Lightweight and Modular Resource Leak Checking (Extended Version)

Narges Shadab[1], Pritam Gharat[2], Shrey Tiwari[3], Michael D. Ernst[4], Martin Kellogg[5], Shuvendu K. Lahiri[6], Akash Lal[2], Manu Sridharan[1]

[1*]University of California, Riverside, USA.
[2*]Microsoft Research, India.
[3*]Carnegie Mellon University, USA.
[4*]University of Washington, USA.
[5*]New Jersey Institute of Technology, USA.
[6*]Microsoft Research, USA.

Contributing authors: Nshad001@ucr.edu; pritamgharat@microsoft.com; Smtiwari@andrew.CMU.edu; mernst@cs.washington.edu; martin.kellogg@njit.edu; shuvendu.lahiri@microsoft.com; akashl@microsoft.com; manu@cs.ucr.edu;

Abstract

A resource leak occurs when a program allocates a resource but fails to deallocate it. Resource leaks cause resource starvation, slowdowns, and crashes. Previous techniques to prevent resource leaks are either unsound, imprecise, inapplicable to existing code, slow, or a combination of these. We present a resource leak checking approach that is applicable, sound, precise, and fast. Our key insight is that leak detection can be reduced to an accumulation problem, a class of typestate problems amenable to sound and modular checking without whole-program alias analysis. The precision of an accumulation analysis can be improved with targeted aliasing information, and we augmented our baseline checker with three such novel techniques: a lightweight ownership transfer system; a specialized resource alias analysis; and a system to create a fresh obligation when a non-final resource field is updated. Our approach occupies a unique slice of the design space: it is sound and runs relatively quickly (taking minutes on programs that a state-of-the-art approach took hours to analyze). Moreover, our approach generalizes to multiple analysis backends. The Resource Leak Checker revealed 49 real resource leaks in widely-deployed software; RLC# revealed 24 real resource leaks in five programs, including three Azure microservices. Both implementations scale well, have manageable false positive rates (comparable to heuristic bug-finders), and impose only a small annotation burden (about 1/6000 LoC) for developers. This is an extended version of an ESEC/FSE 2021 publication. The key new contribution of this work is the introduction of the RLC# tool for checking of C# code. We describe the implementation of RLC# as a reachability-based analysis built on CodeQL (quite different than the previous approach) and present an evaluation of its effectiveness.

Keywords: Resource Leak, CodeQL, Checker Framework, Static Analysis

1 Introduction

A resource leak occurs when some finite resource managed by the programmer is not explicitly disposed of. In an unmanaged language like C, that explicit resource might be memory; in a managed language like Java, it might be a file descriptor, a socket, or a database connection. Resource leaks continue to cause severe failures, even in modern, heavily-used applications [16]. This state-of-the-practice does not differ much from two decades ago [45]. Microsoft engineers consider resource leaks to be one of the most significant development challenges [25]. Preventing resource leaks remains an urgent, difficult, open problem.

An ideal tool to prevent resource leaks would:

- be *applicable* to existing code with few changes;
- be *sound*, so that undetected resource leaks do not slip into the program;
- be *precise*, so that developers are not bothered by excessive false positive warnings; and
- be *fast*, so that it scales to real-world programs and developers can use it regularly.

Extant approaches fail at least one of these criteria. Language-based features may not apply to all uses of resource variables: Java's try-with-resources statement [28], for example, can only close resource types that implement the `java.lang.AutoCloseable` interface, and cannot handle common resource usage patterns that span multiple procedures. Heuristic bug-finding tools for leaks, such as those built into Java IDEs including Eclipse [9] and IntelliJ IDEA [19], are fast and applicable to legacy code, but they are unsound. Inter-procedural typestate or dataflow analyses [42, 49] achieve more precise results—though they usually remain unsound—but their whole-program analysis can require hours to analyze a large-scale Java program. Finally, ownership type systems [5] as employed in languages like Rust [23] can prevent nearly all resource leaks (see section 10.2), but using them would require a significant rewrite for a legacy codebase, a substantial task which is often infeasible.

This paper presents an approach to resource leak checking that is simultaneously applicable, sound, precise, and fast. It is an extended version of Kellogg et al. [21], which introduced the core techniques of the approach and presented and evaluated an implementation for checking Java code. In this extended version, we introduce a new tool RLC#, which shares the core analysis approach of Kellogg et al., but features an entirely different style of implementation. RLC# finds resource leaks in C# programs, utilizing CodeQL's dataflow module to perform a reachability-style analysis for checking leaks.

The goal of a leak detector for a Java-like language is to ensure that required methods (such as `close()`) are called on all relevant objects; we deem this a *must-call* property. Verifying a must-call property requires checking that required methods (or *must-call obligations*) have been called at any point where an object may become unreachable. A static verifier does this by computing an under-approximation of invoked methods. Our key insight is that checking of must-call properties is an *accumulation problem*, and hence does not require heavyweight whole-program analysis.

An accumulation analysis [22] is a special-case of typestate analysis [37]. Typestate analysis attaches a finite-state machine (FSM) to each program element of a given type, and transitions the state of the FSM whenever a relevant operation is performed. In an accumulation analysis, the order of operations performed cannot change what is subsequently permitted, and executing more operations cannot add additional restrictions. Unlike arbitrary typestate analyses, accumulation analyses can be built in a sound, modular fashion without *any* whole-program alias analysis, improving scalability and usability.

Prior work [20] presented an accumulation analysis for verifying that certain methods are invoked on each object before a specific call (e.g., `build()`). Resource leak checking is similar in that certain methods must be invoked on each object before it becomes unreachable. An object becomes unreachable when its references go out of scope or are overwritten. By making an analogy between object-unreachability points and method calls, we show that resource leak checking is an accumulation problem and hence is amenable to sound, modular, and lightweight analysis.

There are two key challenges for this leak-checking approach. First, due to subtyping, the declared type of a reference may not accurately represent its must-call obligations; we devised a simple type system to soundly capture these obligations. Second, the approach is sound, but highly imprecise without targeted reasoning about aliasing. The most important patterns to handle are:

- copying of resources via parameters and returns, or storing of resources in final fields (the RAII pattern [38]);
- wrapper types, which share their must-call obligations with one of their fields; and,
- resources in non-final fields, which might be lazily initialized or written more than once.

To address this need, we extend it with three sound techniques to improve precision:

- a lightweight ownership transfer system, which indicates which reference is responsible for resolving a must-call obligation. Unlike typical ownership type systems, our approach does not impact the privileges of non-owning references.
- resource aliasing, for cases in which a resource's must-call obligations can be resolved by closing one of multiple references.
- a system for creating new obligations outside constructors, which allows our system to handle lazy initialization or re-initialization.

Variants of some of these ideas exist in previous work. We bring them together in a general, modular manner, with full verification and the ability for programmers to easily extend checking to their own types and must-call properties. Our approach occupies a novel point in the design space for a leak detector: unlike most prior work, it is sound; it is an order of magnitude faster than state-of-the-art whole-program analyses; it has a false positive rate similar to a state-of-the-practice heuristic bug-finder; and, though it does require manual annotations from the programmer, its annotation burden is reasonable: about 1 annotation for every 6,000 lines of non-comment, non-blank code. Moreover, it can be implemented on top of multiple analysis backends with different underlying analysis strategies: we present both a pluggable typechecker targeting Java and an implementation of our approach that targets C#, using the reachability engine from CodeQL [6] as a backend.

The contributions of this work shared with the previous paper [21] are:

- the insight that the resource leak problem is an accumulation problem, and an analysis approach based on this fact (section 2).
- three innovations that improve the precision of our analysis via targeted reasoning about aliasing: a lightweight ownership transfer system (section 3), a lightweight resource-alias tracking analysis (section 4), and a system for handling lazy or multiple initialization (section 5).
- an open-source implementation for Java, called the Resource Leak Checker or RLC (section 6).
- an empirical evaluation of the RLC: case studies on heavily-used Java programs (section 7.1), an ablation study that shows the contributions of each innovation to the RLC's precision (section 7.2), and a comparison to other state-of-the-art approaches that demonstrates the unique strengths of our approach (section 7.3).

The key new contributions of this extended version are:

- RLC#, a new implementation of our resource leak checking approach for C#. The implementation of RLC# is based on reachability analysis, leveraging the dataflow module of CodeQL [6], quite a different approach from the RLC for Java. This new approach is described in sections 8.2 through 8.4.
- A comparison of the RLC and RLC# approaches, including a discussion of the challenges arising from the differing designs of C# and Java (section 8.6).
- An experimental evaluation of RLC#, showing its effectiveness (section 8.7).

2 Core Accumulation Analysis

This section presents a sound, modular, accumulation-based resource leak checker ("the Resource Leak Checker" or "the RLC"). Sections 3–5 soundly enhance its precision. The RLC is composed of three cooperating analyses:

1. a taint tracking type system (section 2.2) computes a conservative *overapproximation* of the set of methods that might need to be called on each expression in the program.
2. an accumulation type system (section 2.3) computes a conservative *underapproximation* of the set of methods that are actually called on each expression in the program.
3. a dataflow analysis (section 2.4) checks consistency of the results of the two above-mentioned type systems and provides a platform for targeted alias reasoning. It issues an error if some method that might need to be called on an expression is not always invoked before the expression goes out of scope or is overwritten.

2.1 Background on Pluggable Types

Sections 2.2 and 2.3 describe *pluggable type systems* [13] that are layered on top of the type system of the host language. Types in a pluggable type system are composed of two parts: a *type qualifier* and a base type. The type qualifier is the part of the type that is unique to the pluggable type system; the base type is a type from the host language. Our implementation is for Java (see section 6), so we use the Java syntax for type qualifiers: "`@`" before a type indicates that it is a type qualifier, and a type without "`@`" is a base type. This paper sometimes omits the base type when it is obvious from the context.

A type system checks the programmer-written types. Our system requires the programmer to write types on method signatures, but within method bodies it uses flow-sensitive type refinement, a dataflow analysis that performs type inference. This permits an expression to have different types on different lines of the program.

2.2 Tracking Must-Call Obligations

The Must Call type system tracks which methods might need to be called on a given expression. This type system—and our entire analysis—is not specific to resource leaks. Another such property is that the `build()` method of a builder [14] should always be called.

The Must Call type system supports two qualifiers: `@MustCall` and `@MustCallUnknown`. The `@MustCall` qualifier's arguments are the methods that the annotated value must call. The declaration `@MustCall({"a"}) Object obj` means that before `obj` is deallocated, `obj.a()` might need to be called. The RLC conservatively requires all these methods to be called, and it issues a warning if they are not.

For example, consider fig. 1. The expression `null` has type `@MustCall({})`—it has no obligations to call any particular methods—so `s` has that type after its initialization. The `new` expression has type `@MustCall("close")`, and therefore `s` has that type after the assignment. At the start of the `finally` block, where both values for `s` flow, the type of `s` is their least upper bound, which is `@MustCall("close")`.

Part of the type hierarchy appears in fig. 2. All types are subtypes of `@MustCallUnknown`. The

```
Socket s = null;
try {
  s = new Socket(myHost, myPort);
} catch (Exception e) { // do nothing
} finally {
  if (s != null) {
    s.close();
  }
}
```

Fig. 1: A safe use of a `Socket` resource.

subtyping rule for `@MustCall` type qualifiers is:

$$\frac{A \subseteq B}{\texttt{@MustCall(A)} \sqsubseteq \texttt{@MustCall(B)}}$$

The default type qualifier is `@MustCall({})` for base types without a programmer-written type qualifier.[1] Our implementation provides JDK annotations which require that every object of `Closeable` type must have the `close()` method called before it is deallocated, with exceptions for types that do not have an underlying resource, e.g., `ByteArrayOutputStream`.

