LeakChecker: Practical Static Memory Leak Detection for Managed Languages

Dacong Yan
Ohio State University

Guoqing Xu
UC Irvine

Shengqian Yang
Ohio State University

Atanas Rountev
Ohio State University

ABSTRACT

Static detection of memory leaks in a managed language such as Java is attractive because it does not rely on any leak-triggering inputs, allowing compile-time tools to find leaks before software is released. A long-standing issue that prevents practical static memory leak detection for Java is that it can be very expensive to statically determine object liveness in large applications. We present a novel (and the first practical) static leak detection technique that bypasses this problem by considering a common leak pattern. In many cases severe leaks occur in loops where, in each iteration, some objects created by the iteration are unnecessarily referenced by objects external to the loop. These unnecessary references are never used in later loop iterations. Based on this insight, we shift our focus from computing liveness, which is very difficult to achieve precisely and efficiently for large programs, to the easier goal of identifying objects that flow out of a loop but never flow back in. We formalize this analysis using a type and effect system and present its key properties. The analysis was implemented in a tool called LeakChecker and used to detect leaks in eight real-world programs, such as Eclipse, Derby, and log4j. LeakChecker not only identified known leaks, but also discovered new ones whose causes were unknown beforehand, while exhibiting a false positive rate suitable for practical use.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, experimentation, measurement

Keywords

LeakChecker, memory leak detection, static analysis

CGO '14, February 15 - 19 2014, Orlando, FL, USA

http://dx.doi.org/10.1145/2544137.2544151

1. INTRODUCTION

In managed languages such as Java and C#, developers do not need to worry about memory correctness issues such as dangling pointers and double free errors. However, it remains challenging to avoid leaks. A memory leak in a managed language is caused by keeping unnecessary references to objects that are no longer used. These objects cannot be reclaimed by the garbage collector (GC), often leading to severe performance degradation and even program crashes.

Problems and Motivation Static analysis techniques [19, 36, 20, 26, 8, 22, 32] have been widely used to detect memory leaks for unmanaged languages such as C and C++. The explicit memory management in such languages allows the formulation of leak detection as a reachability problem—a control-flow path that creates an object but does not free it may reveal a leak. This formulation cannot be adopted for managed languages, because object deallocation is done automatically by GC. To the best of our knowledge, [29] and [10] are the only two techniques that can statically detect Java memory leaks. At the core of [29] is an algorithm to detect live regions of arrays, while [10] uses shape analysis to identify the objects that are reachable but no longer used. However, there is no evidence that precisely computing array live regions [29] or performing bi-abduction [10] can scale to large-scale applications such as Eclipse, and any attempt to trade off precision for efficiency can lead to reports that are of little value due to great numbers of false warnings. In addition, no evaluation is provided in [29] and [10], and thus, their effectiveness is unclear.

Dynamic analysis [12, 24, 18, 6, 27, 21, 38, 37] is typically used to find memory leaks in managed languages. Existing dynamic analyses are debugging techniques that require appropriate test inputs and can detect leaks only if they are triggered in a test execution. It can be very difficult to find such leak-triggering test inputs, especially during development and in-house testing, when it may be complicated to set up appropriate inputs and execution environments. This is particularly the case with component-based software (such as the development of Eclipse plugins and smartphone apps): components are developed separately and tested only in simulated environments; problems may be seen only after they are shipped and start communicating with other components in production settings. In this regard, a static analysis is highly desirable because it can detect leaks without running a program, thereby leading to improved software quality.

Challenges The major challenge for developing a static

memory leak detector for managed languages is the difficulty of precisely computing *object liveness* properties. Even with highly precise heap modeling and data-flow analysis, it is still expensive to determine precisely whether an object would be used after a certain point in the program execution. The second challenge is that detecting and reporting unnecessary references at a low level (e.g., at heap reads and writes) can be of very limited help because such reads and writes can be far away from the root cause of the leak. For example, reporting the statement that writes an object into a `HashMap.Entry` without any context does not provide any suggestion as to how the leak can be fixed. A useful static analysis should be more informative and should provide high-level information more closely related to the application semantics and the cause of the leak. In addition, such precise static analysis should have practical cost when analyzing real-world large-scale applications.

Insight We present LeakChecker, the first practical static leak detector for Java that overcomes all these challenges by exploiting developer insight, and by identifying and reporting unnecessary references at a higher level of abstraction. An important observation that motivates the design of LeakChecker is that a severe leak is often related to frequently occurring program events. If each such event does not appropriately clean up a small number of references, unnecessary references can quickly accumulate and cause the memory footprint to grow. These events include, for example, database transactions, processing of user requests in web servers, iterative refinements of certain program properties in a static analysis, etc. For example, in Eclipse 3.2, a number of objects are unnecessarily kept alive every time a diff between two zip files is performed [6, 21, 37]. Comparing a few large zip files can quickly make Eclipse run out of memory. In general, an object created by one event instance may escape this instance and be used by future instances of the event. However, if such an escaping object is never used by future event instances, it is very likely to be a leaking object.

Events are often generated by loops. LeakChecker *analyzes each important loop in a program and detects the objects that escape one iteration of the loop and never flow back into any later iteration from the memory locations to which they escape.* A large-scale application may have a large number of loops, and precise analysis of each one can be prohibitively expensive. The developer usually has a clear understanding of which loop is the "main" event loop and which loop contains performance-critical computations; these loops are given as input to LeakChecker.

To further improve tool usefulness, the developer can also specify a repeatedly-executed code region (not a loop) for checking. This is particularly suitable for component-based software where the developer of a component does not have access to the event loop. For example, the developer of an Eclipse [13] plugin could specify the method that achieves the core plugin functionality as a checkable region—this method may be invoked in an invisible loop located in another plugin or in the framework.

The benefit of using a developer's specification is twofold: (1) memory leak detection can be performed within a relatively small scope, leading to improved practicality and scalability; and (2) the reported leaks are easy to understand and fix, because their root causes are very likely to be the operations that store the leaking objects into objects created outside of the region. Once the important loops and code regions are specified by the tool user, the rest of the approach is fully automated. Because any repeatedly-executed code region can be thought of as the body of an (artificial) loop, the paper discusses only leaks in a loop.

Analysis Technique LeakChecker attempts to identify a path p_{out}—a sequence of statements that write to heap objects—through which an object escapes a loop iteration, as well as a path p_{in} of heap read statements through which an object flows back into a loop iteration. Identification of these two paths considers inter-procedural control flow with properly matched method calls and returns. Objects that only flow out through p_{out} but do not flow back in through an appropriate p_{in} are immediately considered leaking. For objects with a proper p_{in}, we further consider two conditions. First, p_{out} and p_{in} should be related to the same outside object. This condition holds when, for the last heap write statement $a.fld = b$ in p_{out} and the first heap read statement $c = d.fld$ in p_{in}, variables a and d may point to the same outside object. Second, we use an *extended recency abstraction* to check whether the loop iteration associated with p_{out} occurs earlier than the one associated with p_{in}. To achieve this, the analysis distinguishes objects by the loop iteration in which they are created. If both conditions hold, the object is considered to be properly shared between iterations; otherwise, it is reported as leaking. The analysis is formalized as a type and effect system described in Section 3. The analysis implementation (Section 4), employs a demand-driven context-free-language (CFL)-reachability formulation to explore p_{out} and p_{in} individually for each object created inside the loop, without requiring an initial whole-program analysis. This on-demand nature is particularly suitable for analyzing partial programs and components.

Key to the success of LeakChecker is the shift of focus from computing object liveness, which is very difficult to achieve precisely and efficiently for large programs, to the easier goal of identifying objects that flow out of a loop but never flow back in. This leak pattern is inspired by experience from dynamic leak detectors (e.g., [24, 21, 37]), where repeatedly-executed code regions and objects escaping from them are often shown to be the culprits. LeakChecker was implemented using the Soot analysis framework [34] and evaluated on eight large Java applications that have memory leaks. The tool found both known and unknown leaks in all applications, and reported comprehensive context information that can help to quickly identify their root causes. These promising initial findings demonstrate that the proposed static checking technique can be used successfully to find potential leaks during real-world software development.

