codeguru because it is only accessible via a web service. On average, resource-leak detection takes 31 seconds per program, but this varies widely across the four detectors. PMD and Spotbugs are very fast, whereas Infer and CF take much longer. The overall repair time is 14 seconds per program. A majority of this time (13 seconds) is taken by the WALA setup, whereas *RLFixer*, as described in Section 3, takes just 1 second per program because of its demand-driven design. Its fix time per leak is even lesser. The WALA setup includes tasks like constructing the class-hierarchy, call-graph, basic-blocks, etc., but a majority of the time is taken for call-graph construction.

Call-graph construction is unavoidable for any inter-procedural analysis, but we could eliminate this component by integrating *RLFixer* with a WALA based resource-leak detector. Since *RLFixer* is implemented in the WALA framework, such a design allows *RLFixer* to reuse the WALA setup information from the resource-leak detector phase. This could bring the total fix time down to just 1 second.

A direct comparison of repair times with Footpatch is not meaningful because Footpatch suggests very few fixes, and it is hard to factor out the time taken by unsuccessful fix attempts and fixes for other kinds of bugs. However, results from the Footpatch paper show that it takes several minutes per Java program, which is at least an order of magnitude larger than *RLFixer*. There are two reasons why *RLFixer* is faster: it is demand-driven, and unlike other Footpatch, it does not need to perform a search over possible repair codes; it constructs fixes from repair templates.

6.5 Threats to Validity

The first threat to validity is that the human volunteers who participated in the experiment presented in Section 6.2 could make mistakes in their evaluation of the fixes. We mitigate this threat by averaging scores over multiple volunteers and and a large number of leaks (150 in total) from different tools. Furthermore, we also re-ran the resource-leak detectors on the fixed code to confirm that the resource-leak warning disappeared.

The second threat is that our evaluation was carried out on Java-8 programs from the NJR-1 dataset. The assumption is that our results will generalize to other Java benchmarks.

The third threat is the applicability of RLFixer's approach to other langauges and platforms, since RLFixer's design and our experiments only focus on Java code.

7 RELATED WORK

The research direction closest to this work is automated program repair, and one can split this category into general-purpose, special-purpose, and linter-based repair tools. More distantly related are escape analysis and repairing Android resource leaks. We discuss each of these in turn.

General-purpose repair tools. General-purpose repair tools aim to fix a wide-variety of program errors. Most of these tools are test-based techniques, and can be split into three paradigms. The first paradigm, generate-and-validate [28, 33, 45, 46, 62], generates candidate patches by searching through existing patches and code. The second is the deep-learning based paradigm [22, 29, 39, 40, 65, 67] that uses deep-learning to find patches, often by applying Neural Machine Translation models from NLP. The third semantics-based paradigm [8, 34, 35, 49, 51] generates patches by casting the repair problem as a constraint satisfaction problem. Ultimately, all three paradigms validate each patch by checking if it passes the previously failing test case. These paradigms cannot be applied to resource-leaks because resource-leaks do not cause test-failures.

Footpatch [61] is the only general-purpose tool which can be applied to resource-leaks because it relies on the Infer static analysis tool instead of tests to verify the fix. It generates patches by searching the same code-base for program fragments that address the given bug class. It is semi-specialized to heap errors including null-pointer errors, resource leaks, etc. However, Footpatch has three shortcomings, compared to *RLFixer*. Firstly, *RLFixer* has an order-of-magnitude better fixable-rate and fix-correctness (see Section 6.3). Footpatch pays for its generality with a lower fix quality. Secondly, *RLFixer's* repair templates, by design, do not modify the semantics of the program or introduce any new errors. On the other hand, Footpatch's notion of correctness is limited to re-running Infer to ensure the leak disappreas. This does not ensure that the semantics of the program are unmodified and no new errors are introduced. Thirdly, *RLFixer* is also much faster; it takes seconds instead of minutes or hours.

