Lightweight and Modular Resource Leak Verification

Martin Kellogg*
University of Washington, Seattle, USA
kelloggm@cs.washington.edu

Narges Shadab*
University of California, Riverside, USA
nshad001@ucr.edu

Manu Sridharan
University of California, Riverside, USA
manu@cs.ucr.edu

Michael D. Ernst
University of Washington, Seattle, USA
mernst@cs.washington.edu

ABSTRACT

A resource leak occurs when a program allocates a resource, such as a socket or file handle, 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.

Static detection of resource leaks requires checking that deallocation methods are always invoked on relevant objects before they become unreachable. 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 the need for a heavyweight, whole-program alias analysis. We developed a baseline leak checker via this approach. The precision of an accumulation analysis can be improved by computing 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). We implemented our techniques for Java in an open-source tool called the Resource Leak Checker. The Resource Leak Checker revealed 49 real resource leaks in widely-deployed software. It scales well, has a manageable false positive rate (comparable to the high-confidence resource leak analysis built into the Eclipse IDE), and imposes only a small annotation burden (1/1500 LoC) for developers.

CCS CONCEPTS

• **Software and its engineering → Software verification**.

KEYWORDS

Pluggable type systems, accumulation analysis, static analysis, typestate analysis, resource leaks

ACM Reference Format:

Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2021. Lightweight and Modular Resource Leak Verification. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468576

*Both authors contributed equally to this research.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468576

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 Java applications [15]. This state-of-the-practice does not differ much from two decades ago [36]. Microsoft engineers consider resource leaks to be one of the most significant development challenges [23]. Preventing resource leaks remains an urgent, difficult, open problem.

Ideally, a tool for preventing resource leaks would be:

- *applicable* to existing code with few code changes,
- *sound*, so that undetected resource leaks do not slip into the program;
- *precise*, so that developers are not bothered by excessive false positive warnings; and
- *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 [25], 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 [8] and IntelliJ IDEA [19], are fast and applicable to legacy code, but they are unsound. Interprocedural typestate or dataflow analyses [34, 41] 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 [21] can prevent nearly all resource leaks (see section 9.2), but using them would require a significant rewrite for a legacy codebase, a substantial task which is often infeasible.

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. Our contribution is a resource leak verifier that leverages this insight to satisfy all four requirements: it is applicable, sound, precise, and fast.

An accumulation analysis [20] is a special-case of typestate analysis [30]. 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 build in a sound, modular fashion without *any* whole-program alias analysis, improving scalability and usability.

Recent 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 (more so than in previous work [20]) 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 [31]);
- wrapper types, which share their must-call obligations with one of their fields; and,
- storing resources in non-final fields, which might be lazily initialized or written more than once.

To address this need, we introduced an intra-procedural dataflow analysis for alias tracking, and extended it with three sound techniques to improve precision:

- a lightweight ownership transfer system. This system 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 when a resource's must-call obligations can be resolved by closing one of multiple references.
- a system for creating new obligations at locations other than the constructor, 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 1,500 lines of non-comment, non-blank code.

Our contributions 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 (section 6).
- an empirical evaluation: case studies on heavily-used Java programs (section 7.1), an ablation study that shows the contributions of each innovation to the Resource Leak Checker'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).

2 LEAK DETECTION VIA ACCUMULATION

This section presents a sound, modular, accumulation-based resource leak checker ("the Resource Leak Checker"). Sections 3–5 soundly enhance its precision.

The Resource Leak Checker 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* [12] 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 context.

A type system checks 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 A type system for 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 [13] should always be called.

The Must Call type system supports two qualifiers: `@MustCall` and `@MustCallUnknown`. The `@MustCall` qualifier's arguments are

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

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

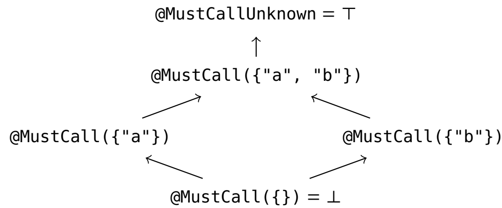

Figure 2: Part of the `MustCall` type hierarchy for representing which methods must be called; the full hierarchy is a lattice of arbitrary size. If an expression's type has qualifier `@MustCall({"a", "b"})`, then the methods "a" and "b" might need to be called before the expression is deallocated. Arrows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

the 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 Resource Leak Checker 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 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 subtyping relationship 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 A type system for 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