2. OVERVIEW

This section presents an overview of the static analysis used in LeakChecker. Figure 1 shows a simple example adapted from the SPECjbb2000 benchmark. `Order` objects are created (line 5) and processed by a `Transaction` (line 6). Each transaction contains `Customers` (lines 12–15), and order processing saves the `Order` in field `curr` of the transaction (line 19) and adds it into one of the `Customer`s (line 22). Before an `Order` is processed, the transaction first displays its current order in `this.curr` (lines 26–30), which is set in the previous iteration. At the end of `display`, the current order is removed from `curr` (line 29). While the developer thinks this removal will make the `Order` object

unreachable, he/she forgets to clean up references from the `Customer` object. These unnecessary references can lead to a severe memory leak.

Extended recency abstraction We first define a new abstraction for heap objects, called *extended recency abstraction* (ERA), which will be computed by the type and effect system described in Section 3. ERA extends the traditional notion of recency abstraction [25, 3] by distinguishing the objects that are carried over from one iteration to another from those that escape the loop but never flow back in. The terms "object" and "allocation site" will be used to refer to a static abstraction of heap objects (the *new* expression that created the object), while "instance" will refer to a run-time instance of the abstraction. The ERA for an object can have one of four abstract values: $\bar{\mathbf{o}}$ (outside), $\bar{\mathbf{c}}$ (current), $\bar{\mathbf{f}}$ (future), and $\top$ (unknown). For a particular loop l, an object whose ERA is $\bar{\mathbf{o}}$ with respect to l must be created outside l; otherwise, the object is created inside the loop. If the object's ERA is $\bar{\mathbf{c}}$, the object must be *iteration-local*. To illustrate, consider a run-time iteration i of loop l where an instance of an allocation site a is created. If a's ERA is $\bar{\mathbf{c}}$, this guarantees that the instance must die before iteration i finishes. If a's ERA is $\bar{\mathbf{f}}$, this instance may escape iteration i, and if it does escape, it may be used in the loop (i.e., it may flow back) in a later iteration. Finally, a's ERA value of $\top$ implies that this instance may escape iteration i, and if it does escape, it will *not* be used in a later iteration.

Example We use a_k to denote the allocation site at line k of Figure 1. Consider ERAs with respect to the loop at line 3. The ERAs for `Transaction` a_2, array a_{10}, `Customer` a_{13}, and `Order` array a_{34} are all $\bar{\mathbf{o}}$, because they are created before the loop starts. The ERA for a_5 is $\bar{\mathbf{f}}$ because every instance of `Order` escapes the iteration in which it is created, and is used in the next iteration (at line 26). We are particularly interested in objects whose ERA is $\bar{\mathbf{f}}$ or $\top$, because iteration-local objects can never be leaks for the loop.

Transitive flows-out relationship To understand the reference flow, we compute a transitive flows-out relationship $d \succeq_g^* b$ between an object d whose ERA is $\bar{\mathbf{f}}$ or $\top$ and an object b whose ERA is $\bar{\mathbf{o}}$. This relationship indicates that a run-time instance of d may still be live after its creating iteration finishes, because it is inside a data structure that is saved in field g of an instance of the outside object b. In other words, it is the reference $b.g$ that prevents this instance of d from being garbage collected. Note that b must be the closest outside object in this reference path—there does not exist any other outside object c such that $d \succeq_{g'}^* c$ and c can be reached from b. In our example, there are two flows-out relationships for loop l: $a_5 \succeq_{curr}^* a_2$ and $a_5 \succeq_{elem}^* a_{34}$. Here, as typically done in prior work, *elem* denotes an artificial field of the array object representing all array elements. For loop 12, only one flows-out relationship exists: $a_{13} \succeq_{elem}^* a_{10}$.

Transitive flows-in relationship We are also interested in how an object flows into the loop from a field of an outside object. Each flows-in relationship is of the form $d \preceq_g^* b$, indicating that the data structure containing an instance of d created from a previous iteration of the loop is carried over to the current iteration by field g of an instance of the outside object b. The ERA for b must be $\bar{\mathbf{o}}$, and we are interested only in d whose ERA is either $\bar{\mathbf{f}}$ or $\top$. Similarly, b must be the closest outside object in this reference path—there does not exist any other outside object c such that $d \preceq_{g'}^* c$ and c can be reached from b. In the example shown in Figure 1, one flows-in relationship ($a_5 \preceq_{curr}^* a_2$) exists for the loop at line 3, implying that an `Order` instance created in one iteration of the loop may be carried over to a later iteration through field `curr` of the `Transaction` instance.

```
1       static void main(String[] args) {
2         Transaction t = new Transaction();
3         for (int i = 0; i < N; i++) {
4           t.display();
5           Order order = new Order(...);
6           t.process(order);
7         }
8       }
9       class Transaction {
10        Customer[] customers = new Customer[...];
11        Transaction() {
12          for (int i = 0; i < numCusts; i++) {
13            Customer newCust = new Customer(...);
14            customers[i] = newCust;
15          }
16        }
17        Order curr;
18        void process(Order p) {
19          this.curr = p;
20          Customer[] custs = this.customers;
21          Customer c = custs[p.custId];
22          c.addOrder(p);
23          ...// process order
24        }
25        void display() {
26          Order o = this.curr;
27          if(o != null) {
28             ... // display o
29             this.curr = null; //remove o
30          }
31        }
32      }
33      class Customer {
34        Order[] orders = new Order[...];
35        void addOrder(Order y) {
36          Order[] arr = this.orders;
37          arr[...] = y;
38        }
39      }
```

Figure 1: An example adapted from SPECjbb2000.

Leak detection First, any object whose ERA is $\top$ is considered by LeakChecker as a potential leak because it may not flow back into the loop. Second, for each flows-out relationship $d \succeq_g^* b$ such that d's ERA is $\bar{\mathbf{f}}$ and b's ERA is $\bar{\mathbf{o}}$, if there does not exist a corresponding flows-in relationship $d \preceq_g^* b$, LeakChecker considers d as a potential leak because it is saved in a field from which it is never retrieved and used. This field unnecessarily maintains a reference that may keep instances of d from being garbage collected. In our example, the `Order` object a_5 will be recognized and reported as a leaking object, because it has two flows-out relationships but only one flows-in relationship. The reference edge from a_{34} to a_5 is a redundant edge because a_5 is never retrieved from this edge. While in this example the allocation sites are used directly to represent objects, LeakChecker is a context-sensitive analysis that uses a CFL-reachability formulation [30] to distinguish objects created by the same allocation site under different calling contexts. The analysis reports each leaking object (e.g., a_5), the redundant reference edge (e.g., $a_{34}.elem$), and the calling context under which the leaking object is saved through the edge (a by-product of the context-sensitive CFL-reachability computation).

LeakChecker soundness The first phase of the analysis computes ERA for each object and the two kinds of flow relationships, and the second phase matches these relationships to find leaks. The first phase is sound: any object

that flows out of/into a loop through an outside object is correctly identified and the relationships are appropriately classified. The second phase is unsound, due to the matching of flows-in and flows-out relationships. For example, even if a flows-in relationship ($d \preceq_g^* b$) matches a flows-out relationship ($d \succeq_g^* b$), which indicates that d is not a leak, the two run-time instances of b (that an instance of d is added into and retrieved from, respectively) may not be the same. In addition, if g represents an array element (i.e., *elem*), the two heap locations may be different. While in the first case a must-alias analysis could be used to verify whether the two instances of b are the same, for the second case a practical static analysis often cannot precisely handle array indices. Although it may be possible to perform more precise analysis of array indices (e.g., [31]), these techniques are generally expensive and cannot scale to large applications.