Special-purpose repair tools. Special-purpose repair tools, as opposed to general-purpose tools, focus on repairing a single kind of error; this enables them to produce much higher quality fixes. Most of these tools report fixable-rates of 40-70% and a high fix-correctness, which is very similar to what we see with *RLFixer*, and this is usually significantly higher than what general-purpose tools can achieve. The kind of errors tackled by special-purpose tools include null-pointer errors [38, 66], integer overflows [16, 50], buffer overflows [55], concurrency errors [7, 30, 41], performance bugs [54], and memory leaks [23, 27, 37].

Among the existing special-purpose repair tools, memory-leak fixing [23, 27, 37] is the closest to *RLFixer* because it has a similar specification: repair the leak without modifying the program's semantics. However, these tools focus on C programs, and memory-leaks present different challenges than resource-leaks. For example, features such as Java's exception mechanism, its reliance on try-catch blocks for resource handling, resource aliasing, and the presence of class fields are some challenges in Java resource-leak fixing that do not appear when dealing with memory leaks in C.

Unlike all these specialized repair tools, *RLFixer* focuses on resource-leaks, a problem that has not been tackled by any specialized tool before. Additionally, *RLFixer's* demand-driven design makes it significantly faster than most special-purpose tools. Most other tools use time-budgets of a few minutes or more per program, whereas *RLFixer* finishes in 1 second, plus the 13 seconds it takes for WALA to setup the call-graph, etc.

Linter-based repair tools. Linters scan code for style or coding-convention violations using pattern matching on the AST (Abstract Syntax Tree). Linter-based repair tools [11, 21, 47, 48] use a similar AST pattern-matching approach to apply repairs for a linter's warnings. These tools vary in the coding-conventions they target, and in their method of learning repair patterns. Phoenix [11] mines patches from a corpus of GitHub programs, and uses it to learn generalized repair strategies that are represented as executable programs in a domain specific language (DSL). Styler [47] similarly learns fix patterns for code-formatting violations from a corpus, but it learns using an LSTM neural network. Getafix [9] applies a hierarchical clustering algorithm to effectively summarize fix patterns, and then uses a novel ranking technique based on past human fixes to pick the most plausible fix. TFix [12] formulates the linter-repair problem as a text-to-text prediction task and then uses a pre-trained text-to-text Transformer model to generate fixes. SpongeBugs [48] and Sorald [21] create manually defined fix templates for a handful of linter violations.

The errors targetted by linter-based repair tools are often local and can be represented using AST patterns. Hence, unlike *RLFixer*, their techniques will not work for a more complex bug such as resource-leaks which requires data-flow tracking and an inter-procedural analysis.

Escape analysis. Escape analysis [18, 63] is a research direction that sounds similar to our *resource escape analysis* from Section 3.3, but it has very different designs and goals. Escape analysis characterizes how objects allocated in one region of the program escape to code outside this region. It cares less about the kind of program construct (such as an array or field) it escapes to. On the other hand, our *resource escape analysis* computes the kinds of program constructs (such as a field or parameter) that a resource aliases with, and has no concept of regions. Hence, the two analyses end up having different abstract domains, constraints, and design decisions.

Repairing Android Resource Leaks. Android Resource Leaks are leaks involving event-driven control flow from Android events, and are different from the Java resource leaks discussed in this paper, which involve sequential control flow. Let us take a closer look at how these two kinds of leaks differ to understand why they need different kinds of repair tools. An Android application is an event-driven system with event-handlers responding to a sequence of events such as user-interaction or the application life-cycle events. For example, Android defines the event handlers `onPause` and `onDestroy` for when the user pauses and closes an application, respectively. *Android Resource Leak* detectors [44, 64] model these event sequences and find ones that can leak some Android resource. For example, if a resource is not closed in the `onPause` or `onDestroy` event handlers, we may get an *Android Resource Leak.* Liu et. al [43] prepare a database of such *Android Resource Leaks. Android Resource Leak* repair tools such as [10, 13, 42] then suggest the correct event-handler to close the resource in. Hence, all these leak detection and repair tools for *Android Resource Leaks* focus exclusively on Android's event-driven control flow. On the other hand, tools such as *RLFixer* and Footpatch [61] focus on Java resource leaks resulting from the control-flow in sequential Java code. Thus, they solve a completely different problem than *Android Resource Leak* repair tools.