2.3 Tracking Called Methods

The Called Methods type system tracks a conservative underapproximation of which methods have been called on an expression. It is an extension of a similar system from prior work [20]. The primary difference in our version is that a method is considered called even if it throws an exception—a necessity in Java because the `close()` method in `java.io.Closeable` is specified to possibly throw an `IOException`. In the prior work, a method was only considered "called" when it terminated successfully. The remainder of this section is a brief summary of the prior work [20].

The checker is an accumulation analysis whose accumulation qualifier is `@CalledMethods`. The type `@CalledMethods(`A`) Object` represents an object on which the methods in the set A have definitely been called; other methods not in A might also have been called. The subtyping rule is:

$$\frac{B \subseteq A}{\texttt{@CalledMethods(A)} \sqsubseteq \texttt{@CalledMethods(B)}}$$

[1] For unannotated local variable types, flow-sensitive type refinement infers a qualifier.

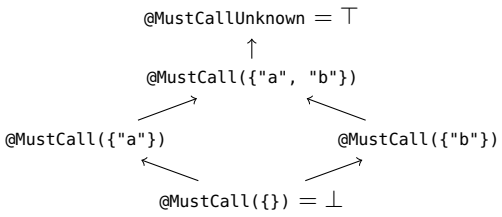

Fig. 2: Part of the `MustCall` type hierarchy for representing which methods must be called; the full hierarchy is a lattice of arbitrary size. If an expression's type has qualifier `@MustCall({"a", "b"})`, then the methods "a" and "b" might need to be called before the expression is deallocated. Arrows represent subtyping relationships.

The top type is `@CalledMethods({})`. The qualifier `@CalledMethodsBottom` is a subtype of every `@CalledMethods` qualifier.

Thanks to flow-sensitive type refinement, Called Methods types are inferred within method bodies. In fig. 1 the type of `s` is initially `@CalledMethods({})`, but it transitions to `@CalledMethods("close")` after the call to `close`.

2.4 Consistency Checking

Given `@MustCall` and `@CalledMethods` types, the Must Call Consistency Checker ensures that the `@MustCall` methods for each object are always invoked before it becomes unreachable, via an intraprocedural dataflow analysis. We employ dataflow analysis to enable targeted reasoning about aliasing, crucial for precision. Here, we present a simple, sound version of the analysis. Sections 3–5 describe sound enhancements to this approach.

Language. For simplicity, we present the analysis over a simple assignment language in three-address form. An expression e in the language is `null`, a variable `p`, a field read `p.f`, or a method call `m(p1,p2,...)` (constructor calls are treated as method calls). A statement s takes one of three forms: `p = e`, where e is an expression; `p.f = p'`, for a field write; or `return p`. Methods are represented by a control-flow graph (CFG) where nodes are statements and edges indicate possible control flow. We elide control-flow predicates because the consistency checker is path-insensitive.

For a method CFG, *CFG.statements* is the statements, *CFG.formals* is the formal parameters, *CFG.entry* is its entry node, *CFG.exit* is its exit node, and *CFG.succ* is its successor relation. For a statement s of the form `p = e`, $s.LHS = p$ and $s.RHS = e$.

Algorithm 1 Finding unfulfilled `@MustCall` obligations in a method. Algorithm 2 defines helper functions.

```
1:  procedure FindMissedCalls(CFG)
2:    // D maps each statement s to a set of dataflow
3:    // facts reaching s. Each fact is of the form ⟨P, e⟩,
4:    // where P is a set of variables that must-alias e
5:    // and e is an expression with a nonempty must-
6:    // call obligation.
7:    D ← InitialObligations(CFG)
8:    while D has not reached fixed point do
9:      for s ∈ CFG.statements, ⟨P, e⟩ ∈ D(s) do
10:       if s is exit then
11:         report a must-call violation for e
12:       else if ¬MCSatisfiedAfter(P, s) then
13:         kill ← s assigns a variable ? {s.LHS} : ∅
14:         gen ← CreatesAlias(P, s) ? {s.LHS} : ∅
15:         N ← (P − kill) ∪ gen
16:         ∀t ∈ CFG.succ(s) . D(t) ← D(t) ∪ {⟨N, e⟩}
17: procedure InitialObligations(CFG)
18:   D ← {s ↦ ∅ | s ∈ CFG.statements}
19:   for p ∈ CFG.formals,
          t ∈ CFG.succ(CFG.entry) do
20:     if HasObligation(p) then
21:       D(t) ← D(t) ∪ {⟨{p}, p⟩}
22:   for s ∈ CFG.statements of the form
          p = m(p1, p2, ...) do
23:     ∀t ∈ CFG.succ(s) .
          D(t) ← D(t) ∪ FactsFromCall(s)
24:   return D
```

Pseudocode. Algorithm 1 gives the pseudocode for the basic version of our checker, with helper functions in algorithm 2. At a high level, the dataflow analysis computes a map D from each statement s in a CFG to a set of facts of the form $\langle P, e\rangle$, where P is a set of variables and e is an expression. The meaning of D is as follows: if $\langle P, e\rangle \in D(s)$, then e has a declared `@MustCall` type, and all variables in P are *must aliases* for the value of e at the program point before s. Computing a set of must aliases is useful since any must alias may be used to satisfy the must-call obligation of e. Using D, the analysis finds any e that does not have its `@MustCall` obligation fulfilled, and reports an error.

Algorithm 1 proceeds as follows. Line 7 invokes InitialObligations to initialize D. Only formal parameters or method calls can introduce obligations to be checked (reads of local variables or fields cannot). The fixed-point loop iterates over all facts $\langle P, e\rangle$ present in any $D(s)$ (our implementation uses a worklist for efficiency). If s is the exit node (line 10), the obligation for e has not been

Algorithm 2 Helper functions for algorithm 1. Except for MCAfter and CMAfter, all functions will be replaced with more sophisticated versions in sections 3–5.

```
1:  // Does e introduce a must-call obligation to check?
2:  procedure HasObligation(e)
3:    return e has a declared @MustCall type
4:  // s must be a call statement p = m(p1, p2, ...)
5:  procedure FactsFromCall(s)
6:    p ← s.LHS, c ← s.RHS
7:    return HasObligation(c) ? {⟨{p}, c⟩} : ∅
8:  // Is must-call obligation for P satisfied after s?
9:  procedure MCSatisfiedAfter(P, s)
10:   return ∃p ∈ P.
          MCAfter(p, s) ⊆ CMAfter(p, s)
11: // Does s introduce a must-alias for a var in P?
12: procedure CreatesAlias(P, s)
13:   return ∃q ∈ P . s is of the form p = q
14: procedure MCAfter(p, s)
15:   return methods in @MustCall type of p after s
16: procedure CMAfter(p, s)
17:   return methods in @CalledMethods type of p after s
```

satisfied, and an error is reported. Otherwise, the algorithm checks if the obligation for e is satisfied after s (line 12). For the basic checker, MCSatisfiedAfter in algorithm 2 checks whether there is some $p \in P$ such that after s, the set of methods in p's `@MustCall` type are contained in the set of methods in its `@CalledMethods` type; if true, all `@MustCall` methods have already been invoked. This check uses the inferred flow-sensitive `@MustCall` and `@CalledMethods` qualifiers described in sections 2.2 and 2.3.

If the obligation for e is not yet satisfied, the algorithm propagates the fact to successors with an updated set N of must aliases. N is computed in a standard gen-kill style on lines 13–15. The kill set simply consists of whatever variable (if any) appears on the left-hand side of s. The gen set is computed by checking if s creates a new must alias for some variable in P, using the CreatesAlias routine. Since our analysis is accumulation, CreatesAlias could simply return false without impacting soundness. In algorithm 2, CreatesAlias handles the case of a variable copy where the right-hand side is in P. (Section 4 presents more sophisticated handling.) Finally, line 16 propagates the new fact to successors. The process continues until D reaches a fixed point.

Example. To illustrate our analysis, fig. 3 shows a simple program (irrelevant details elided) and its corresponding CFG. The CFG shows the dataflow

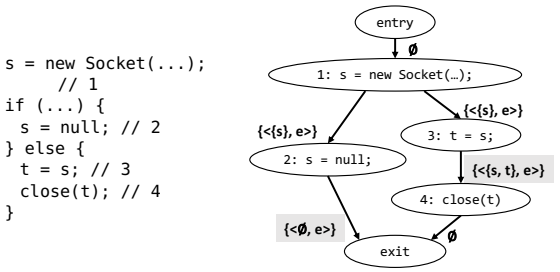

Fig. 3: CFG and code for illustrating algorithm 1. 'e' is "`new Socket(...)`". Non-shaded facts are created by InitialObligations, and shaded facts are propagated by the fixed-point loop.

facts propagated along each edge. For initialization, statement 1 introduces the fact $\langle\{s\}, e\rangle$ (where e is the `new Socket(...)` call) to $D(2)$ and $D(3)$. At statement 2, s is killed, causing $\langle\emptyset, e\rangle$ to be added to $D(exit)$. This leads to an error being reported for statement 1, as the socket is not closed on this path. Statement 3 creates a must alias t for s, causing $\langle\{s, t\}, e\rangle$ to be added to $D(4)$. For statement 4, MCSatisfiedAfter($\{s, t\}$, `close(t)`) holds, so no facts are propagated from 4 to *exit*.

3 Lightweight Ownership

Section 2 describes a sound accumulation-based checker for resource leaks. However, that checker often encounters false positives in cases where an `@MustCall` obligation is satisfied in another procedure via parameter passing, return values, or object fields. Consider the following code that safely closes a `Socket`:

```
void example(String myHost, int myPort) {
  Socket s = new Socket(myHost, myPort);
  closeSocket(s);
}
void closeSocket(@Owning @MustCall("close") Socket t) {
  t.close();
}
```

The `closeSocket()` routine takes ownership of the socket—that is, it takes responsibility for closing it. The checker described by section 2 would issue a false positive on this code, because it would warn when `s` goes out of scope at the end of `example()`.

This section describes a *lightweight ownership transfer* technique for reducing false positives in such cases. Programmers write annotations like `@Owning` that transfer an obligation from one expression to another. Programmer annotations

cannot introduce any checker unsoundness; at worst, incorrect `@Owning` annotations will cause false positive warnings. Unlike an ownership type system like Rust's (see section 10.2), lightweight ownership transfer imposes no restrictions on what operations can be performed through an alias, and hence has a minimal impact on the programming model.

3.1 Ownership Transfer

`@Owning` is a declaration annotation, not a type qualifier; it can be written on a declaration such as a parameter, return, field, etc., but not on a type. A pseudo-assignment to an `@Owning` lvalue transfers the right-hand side's `@MustCall` obligation. More concretely, in the Must Call Consistency Checker (section 2.4), at a pseudo-assignment to an lvalue with an `@Owning` annotation, the right-hand side's `@MustCall` obligation is treated as satisfied.

The MCSatisfiedAfter(P, s) and HasObligation(e) procedures of algorithm 2 are enhanced for ownership transfer as follows:

---

procedure MCSatisfiedAfter(P, s)
return $\exists p \in P$.
MCAfter(p, s) $\subseteq$ CMAfter(p, s)
$\vee$ (s is `return p` $\wedge$ OwningReturn(CFG))
$\vee$ PassedAsOwningParam(s, p)
$\vee$ (s is `q.f = p` $\wedge$ `f` is `@Owning`)
procedure HasObligation(e)
return e has a declared `@MustCall` type and e's declaration is `@Owning`
procedure OwningReturn(CFG)
return CFG's return declaration is `@Owning`
procedure PassedAsOwningParam(s,p)
return s passes p to an `@Owning` parameter of its callee

Section 3.2 discusses checking of `@Owning` fields.