Despite these sources of unsoundness, LeakChecker has not missed any known leaks in our studies on eight large applications. This is because in order to have severe effects on program performance, a leak has to exhibit *sustained* behaviors: an allocation site keeps creating instances that escape the loop and are no longer used. Very often these instances escape to an outside container, and this container is never read by later loop iterations. We have not seen any case where only a fixed set of elements are retrieved from the container (but a growing number of elements is untouched), which would cause LeakChecker to miss a sustained leak.

LeakChecker precision For each analyzed loop, the approach can precisely identify the objects that escape the loop through the references that are never read again in the loop. However, these references may be used later after the loop terminates, leading to an imprecise leak report. Hence, the precision of the analysis relies on the appropriate selection of loops to be checked. In the real-world (e.g., enterprise) applications that often suffer from leaks, this task is relatively easy; the case studies from Section 5 illustrate this point. For example, it is natural to select loops that create transactions in a database system, or that process events in an event-based system. To find leaks in an Eclipse plugin, we can use an artificial loop containing the body of a plugin interface method. Of course, it can be difficult to specify such a loop in certain applications, such as program analysis tools, because they often save all objects created in one phase and carry them over to another phase for further processing. However, these applications often do not run repeated tasks, and leaks may not have as significant of an impact in them as in business applications that exhibit repeated behavior.

Another source of imprecision is the lack of precise handling of destructive updates. For example, suppose an inside object flows to a field of an outside object and later this field is assigned `null` without being read in between. If the analysis cannot perform a strong update at the `null` assignment, the flows-out without a matching flows-in will be considered a symptom of a leak and a false warning will be reported. In practice, however, cases in which a reference is removed without being read are quite uncommon. Finally, an unused reference does not necessarily imply that the referenced object is no longer used. The object may be loaded from other (necessary) references and used in later iterations of the loop. In such a case, although the reported object is a false leak, this information is still useful because the redundant reference is worth inspecting and fixing.

Variable	b, c	$\in$	$\mathbb{V}$
Alloc Site	a	$\in$	$\mathbb{A}$
Instance Field	g	$\in$	$\mathbb{F}$
Loop Label	l	$\in$	$\mathbb{L}$
Statement	s	$\in$	$\mathbb{S}$

$s ::= b = c \mid b = new\ a \mid b = c.g \mid b.g = c \mid b = \mathsf{null} \mid s\ ;\ s \mid \text{if (*) then } s \text{ else } s \mid \text{while}^l \text{ (*) do } s$

Iteration Count	j	$::=$	$0 \mid 1 \mid 2 \mid \ldots \quad \in \mathbb{N}$
Iteration Map	ν	$\in$	$\mathbb{L} \to \mathbb{N}$
Loop State	π	$::=$	$\langle l\ ,\ j \rangle \quad l \in \mathbb{L} \cup \{0\}$
Labeled Object	$\hat{o}$	$::=$	$o^{\pi} \quad \in \mathbb{P}$
Heap	σ	$\in$	$\mathbb{P} \times \mathbb{F} \to \mathbb{P} \cup \{\bot\}$
Environment	ρ	$\in$	$\mathbb{V} \to \mathbb{P} \cup \{\bot\}$
Heap Store Effect	Ψ	$::=$	$\emptyset \mid \Psi \cup \{\hat{o}_1 \succ_g^j \hat{o}_2\}$
Heap Load Effect	Ω	$::=$	$\emptyset \mid \Omega \cup \{\hat{o}_1 \prec_g^j \hat{o}_2\}$

Figure 2: A `while` language: syntax and semantic domains.

3. MEMORY LEAK DETECTION

This section formalizes the notion of a memory leak and formally defines LeakChecker's core analysis to find such leaks. First, we define a simple Java-like `while` language, its abstract syntax, and its operational semantics. Using this semantics, we formally define what we mean by a loop-related memory leak in an object-oriented program. Second, we present a type and effect system that abstracts the concrete objects and the flows-in/flows-out relationships. Finally, the memory leak detection algorithm is presented based on the abstract effects computed by the type and effect system.

3.1 A Type and Effect System

Language The abstract syntax of the `while` language and its semantic domains are shown in Figure 2. This language has all important features of an object-oriented language except function calls. They are eluded in this section to ease the formal development. In our implementation, call semantics and calling context sensitivity are modeled by the CFL-reachability formulation that treats the entry and the exit of the same method as a pair of balanced parentheses; this technical issue is elaborated later.

Each loop has a label l and an iteration count j that is incremented per iteration. Map ν maps each loop to its current iteration count. Each run-time object $\hat{o}$ is a regular object annotated with a loop state $\langle l, j \rangle$, indicating that the object is created in the j-th iteration of loop l. If $\hat{o}$ is created in the loop, its j is a positive number; otherwise, its j is always 0. Environment ρ maps a variable b to the heap object $\hat{o}$ pointed-to by b. Heap σ records when a instance field g of one heap object $\hat{o}_2$ points to another heap object $\hat{o}_1$. Both ρ and σ are augmented with $\bot$, representing a `null` value.

A concrete heap store effect captures a reference relationship $\hat{o}_1 \succ_g^j \hat{o}_2$, representing that (a reference to) $\hat{o}_1$ is saved in instance field g of object $\hat{o}_2$ in the j-th iteration of the loop. A concrete heap load effect captures a retrieval action $\hat{o}_1 \prec_g^j \hat{o}_2$ where $\hat{o}_1$ is obtained from field g of $\hat{o}_2$ in the j-th iteration. These two kinds of effects will be employed to compute the transitive flows-out and flows-in relationships.

Concrete Semantics Figure 3 shows the semantics of the language. A judgment $s, \nu, \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega$ starts with a statement s, followed by loop iteration map ν, heap σ, and environment ρ. The execution of s terminates with an iteration map ν', heap σ', environment ρ', heap store effect set Ψ, and heap load effect set Ω.

$$\frac{\sigma' = \sigma[\lambda g.(\hat{o}.g \mapsto \bot)] \quad \hat{o} = (\textbf{if } a \text{ is outside loop } l \textbf{ then } o^{\langle 0,0\rangle} \textbf{ else } o^{\langle l,\nu(l)\rangle})}{b = \text{new } a, \nu, \sigma, \rho \Downarrow \nu, \sigma', \rho[b \mapsto \hat{o}], \emptyset, \emptyset} \quad \text{(New)}$$

$$b = c, \nu, \sigma, \rho \Downarrow \nu, \sigma, \rho[b \mapsto \rho(c)], \emptyset, \emptyset \quad \text{(Assign)}$$

$$b = \text{null}, \nu, \sigma, \rho \Downarrow \nu, \sigma, \rho[a \mapsto \bot], \emptyset, \emptyset \quad \text{(Assign-Null)}$$

$$\frac{\hat{o}_c = \rho(c) \qquad \hat{o}_b = \sigma(\hat{o}_c.g) \qquad \Omega = (\textbf{if } \hat{o}_b = \bot \textbf{ then } \emptyset \textbf{ else } \{\hat{o}_b \prec_g^{\nu(l)} \hat{o}_c\})}{b = c.g, \nu, \sigma, \rho \Downarrow \nu, \sigma, \rho[b \mapsto \hat{o}_b], \emptyset, \Omega} \quad \text{(Load)}$$

$$\frac{\rho(c) = \hat{o}_c \qquad \rho(b) = \hat{o}_b \qquad \Psi = (\textbf{if } \hat{o}_b = \bot \textbf{ then } \emptyset \textbf{ else } \{\hat{o}_b \succ_g^{\nu(l)} \hat{o}_c\})}{c.g = b, \nu, \sigma, \rho \Downarrow \nu, \sigma[\hat{o}_c.g \mapsto \hat{o}_b], \rho, \mu, \Psi, \emptyset} \quad \text{(Store)}$$