8 CONCLUSION

Resource leaks are an important bug type that need better tool-support for automated fix suggestions. In this paper, we introduced RLFixer, the first specialized repair tool for resource leaks. We highlighted several challenges for the resource leak problem, including identifying resource-leaks that are infeasible to solve, identifying resource aliases, and constructing fixes that do not modify the semantics of the existing program. We then discussed how *RLFixer* tackles these challenges using a new demand-driven static analysis called *resource escape analysis*. Finally, we experimentally showed that RLFixer repairs a majority of resource-leaks in our benchmarks with near perfect correctness and very low repair time.

There are two interesting future directions that we forsee. The first is to investigate if there are any special cases of field or data-structure escapes that are feasible to fix. The second is applying RLFixer's templates and resource alias analysis to other object oriented languages like Python or C#, which have a similar try-catch-finally exception handling style.

9 DATA AVAILABILITY AND EXPERIMENT REPLICATION

The dataset used for this paper, NJR-1, is publicly available at the following link: (https://doi.org/10.5281/zenodo.3897691). The anonymized artifact for the paper, including the source code, experimental results, and detailed documentation, are publicly available at the following repository: (https://doi.org/10.5281/zenodo.7592371). The artifact also includes a VM image that comes with pre-installed dependencies, and can be used to quickly reproduce the results of the paper by running a few simple scripts.

ACKNOWLEDGMENTS

This work was supported by the U.S. NSF Award 1823360, and the ONR Award N00014-18-1-2037. We also thank the FSE'23 reviewers and Michael Abir for their constructive comments that helped improve the paper.

REFERENCES

[1] 2002. PMD Source Code Analyzer. https://pmd.github.io.

[2] 2015. IBM, "T.J. Watson Libraries for Analysis (WALA),". http://wala.sourceforge.net.

[3] 2017. SpotBugs Static Analysis Tool. https://spotbugs.github.io.

[4] 2019. JavaParser. https://javaparser.org.

[5] 2020. Amazon Codeguru Reviewer. https://aws.amazon.com/codeguru/.

[6] 2022. Common Weakness Enumeration (CWE-400). https://cwe.mitre.org/data/definitions/400.html.

[7] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 289–299. https://doi.org/10.1109/ICSE.2017.34

[8] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2021. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2162–2181. https://doi.org/10.1109/TSE.2019.2944914

[9] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (oct 2019), 27 pages. https://doi.org/10.1145/3360585

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

[11] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

[12] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. https://proceedings.mlr.press/v139/berabi21a.html

[13] Bhargav Nagaraja Bhatt and Carlo A. Furia. 2022. Automated Repair of Resource Leaks in Android Applications. *J. Syst. Softw.* 192, C (oct 2022), 19 pages. https://doi.org/10.1016/j.jss.2022.111417

[14] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11.

[15] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-Fly Patch Validation for All. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1123–1134. https://doi.org/10.1109/ICSE43902.2021.00104

[16] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2017. IntPTI: Automatic Integer Error Repair with Proper-Type Inference. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 996–1001.

[17] Sigmund Cherem and Radu Rugina. 2007. Uniqueness Inference for Compile-Time Object Deallocation. In *Proceedings of the 6th International Symposium on Memory Management* (Montreal, Quebec, Canada) *(ISMM '07)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1296907.1296923

[18] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/320384.320386

[19] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[20] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. 2022. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering* 48, 3 (2022), 835–847. https://doi.org/10.1109/TSE.2020.3004525

[21] Khashayar Etemadi Someoliayi, Nicolas Yves Maurice Harrand, Simon Larsén, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, and Martin Monperrus. 2022. Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–1. https://doi.org/10.1109/TDSC.2022.3167316

[22] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. https://doi.org/10.1145/3540250.3549098

[23] 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 *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 459–470.