Constructor returns are always `@Owning`. The RLC's default for unannotated method returns is `@Owning`, and for unannotated parameters and fields is `@NotOwning`. These assumptions coincide well with coding patterns we observed in practice, reducing the annotation burden for programmers. Further, this treatment of parameter and return types ensures sound handling of unannotated third-party libraries: any object returned from such a library is tracked by default, and the checker never assumes that passing an object to an unannotated library satisfies its obligations.

3.2 Final Owning Fields

Class-level checking is required for `@Owning` fields, as the code satisfying their `@MustCall` obligations usually spans multiple procedures. This section handles final fields,[2] which cannot be overwritten after initialization of the enclosing object. When checking non-final fields, the checker must ensure that overwriting the field is safe (see Section 5.1).

For final fields, our checking enforces the "resource acquisition is initialization (RAII)" programming idiom [38]. Some destructor-like method `d()` must ensure the field's `@MustCall` obligation is satisfied, and the enclosing class must have an `@MustCall("d")` obligation to ensure the destructor is called. More formally, consider a final `@Owning` field f declared in class C, where f has type `@MustCall("m")`. To modularly verify that f's `@MustCall` obligation is satisfied, the RLC checks the following conditions:

1. All C objects must have a type `@MustCall("d")` for some method `C.d()`.
2. `C.d()` must always invoke `this.f.m()`, thereby satisfying f's `@MustCall` obligation.

Condition 1 is checked by inspecting the `@MustCall` annotation on class C. Condition 2 is checked by requiring an appropriate `@EnsuresCalledMethods` postcondition annotation on `C.d()`, which is then enforced by section 2.3's checker.

4 Resource aliasing

This section introduces a sound, lightweight, specialized must-alias analysis that tracks *resource alias* sets—sets of pointers that definitely correspond to the same underlying system resource. Closing one alias also closes the others. Thus, the RLC can avoid issuing false positive warnings about resources that have already been closed through a resource alias.

4.1 Wrapper Types

Java programs extensively use *wrapper types*. For example, the Java `BufferedOutputStream` wrapper adds buffering to some delegate `OutputStream`, which may or may not represent a resource that

[2]The Resource Leak Checker considers all static fields as non-owning, so no assignment to them can fulfill a must-call obligation. Our case studies showed no assignments of expressions with non-empty must-call obligations to static fields. Handling owning static fields is left for future work.

needs closing. The wrapper's `close()` method invokes `close()` on the delegate. Wrapper types introduce two additional complexities for `@MustCall` checking:

1. If a delegate has no `@MustCall` obligation, the corresponding wrapper object should also have no obligation.
2. Satisfying the obligation of *either* the wrapped object or the wrapper object is sufficient.

For example, if a `BufferedOutputStream` b wraps a stream with no underlying resource (e.g., a `ByteArrayOutputStream`), b's `@MustCall` obligation should be empty, as b has no resource of its own. By contrast, if b wraps a stream managing a resource, like a `FileOutputStream` f, then `close()` must be invoked on *either* b or f.

Previous work has shown that reasoning about wrapper types is required to avoid excessive false positive and duplicate reports [42, 9]. Wrapper types in earlier work were handled with hard-coded specifications of which library types are wrappers, and heuristic clustering to avoid duplicate reports for wrappers [42]. Our technique handles wrapper types more generally by tracking *resource aliases*. Two references r_1 and r_2 are resource aliases if r_1 and r_2 are must-aliased pointers, or if satisfying r_1's `@MustCall` obligation also satisfies r_2's obligation and vice-versa.

Introducing resource aliases. To indicate where an API method creates a resource-alias relationship between distinct objects, the programmer writes a pair of `@MustCallAlias` qualifiers: one on a parameter of a method, and another on its return type. For example, one constructor of `BufferedOutputStream` is:

```
@MustCallAlias BufferedOutputStream(@MustCallAlias
    OutputStream arg0);
```

`@MustCallAlias` annotations are verified, not trusted; see section 4.3.

At a call site to an `@MustCallAlias` method, there are two effects. First, the must-call type of the method call's return value is the same as that of the `@MustCallAlias` argument. If the type of the argument has no must-call obligations (like a `ByteArrayOutputStream`), the returned wrapper has no must-call obligations.

Second, the Must Call Consistency Checker (section 2.4) treats the `@MustCallAlias` parameter and return as aliases. For our section 2.4 pseudocode, this version of CreatesAlias from algorithm 2 handles resource aliases:

---

procedure CreatesAlias(P, s)
  return $\exists$`q` $\in P$. s is of the form `p = q`
    $\vee$ IsMustCallAliasParam(s, q)
procedure IsMustCallAliasParam(s, p)
  return s passes p to an `@MustCallAlias` parameter of its callee

4.2 Beyond Wrapper Types

`@MustCallAlias` can also be employed in scenarios beyond direct wrapper types, a capability not present in previous work on resource leak detection. In certain cases, a resource gets shared between objects via an intermediate object that cannot directly close the resource. For example, `java.io.RandomAccessFile` (which must be closed) has a method `getFd()` that returns a `FileDescriptor` object for the file. This file descriptor cannot be closed directly—it has no `close()` method. However, the descriptor can be passed to a wrapper stream such as `FileOutputStream`, which if closed satisfies the original must-call obligation. By adding `@MustCallAlias` annotations to the `getFd()` method, our technique can verify code like the below (adapted from Apache Hadoop [39]):

```
RandomAccessFile file = new RandomAccessFile(myFile,
    "rws");
FileInputStream in = null;
try {
  in = new FileInputStream(file.getFD());
  // do something with in
  in.close();
} catch (IOException e){
  file.close();
}
```

Because the must-call obligation checker (section 2.2) treats `@MustCallAlias` annotations polymorphically, regardless of the associated base type, the RLC can verify that the same resource is held by the `RandomAccessFile` and the `FileInputStream`, even though it is passed via a class without a `close()` method.

4.3 Verification of `@MustCallAlias`

A pair of `@MustCallAlias` annotations on `m`'s return type and its parameter `p` can be verified if either of the following holds:

1. `p` is passed to another method or constructor in an `@MustCallAlias` position, and `m` returns that method's result, or the call is a `super()` constructor call annotated with `@MustCallAlias`.

2. `p` is stored in an `@Owning` field of the enclosing class. (`@Owning` field verification is described in sections 3.2 and 5.1.)

These verification procedures permit a programmer to soundly specify a resource-aliasing relationship in their own code, unlike prior work that relied on a hard-coded list of wrapper types.

5 Creating New Obligations

Every constructor of a class that has must-call obligations implicitly creates obligations for the newly-created object. However, non-constructor methods may also create obligations when re-assigning non-final owning fields or allocating new system-level resources. To handle such cases soundly, we introduce a method post-condition annotation, `@CreatesMustCallFor`, to indicate expressions for which an obligation is created at a call. At each call-site of a method annotated as `@CreatesMustCallFor(`*expr*`)`, the RLC removes any inferred Called Methods information about *expr*, reverting to `@CalledMethods({})`.

When checking a call to a method annotated as `@CreatesMustCallFor(`*expr*`)`, the Must Call Consistency Checker (1) treats the `@MustCall` obligation of *expr* as *satisfied*, and (2) creates a fresh obligation to check. We update the FactsFromCall and MCSatisfiedAfter procedures of algorithm 2 as follows ([...] stands for the cases shown previously, including those in section 3.1):

---

procedure FactsFromCall(s)
$p \leftarrow s.LHS, c \leftarrow s.RHS$
return $\{\langle\{p_i\}, c\rangle \mid p_i \in \text{CMCFTargets}(c)\}$
$\cup$ (HasObligation(c) ? $\{\langle\{p\}, c\rangle\} : \emptyset$)
procedure MCSatisfiedAfter(P, s)
return $\exists p \in P.$ [...] $\vee\ p \in$ CMCFTargets(s)
procedure CMCFTargets(c)
return { p_i | p_i passed to an `@CreatesMustCallFor` target for c's callee }

This change is sound: the checker creates a new obligation for calls to `@CreatesMustCallFor` methods, and the must-call obligation checker (section 2.2) ensures the `@MustCall` type for the target will have a *superset* of any methods present before the call. There is an exception to this check: if an `@CreatesMustCallFor` method is invoked within a method that has an `@CreatesMustCallFor` annotation with the same target—imposing the obligation on its caller—then the new obligation can be treated as satisfied immediately.

5.1 Non-Final, Owning Fields

`@CreatesMustCallFor` allows the RLC to verify uses of non-final fields that contain a resource, even if they are re-assigned. Consider the following example:

```
@MustCall("close") // sets default qual. for uses of
      SocketContainer
class SocketContainer {
  private @Owning Socket sock;
  public SocketContainer() { sock = ...; }
  void close() { sock.close() };
  @CreatesMustCallFor("this")
  void reconnect() {
    if (!sock.isClosed()) {
      sock.close();
    }
    sock = ...;
  }
}
```

In the lifetime of a `SocketContainer` object, `sock` might be re-assigned arbitrarily many times: once at each call to `reconnect()`. This code is safe, however: `reconnect()` ensures that `sock` is closed before re-assigning it.

The RLC must enforce two new rules to ensure that re-assignments to non-final, owning fields like `sock` in the example above are sound:

- any method that re-assigns a non-final, owning field of an object must be annotated with an `@CreatesMustCallFor` annotation that targets that object.
- when a non-final, owning field f is re-assigned at statement s, its inferred `@MustCall` obligation must be contained in its `@CalledMethods` type at the program point before s.

The first rule ensures that `close()` is called after the last call to `reconnect()`, and the second rule ensures that `reconnect()` safely closes `sock` before re-assigning it. Because calling an `@CreatesMustCallFor` method like `reconnect()` resets local type inference for called methods, calls to `close` before the last call to `reconnect()` are disregarded.

5.2 Unconnected Sockets

`@CreatesMustCallFor` can also handle cases where object creation does not allocate a resource, but the object will allocate a resource later in its lifecycle. Consider the no-argument constructor to `java.net.Socket`. This constructor does not allocate an operating system-level socket, but instead just creates the container object, which permits the programmer to e.g. set options which will be used when creating the physical socket. When such

Table 1: Verifying the absence of resource leaks. Throughout, "LoC" is lines of non-comment, non-blank Java code. "Resources" is the number of resources created by the program. "Resource leaks" are true positive warnings. "False positives" are where the tool reported a potential leak, but manual analysis revealed that no leak is possible. "Annotations" and "code changes" are the number of edits to program text; see section 7.1.3 for details. "Wall-clock time" is the median of five trials. "zookeeper", "hadoop", and "hbase" are Apache projects.

	LoC	Resources	Resource leaks	False positives	Annotations	Code changes	Wall-clock time
zookeeper:zookeeper-server	45,248	177	13	48	122	5	1m 24s
hadoop:hadoop-hdfs-project/hadoop-hdfs	151,595	365	23	49	117	13	16m 21s
hbase:hbase-server, hbase-client	220,828	55	5	22	45	5	7m 45s
plume-lib/plume-util	10,187	109	8	2	2	19	0m 15s
Total	427,858	706	49	121	286	42	-

a `Socket` is created, it initially has no must-call obligation; it is only when the `Socket` is actually connected via a call to a method such as `bind()` or `connect()` that the must-call obligation is created.