$$\frac{s_1, \nu, \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega \qquad s_2, \nu', \sigma', \rho' \Downarrow \nu'', \sigma'', \rho'', \Psi', \Omega'}{s_1; s_2, \nu, \sigma, \rho \Downarrow \nu'', \sigma'', \rho'', \Psi \cup \Psi', \Omega \cup \Omega'} \quad \text{(Comp)}$$

$$\frac{s_1, \nu, \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega}{\text{if } (*) \text{ then } s_1 \text{ else } s_2, \nu, \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega} \quad \text{(If-Else-1)}$$

$$\frac{s, \nu[l \mapsto \nu(l)+1], \sigma, \rho \Downarrow \nu', \sigma', \rho', \Psi, \Omega \qquad \text{while}^l\ (*) \text{ do } s, \nu', \sigma', \rho' \Downarrow \nu'', \sigma'', \rho'', \Psi', \Omega'}{\text{while}^l\ (*) \text{ do } e, \nu, \sigma, \rho \Downarrow \nu'', \sigma'', \rho'', \Psi \cup \Psi', \Omega \cup \Omega'} \quad \text{(While)}$$

Figure 3: Concrete operational semantics.

Rules Assign, Comp, If-Else1, If-Else2 (not shown), and While are standard. In rule new, the loop state pair associated with each run-time object o is $\langle l, \nu(l)\rangle$, where l is the loop in which the object is allocated and $\nu(l)$ is the current iteration count of l. If the allocation site is not in any user-specified loop, $l = 0$, indicating that any object created here is an outside object for any loop. At each store $c.g = b$ into the heap, an effect $\hat{o}_b \succ_g^j \hat{o}_c$ is recorded in Ψ, while at each load $b = c.g$ from the heap, an effect $\hat{o}_b \prec_g^j \hat{o}_c$ is recorded in Ω. Note that if the loop iteration count k of the retrieved object $\hat{o}_b$ is $< j$, the load retrieves an object created in a previous iteration. The operational definition of a memory leak is as follows:

Definition 1. (Leaking Object) *A run-time object $o^{\langle l,j\rangle}$ is the root of an escaping data structure during the execution of loop l if there exists a heap store effect $o^{\langle l,j\rangle} \succ_g^k q^{\langle 0,0\rangle} \in \Psi$. An object $r^{\langle l,j'\rangle}$ is a leaking object if*

$$r^{\langle l,j'\rangle} \succ^* o^{\langle l,j\rangle} \wedge \begin{pmatrix} {}^{(1)}\nexists(m > k) : o^{\langle l,j\rangle} \prec_g^m q^{\langle 0,0\rangle} \in \Omega \ \vee \\ {}^{(2)}\nexists(n > j' \wedge \hat{w} \in \mathbb{P}) : r^{\langle l,j'\rangle} \prec_{g'}^n \hat{w} \in \Omega \end{pmatrix}$$

where $\succ^*$ is the transitive closure of relation $\succ$.

This definition formally describes the leaking objects targeted by our approach. If an inside object o is assigned to a field g of an outside object q in iteration k, o is considered to be the root of an escaping data structure. Any inside object r that is transitively reachable from o (i.e., $r^{\langle l,j'\rangle} \succ^* o^{\langle l,j\rangle}$) is thus considered escaping. The escaping inside object r is a leaking object if (1) the root o of the data structure is leaking, that is, o is never loaded back in any later iteration through $q.g$, or (2) r itself never flows back to the loop in a later iteration. The second condition represents a scenario where a subset of this escaping data structure may flow back into the loop, but this subset does not include r. The formulation does not consider nested loops; although object flow across iterations of nested loops can be easily modeled, we have not found it useful in detecting real-world leaks.

Ext. Recency Abst.	$\tilde{j}$	::=	$\bar{\mathbf{o}} \mid \bar{\mathbf{c}} \mid \bar{\mathbf{f}} \mid \top$	$\in \mathbb{N}_{\mathbb{A}}$
Loop State Abst.	$\tilde{\pi}$	::=	$\langle l, \tilde{j}\rangle \qquad l$	$\in \mathbb{L} \cup \{0\}$
Type	$\tilde{\tau}$	::=	$o^{\tilde{\pi}}$	$\in \mathbb{T}$
Type Environment	Γ	$\in$	$\mathbb{V} \to \mathbb{T} \cup \{\bot, \top\}$	
Type Heap	H	$\in$	$\mathbb{T} \times \mathbb{F} \to \mathbb{T} \cup \{\bot, \top\}$	
Abst. Store Effect	$\tilde{\Psi}$	::=	$\emptyset \mid \tilde{\Psi} \cup \{\tilde{\tau}_1 \succeq_g \tilde{\tau}_2\}$	
Abst. Load Effect	$\tilde{\Omega}$	::=	$\emptyset \mid \tilde{\Omega} \cup \{\tilde{\tau}_1 \preceq_g \tilde{\tau}_2\}$	

(a)

(1) $\tilde{\tau} \sqsubset \hat{o} \quad \Leftrightarrow \quad \tilde{\tau} = \top \vee (\tilde{\tau} = \bot \wedge \hat{o} = \bot) \wedge (\hat{o}.o = \tilde{\tau}.o \wedge \tilde{\tau}.\tilde{\pi} \sqsubset \hat{o}.\pi)$

(2) $\tilde{\pi} \sqsubset \pi \quad \Leftrightarrow \quad \tilde{\pi}.l = \pi.l \wedge (\pi.l = 0 \vee \tilde{\pi}.\tilde{j} \sqsubset \pi.j)$

(3) $\tilde{j} \sqsubset j \quad \Leftrightarrow \quad (j = 0 \wedge \tilde{j} = \bar{\mathbf{o}}) \vee (j > 0 \wedge \tilde{j} \neq \bar{\mathbf{o}})$

(4) $\tilde{\Psi} \sqsubset \Psi \quad \Leftrightarrow \quad \forall(\hat{p_1} \succ_g^j \hat{p_2}) \in \Psi : \exists(\tilde{\tau}_1 \succeq_g \tilde{\tau}_2) \in \tilde{\Psi} : (\tilde{\tau}_1 \sqsubset \hat{p_1}) \wedge (\tilde{\tau}_2 \sqsubset \hat{p_2}) \wedge (\hat{p_1}.\pi.j \neq j \Rightarrow (\tilde{\tau}_1.\tilde{\pi}.\tilde{j} = \bar{\mathbf{f}} \vee \tilde{\tau}_1.\tilde{\pi}.\tilde{j} = \top)))$

(5) $\tilde{\Omega} \sqsubset \Omega \quad \Leftrightarrow \quad \forall(\hat{p_1} \prec_g^j \hat{p_2}) \in \Omega : \exists(\tilde{\tau}_1 \preceq_g \tilde{\tau}_2) \in \tilde{\Omega} : (\tilde{\tau}_1 \sqsubset \hat{p_1}) \wedge (\tilde{\tau}_2 \sqsubset \hat{p_2}) \wedge (\hat{p_1}.\pi.j \neq j \Rightarrow (\tilde{\tau}_1.\tilde{\pi}.\tilde{j} = \bar{\mathbf{f}} \vee \tilde{\tau}_1.\tilde{\pi}.\tilde{j} = \top)))$

(6) $\Gamma \sqsubset \rho \quad \Leftrightarrow \quad (\forall v \in \text{Dom}(\rho) : \Gamma(v) \sqsubset \rho(v))$

(7) $\mathsf{H} \sqsubset \sigma \quad \Leftrightarrow \quad (\forall \hat{o}.g \in \text{Dom}(\sigma) : \exists \tilde{\tau}.g \in \text{Dom}(\mathsf{H}) : \tilde{\tau} \sqsubset \hat{o} \wedge \mathsf{H}(\tilde{\tau}.g) \sqsubset \sigma(\hat{o}.g))$

(b)

Figure 4: Abstract semantic domains: (a) types and abstract effects; (b) abstraction details.