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 facts reaching*
3: // *s. Each fact is of the form* $\langle P, e \rangle$*, where P is a set of variables*
4: // *that must-alias e and e is an expression with a nonempty*
5: // *must-call obligation.*
6: $D \leftarrow$ InitialObligations(CFG)
7: **while** D has not reached fixed point **do**
8: **for** $s \in CFG.statements$, $\langle P, e \rangle \in D(s)$ **do**
9: **if** s is *exit* **then**
10: report a must-call violation for e
11: **else if** $\neg$MCSatisfiedAfter(P, s) **then**
12: $kill \leftarrow s$ assigns a variable ? $\{s.LHS\} : \emptyset$
13: $gen \leftarrow$ CreatesAlias(P, s) ? $\{s.LHS\} : \emptyset$
14: $N \leftarrow (P - kill) \cup gen$
15: $\forall t \in CFG.succ(s)\ .\ D(t) \leftarrow D(t) \cup \{\langle N, e \rangle\}$
16: **procedure** InitialObligations(CFG)
17: $D \leftarrow \{s \mapsto \emptyset \mid s \in CFG.statements\}$
18: **for** $p \in CFG.formals$, $t \in CFG.succ(CFG.entry)$ **do**
19: **if** HasObligation(p) **then**
20: $D(t) \leftarrow D(t) \cup \{\langle \{p\}, p \rangle\}$
21: **for** $s \in CFG.statements$ of the form `p = m(p1, p2, ...)` **do**
22: $\forall t \in CFG.succ(s)\ .\ D(t) \leftarrow D(t) \cup$ FactsFromCall(s)
23: **return** D

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: $p \leftarrow s.LHS, c \leftarrow s.RHS$
7: **return** HasObligation(c) ? $\{\langle \{p\}, c \rangle\} : \emptyset$
8: // *Is the must-call obligation for P satisfied after s?*
9: **procedure** MCSatisfiedAfter(P, s)
10: **return** $\exists p \in P$. MCAfter$(p, s) \subseteq$ CMAfter(p, s)
11: // *Does s introduce a must-alias for a var in P?*
12: **procedure** CreatesAlias(P, s)
13: **return** $\exists \mathrm{q} \in P$. 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

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

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

called. The subtyping rule is:

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

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 intra-procedural 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 $s.RHS = e$.

Pseudocode. Algorithm 1 gives 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 6 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 9), the obligation for e has not been satisfied, and an error is reported. Otherwise, the algorithm checks if the obligation for e is satisfied after s (line 11). 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

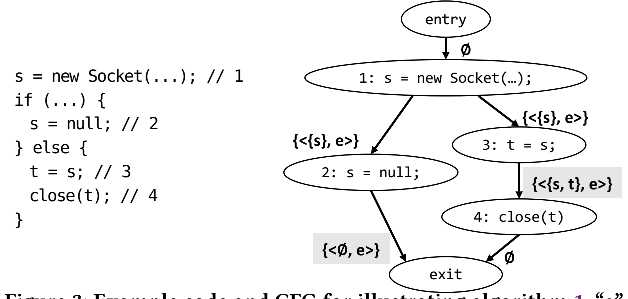

Figure 3: Example code and CFG for illustrating algorithm 1. "e" is "`new Socket(...)`". Non-shaded facts are created by InitialObligations, and shaded facts are propagated by the fixed-point loop.

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 12–14. 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 15 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 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 TRANSFER

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 9.2), 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 Resource Leak Checker'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

Additional 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).

[2]The Resource Leak Checker treats all static fields as non-owning, meaning that no assignment to one can satisfy a must-call obligation. In our case studies, we did not observe any assignments of expressions with non-empty must-call obligations to static fields. We leave handling owning static fields to future work.

For final fields, our checking enforces the "resource acquisition is initialization (RAII)" programming idiom [31]. 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 Resource Leak Checker 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 the Called Methods Checker (section 2.3).

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 Resource Leak Checker 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 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 [8, 34]. 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 [34].

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 [32]):

```
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 Resource Leak Checker 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 reassigning 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 Resource Leak Checker 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 $\{ p_i \mid p_i$ 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 Resource Leak Checker 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 = ...;
  }
}
```

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 7.1.2 for details. "Wall-clock time" is the median of five trials.