If all `Sockets` are treated as `@MustCall({"close"})`, a false positive would be reported in code such as the below, which operates on an unconnected socket (simplified from real code in Apache Zookeeper [40]):

```
static Socket createSocket() {
  Socket sock = new Socket();
  sock.setSoTimeout(...);
  return sock;
}
```

The call to `setSoTimeout` can throw a `SocketException` if the socket is actually connected when it is called. Using `@CreatesMustCallFor`, however, the RLC can soundly show that this socket is not connected: the type of the result of the no-argument constructor is `@MustCall({})`, and `@CreatesMustCallFor` annotations on the methods that actually allocate the socket—`connect()` or `bind()`—enforce that as soon as the socket is open, it is treated as `@MustCall("close")`.

6 Java Implementation

We implemented the Resource Leak Checker on top of the Checker Framework [29], an industrial-strength framework for building pluggable type systems for Java. The checkers which propagate and infer `@MustCall` and `@CalledMethods` annotations are implemented directly as Checker Framework type-checkers. The Must Call Consistency Checker (algorithm 1) is implemented as a post-analysis pass over the control-flow graph produced by the Checker Framework's dataflow analysis, and is invoked when the other two checkers terminate. The framework provides the checkers with flow-sensitive local type inference, support for Java generics and qualifier polymorphism, and other conveniences. Our implementation is open-source and distributed as part of the Checker Framework (https://checkerframework.org/) from version 3.15.0.

7 Evaluation

Our evaluation has three parts:

- case studies on open-source projects, which show that our approach is scalable and finds real resource leaks (section 7.1).
- an evaluation of the importance of lightweight ownership, resource aliasing, and obligation creation (section 7.2).
- a comparison to previous leak detectors: both a heuristic bug finder and a whole-program analysis (section 7.3).

All code and data for our experiments described in this section, including the RLC's implementation, experimental machinery, and annotated versions of our case study programs, are publicly available at https://doi.org/10.5281/zenodo.4902321.

7.1 Case Studies

We selected 3 open-source projects that were analyzed by prior work [49]. For each, we selected and analyzed one or two modules with many uses of leakable resources. We used the latest version of the source code that was available when we began. We also analyzed an open-source project maintained by one of the authors, to simulate the RLC's expected

```
public InputStream getInputStreamForSection(
   FileSummary.Section section, String compressionCodec)
   throws IOException {
  FileInputStream fin = new FileInputStream(filename);
  FileChannel channel = fin.getChannel();
  channel.position(section.getOffset());
  InputStream in = new BufferedInputStream(
    new LimitInputStream(fin, section.getLength()));
  in = FSImageUtil.wrapInputStreamForCompression(conf,
     compressionCodec, in);
  return in;
}
```

Fig. 4: A resource leak that the RLC found in Hadoop. Hadoop's developers merged our fix [32].

```
Optional<ServerSocket> createServerSocket(...) {
 ServerSocket serverSocket;
 try {
  if (...) {
    serverSocket = new ServerSocket();
    serverSocket.setReuseAddress(true);
    serverSocket.bind(...);
    return Optional.of(serverSocket);
  }
 } catch (IOException e) {
  // log an error
 }
 return Optional.empty();
}
```

Fig. 5: Code from the ZooKeeper that causes the Resource Leak Checker to issue a false positive.

use case, where the user is already familiar with the code under analysis (see section 7.1.5).

For each case study, our methodology was as follows. (1) We modified the build system to run the RLC on the module(s), analyzing uses of resource classes that are defined in the JDK. It also reports the maximum possible number of resources (references to JDK-defined classes with a non-empty `@MustCall` obligation) that could be leaked: each obligation at a formal parameter or method call. (2) We manually annotated each program with must-call, called-methods, and ownership annotations (see section 7.1.3). (3) We iteratively ran the analysis to correct our annotations. We measured the run time as the median of 5 trials on a machine running Ubuntu 20.04 with an Intel Core i7-10700 CPU running at 2.90GHz and 64GiB of RAM. Our analysis is embarrassingly parallel, but our implementation is single-threaded because javac is single-threaded. (4) We manually categorized each warning as revealing a real resource leak (a true positive) or as a false positive warning about safe code that our tool is unable to prove correct. At least two authors agreed on each categorization.

Table 1 summarizes the results. The RLC found multiple serious resource leaks in every program. The RLC's overall precision on these case studies is 29% (49/170). Though there are more false positives than true positives, the number is small enough to be examined by a single developer in a few hours. The annotations in the program are also a benefit: they express the programmer's intent and, as machine-checked documentation, they cannot become out-of-date.

7.1.1 False Negatives

The primary sources of unsoundness (i.e., false negatives or missed alarms) in our resource leak checker are 1) unchecked exceptions and 2) potential bugs in the implementation. Java supports both checked and unchecked exceptions; the RLC "handles" unchecked exceptions by assuming that they cannot occur (which is clearly unsound), but because the unchecked exceptions in Java mostly indicate run-time problems from which the program cannot recover (e.g., running out of memory) this choice is not problematic in practice. Like any practical implementation, the RLC may have bugs. In practice, we have only encountered unsoundness caused by bugs in our implementation.

7.1.2 True and False Positive Examples

This section gives examples of warnings reported by the RLC. Figure 4 shows code from Hadoop. If an IO error occurs any time between the allocation of the `FileInputStream` in the first line of the method and the `return` statement at the end—for example, if `channel.position(section.getOffset())` throws an `IOException`, as it is specified to do—then the only reference to the stream is lost. Hadoop's developers assigned this issue a priority of "Major" and accepted our patch [32]. One developer suggested using a try-with-resources statement instead of our patch (which catches the exception and closes the stream), but we pointed out that the file needs to remain open if no error occurs so that it can be returned.

The most common reason for false positives (which caused 22% of the false positives in our case studies) was a known bug in the Checker Framework's type inference algorithm for Java generics, which the Checker Framework developers are working to fix [26]. The second most common reason (causing 15%) was a generic container object like `java.util.Optional` taking ownership

Annotation	Count
`@Owning` and `@NotOwning`	98
`@EnsuresCalledMethods`	54
`@MustCall`	53
`@MustCallAlias`	41
`@CreatesMustCallFor`	40
Total	286

Table 2: The annotations we wrote in the case studies.

of a resource, such as the example in fig. 5. Our lightweight ownership system does not support transferring ownership to generic parameters, so the RLC issues an error when `Optional.of` is returned. In this case, the use of the `Optional` class is unnecessary and complicates the code [10]. If `Optional` was replaced by a nullable Java reference, the RLC could verify this code. Future work should expand the lightweight ownership system to support Java generics. The third most common reason (causing 8%) is nullness reasoning: some resource is closed only if it is non-null, but our checker expects the resource to be closed on every path. Our checker handles simple comparisons with `null` (as in fig. 1), but future work could incorporate more complex nullness reasoning [29].

While the false positive rate of the RLC *is* high when considered absolutely, anecdotally we have found that the signal-to-noise ratio is high enough that motivated developers are willing to tolerate it—e.g., by the Checkstyle project [41]. Moreover, resource leak checking is a fundamentally hard problem: extant heuristic bug-finding tools do not do much better than the RLC in terms of false positive ratio, but find far fewer real bugs (see section 7.3).

7.1.3 Annotations and Code Changes

We wrote about one annotation per 1,500 lines of code (table 2). (In other work [33], we developed a system to infer many of these annotations.) We also made 42 small, semantics-preserving changes to the programs to reduce false positives from our analysis. In 19 places in plume-util, we added an explicit `extends` bound to a generic type. The Checker Framework uses different defaulting rules for implicit and explicit upper bounds, and a common pattern in this benchmark caused our checker to issue an error on uses of implicit bounds. In 18 places, we made a field `final`; this allows our checker to verify the usage of the field without using the stricter rules for non-final owning fields given in section 5. In 9 of those cases, we also removed assignments of `null` to the field after it was closed; in 1 other we added an `else` clause in the constructor that assigned the field a `null` value. In 3 places, we re-ordered two statements to remove an infeasible control-flow-graph edge. In 2 places, we extracted an expression into a local variable, permitting flow-sensitive reasoning or targeting by an `@CreatesMustCallFor` annotation.

7.1.4 Inference of Annotations

Although the number of annotations is small, manually adding them is a time-consuming task. In other work, we developed a novel technique to automatically infer resource management annotations for programs [34], which enhances the applicability of the specify-and-check verification tools described herein. Inference in this domain is challenging because resource management annotations differ significantly from the types most inference techniques target. Additionally, for practical effectiveness, we need a technique that can infer the resource management annotations intended by the developer, even when the code does not fully adhere to that specification. We address these challenges with a set of inference rules designed to capture real-world coding patterns, resulting in an effective fixed-point-based inference algorithm. Implementations of this inference technique are available for both the checkers [35]. In an experimental evaluation, this inference technique inferred 85.5% of the annotations that programmers had written manually for a suite of benchmarks. Further, the verifier issued nearly the same rate of false alarms with the maually-written and automatically-inferred annotations.

7.1.5 Simulating the User Experience

To simulate the experience of a typical user who understands the codebase being analyzed, one author used the RLC to analyze plume-util, a 10kLoC library he wrote 23 years ago. The process took about two hours, including running the tool, writing annotations, and fixing the 8 resource leaks that the tool discovered. The annotations were valuable enough that they are now committed to that codebase, and the RLC runs in CI to prevent the introduction of new resource leaks. This example is suggestive that the programmer effort to use our tool is reasonable.

Table 3: False positives in our case studies ("RLC") and without lightweight ownership ("w/o LO"), resource aliasing ("w/o RA"), and obligation creation ("w/o OC").

Project	w/o LO	w/o RA	w/o OC	RLC
apache/zookeeper	117	158	47	48
apache/hadoop	97	184	58	49
apache/hbase	82	93	26	22
plume-lib/plume-util	4	11	2	2
Total	300	446	133	121

7.2 Evaluating Our Enhancements

Lightweight ownership (section 3), resource aliasing (section 4), and obligation creation (section 5) reduce false positive warnings and improve the RLC's precision. To evaluate the contribution of each enhancement, we individually disabled each feature and re-ran the experiments of section 7.1. Table 3 shows that each of lightweight ownership and resource aliases prevents more false positive warnings than the total number of remaining false positives on each benchmark. The system for creating new obligations at points other than constructors reduces false positives by a small amount: non-final, owning field re-assignments are rare. Although `@CreatesMustCallFor` annotations permit verification of some uses of non-final, owning fields, our experience suggests that its discipline is too restrictive for most real programs, and so we still view finding a better way to verify such patterns as an open problem.

7.3 Comparison to Other Tools

Our approach represents a novel point in the design space of resource leak checkers. This section compares our approach with two other modern tools that detect resource leaks:

- The analysis built into the Eclipse Compiler for Java (ecj), which is the default approach for detecting resource leaks in the Eclipse IDE [9]. We used version 4.18.0.
- Grapple [49], a state-of-the-art typestate checker that leverages whole-program alias analysis.

In brief, both of the above tools are unsound and missed 87–93% of leaks. Both tools neither require nor permit user-written specifications, a plus in terms of ease of use but a minus in terms of documentation and flexibility. Eclipse is very fast (nearly instantaneous) but has low precision (25% for high-confidence warnings, *much* lower if all warnings are included). Grapple is more precise (50% precision), but an order of magnitude slower than the RLC. The RLC had 100% recall and 26% precision. Users can select whichever tool matches their priorities. Tables 4 and 5 quantitatively compare the tools. Our comparison uses parts of the 3 case study programs that Grapple was run on in the past; see section 7.3.2 for details.