Abstract Semantics We develop a type and effect system that uses an abstract semantics to conservatively approximate the two heap effects. Figure 4(a) shows the types and effects used to abstract the concrete semantics. Iteration counts j are abstracted by ERA $\tilde{j}$ (which can have four abstract values $\bar{\mathbf{o}}$, $\bar{\mathbf{c}}$, $\bar{\mathbf{f}}$, and $\top$), objects $\hat{o}$ are abstracted by types $\tilde{\tau}$, environment ρ is abstracted by type environment Γ, heap σ is abstracted by type heap H, and the two concrete effects in Ψ and Ω are abstracted by the abstract effects in $\tilde{\Psi}$ and $\tilde{\Omega}$. Type environment Γ and type heap H are augmented with $\bot$ and $\top$, which represent, respectively, no type and any type.

Details of how the concrete semantic domains are abstracted can be found in Figure 4(b), where $\sqsubset$ denotes the abstraction relation. In particular, rules (4) and (5) show how the two heap effects are abstracted. An abstract store effect $\tilde{\tau}_1 \preceq_g \tilde{\tau}_2 \in \tilde{\Omega}$ appropriately abstracts concrete effect $\hat{p_1} \prec_g^j \hat{p_2} \in \Omega$ if types $\tilde{\tau}_1$ and $\tilde{\tau}_2$ appropriately abstract $\hat{p_1}$ and $\hat{p_2}$, respectively. In addition, if this store happens in an iteration different from the one where $\hat{p_1}$ is created ($\hat{p_1}.\pi.j \neq j$), the ERA of $\tilde{\tau}_1$ must be either $\bar{\mathbf{f}}$ or $\top$.

The abstract semantics of our analysis is shown in Figure 5 and Figure 6. We discuss only a few important rules. When an allocation site is executed in an iteration, the ERA of the type is set to $\bar{\mathbf{c}}$, indicating that this object is created in the current iteration of the loop (rule TNew). In the beginning of each iteration, the abstract loop state $\tilde{\pi}$ of each type in environment Γ is incremented by rule TWhile using operator $\oplus$ (whose definition is shown in rule (6) of Figure 6). This sets the ERA of each existing loop object (created in previous iterations) to $\top$. If an existing object is iteration-local and cannot escape to the current iteration, its ERA will be updated back to $\bar{\mathbf{c}}$ when its allocation site is encountered again. If the object escapes the loop and flows into the current iteration via a load, its ERA is then updated to $\bar{\mathbf{f}}$ by rule TLoad, indicating that the object is used in an iteration different from the one where it is created. If the object escapes the loop and never flows back in, its ERA will remain $\top$.

At each control flow merge point, type joins are performed

$$\frac{\Gamma' = \Gamma[b \mapsto \tilde{\tau}] \qquad \mathsf{H}' = \mathsf{H}[\lambda g.(\tilde{\tau}.g \mapsto \bot)] \quad \tilde{\tau}.o = a \qquad \tilde{\tau}.\tilde{\pi} = (\textbf{if } a \text{ is outside loop } l \textbf{ then } \langle 0\ ,\ \bar{\mathbf{o}}\rangle \textbf{ else } \langle l\ ,\ \bar{\mathbf{c}}\rangle)}{\Gamma, \mathsf{H} \vdash b = \text{new } a : \Gamma', \mathsf{H}', \emptyset, \emptyset} \text{ (TNew)}$$

$$\Gamma, \mathsf{H} \vdash b = c : \Gamma[b \mapsto \Gamma(c)], \mathsf{H}, \emptyset, \emptyset \quad \text{(TAssign)}$$
$$\Gamma, \mathsf{H} \vdash b = \text{null} : \Gamma'[b \mapsto \bot], \mathsf{H}, \emptyset, \emptyset \quad \text{(TAssign-Null)}$$

$$\frac{\tilde{\tau_c} = \Gamma(c) \quad \tilde{\tau_b} = \mathsf{H}(\tilde{\tau_c}.g) \qquad \tilde{\tau_b}' = (\textbf{if } \tilde{\tau_b}.\tilde{\pi}.\tilde{j} = \top \textbf{ then } \tilde{\tau_b}.o^{\langle \tilde{\tau_b}.\tilde{\pi}.l, \bar{\mathbf{f}}\rangle} \textbf{ else } \tilde{\tau_b}) \quad \tilde{\Omega} = (\textbf{if } \tilde{\tau_b}' \neq \bot \wedge \tilde{\tau_c} \neq \bot \textbf{ then } \{\tilde{\tau_b}' \preceq_g \tilde{\tau_c}\} \textbf{ else } \emptyset)}{\Gamma, \mathsf{H} \vdash b = c.g : \Gamma[b \mapsto \tilde{\tau_b}'], \mathsf{H}, \emptyset, \tilde{\Omega}} \text{ (TLoad)}$$

$$\frac{\tilde{\tau_c} = \Gamma(c) \quad \tilde{\tau_b} = \Gamma(b) \qquad \tilde{\Psi} = (\textbf{if } \tilde{\tau_b} \neq \bot \wedge \tilde{\tau_c} \neq \bot \textbf{ then } \{\tilde{\tau_b} \succeq_g \tilde{\tau_c}\} \textbf{ else } \emptyset)}{\Gamma, \mathsf{H} \vdash c.g = b : \Gamma, \mathsf{H}[\tilde{\tau_c}.g \mapsto \tilde{\tau_b}], \tilde{\Psi}, \emptyset} \text{ (TStore)}$$

$$\frac{\Gamma, \mathsf{H} \vdash s_1 : \Gamma', \mathsf{H}', \tilde{\Psi}, \tilde{\Omega} \qquad \Gamma', \mathsf{H}' \vdash s_2 : \Gamma'', \mathsf{H}'', \tilde{\Psi}', \tilde{\Omega}'}{\Gamma, \mathsf{H} \vdash s_1; s_2 : \Gamma'', \mathsf{H}'', \tilde{\Psi} \cup \tilde{\Psi}', \tilde{\Omega} \cup \tilde{\Omega}'} \text{ (TComp)}$$

$$\frac{\Gamma, \mathsf{H} \vdash s_1 : \Gamma', \mathsf{H}', \tilde{\Psi}, \tilde{\Omega} \qquad \Gamma, \mathsf{H} \vdash s_2 : \Gamma'', \mathsf{H}'', \tilde{\Psi}', \tilde{\Omega}'}{\Gamma, \mathsf{H} \vdash \text{if } (*) \text{ then } s_1 \text{ else } s_2 : \Gamma' \uplus \Gamma'', \mathsf{H}' \uplus \mathsf{H}'', \tilde{\Psi} \cup \tilde{\Psi}', \tilde{\Omega} \cup \tilde{\Omega}'} \text{ (TIf-Else)}$$

$$\frac{\Gamma[\lambda v.(v \mapsto \Gamma(v).o^{\Gamma(v).\tilde{\pi} \oplus 1})], \mathsf{H} \vdash e : \Gamma, \mathsf{H}, \tilde{\Psi}, \tilde{\Omega}}{\Gamma, \mathsf{H} \vdash \text{while}^j\ (*) \text{ do } e : \Gamma, \mathsf{H}, \tilde{\Psi}, \tilde{\Omega}} \text{ (TWhile)}$$

Figure 5: Type rules.

(rule TIf-Else). The definition of the join operator $\uplus$ can be found in rules (1)–(5) in Figure 6. A finite-height type lattice can be defined based on the join operations, with $\top$ and $\bot$ as the maximum and minimum types in the lattice. Types with different allocation sites are not comparable. Because joining any type with $\top$ results in $\top$, LeakChecker reports a potential leak as long as there exists a control flow path in which an object escapes the loop but does not flow back. Abstract effects are recorded by rules TLoad and TStore. Rule TWhile describes a fixed-point computation—the analysis of the loop does not terminate until the type of each object does not change any more.