	LoC	Resources	Resource leaks	False positives	Annota-tions	Code changes	Wall-clock time
apache/zookeeper:zookeeper-server	45,248	177	13	48	122	5	1m 24s
apache/hadoop:hadoop-hdfs-project/hadoop-hdfs	151,595	365	23	49	117	13	16m 21s
apache/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	-

In the lifetime of a `SocketContainer` object, `sock` might be reassigned 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 Resource Leak Checker 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 inferece 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 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 [33]):

```
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 Resource Leak Checker 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 IMPLEMENTATION

We implemented the Resource Leak Checker on top of the Checker Framework [26], 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 Resource Leak Checker's implementation, our experimental machinery, and the annotated versions of our case study programs, are publicly available at https://doi.org/10.5281/zenodo.4902321.

7.1 Case studies on open-source projects

We selected 3 popular open-source projects that were analyzed by prior work [41]. 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 a smaller open-source project maintained by one of the authors, to simulate the Resource Leak Checker's expected use case, where the user is already familiar with the code under analysis (see section 7.1.3).

```
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;
}
```

Figure 4: A resource leak that the Resource Leak Checker found in Hadoop. Hadoop's developers accepted our pull request to fix it [28].

For each case study, our methodology was as follows. (1) We modified the build system to run the Resource Leak Checker 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 7.1.2). (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 Resource Leak Checker found multiple serious resource leaks in every program. The Resource Leak Checker'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.

At the time of writing, the developers of the case study programs have validated and accepted patches for 16 resource leaks discovered by our tool, including at least one for each project. No patches we have submitted this way have been rejected.

7.1.1 *True and false positive examples.*

This section gives examples of warnings reported by the Resource Leak Checker.

Figure 4 contains 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 [28]. 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