[24] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and Resource Leak Defects and Their Repairs in Java Projects. *Empirical Softw. Engg.* 25, 1 (jan 2020), 678–718. https://doi.org/10.1007/s10664-019-09731-8

[25] Denis Gopan, Thomas Reps, and Mooly Sagiv. 2005. A Framework for Numeric Analysis of Array Operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 338–350. https://doi.org/10.1145/1040305.1040333

[26] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 364–375. https://doi.org/10.1145/1133981.1134024

[27] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: Scalable, Precise, and Safe Memory-Error Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 271–283. https://doi.org/10.1145/3377811.3380323

[28] 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* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[29] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[30] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-Violation Fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[31] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681.

[32] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2021. Lightweight and Modular Resource Leak Verification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/3468264.3468576

[33] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 802–811.

[34] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 376–379. https://doi.org/10.1145/3092703.3098225

[35] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[36] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D. Le, and Quyet Thang Huynh. 2022. AutoPruner: Transformer-Based Call Graph Pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 520–532. https://doi.org/10.1145/3540250.3549175

[37] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/3236024.3236079

[38] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2022. NPEX: Repairing Java Null Pointer Exceptions without Tests. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1532–1544. https://doi.org/10.1145/3510003.3510186

[39] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[40] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-Based Approach for Automated Program Repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 511–523. https://doi.org/10.1145/3510003.3510177

[41] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and Generating High Quality Patches for Concurrency Bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 715–726. https://doi.org/10.1145/2950290.2950309

[42] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. 2016. Fixing Resource Leaks in Android Apps with Light-Weight Static Analysis and Low-Overhead Instrumentation. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 342–352. https://doi.org/10.1109/ISSRE.2016.15

[43] 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 (12 2019), 1–49. https://doi.org/10.1007/s10664-019-09715-8

[44] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. 2016. Understanding and Detecting Wake Lock Misuses for Android Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 396–409. https://doi.org/10.1145/2950290.2950297

[45] 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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[46] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (jan 2016), 298–312. https://doi.org/10.1145/2914770.2837617

[47] Benjamin Loriot, Fernanda Madeiral, and Martin Monperrus. 2022. Styler: Learning Formatting Conventions to Repair Checkstyle Violations. *Empirical Softw. Engg.* 27, 6 (nov 2022), 36 pages. https://doi.org/10.1007/s10664-021-10107-0

[48] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. 2020. SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. *Journal of Systems and Software* 168 (2020), 110671. https://doi.org/10.1016/j.jss.2020.110671

[49] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[50] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. 2021. IntRepair: Informed Repairing of Integer Overflows. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2225–2241. https://doi.org/10.1109/TSE.2019.2946148

[51] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[52] H.D. Pande, W.A. Landi, and B.G. Ryder. 1994. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering* 20, 5 (1994), 385–403. https://doi.org/10.1109/32.286418

[53] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (mar 2018), 58–66. https://doi.org/10.1145/3188720

[54] Marija Selakovic and Michael Pradel. 2015. Poster: Automatically Fixing Real-World JavaScript Performance Bugs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 811–812. https://doi.org/10.1109/ICSE.2015.260

[55] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, USA, 124–135. https://doi.org/10.1109/DSN.2014.25

[56] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-Driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817

[57] 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* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 535–544. https://doi.org/10.1145/1806799.1806876

[58] Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. 2020. *NJR-1 Dataset*. https://doi.org/10.5281/zenodo.4839913

[59] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. 2022. Striking a Balance: Pruning False-Positives from Static Call Graphs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2043–2055. https://doi.org/10.1145/3510003.3510166

[60] Akshay Utture and Jens Palsberg. 2022. Fast and Precise Application Code Analysis Using a Partial Library. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 934–945. https://doi.org/10.1145/3510003.3510046

[61] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[62] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[63] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 187–206. https://doi.org/10.1145/320384.320400

[64] 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 Transactions on Software Engineering* 42, 11 (2016), 1054–1076. https://doi.org/10.1109/TSE.2016.2547385

[65] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. https://doi.org/10.1145/3540250.3549101

[66] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 512–523. https://doi.org/10.1109/ICSE.2019.00063

[67] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. https://doi.org/10.1145/3510003.3510222

Received 2023-02-02; accepted 2023-07-27