Example Consider the following simple example:

```
b = new o1; while^l (...) do {
                    c = new o2; d = new o3; e = new o4;
                    m = b.g; if(...) n = m.h;
                    if(...) {b.g = d; d.h = e;} }
```

When our analysis terminates, the abstract semantic domains contain the following values:

$$\begin{aligned}
\Gamma &= [b \mapsto o_1^{\langle 0, \bar{\mathbf{o}}\rangle}, c \mapsto o_2^{\langle l, \bar{\mathbf{c}}\rangle}, d \mapsto o_3^{\langle l, \bar{\mathbf{f}}\rangle}, \\
&\quad e \mapsto o_4^{\langle l, \top\rangle}, m \mapsto o_3^{\langle l, \bar{\mathbf{f}}\rangle}, n \mapsto o_4^{\langle l, \top\rangle}], \\
\mathsf{H} &= [o_1^{\langle 0, \bar{\mathbf{o}}\rangle}.g \mapsto o_3^{\langle l, \bar{\mathbf{f}}\rangle}, o_3^{\langle l, \bar{\mathbf{f}}\rangle}.h \mapsto o_4^{\langle l, \top\rangle}], \\
\tilde{\Psi} &= \{o_3^{\langle l, \bar{\mathbf{f}}\rangle} \succeq_g o_1^{\langle 0, \bar{\mathbf{o}}\rangle}, o_4^{\langle l, \top\rangle} \succeq_h o_3^{\langle l, \bar{\mathbf{f}}\rangle}\}, \\
\tilde{\Omega} &= \{o_3^{\langle l, \bar{\mathbf{f}}\rangle} \preceq_g o_1^{\langle 0, \bar{\mathbf{o}}\rangle}, o_4^{\langle l, \top\rangle} \preceq_h o_3^{\langle l, \bar{\mathbf{f}}\rangle}\}.
\end{aligned}$$

The ERAs for o_1, o_2, o_3, and o_4 are $\bar{\mathbf{o}}$, $\bar{\mathbf{c}}$, $\bar{\mathbf{f}}$, and $\top$, respectively. Here o_1 is an outside object and o_2 is an iteration-local object. Both o_3 and o_4 may escape the loop, and thus their ERA is changed to $\top$ by rule TWhile. If o_3 escapes, it must be used in a later iteration (via $m = b.g$), and thus, its ERA is updated to $\bar{\mathbf{f}}$. While o_4's ERA is also updated to $\bar{\mathbf{f}}$ at load $n = m.h$, it is changed back to $\top$ by the environment join at the end of the first `if` statement (because there exists a CFG path in which it does not flow back into the loop).

[Type Join]

$$(1)\ \tilde{\tau_1} \uplus \tilde{\tau_2} = \begin{cases} \tilde{\tau_1} & \text{if } \tilde{\tau_2} = \bot \\ \tilde{\tau_2} & \text{if } \tilde{\tau_1} = \bot \\ (\tilde{\tau_1}.o)^{\tilde{\tau_1}.\tilde{\pi} \uplus \tilde{\tau_2}.\tilde{\pi}} & \text{if } \tilde{\tau_1}.o = \tilde{\tau_2}.o \\ \top & \text{otherwise} \end{cases}$$

$$(2)\ \tilde{\pi_1} \uplus \tilde{\pi_2} = \begin{cases} \langle \tilde{\pi_1}.l\ ,\ \tilde{\pi_1}.\tilde{j} \uplus \tilde{\pi_2}.\tilde{j}\rangle & \text{if } \tilde{\pi_1}.l = \tilde{\pi_2}.l \\ \langle 0, 0\rangle & \text{otherwise} \end{cases}$$

$$(3)\ \tilde{j_1} \uplus \tilde{j_2} = \begin{cases} \tilde{j_1} & \text{if } \tilde{j_1} = \tilde{j_2} \\ \top & \text{otherwise} \end{cases}$$

$$(4)\ \Gamma_1 \uplus \Gamma_2 = \Gamma_3, \text{ where } \forall v \in \text{Dom}(\Gamma_3),$$
$$\Gamma_3(v) = \begin{cases} \Gamma_1(v) & \text{if } v \in \text{Dom}(\Gamma_1) \text{ and } v \notin \text{Dom}(\Gamma_2) \\ \Gamma_2(v) & \text{if } v \in \text{Dom}(\Gamma_2) \text{ and } v \notin \text{Dom}(\Gamma_1) \\ \Gamma_1(v) \uplus \Gamma_2(v) & \text{if } v \in \text{Dom}(\Gamma_1) \cap \text{Dom}(\Gamma_2) \end{cases}$$

$$(5)\ \mathsf{H}_1 \uplus \mathsf{H}_2 = \mathsf{H}_3, \text{ where } \forall \tilde{\tau}.g \in \text{Dom}(\mathsf{H}_3), \mathsf{H}_3(\tilde{\tau}.g) =$$
$$\begin{cases} \mathsf{H}_1(\tilde{\tau}.g) & \text{if } \tilde{\tau}.g \in \text{Dom}(\mathsf{H}_1) \text{ and } \tilde{\tau}.g \notin \text{Dom}(\mathsf{H}_2) \\ \mathsf{H}_2(\tilde{\tau}.g) & \text{if } \tilde{\tau}.g \in \text{Dom}(\mathsf{H}_2) \text{ and } \tilde{\tau}.g \notin \text{Dom}(\mathsf{H}_1) \\ \mathsf{H}_1(\tilde{\tau}.g) \uplus \mathsf{H}_2(\tilde{\tau}.g) & \text{if } \tilde{\tau}.g \in \text{Dom}(\mathsf{H}_1) \cap \text{Dom}(\mathsf{H}_2) \end{cases}$$

[Operator $\oplus$]

$$(6)\ \tilde{\pi} \oplus 1 = \begin{cases} \tilde{\pi} & \text{if } \tilde{\pi}.\tilde{j} = \bar{\mathbf{o}} \\ \langle \tilde{\pi}.l\ ,\ \top\rangle & \text{otherwise} \end{cases}$$

Figure 6: Join operations on types and domains.

3.2 Leak Detection

Based on the abstract load and store effects computed by the type and effect system, leaks can be detected as follows.

Definition 2. (Flows-out Relation $\succeq_g^*$ and Flows-in Relation $\preceq_g^*$) *Suppose $\succeq^*$ and $\preceq^*$ are the transitive closures of relations $\succeq$ and $\preceq$, respectively. A pair $(\tilde{\tau}_1, \tilde{\tau}_2) \in$ flows-out relation $\succeq_g^*$ if*
$\tilde{\tau_1}.\tilde{\pi}.\tilde{j} \neq \bar{\mathbf{o}} \wedge \tilde{\tau_2}.\tilde{\pi}.\tilde{j} = \bar{\mathbf{o}} \wedge \exists \tilde{\tau_3} : \tilde{\tau_3}.\tilde{\pi}.\tilde{j} \neq \bar{\mathbf{o}} \wedge \tilde{\tau_1} \succeq^* \tilde{\tau_3} \wedge \tilde{\tau_3} \succeq_g \tilde{\tau_2}$.
Similarly, a pair $(\tilde{\tau}_1, \tilde{\tau}_2) \in$ flows-in relation $\preceq_g^*$ if
$\tilde{\tau_1}.\tilde{\pi}.\tilde{j} \neq \bar{\mathbf{o}} \wedge \tilde{\tau_2}.\tilde{\pi}.\tilde{j} = \bar{\mathbf{o}} \wedge \exists \tilde{\tau_3} : \tilde{\tau_3}.\tilde{\pi}.\tilde{j} \neq \bar{\mathbf{o}} \wedge \tilde{\tau_1} \preceq^* \tilde{\tau_3} \wedge \tilde{\tau_3} \preceq_g \tilde{\tau_2}$.