```
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();
}
```

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

Table 2: The annotations we wrote in the case study programs.

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

Checker Framework's type inference algorithm for Java generics, which the Checker Framework developers are working to fix [24]. The second most common reason (causing 15%) was a generic container object like `java.util.Optional` taking ownership of a resource, such as the example in fig. 5. Our lightweight ownership system does not support transferring ownership to generic parameters, so the Resource Leak Checker issues an error when `Optional.of` is returned. In this case, the use of the `Optional` class is unnecessary and complicates the code [9]. If `Optional` was replaced by a nullable Java reference, the Resource Leak Checker 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 [26].

7.1.2 *Annotations and code changes.*

We wrote about one annotation per 1,500 lines of code (table 2).

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 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.

Table 3: The number of false positives in our case studies ("RLC") and when disabling each of 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	54	48
apache/hadoop	97	184	52	49
apache/hbase	82	93	26	22
plume-lib/plume-util	4	11	3	2
Total	300	446	135	121

7.1.3 Simulating the user experience. To simulate the experience of a typical user who understands the codebase being analyzed, one author used the Resource Leak Checker 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 Resource Leak Checker 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.

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 Resource Leak Checker'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 smaller amount: non-final, owning field reassignments are rare.

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 [8]. We used version 4.18.0.
- Grapple [41], 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 Resource Leak Checker. The Resource Leak Checker 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: Eclipse, Grapple, and the Resource Leak Checker. As is standard, 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 Resource Leak Checker 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" [41]. 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 [35] to achieve both context- and path-sensitivity. Grapple contains four checkers, of which two are useful for detecting resource leaks. Unlike the Resource Leak Checker, Grapple is unsound, as it performs a fixed bounded unrolling of loops to make path sensitivity tractable. The Resource Leak Checker 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 [40].

Table 5: Run times of resource leak checking tools.

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

The Grapple authors evaluated their tool on earlier versions of the first three case study programs in section 7.1 [41]. 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 the 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 one of the tools. For example, Grapple analyzed test code, but our experiments did not write annotations in test code nor type-check it. 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 Resource Leak Checker. Some of Grapple's false positives are reports about types like `java.io.StringWriter` that have no underlying resource that needs to be closed. (These reports were mis-classified as true positives in [41], 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). On the flip side, the Resource Leak Checker has lower precision (more false positives) than Grapple, and it requires effort to annotate the source code.

Grapple runs can take many hours (run times are from [41]), whereas the Resource Leak Checker runs in minutes (table 5). Further, Grapple is not modular, so if the user edits their program, Grapple must be re-run from scratch [40]. After a code edit, the Resource Leak Checker only needs to re-analyze modified code (and possibly its dependents if the modified code's interface changed).

8 LIMITATIONS AND THREATS TO VALIDITY

Like any tool that analyzes source code, the Resource Leak Checker only gives guarantees for code that it checks: 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 Resource Leak Checker 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 Resource Leak Checker assumes that 1) any errors issued can be ignored, and 2) all annotations written by the programmer are correct.

The Resource Leak Checker 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 Resource Leak Checker to miss resource leaks.

The results of our experiments may not generalize, compromising the external validity of the experimental results. The Resource Leak Checker may produce more false positives, require more annotations, or be more difficult to use if applied to other programs. Case studies on legacy code represents a worst case for a source code analysis tool. Using the Resource Leak Checker 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 Resource Leak Checker or in the design of its analyses. We have mitigated this threat with code review and an extensive test suite for the Resource Leak Checker: 119 test classes containing 3,776 lines of non-comment, non-blank code. This test suite is publicly available and distributed with the Resource Leak Checker.

9 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.

9.1 Analysis-based approaches

Static analysis. Tracker [34] performs inter-procedural dataflow analysis to detect resource leaks, with various additional features to make the tool practical, including issue prioritization and handling of wrapper types. Tracker avoids whole-program alias analysis to improve scalability, instead using a local, access-path-based approach. While Tracker scales to large programs, it is deliberately unsound, unlike the Resource Leak Checker.

The Eclipse Compiler for Java includes a dataflow-based bug-finder for resource leaks [8]. Its analysis uses a fixed set of ownership heuristics and a fixed list of wrapper classes; unlike the Resource Leak Checker, it is unsound. It is very fast. Similar analyses—with similar trade-offs compared to the Resource Leak Checker—exist in other heuristic bug-finding tools, including SpotBugs [29], PMD [27], and Infer [18]. Section 7.3.1 experimentally evaluates the Eclipse analysis. Relda and Relda2 [17, 38] are unsound resource-leak detection approaches that are specialized to the Android framework with call graphs that model the framework's use of callbacks for releasing resources.

Typestate analysis [11, 30] can be used to find resource leaks. Grapple [41] 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 Resource Leak Checker, Grapple is more precise but suffers from unsoundness and longer run times. Section 7.3.2 gives a more detailed comparison to Grapple.

The CLOSER [7] automatically inserts Java code to dispose of resources when they are no longer "live" according to its dataflow analysis. Their approach requires an expensive alias analysis for soundness, as well as manually-provided aliasing specifications for linked libraries. The Resource Leak Checker uses accumulation analysis [10, 20] to achieve soundness without the need for a whole-program alias analysis.

Dynamic analysis. Some approaches use dynamic analysis to ameliorate leaks. Resco [6] 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 [37, 39]. 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 Resource Leak Checker verifies that no resource leaks are present.

Data sets and surveys. The DroidLeaks benchmark [22] is a set of Android apps with known resource leaks. Unfortunately, it includes only the compiled apps. The Resource Leak Checker runs on source code, so we were unable to run the Resource Leak Checker on DroidLeaks. Ghanavati et al. [15] 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 Resource Leak Checker to more programs from their study.