Table 4: Comparison of resource leak checking tools: **Ecl**ipse, **Gr**apple, and the **R**esource **L**eak **C**hecker. Recall is the ratio of reported leaks to all leaks present in the code, and precision is the ratio of true positive warnings to all tool warnings. Different tools were run on different versions of the case study programs. The number of leaks and the recall are computed over the code that is common to all versions of the programs, so recall is directly comparable within rows. Precision is computed over the code version analyzed by each tool, so it may not be directly comparable within rows. Eclipse reports no high-confidence warnings for JDK types in HBase.

		Recall			**Precision***		
Project	leaks	Ecl	Gr	RLC	Ecl	Gr	RLC
ZooKeeper	6	17%	17%	100%	33%	67%	21%
HDFS	7	14%	0%	100%	20%	71%	32%
HBase	2	0%	0%	100%	-	35%	19%
Total	15	13%	7%	100%	25%	50%	26%

7.3.1 Eclipse

The Eclipse analysis is a simple dataflow analysis augmented with heuristics. Since it is tightly integrated with the compiler, it scales well and runs quickly. It has heuristics for ownership, resource wrappers, and resource-free closeables, among others; these are all hard-coded into the analysis and cannot be adjusted by the user. It supports two levels of analysis: detecting high-confidence resource leaks and detecting "potential" resource leaks (a superset of high-confidence resource leaks).

We ran Eclipse's analysis on the exact same code that we ran the RLC on for section 7.1 (excluding the plume-util case study). Table 4 reports results for a subset of the code; this paragraph reports results for the full code. In "high-confidence" mode on the three projects, Eclipse reports 8 warnings related to classes defined in the JDK: 2 true positives (thus, it misses 39 real resource leaks) and 6 false positives. In "potential" leak mode, the analysis reports many more warnings. Thus, we triaged only the 180 warnings about JDK classes from the ZooKeeper benchmark. Among these were 3 true positives (it missed 10

real resource leaks) and 177 false positives (2% precision). The most common cause of false positives was the unchangeable, default ownership transfer assumption at method returns, leading to a warning at each call that returns a resource-alias, such as `Socket#getInputStream`.

7.3.2 Grapple

Grapple is a modern typestate-based resource leak analysis "designed to conduct precise and scalable checking of finite-state properties for very large codebases" [49]. Grapple models its alias and dataflow analyses as dynamic transitive-closure computations over graphs, and it leverages novel path encodings and techniques from predecessor-system Graspan [44] to achieve both context- and path-sensitivity. Grapple contains four checkers, of which two can detect resource leaks. Unlike the RLC, Grapple is unsound, as it performs a fixed bounded unrolling of loops to make path sensitivity tractable. The RLC reports violations of a user-supplied specification (which takes effort to write but provides documentation benefits), so it can ensure that a library is correct for all possible clients. By contrast, Grapple checks a library in the context of one specific client; it only reports issues in methods reachable from entry points (like a `main()` method) in a whole-program call graph [48].

The Grapple authors evaluated their tool on earlier versions of the first three case study programs in section 7.1 [49]. Unfortunately, a direct comparison on our benchmark versions is not possible, because Grapple's leak detector currently cannot be run (by us or by the Grapple authors) due to library incompatibilities and bitrot in the implementation. The Grapple authors provided us with the finite-state machine (FSM) specifications used in Grapple to detect resource leaks, and also details of all warnings issued by Grapple in the versions of the benchmarks they analyzed.

We used the following methodology to permit a head-to-head comparison. We started with all warnings issued by either tool. We disregarded any warning about code that is not present identically in the other version of the target program (due to refactoring, added code, bug fixes, etc.). We also disregarded warnings about code that is not checked by both tools. For example, Grapple analyzed test code, but our experiments did not write annotations in test code nor type-check it.

Table 5: Run times of the resource leak checking tools.

Project	Eclipse	Grapple	RLC
ZooKeeper	<5s	1h 07m 02s	1m 24s
HDFS	<5s	1h 54m 52s	16m 21s
HBase	<5s	33h 51m 59s	7m 45s

```
isSourceNode(DataFlow::Node node) {
 exists(ObjectCreation o | o.getType() in RType)
 or
 exists(Call c, Callable m, Attribute a |
  m = c.getARuntimeTarget() and a = m.getAnAttribute()
  and a.getType().hasName("Owning"))
 or
 exists(Call c, Callable m, Attribute a |
  m = c.getARuntimeTarget() and a = m.getAnAttribute()
  and a.getType().hasName("CreatesMust..."))
 or
 exists(Parameter p, Attribute a | a = p.getAttribute()
  and a.getType().hasName("Owning"))
}
```

Fig. 6: CodeQL predicate for source node

The remaining warnings pertain to resource leaks in identical code that both tools ought to report. For each remaining warning, we manually identified it as a true positive (a real resource leak) or a false positive (correct code, but the tool cannot determine that fact). Table 4 reports the precision and recall of Eclipse, Grapple, and the RLC. Some of Grapple's false positives are reports about types like `java.io.StringWriter` with no underlying resource that must be closed. (These reports were mis-classified as true positives in [49], which is one reason the numbers there differ from table 4.) Grapple's false negatives might be due to analysis unsoundness or gaps in API modeling (e.g., Grapple does not include FSM specifications for `OutputStream` classes).

Grapple runs can take many hours (run times are from [49]), whereas the RLC runs in minutes (table 5). Further, Grapple is not modular, so if the user edits their program, Grapple must be re-run from scratch [48]. After a code edit, the RLC only needs to re-analyze modified code (and possibly its dependents if the modified code's interface changed).

8 RLC#

Based on the Resource Leak Checker (RLC) for Java, we designed and developed a similar checker for C# code named RLC#, using CodeQL for dataflow analysis. While both RLC and RLC#

address resource leak checking as an accumulation problem, the core difference lies in their implementation strategies. The RLC directly solves the accumulation problem, whereas RLC# reduces it to a reachability problem. RLC# leverages CodeQL's local dataflow engine and adapts the pluggable type system introduced by RLC to suit the C# language: for example, the Java RLC implements its type qualifiers as annotations; RLC# implements its qualifiers via C# attributes. This section briefly introduces CodeQL, then details RLC#'s design and implementation. Subsequently, we present an evaluation of RLC# and discuss its limitations.

8.1 An Overview of CodeQL

CodeQL [6] is widely used taint-tracking tool for security analysis. The core components of CodeQL, which trace data flow from sources (where data enters a program) to sinks (where data is used), are designed to detect security issues such as SQL injection and cross-site scripting. CodeQL offers a generic configuration for data flow analysis, which we adapted for resource leak checking. By leveraging this flexibility, we can trace the flow of resources within C# programs, identifying potential leaks. This innovative application of CodeQL's dataflow module shows its versatility beyond traditional security applications: we can reason about resource management with the same robust infrastructure used for security analysis.

CodeQL supports a wide range of languages and frameworks; it treats code as data and uses a *query language (QL)* for pattern-matching to extract information from a CodeQL database, which is a relational model of the source files. A simple query that finds redundant `if` statements in the source code is given below.

```
from IfStmt ifstmt, BlockStmt block
where ifstmt.getThen() = block and block.isEmpty()
select ifstmt, "if-stmt is redundant."
```

In this query, the `from` clause declares variables, the `where` clause defines the logical conditions, and the `select` clause specifies the results for variables that meet the `where` clause conditions.

CodeQL users can define queries to discover various types of coding errors. The DataFlow module in CodeQL performs a taint-style dataflow analysis with sources and sinks. A dataflow graph is constructed that models dataflow during program execution, with nodes representing elements like expressions and parameters, and edges representing the flow between them.

```
isSink(DataFlow::Node node) {
 exists(Callable c, Call call |
 c = call.getARuntimeTarget() and
 (c.hasName("Close") or c.hasName("Dispose")))
 or
 exists(Callable m, Expr e, Attribute a |
  m.canReturn(e) and a = m.getAnAttribute() and
  a.getType().hasName("Owning"))
 or
 exists(Call c, Parameter p, Callable m, Attribute a |
  m = c.getARuntimeTarget() and p = m.getParameter(_)
  and a = p.getAnAttribute() and
  a.getType().hasName("Owning"))
 or
 exists(Call c, Callable c, Attribute a |
  m = c.getARuntimeTarget() and a = m.getAnAttribute()
  and a.getType().hasName("EnsuresCalledMethods"))
 or
 exists(Call c, Callable m, Attribute a |
  m = c.getARuntimeTarget() and a = m.getAnAttribute()
  and a.getType().hasName("CreatesMustCallFor"))
}
```

Fig. 7: CodeQL predicate for sink node

Users of the DataFlow module need only define the desired nodes and edges using predicates. For instance, to detect null-pointer dereferences, a source node could be an assignment of `null` to a reference or a method call that returns `null`, while a sink node could be a dereference operation on the same reference. For resource leak checking, the CodeQL predicates for source and sink nodes are defined in fig. 6 and fig. 7 (explained in section 8.2). CodeQL also includes other nodes in the dataflow graph, which may represent aliases. It represents the dataflow between the nodes as edges but also allows for the addition of explicit edges to capture additional dataflow. To address challenges such as unavailable source code, alias information, and scalability, CodeQL provides both local and global dataflow analysis. Local dataflow analysis is confined to a single method and is faster and more precise, whereas global dataflow analysis spans multiple methods and includes dataflow through method calls. CodeQL's local dataflow analysis is flow-sensitive, meaning it tracks the flow of data through the program, considering the order of statements and control flow.

Local dataflow computes may-alias information, while sound resource leak detection requires must-alias information (see section 2.4). RLC# treats the may-alias information from local dataflow as

Fig. 8: A C# example to demonstrate the working of RLC#.

```
1. [MustCall("Dispose")]
2. public class Container() {
3.
4.  [Owning]
5.  private readonly Socket sock;
6.
7.  [MustCallAlias]
8.  public Container([MustCallAlias] Socket s) {
9.    sock = s;
10.  }
11.
12. [EnsuresCalledMethods("sock","close(sock)")]
13. public void Dispose() {
14.   closeSocket(socket);
15. }
16. [Owning]
17. public static Socket createSocket() {
18.   return new Socket(...);
19. }
20.
21. public void close([Owning] Socket s) {
22.   s.Dispose();
23. }
24.
```

```
25. [CreatesMustCallFor("sock")]
26. public void reset() {
27.   if (this.sock != null) {
28.     this.sock.Dispose();
29.   }
30.   this.sock = createSocket();
31. }
32.}
33. public static void Main() {
34.   try {
35.     Socket s = Container.createSocket();
36.     Container c = new Container(s);
37.     ...
41.     c.reset();
42.     ...
55.     c.Dispose();
56.   }
57.   catch(...) {
58.     c.Dispose();
59.   }
60. }
```

must information, to re-use the tuned DataFlow module, at the cost of potential unsoundness.

For RLC#, nodes are defined using CodeQL's code-pattern-matching capabilities, and local dataflow is used for intraprocedural analysis. Like RLC, RLC# uses the lightweight ownership transfer system, a lightweight resource-alias tracking analysis, and a system for handling lazy or multiple initialization for capturing interprocedural (global) dataflow.

8.2 RLC# Query Design

RLC# uses CodeQL's local dataflow analysis to check must-call properties by checking for the presence of a dataflow path between the node where a resource is acquired (referred to as the *source* node) and the node where it is released (referred to as the *sink* node), thereby framing the issue as a reachability problem. Interprocedural dataflow is tracked using attributes similar to Java annotations. These attributes are derived from RLC's pluggable type systems (section 2), lightweight ownership transfer (section 3), resource-alias tracking (section 4), and mechanisms to handle lazy or multiple initializations (section 5).