As discussed in Section 2, $\tilde{\tau}_1 \succeq_g^* \tilde{\tau}_2$ if $\tilde{\tau}_1$ represents an inside object, $\tilde{\tau}_2$ represents an outside object, and there exists a sequence of store effects that connects them. Field g is the field of $\tilde{\tau}_2$ through which the leaking data structure is saved. Based on the definitions of $\succeq_g^*$ and $\preceq_g^*$, we give the following definition of a memory leak.

Definition 3. (Memory Leak) *An object o is a leaking object, if it has a type $\tilde{\tau}$ such that*
$\tilde{\tau}.\tilde{\pi}.\tilde{j} = \top \vee (\tilde{\tau}.\tilde{\pi}.\tilde{j} = \bar{\mathbf{f}} \wedge \exists (\tilde{\tau}, \tilde{\tau}') \in \succeq_g^* : (\tilde{\tau}, \tilde{\tau}') \notin \preceq_g^*)$

In the above example, object o_4 is a leaking object, because its ERA is $\top$.

4. IMPLEMENTATION

Calls and Calling Context LeakChecker is implemented based on the Soot Java program analysis framework [34]. A demand-driven CFL-reachability formulation of points-to analysis [30] is used to identify leaking objects interprocedurally and with modeling of calling context. In such a formulation, program semantics is encoded as a flow graph in which nodes represent variables and edges represent propagation of object references. Points-to relationships are determined by traversing the graph, and flows-in/out information is derived from them. For example, at a heap store statement $c.g = b$, the points-to sets of c and b are computed on demand to identify flows-out pairs of objects. The analysis is calling-context-sensitive in that edge labels representing interprocedural control flow—i.e., method calls and

Case	*Mtds*	*Stmts*	*Time* (s)	*LO*	*LS*	*FP*	*FPR*
SPECjbb2000	4717	97387	82	95	21	8	38.1%
Eclipse Diff	26300	400647	1638	309	7	3	42.9%
Eclipse CP	30487	466307	1000	123	7	4	57.1%
Mckoi	20373	342041	1985	450	18	17	94.4%
MySQL Connector/J	11868	210053	1059	181	15	9	60%
log4j	3385	62568	35	10	4	0	0%
FindBugs	3817	70177	82	72	9	5	55.6%
Derby	8661	147899	700	165	8	4	50%

Table 1: Analysis results.

returns—along a traversed graph path are required to satisfy a matching parentheses property (defined by a context-free language, thus the term CFL-reachability). With this addition, objects are distinguished not only by their allocation sites and their ERA, but also by their calling contexts. When a leaking object is detected, its allocation site, the field through which it escapes, and the calling context under which it escapes are all reported.

Flow into Library Methods Many popular Java data structures, such as `HashMap` and `ArrayList`, use arrays to store objects. These arrays are read in certain operations that are not meant to retrieve objects. For example, in method `HashMap.put`, entries that match the hashcode of the given key are read from the array to determine whether the key already exists. If a loop calls `put` and these reads are treated as regular object retrievals, LeakChecker may miss some leaks. To avoid this, we distinguish application code and library code, and use a stronger condition to identify leaking objects: if an object is read from the heap by a library class, a flows-in relationship (as defined earlier) exists only when the object is returned to the application code, accounting for calling context. Hence, even if the array is read in `HashMap.put`, LeakChecker does not generate a flows-in relationship because the loaded object is not returned by the method. This treatment is used not only for array objects, but also for objects that are not of an array type. This stronger condition for leak identification is applied by our approach to all library methods in the standard Java libraries.

Pivot Mode For any two leaking objects o_1 and o_2 such that $o_1 \succeq^* o_2$, object o_2 is more likely to be the root of a leaking data structure, and object o_1 can not be garbage collected as long as o_2 is unnecessarily kept alive. In such a case, the leak can be understood and fixed by just inspecting o_2 and removing its unnecessary reference(s). LeakChecker provides a *pivot mode* under which leaking objects such as o_1 are omitted from the leak report. The experiments described in Section 5 use this mode.

5. EMPIRICAL EVALUATION

We evaluated LeakChecker on eight large programs. Some programs (*log4j*, *FindBugs*, and *Derby*) have never been studied before, while leaks in others were discussed in existing work [33, 7, 37]. These programs cover a variety of domains, including enterprise trading, software development, database management, logging, and static program analysis.

5.1 Summary of Results

All experiments were performed on a machine with a 3.4 GHz Quad Core Intel i7-2600 processor, and the analysis was run with a maximum Java heap size of 4 GB. Characteristics of the studied programs and a summary of leak detection results are shown in Table 1. The table shows the number of reachable methods in the call graph (Mtds), the number of Soot's Jimple statements in these methods (Stmts), LeakChecker's analysis time (Time) in seconds, the number of context-sensitive allocation sites in analyzed loops (LO), the number of reported context-sensitive leaking allocation sites (LS), the number of false positives (FP), and the false positive rate (FPR = FP / LS). For each studied program, one suspicious loop was specified for checking.

Due to the client-driven nature of the analysis (checking user-specified loops), LeakChecker is able to quickly detect leaks for all the applications, including large programs such as Eclipse. The approach does not generate many leak warnings, so we verified each warning manually. With detailed leak reports, we pinpointed the root cause of the leak and fixed the underlying defect in less than 2 hours for each report. LeakChecker's average false positive rate is 49.8%, which indicates that is may be suitable for practical use.

5.2 Case Studies

We performed case studies on all eight programs, but due to space limitations only six of them are discussed below.

SPECjbb2000 SPECjbb2000 is a transaction-based system. It has a `TransactionManager` class that runs different types of transactions, and the transaction-creating method is only a few calls away from the `main` method. It contains a loop that, in each iteration, retrieves a command from an input map, and then creates and runs a transaction whose type corresponds to the command received. Thus it is natural to apply LeakChecker on this loop. The tool reported 5 allocation sites (corresponding to 21 context-sensitive allocation sites), among which 4 (under 6 different calling contexts) can be immediately excluded because the outside heap locations they flow to are overwritten in each iteration of the loop. The remaining site allocates `longBTreeNode` objects. These objects are created to hold element objects, when the elements are added to a `longBTree` container data structure. We focused our efforts on these `longBTreeNode` objects.

We found that the calling contexts are particularly useful in understanding the root cause of this problem. In the report, `longBTreeNode` objects are shown to be created under 15 different calling contexts. We first examined the top call sites in these calling contexts. There are only 3 distinct top call sites, and they are all in the method enclosing the specified loop. These call sites correspond to the processing of 3 types of commands: `new_order`, `multiple_orders`, and `payment`. The last one, the only one that is irrelevant to the leak, indicates that `History` objects, a representation of payment history, are saved in the long-lived `Warehouse` objects. However, we found that every time a new `History` object is added, the oldest one will be removed. Through this particular leaking context, these `History` objects cannot cause constant increase in memory footprint.

Excluding `payment` commands, we were left with 13 calling contexts, all related to processing of new orders. They indicate that `Order` objects are kept alive unnecessarily by `longBTreeNode` objects. One such relevant context-sensitive allocation site is shown below:

```
* Leaking Object (longBTreeNode; createlongBTreeNode(...), ln 102)
  Context - at longBTreeNode.Insert(long,Object), ln 760
            at longBTree.put(long,Object), ln 1521
            at District.addOrder(Order), ln 264
            at NewOrderTransaction.process(), ln 293
            at TransactionManager.go(), ln 296
* Outside Object (longBTree; createLongBTree(...), ln 790)
  Context - at District.initDistrict(short,byte), ln 184
            at District.createDistrict(...), ln 100
            at Warehouse.setUsingRandom(short), ln 396
```

```
            at Company.loadWarehouseTable(), ln 761
* Heap Write (r0.<longBTree: longBTreeNode root> = r5)
  Context - at longBTree.put(long,Object), ln 1521
            at District.addOrder(Order), ln 264
            at NewOrderTransaction.process(), ln 293
            at TransactionManager.go(), ln 296
```

`Order` object is stored in the newly created `longBTreeNode` object, which is inserted into the `longBTree` and later itself becomes the root of the `longBTree`:

```
btree.root = btree.root.Insert(key, order)
```

The `longBTree` object is stored in a field of a long-lived outside `District` object to represent orders processed through this district. Thus, `Order` objects are kept alive and leaking.