9.2 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 ownership types in Rust [21] 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 [21]; 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 Resource Leak Checker has less power than Rust's ownership types; it cannot prevent double-free or use-after-free defects. But, the Resource Leak Checker'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 [25] was discussed in section 1. Java also provides finalizer methods [16, 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 [36] 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 Resource Leak Checker.

Previous work has performed modular typestate analysis for annotated Java programs [4] or proposed typestate-oriented programming languages with modular typestate checking [1, 14]. The type systems of these approaches can express arbitrary typestate properties, beyond what can be checked with the Resource Leak Checker. However, these systems impose restrictions on aliasing and a higher type annotation burden than the Resource Leak Checker, making adoption for existing code more challenging.

10 CONCLUSION

We have developed a new, sound, modular approach for detecting and preventing resource leaks in large-scale Java 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.

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.

In future work, we plan to develop improved inference techniques for lightweight ownership annotations. Since these annotations can be added anywhere without impacting soundness (they are verified, not trusted), genetic search and machine-learning techniques could be used to introduce them, using the warnings emitted by the Resource Leak Checker as the fitness function.

Acknowledgments Thanks to Rashmi Mudduluru, Ben Kushigian, Chandrakana Nandi, and the anonymous reviewers for their comments on earlier versions of this paper. This research was supported in part by the National Science Foundation under grants CCF-2005889 and CCF-2007024, and also by a gift from Oracle Labs and a Google Research Award.

REFERENCES

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *OOPSLA Companion: Object-Oriented Programming Systems, Languages, and Applications*. Orlando, FL, USA, 1015–1022.

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

[3] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. 2015. 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*. Lincoln, NE, USA, 669–679.

[4] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. In *OOPSLA 2007, Object-Oriented Programming Systems, Languages, and Applications*. Montreal, Canada, 301–320.

[5] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, Berlin, Heidelberg.

[6] Ziying Dai, Xiaoguang Mao, Yan Lei, Xiaomin Wan, and Kerong Ben. 2013. Resco: Automatic collection of leaked resources. *IEICE TRANSACTIONS on Information and Systems* 96, 1 (2013), 28–39.

[7] Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. 2008. The CLOSER: automating resource management in Java. In *International symposium on Memory management*. 1–10.

[8] Eclipse developers. 2020. 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. Accessed 3 February 2021.

[9] Michael D. Ernst. 2016. Nothing is better than the Optional type. https://homes.cs.washington.edu/~mernst/advice/nothing-is-better-than-optional.html.

[10] Manuel Fähndrich and K. Rustan M. Leino. 2003. Heap Monotonic Typestates. In *IWACO 2003: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*. Darmstadt, Germany.

[11] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM TOSEM* 17, 2, Article Article 9 (2008), 34 pages.

[12] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. Atlanta, GA, USA, 192–203.

[13] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995. *Design Patterns*. Addison-Wesley, Reading, MA.

[14] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4 (2014), 12:1–44.

[15] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and resource leak defects and their repairs in Java projects. *Empirical Software Engineering* 25, 1 (2020), 678–718.

[16] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional.

[17] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *Automated Software Engineering (ASE)*. IEEE, 389–398.

[18] Infer developers. 2021. Resource leak in Java. https://fbinfer.com/docs/checkers-bug-types#resource-leak-in-java. Accessed 4 February 2021.

[19] JetBrains. 2020. List of Java Inspections. https://www.jetbrains.com/help/idea/list-of-java-inspections.html#resource-management. Accessed 5 February 2021.

[20] Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. 2020. Verifying Object Construction. In *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*. Seoul, Korea.

[21] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. https://doc.rust-lang.org/1.50.0/book/

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

[23] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. 2015. 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)*. Bergamo, Italy.

[24] Suzanne Millstein. 2016. Implement Java 8 type argument inference. https://github.com/typetools/checker-framework/issues/979. Accessed 17 April 2020.

[25] Oracle. 2020. The try-with-resources Statement (The Java Tutorials). https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html. Accessed 24 February 2021.

[26] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*. Seattle, WA, USA, 201–212.

[27] PMD developers. 2021. CloseResource. https://pmd.github.io/pmd-6.31.0/pmd_rules_java_errorprone.html#closeresource. Accessed 4 February 2021.

[28] Narges Shadab. 2021. HDFS-15791. Possible Resource Leak in FSImageFormatProtobuf. https://github.com/apache/hadoop/pull/2652. Accessed 16 June 2021.

[29] SpotBugs developers. 2021. 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-unsatisfied-obligation. Accessed 4 February 2021.

[30] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE* SE-12, 1 (Jan. 1986), 157–171.

[31] Bjarne Stroustrup. 1994. 16.5, Resource Management. In *The design and evolution of C++*. Pearson Education India, 388–389.

[32] The Apache Hadoop developers. 2018. 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. Accessed 22 February 2021.

[33] The Apache ZooKeeper developers. 2020. Learner.java. https://github.com/apache/zookeeper/blob/c42c8c94085ed1d94a22158fbdfe2945118a82bc/zookeeper-server/src/main/java/org/apache/zookeeper/server/quorum/Learner.java#L465. Accessed 24 February 2021.

[34] Emina Torlak and Satish Chandra. 2010. Effective interprocedural resource leak detection. In *International Conference on Software Engineering (ICSE)*. 535–544.

[35] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing (Harry) Xu, and Ardalan Amiri Sani. 2017. 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)*. 389–404.

[36] Westley Weimer and George C Necula. 2004. Finding and preventing run-time error handling mistakes. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*. 419–431.

[37] Haowei Wu, Yan Wang, and Atanas Rountev. 2018. Sentinel: generating GUI tests for Android sensor leaks. In *International Workshop on Automation of Software Test (AST)*. IEEE, 27–33.

[38] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. 2016. Relda2: an effective static analysis tool for resource leak detection in Android apps. In *Automated Software Engineering (ASE)*. IEEE, 762–767.

[39] Hailong Zhang, Haowei Wu, and Atanas Rountev. 2016. Automated test generation for detection of leaks in Android applications. In *International Workshop on Automation of Software Test (AST)*. 64–70.

[40] Zhiqiang Zuo. 2021. Personal communication.

[41] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *EuroSys*. 1–17.