RLC#'s algorithm has three key steps: (a) defining source and sink nodes for dataflow, (b) checking the existence of a path between each source and corresponding sinks in a *dataflow graph*, and (c) ensuring that a sink exists along every path from the source to the method's exit in the *control flow graph*. Since we use CodeQL's local dataflow analysis, both the source and sink are within the same method. For calls to other methods, the query examines the attributes added to the method boundaries instead of their bodies to capture global dataflow. This approach makes RLC# modular, similar to RLC.

In this section, we outline the design of RLC# by defining the resource type, specifying the source and sink nodes, and explaining how to verify the must-call property by checking for dataflow between these nodes. Additionally, we provide implementation details of RLC# in CodeQL.

Resource Type. We define *Resource Type* (RType) as a set of types that represent resources. A type `t` belongs to RType if and only if it satisfies one of the following conditions:

- `t` implements the `System.IDisposable` interface, or
- `t` has a `MustCall` attribute associated to its definition, or
- `t` is a `CollectionType` (like Vectors, Arrays) where the type of the elements is in RType,[3] or

[3]Similar to RLC, RLC# handles collection types conservatively. As a result, warnings related to resource leaks for collection types are often reported even if a sink node exists for them. This conservative approach is due to the difficulty in ensuring that all resources within the collection are properly released, as the analysis is not path-sensitive. Path sensitivity would require tracking the exact execution paths to confirm that

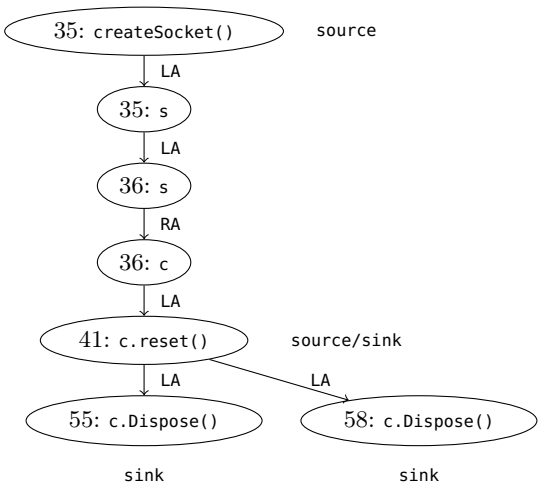

Fig. 9: DataFlow graph for procedure `main` in Figure 8. `LA` and `RA` represent local alias and resource alias, respectively. The numbers in each node refer to line numbers in the source code.

- `t` is a subtype of another type in RType.

Modelling Source and Sink Nodes. RLC# harnesses CodeQL's code-pattern-matching capabilities to specify the source and sink nodes, which are method calls with specific attributes. This design, like the Java RLC, has the benefit of modularity: interprocedural dataflow facts are communicated via the summaries (i.e., attributes), so the analysis is usually fast.

Sources: A source node is any of the following:

1. A `new` expression (call to a constructor) that allocates a resource (e.g. line 18 in Figure 8).
2. A call to a method whose return type has the `Owning` attribute associated to it (e.g. lines 30 and 35 in Figure 8). The `Owning` attribute on a method's return type indicates that a resource is allocated within the method and returned to the caller. Therefore, a call to such a method constitutes a source node.
3. A call to a method with `CreatesMustCallFor` attribute (e.g. line 41 in Figure 8). The invocation of a method with the `CreatesMustCallFor` attribute signifies the allocation of a new resource within the method; so, such an invocation is a source node.
4. A parameter with an `Owning` attribute (e.g. line 21 in Figure 8), which indicates that the parameter is the owner of a resource (hence obligated to release the resource); thereby making it a source node.

The CodeQL predicate `isSource` identifies the source node in Figure 6. The CodeQL pattern identifies the `new` expression (1st case above) as an `ObjectCreation` (`o`); the source node is any `o` whose type is a RType. The predicate identifies the source node as a call `c` where the called method `m` has an `Owning` attribute (2nd case). A call `c` to a method `m` with the `CreatesMustCallFor` attribute `a` is identified as a source node (third case). A parameter `p` is checked for an `Owning` attribute to determine if it is a source node (fourth case).

The CodeQL predicate `getARuntimeTarget` returns a set of methods that could be invoked at the call node `c`. CodeQL computes an over-approximation of the call graph (during the CodeQL database creation), which includes virtual calls. As a result, the set of methods returned by `getARuntimeTarget` encompasses all possible methods that could be called at `c`. Additionally, CodeQL provides a predicate `getATarget` that excludes callees for virtual calls, focusing only on non-virtual call targets. The Java RLC handles this differently: it does not directly account for virtual method calls within its primary analysis. Instead, a separate check ensures that method overrides respect the standard principles of subtyping. The Java approach requires additional checks, but avoids the need to build a call graph, potentially improving scalability and modularity.

Sinks: The specification of a sink node in a dataflow graph for a resource leak checker includes:

1. A call to the `Close` or `Dispose` method is used to release a resource in C# (e.g. line 22 in Figure 8).
2. A return expression within a method having an `Owning` attribute on its return-type (e.g. line 18 in Figure 8). This expression passes a reference to a resource back to the caller, transferring the responsibility of releasing the resource to the caller, thus forming the sink node.
3. A call to a method with a parameter that has an `Owning` attribute (e.g. line 14 in Figure 8). In this scenario, the ownership transfer occurs to the callee's parameter, making the callee responsible for releasing the resource through that parameter.

every resource in the collection is released, which is complex and computationally expensive.

4. A call to a method with an attribute `EnsuresCalledMethods` (e.g. line 13 in Figure 8) is considered a sink node. The first argument of this attribute holds the ownership, and the resource is released within the method by the call expression specified as the second argument of the attribute. For example, in Figure 8, the `EnsuresCalledMethods` attribute (line 12) has the field `sock` as its first argument, which owns the resource. The resource is released by the call expression `close(sock)`, which is the second argument of the attribute.
5. A call to a method with the attribute `CreatesMustCallFor` (e.g., line 41 in Figure 8) is considered both a sink node and a source node. This dual role arises because the method allocates a new resource, making it a source node, but also releases an older resource before the new allocation, making it a sink node. A resource leak is reported within this method if the older resource is not released before the new resource is allocated.

   In our example, the older resource is released on line 28 before the new resource is allocated on line 30. This approach could potentially result in a false positive, as the older resource might be released by the caller of this method before the method itself allocates a new resource. However, the design expectation is that this method might be called multiple times. To ensure modularity and avoid placing the burden of resource management on the caller, the method ensures that the resource is released before allocating a new one. This practice allows the method to be used more flexibly and reliably in different contexts.

The CodeQL predicate `isSink` is defined to identify the sink node in Figure 7.

8.3 Source-to-Sink Dataflow

In a dataflow graph, a path from a source to a sink node only exists if the resource obtained at the source is released at the sink. Consider this code example:

```
1. Socket s1 = new Socket(...);
2. Socket s2 = new Socket(...);
3. s2.Dispose();
4. s1.Dispose();
```

CodeQL identifies the `new Socket(...)` expression in lines 1 and 2 as source nodes, and the `Dispose` calls on lines 3 and 4 as sink nodes. There's a dataflow path from the source on line 1 to the sink on line 4, and from the source on line 2 to the sink on line 3.

The CodeQL `localFlow` predicate checks for local dataflow between two nodes within the same method. However, it doesn't capture aliases created by callees. The `MustCallAlias` attribute is used to capture such aliasing relationships.

A method that establishes a resource-alias relationship between different objects will have a `MustCallAlias` attribute on its return type and parameter. The parameter and return value are must-aliases, creating an alias relationship between the method call's argument and its return value. For instance, `x` and `y` become resource aliases in the assignment `x = createHandle(y)` if the `createHandle` method's parameter and return-type have the `MustCallAlias` attribute.

```
predicate isResourceAlias(n1, n2) {
 exists(AssignableDefinition def, Call c, Callable m,
  Parameter p, Expr arg |
 c = def.getSource() and m = c.getARuntimeTarget()
 and p = m.getParameter(_) and
 arg = c.getArgumentForParameter(p) and
 n2.asExpr() = def.getTarget() and n1.asExpr() = arg
 and isMustCallAliasMethod(m) and isMustCallAliasPar(p))
}
```

We define `isResourceAlias` predicate to determine if nodes `n1` and `n2` are resource aliases. The predicate checks if an assignment `def` has a call `c` on its right-hand side and if the call's argument `arg` is a resource alias to the assignment's left-hand side (`def.getTarget()`). It also verifies that the corresponding parameter `p` and the callee `m`'s return-type have the `MustCallAlias` attribute.

In Figure 8, within the `Container` constructor (line 8), the parameter `s` and the field `sock` are must-aliases. This creates a resource alias between the `Main` method's local variable `s` and the `Container` class instance `c`, where the constructor call (line 36) occurs.

We define the predicate `isAlias` that captures all the dataflow between nodes `n1` and `n2`.

```
predicate isAlias(n1, n2) {
 DataFlow::localFlow(n1, n2) or isResourceAlias(n1, n2)
 or exists(DataFlow::Node n |
 isAlias(n1, n) and isAlias(n, n2))
}
```

Nodes `n1` and `n2` are either local aliases (`localFlow`) or resource aliases (`isResourceAlias`) or a combination of the two.

	LoC	Resources	TP	FP	Attr.	Time
Lucene.Net	609,754	284	8	12	71	55m 56s
EFCore	883,195	176	0	19	29	76m 27s
Service 1	670,988	149	7	2	23	2m 18s
Service 2	194,765	263	6	3	31	3m 31s
Service 3	170,471	33	3	1	26	3m 0s
Total	2,529,173	905	24	37	180	

Table 6: Verifying resource leaks: "LoC" refers to lines of non-comment, non-blank C# code. "Resources" indicates the number of resources created by an application. "TP" are true positive warnings, while "FP" are false positives where RLC# reported a potential leak, but manual analysis found none. "Attr." are attributes added to the code. Time is the average of three trials.

8.4 Verifying the Must-Call Property

For a source node `src`, confirming the presence of a sink node and a dataflow path between `src` and the sink is necessary but not sufficient. It is essential that the resource is released on all control-flow paths from the source to the method's exit.

In Figure 8, resources are allocated on lines 36 and 41 and released on lines 55 and 58. Removing the call to `c.Dispose()` on line 58 would result in a resource leak because there would be a control-flow path (an exceptional path) where the sink node is absent, despite the existence of a dataflow path between the source and sink nodes.

The `notDisposed` predicate defined below checks whether a resource associated with `src` is released on all control-flow paths. The second parameter `nd` is a control-flow node that changes as the control-flow graph is traversed backward (using CodeQL predicate `getAPredecessor` for backward traversal). The traversal of a control-flow path stops when a sink node or a source node is encountered. If a source node is encountered, it indicates a control-flow path from the source node to the method's exit that does not include a sink. If a sink node is encountered, the path is not further explored, indicating that the resource is released along this path.

```
predicate notDisposed(src, nd) {
 nd = src.getControlFlowNode() or
 notDisposed(src,nd.getAPredecessor()) and not
 exists(DataFlow::Node sink |
  sink.getControlFlowNode() = nd and isAlias(src, sink))
}
```

The CodeQL predicates `getEnclosingCallable` and `getExitPoint` are used to determine the method in which `src` is allocated and its exit point.

8.5 RLC# Example

This section illustrates how RLC# operates on our motivating example. In Figure 8, the type `Container` is identified as a resource type because it has the `MustCall` attribute. The dataflow graph for the method `Main` is given in Figure 9. The source node is a call to the method `createSocket` on line 35, which has the `Owning` attribute on its return type. The expression `s` on line 35 is aliased to the source node `createSocket()`. The local dataflow analysis computes the alias relationship between the expression `s` on line 35 and the expression `s` on line 36 representing the argument being passed to the constructor. The `isResourceAlias` predicate identifies the expressions `c` and `s` on line 36 as resource aliases.

The call to the method `reset` on line 41 is a sink node for the resource allocated on line 35. However, it also serves as a source node since a new resource is allocated on line 30 within the method, as indicated by the `CreatesMustCallFor` attribute on the method. The release of the resource on line 28 within the method `reset`, before reallocating a resource on line 30, is verified when the method `reset` is analyzed independently. For the source node on line 41, the sink nodes are the calls to `Dispose` on lines 55 and 58.

However, the presence of sink nodes for the sources on lines 35 and 41 is not sufficient to ensure the absence of resource leaks. It is necessary to verify that each control flow path from the source to the method's exit has a sink node. For the source on line 35, the sink node is on line 41. In the event of an exception occurring after line 36, the sink node would be the call to `Dispose` on line 58. Similarly, for the source on line 41, the sink node is on line 55; however, in case of an exception, the resource is released on line 58, ensuring no resource leak.

8.6 Comparing RLC# and RLC

RLC and the RLC# share a common approach: a pluggable type system to track interprocedural dataflow. However, they differ in their underlying design philosophies and handling of language-specific features.

RLC models resource leak checking as an accumulation problem, while RLC# addresses it as a reachability problem. This fundamental difference in perspective influences how each checker

Attribute	Count
`Owning`	57
`EnsuresCalledMethods`	41
`MustCallAlias`	14
`MustCall`	44
`CreatesMustCallFor`	19
Total	180

Table 7: The attributes we wrote in the case studies.

approaches the problem of resource management and also its implementation.

In terms of language-dependent distinctions, RLC and RLC# diverge in two main areas:

- Java incorporates both checked and unchecked exceptions, whereas C# only has unchecked exceptions. Both checkers handle unchecked exceptions in a way that is not sound, but this does not affect Java applications significantly as critical exceptions in Java are checked. The impact is more pronounced in C# applications due to the absence of checked exceptions.
- Java employs type erasure for generic types, a feature not supported by C#. Consequently, in C#, attributes must be explicitly associated with each bound of the type parameters, complicating the addition of attributes in the source code for generic types. To circumvent this, attributes are added as logical formulas within the CodeQL query rather than in the source code. This approach identifies the specific locations and program elements in the source code where attributes need to be added, avoiding the repetitive task of building code and creating a new CodeQL database for each new attribute addition.

8.7 Evaluation

We used CodeQL version 2.11.4 to implement RLC#. This section outlines an evaluation of RLC# on both open-source projects and Azure microservices, and compares RLC# with a pre-existing (naive) query from the CodeQL repository.

Case Studies. We selected two open-source C# projects, `Lucene.Net` and `EFCore` by convenience. Additionally, we analyzed three proprietary Azure microservices. Our methodology for each application was as follows: (a) We constructed a CodeQL database for the source code. (b) We manually annotated the source code for each application with appropriate attributes and iteratively ran RLC# to refine our annotations. (c) We manually categorized each RLC# warning as either an actual resource leak (true positive) or a false warning.

	RLC#		NCQ	
	TP	FP	TP	FP
Lucene.Net	8	12	3	42
EFCore	0	19	0	1
Service 1	7	2	0	38
Service 2	6	3	0	234
Service 3	3	1	0	3
Total	24	37	3	318

Table 8: Comparison of RLC# with an existing "Naive CodeQL Query" (NCQ) for resource leaks. "TP" means True Positives (actual resource leaks) and "FP" stands for False Positives.

Results. Table 6 presents our findings. RLC# detected resource leaks in four out of five applications and successfully identified known resource leaks in microservices that previously caused high-impact incidents. RLC#'s precision rate across these applications is 39.34% (24/61). Manual source code annotation with attributes is laborious and time-consuming. To mitigate this, we selectively annotated only library-defined program elements, disregarding verifier warnings for custom types. Our subsequent work [34], extends this approach to custom types and has uncovered more resource leaks using inferred attributes.

The majority of false positives (76%) arise from path sensitivity issues, where resources are conditionally released based on their null status, while RLC# anticipates an unconditional release across all paths. RLC# can handle straightforward null comparisons. A smaller fraction of false positives arises from conservative exception handling (3%) and conservative handling of collection types (4%). The rest involve resources allocated to an `Owning` field (e.g., `f`) without a class-defined disposal method; verification of such resources is not modular and hence not handled by RLC#.

There are three sources of unsoundness in RLC#. First, the local dataflow analysis in RLC# computes may-alias information, whereas sound resource leak detection requires must-alias information. To leverage the existing DataFlow module, RLC# treats may-alias information as must-alias information. Second, C# only supports unchecked exceptions. While the handling of unchecked exceptions by both checkers is not sound, in C# applications the effect is noticeable due to the lack

of checked exceptions. Third, like any practical implementation, RLC# may contain bugs.

Attributes added to Source Code. For every 7,000 lines of code, we introduced one attribute (refer to Table 7). The manual addition of attributes to the source code entails a rebuild of the source code and the creation of a new CodeQL database, as the database creation is not incremental. To circumvent the repetitive generation of the CodeQL database, we incorporated attributes as logical formulas within the query, rather than directly modifying the code. An example of one such formula for an `Owning` parameter `s` is ("`RLCTests`" is the namespace):

```
fileName="RLCTests/SimpleEg.cs" and lineNo="17"
and elementType="Parameter" and elementName="s"
and annotation="Owning"
```

Comparing to An Existing CodeQL Query. We contrast RLC# with an existing CodeQL query "NCQ" for heuristic detection of resource leaks [27]. NCQ relies on CodeQL's local dataflow, which overlooks method calls and field dereferences, and is hence unsound due to untracked interprocedural dataflow and incomplete aliasing information. Unlike RLC#, NCQ does not require adding attributes to the source code and is faster. As seen in table 8, except for Lucene.Net, NCQ fails to detect resource leaks in our benchmarks. Its high false positive rate stems from naively treating every constructor call as resource allocation.

9 Threats to Validity

Like any tools that analyze source code, the RLC and RLC# only give guarantees for the code they check: the guarantee excludes native code, the implementation of unchecked libraries (such as the JDK), and code generated dynamically or by other annotation processors such as Lombok. Though the Checker Framework can handle reflection soundly [3], by default (and in our case studies) the RLC compromises this guarantee by assuming that objects returned by reflective invocations do not carry must-call obligations. (Users can customize this behavior.) Within the bounds of a user-written warning suppression, the RLC assumes that 1) any errors issued can be ignored, and 2) all annotations written by the programmer are correct.

The RLC is sound with respect to specifications of which types have a `@MustCall` obligation that must be satisfied. We wrote such specifications for the Java standard library, focusing on IO-related code in the `java.io` and `java.nio` packages. Any missing specifications of `@MustCall` obligations could lead the RLC to miss resource leaks. Similarly, RLC# is sound for types marked with the `MustCall` attribute, primarily for IO-related code in the `System.IO` package. Also, RLC# does not track resource leaks on exceptional paths if a statement that may throw an exception is not enclosed in a try-catch-finally block or declared with a `throws` clause. It only soundly handles developer-managed exceptions. Finally, in RLC# resource allocation (source nodes) and deallocation (sink nodes) are identified through pattern matching. Missing a source node may cause unsoundness, while missing a sink node could only cause false positives.

The results of our experiments may not generalize, compromising the external validity of the experimental results. The RLC may produce more false positives, require more annotations, or be more difficult to use if applied to other programs. Case studies on legacy code represent a worst case for a source code analysis tool. Using the RLC from the inception of a project would be easier, since programmers know their intent as they write code and annotations could be written along with the code. It would also be more useful, since it would guide the programmers to a better design that requires fewer annotations and has no resource leaks. The need for annotations could be viewed as a limitation of our approach. However, the annotations serve as concise documentation of properties relevant to resource leaks—and unlike traditional, natural-language documentation, machine-checked annotations cannot become out-of-date.

Like any practical system, it is possible that there might be defects in the implementation of the RLC or RLC#, or in the design of their analyses. We have mitigated this threat with code review and extensive test suites: 209 test classes containing 6,418 lines of non-comment, non-blank code for the RLC (publicly-available and distributed with the RLC); and 88 test cases with 4,843 lines of code for RLC# (available with the RLC# code [31]).

10 Related Work

Most prior work on resource leak detection either uses program analysis to detect leaks or adds language features to prevent them. Here we focus on the most relevant work from these categories.

10.1 Analysis-Based Approaches

Static analysis. The most closely related work are our prior conference publications about the RLC [21] and about an inference system for RLC annotations and RLC# attributes [33].

Tracker [42] performs inter-procedural dataflow analysis to detect resource leaks, with various features to make the tool practical, including issue prioritization and handling of wrapper types. Tracker avoids whole-program alias analysis for scalability, instead using a local, access-path-based approach. While Tracker does scale to large programs, it is deliberately unsound, unlike the RLC.

The Eclipse Compiler for Java includes a dataflow-based bug-finder for resource leaks [9]. Its analysis uses a fixed set of ownership heuristics and a fixed list of wrapper classes; unlike the RLC, it is unsound. It is very fast. Similar analyses—with similar trade-offs compared to the RLC—exist in other heuristic bug-finding tools, including SpotBugs [36], PMD [30], and Infer [18]. Section 7.3.1 experimentally evaluates the Eclipse analysis.

Typestate analysis [37, 12] can be used to find resource leaks. Grapple [49] is the most recent system to use this approach, leveraging a disk-based graph engine to achieve unprecedented scalability on a single machine. Compared to the RLC, Grapple is more precise but suffers from unsoundness and longer run times. Section 7.3.2 gives a more detailed comparison to Grapple.

CLOSER [8] automatically inserts Java code to dispose of resources when they are no longer "live" according to its dataflow analysis. It requires an expensive alias analysis for soundness, as well as manually-provided aliasing specifications for linked libraries. The RLC uses accumulation analysis [22, 11] to achieve soundness without the need for a whole-program alias analysis. RLFixer [43] also automatically fixes leaks, but runs existing leak detectors (including the RLC) for fault localization.

Dynamic analysis. Some approaches use dynamic analysis to ameliorate leaks. Resco [7] operates similarly to a garbage collector, tracking resources whose program elements have become unreachable. When a given resource (such as file descriptors) is close to exhaustion, the runtime runs Resco to clean up any resources of that type that are unreachable. With a static approach such as ours, leaks are impossible and a tool like Resco is unnecessary.

Automated test generation can also be used to detect resource leaks. For example, leaks in Android applications can be found by repeatedly running neutral—i.e. eventually returning to the same state—GUI actions [46, 47]. Other techniques detect common misuse of the Android activity lifecycle [2]. Testing can only show the presence of failures, not the absence of defects; the RLC verifies that no resource leaks are present.

Data sets and surveys. The DroidLeaks benchmark [24] is a set of Android apps with known resource leaks. Unfortunately, it includes only the compiled apps. The RLC runs on source code, so we were unable to run the RLC on DroidLeaks. Ghanavati et al. [16] performed a detailed study of resource leaks and their repairs in Java projects, showing the pressing need for better tooling for resource leak prevention. In particular, their study showed that developers consider resource leaks to be an important problem, and that previous static analysis tools are insufficient for preventing resource leaks. We plan to apply the RLC to more programs from their study.

10.2 Language-Based Approaches

Ownership types and Rust. Ownership type systems [5] impose control over aliasing, which in turn enables guaranteeing other properties, like the absence of resource leaks. We do not discuss the vast literature on ownership type systems [5] here. Instead, we focus on Rust [23] as the most popular practical example of using ownership to prevent resource leaks.

For a detailed overview of ownership in Rust, see chapter 4 of [23]; we give a brief overview here. In Rust, ownership is used to manage both memory and other resources. Every value associated with a resource must have a *unique* owning pointer, and when an owning pointer's lifetime ends, the value is "dropped," ensuring all resources are freed. Rust's ownership type system statically prevents not only resource leaks, but also other

important issues like "double-free" defects (releasing a resource more than once) and "use-after-free" defects (using a resource after it has been released). But, this power comes with a cost; to enforce uniqueness, non-owning pointers must be invalidated after an ownership transfer and can no longer be used. Maintaining multiple usable pointers to a value requires use of language features like references and borrowing, and even then, borrowed pointers have restricted privileges.

The RLC has less power than Rust's ownership types; it cannot prevent double-free or use-after-free defects. But, the RLC's lightweight ownership annotations impose *no* restrictions on aliasing; they simply aid the tool in identifying how a resource will be closed. Lightweight ownership is better suited to preventing resource leaks in existing, large Java code bases; adapting such programs to use a full Rust-style ownership type system would be impractical.

Other approaches. Java's try-with-resources construct [28] was discussed in section 1. Java also provides finalizer methods [17, Chapter 12], which execute before an object is garbage-collected, but they should not be used for resource management, as their execution may be delayed arbitrarily. Compensation stacks [45] generalize C++ destructors and Java's try-with-resources, to avoid resource leak problems in Java. While compensation stacks make resource leaks less likely, they do not guarantee that leaks will not occur, unlike the RLC. Previous work has performed modular typestate analysis for annotated Java programs [4] or proposed typestate-oriented programming languages with modular typestate checking [1, 15]. The type systems of these approaches can express arbitrary typestate properties, beyond what can be checked with the RLC. However, these systems impose restrictions on aliasing and a higher type annotation burden than the RLC, making adoption for existing code more challenging.

11 Conclusion

We have developed a sound and modular approach to detecting and preventing resource leaks in large-scale Java and C# programs. The Resource Leak Checker consists of sound core analyses, built on the insight that leak checking is an accumulation problem, augmented by three new features to handle common aliasing patterns: lightweight ownership transfer, resource aliasing, and obligation creation by non-constructor methods. Inspired by The Resource Leak Checker, RLC# detects and prevents resource leaks in large-scale C# programs, leveraging the insight that leak checking can be viewed as a reachability problem.

The Resource Leak Checker discovered 49 resource leaks in heavily-used, heavily-tested Java code. Its analysis speed is an order of magnitude faster than whole-program analysis, and its false positive rate is similar to a state-of-the-practice heuristic bug-finder. It reads and verifies user-written specifications; the annotation burden is about 1 annotation per 1,500 lines of code. RLC# discovered 24 resource leaks in C# code, and its annotation burden is about 1 annotation per 7000 lines of code.

References

[1] Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Typestate-oriented programming. In: OOPSLA Companion: Object-Oriented Programming Systems, Languages, and Applications, pp. 1015–1022. Orlando, FL, USA (2009)

[2] Amalfitano, D., Riccio, V., Tramontana, P., Fasolino, A.R.: Do memories haunt you? an automated black box testing approach for detecting memory leaks in android apps. IEEE Access **8**, 12217–12231 (2020)

[3] Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., d'Amorim, M., Ernst, M.D.: Static analysis of implicit control flow: Resolving Java reflection and Android intents. In: ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering, pp. 669–679. Lincoln, NE, USA (2015)

[4] Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA 2007, Object-Oriented Programming Systems, Languages, and Applications, pp. 301–320. Montreal, Canada (2007)

[5] Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification. Springer, Berlin, Heidelberg (2013)

[6] CodeQL. https://codeql.github.com

[7] Dai, Z., Mao, X., Lei, Y., Wan, X., Ben, K.: Resco: Automatic collection of leaked resources. IEICE TRANSACTIONS on Information and Systems **96**(1), 28–39 (2013)
[8] Dillig, I., Dillig, T., Yahav, E., Chandra, S.: The closer: automating resource management in java. In: International symposium on Memory management, pp. 1–10 (2008)
[9] Eclipse developers: Avoiding resource leaks. https://help.eclipse.org/2020-12/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-avoiding_resource_leaks.htm&cp%3D1_3_9_3 (2020). Accessed 3 February 2021
[10] Ernst, M.D.: Nothing is better than the optional type. https://homes.cs.washington.edu/~mernst/advice/nothing-is-better-than-optional.html (2016)
[11] Fähndrich, M., Leino, K.R.M.: Heap monotonic typestates. In: IWACO 2003: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming, pp. 58–72. Darmstadt, Germany (2003)
[12] Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM TOSEM **17**(2) (2008)
[13] Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: PLDI '99: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, pp. 192–203. Atlanta, GA, USA (1999). DOI 10.1145/301618.301665
[14] Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns. Addison-Wesley, Reading, MA (1995)
[15] Garcia, R., Tanter, E., Wolff, R., Aldrich, J.: Foundations of typestate-oriented programming. ACM Trans. Program. Lang. Syst. **36**(4), 12:1–44 (2014)
[16] Ghanavati, M., Costa, D., Seboek, J., Lo, D., Andrzejak, A.: Memory and resource leak defects and their repairs in java projects. Empirical Software Engineering **25**(1), 678–718 (2020)
[17] Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 8 Edition. Addison-Wesley Professional (2014)
[18] Infer developers: Resource leak in java. https://fbinfer.com/docs/checkers-bug-types#resource-leak-in-java (2021). Accessed 4 February 2021
[19] JetBrains: List of java inspections. https://www.jetbrains.com/help/idea/list-of-java-inspections.html#resource-management (2020). Accessed 5 February 2021
[20] Kellogg, M., Ran, M., Sridharan, M., Schäf, M., Ernst, M.D.: Verifying object construction. In: ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering, pp. 1447–1458. Seoul, Korea (2020)
[21] Kellogg, M., Shadab, N., Sridharan, M., Ernst, M.D.: 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), pp. 181–192. Athens, Greece (2021). DOI 10.1145/3468264.3468576
[22] Kellogg, M., Shadab, N., Sridharan, M., Ernst, M.D.: Accumulation analysis. In: ECOOP 2022 — Object-Oriented Programming, 33rd European Conference, pp. 10:1–10:31. Berlin, Germany (2022). DOI 10.4230/DARTS.8.2.22
[23] Klabnik, S., Nichols, C.: The Rust Programming Language (2018). URL https://doc.rust-lang.org/1.50.0/book/
[24] Liu, Y., Wang, J., Wei, L., Xu, C., Cheung, S.C., Wu, T., Yan, J., Zhang, J.: Droidleaks: a comprehensive database of resource leaks in android apps. Empirical Software Engineering **24**(6), 3435–3483 (2019)
[25] Lo, D., Nagappan, N., Zimmermann, T.: How practitioners perceive the relevance of software engineering research. In: ESEC/FSE 2015: The 10th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp. 415–425. Bergamo, Italy (2015)
[26] Millstein, S.: Implement java 8 type argument inference. https://github.com/typetools/checker-framework/issues/979 (2016). Accessed 17 April 2020
[27] NCQ CodeQL Query for missing Dispose calls. https://github.com/github/codeql/blob/28f8874243bc110099483535633e7f4c9c2738a3/csharp/ql/src/API%20Abuse/NoDisposeCallOnLocalIDisposable.ql (2024). Accessed 15 October 2024

[28] Oracle: The try-with-resources statement (the java tutorials). https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html (2020). Accessed 24 February 2021
[29] Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 201–212. Seattle, WA, USA (2008). DOI 10.1145/1390630.1390656
[30] PMD developers: Closeresource. https://pmd.github.io/pmd-6.31.0/pmd_rules_java_errorprone.html#closeresource (2021). Accessed 4 February 2021
[31] RLC# implementation. https://github.com/microsoft/global-resource-leaks-codeql (2024)
[32] Shadab, N.: Hdfs-15791. possible resource leak in fsimageformatprotobuf. https://github.com/apache/hadoop/pull/2652 (2021). Accessed 16 June 2021
[33] Shadab, N., Gharat, P., Tiwari, S., Ernst, M.D., Kellogg, M., Lahiri, S., Lal, A., Sridharan, M.: Inference of resource management specifications. Proc. ACM Program. Lang. **7**(OOPSLA2, article #282), 1705–1728 (2023)
[34] Shadab, N., Gharat, P., Tiwari, S., Ernst, M.D., Kellogg, M., Lahiri, S.K., Lal, A., Sridharan, M.: Inference of resource management specifications. Proc. ACM Program. Lang. **7**(OOPSLA2) (2023). DOI 10.1145/3622858. URL https://doi.org/10.1145/3622858
[35] Shadab, N., Gharat, P., Tiwari, S., Ernst, M.D., Kellogg, M., Lahiri, S.K., Lal, A., Sridharan, M.: Inference of resource management specifications (2023). DOI 10.5281/zenodo.10438985. URL https://doi.org/10.5281/zenodo.10438985
[36] SpotBugs developers: Obl: Method may fail to clean up stream or resource. https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#obl-method-may-fail-to-clean-up-stream-or-resource-obl (2021). Accessed 4 February 2021
[37] Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE TSE **SE-12**(1), 157–171 (1986)
[38] Stroustrup, B.: 16.5, resource management. In: The design and evolution of C++, pp. 388–389. Pearson Education India (1994)
[39] The Apache Hadoop developers: Storageinfo.java. https://github.com/apache/hadoop/blob/aa96f1871bfd858f9bac59cf2a81ec470da649af/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/common/StorageInfo.java#L246 (2018). Accessed 22 February 2021
[40] The Apache ZooKeeper developers: Learner.java. https://github.com/apache/zookeeper/blob/c42c8c94085ed1d94a22158fbdfe2945118a82bc/zookeeper-server/src/main/java/org/apache/zookeeper/server/quorum/Learner.java#L465 (2020). Accessed 24 February 2021
[41] the Checkstyle developers: Checkstyle's pom.xml file. https://github.com/checkstyle/checkstyle/blob/20733949774a9accb7cd1a15b12da6b0eb795627/pom.xml#L2622 (2025)
[42] Torlak, E., Chandra, S.: Effective interprocedural resource leak detection. In: International Conference on Software Engineering (ICSE), pp. 535–544 (2010)
[43] Utture, A., Palsberg, J.: From leaks to fixes: Automated repairs for resource leak warnings. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 159–171 (2023)
[44] Wang, K., Hussain, A., Zuo, Z., Xu, G.H., Sani, A.A.: Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 389–404 (2017)
[45] Weimer, W., Necula, G.C.: Finding and preventing run-time error handling mistakes. In: Object-oriented programming, systems, languages, and applications (OOPSLA), pp. 419–431 (2004)
[46] Wu, H., Wang, Y., Rountev, A.: Sentinel: generating gui tests for android sensor leaks. In: International Workshop on Automation of Software Test (AST), pp. 27–33. IEEE (2018)
[47] Zhang, H., Wu, H., Rountev, A.: Automated test generation for detection of leaks in android applications. In: International Workshop on Automation of Software Test (AST),

pp. 64–70 (2016)
[48] Zuo, Z.: Personal communication (2021)
[49] Zuo, Z., Thorpe, J., Wang, Y., Pan, Q., Lu, S., Wang, K., Xu, G.H., Wang, L., Li, X.: Grapple: A graph system for static finite-state property checking of large-scale systems code. In: EuroSys, pp. 1–17 (2019)