Eclipse Diff Eclipse is an IDE that allows plugins to be added into a unified platform. Plugins are usually developed separately, but they can interact with each other at run time. It is often unclear to developers how one plugin could be affected by others. For example, the leak in this case manifests only after the structures of two large JAR files are compared multiple times by plugin `org.eclipse.compare`. Files selected for comparison are represented by `ISelection` objects, which are passed into a `runCompare` method, the entry method of this plugin. We created an artificial loop in which `runCompare` is called, and applied the analysis on it.

LeakChecker reported 7 context-sensitive leaking allocation sites. Three of them are for temporary GUI objects (e.g., a temporarily shown dialog to indicate progress of computation) and can be immediately discarded. The rest of them all point to one allocation site that creates `HistoryEntry` objects. The associated contexts indicate that these objects are created when `History.addEntry` is called. `History` records the history of opened editors in a list of `HistoryEntry` objects, and the editors are used to show the comparison results. Calling `runCompare` multiple times would lead to the creation of multiple history entry objects. These objects are added to the list, but not properly cleared. Note that `History` is a class in the platform, and thus it is very difficult for developers to find and fix the bug (in fact, the root cause of this bug was found almost one year after it was reported). LeakChecker started from a code stub that uses the compare plugin, and quickly reported the root cause. To detect this leak using a dynamic analysis, a full-fledged executable program has to be developed to automatically select items in the GUI and trigger the comparison action. This task could be quite challenging for programmers without Eclipse GUI programming/testing experience.

Mckoi Mckoi [23] is an open-source database system. It has a memory leak when used as an embedded application. It is leaking because `DatabaseSystem` objects are kept alive by running threads. We created a simple client that repeatedly establishes a database connection and closes it. When we first ran LeakChecker on the program, there was only one leaking object reported. The reported `LocalBootable` object is a singleton object created only at the first time a connection is established, and the only outside object to which it escapes is the (outside) JDBC driver object. This is a false warning, because at run time it is guaranteed to be one instance of `LocalBootable` created per connection, which cannot be understood by LeakChecker.

LeakChecker fails to detect the leak because threads are not explicitly modeled. To solve the problem, threads that never terminate should be treated as outside objects. However, this is non-trivial as it is generally undecidable to determine whether a thread would terminate. As a workaround, we tag an object as an outside object if (1) it is a thread object (an instance of `java.lang.Thread`) regardless of whether or not it may terminate, and (2) method `start` has been called on this object. After this new modeling was employed, 18 context-sensitive allocation sites were reported. To verify whether they are true leaks, we manually examined the `run` method of each (outside) thread object and found that (1) most of the reported sites are false positives because they escape to threads that must terminate; and (2) the allocation site of `DatabaseSystem` leads to the root cause of the leak, related to non-terminating thread `DatabaseDispatcher`. Due to the lack of a thread termination analysis, we saw a high false positive rate for this program.

log4j log4j [2] is a logging library for Java. When a client application uses `JDBCAppender` to write log messages to a remote database, the memory usage increases significantly. We created a simple program that mimics such a client by sending multiple log requests. Four context-sensitive allocation sites were reported as leaks, all of them related to a list called `removes` in `JDBCAppender`. We inspected the code and found that (1) log requests are first added to a `buffer` list; (2) they are retrieved (but not removed) one by one from the list for processing, and added to `removes` list afterwards; and (3) at the end of one bulk processing, all the request objects in `removes` are removed from `buffer`. However, the `removes` list itself is never cleared, leading to the leak.

FindBugs FindBugs [14] is a static analysis tool in which bug detectors are organized as plugins, while the base framework provides common functionality (e.g., parsing of class files). A leak is exposed when `FindBugs2.execute` is called many times to analyze a large number of JAR files. We created a loop that iterates over a list of JAR files and parses the class files contained in each JAR.

LeakChecker reported 9 leaking allocation sites, 5 of which were obviously irrelevant to the leak. Objects created at these sites are stored into `HashMap` objects reachable from a global `DescriptorFactory` object. Because the `HashMap`s are cleared at the end of the analysis of each JAR file, no objects can be leaking through them. These (false) warnings were reported due to the lack of precise handling of destructive updates. The remaining 4 sites all point to a long-lived `IdentityHashMap` object, to which a number of `MethodInfo` objects are added. However, these `MethodInfo` objects are never used or removed. After inspecting these 4 sites, one can easily fix the leak by appropriately clearing the `IdentityHashMap`.

Derby In Apache Derby 10.2.1.6 [1], a leak can be seen if a `Statement` or a `ResultSet` is not closed after being used in client/server mode. We created a simple loop that executes one SQL query per iteration but does not call `close` on `Statement` or `ResultSet`. Eight leaking allocation sites were reported. Half of them are related to a `Hashtable` in `SectionManager` that saves `ResultSet` objects—these objects are never retrieved, causing the memory leak. All other reported allocation sites are related to saving `Section` objects in a `Stack`. These are false warnings because at the reported sites only one object instance can be created and escape the loop, due to use of the singleton pattern.

5.3 Experience Summary

Our case studies demonstrate that deep implementation knowledge is not required for effective use of LeakChecker.

In the wide variety of programs we studied, loops relevant to leaking behavior can be easily identified/created, even for users unfamiliar with the program. The specified loop can serve as a *client* that interacts with a complicated system. To pinpoint bugs in database systems (e.g., Derby), we only need to create a loop that performs database queries. Similarly, for a plugin-based system such as Eclipse, we can perform checks on plugins, and leak detection can be done regardless of whether the bug is in the plugin or in the base system. This is very useful because it allows testers or performance experts to quickly create the necessary setup to check the code, without the need to dig into the details of a large system, or create leak-triggering test cases. Of course, there may be scenarios where the selection of the loop to be checked is not as straightforward, and additional considerations may be needed: e.g., identifying loops that are likely to frequently invoke important subcomponents of the analyzed component, or using application-specific knowledge to focus on loops whose frequent execution is expected under realistic usage scenarios. In cases where actual run-time frequency information is available, the checking effort could be targeted toward the most frequently executed loops.

A leak report generated by LeakChecker contains both leaking allocation sites and the specific loops they escape from. Understanding why a reported object never flows back into a loop is often sufficient to locate the cause of a defect. Relevant code in the program usually can be easily identified as LeakChecker also reports the calling contexts and escaping store statements for each leaking object. Our experience indicates that, given a detailed LeakChecker report, the developer effort to identify the root cause of a leak is typically small.

In the experiments, most of the false positives were due to internal constraints used by developers to prevent multiple instances of a loop object from escaping the loop. In future work, it is worth investigating how LeakChecker can be extended to detect such code patterns.

6. RELATED WORK

There exists a large body of work on memory leak detection, including both static [19, 36, 20, 26, 8, 22, 32] and dynamic approaches [12, 24, 18, 6, 27, 21, 38, 37]. LeakChecker is the first practical static memory leak detector for managed languages.

Static analyses for memory leak detection Static analysis techniques [19, 36, 20, 26, 8, 22, 32] have been widely used to detect memory leaks for unmanaged languages such as C and C++. The explicit memory management in such languages allows the formulation of leak detection as a reachability problem—any control flow path that creates an object but does not free it may reveal a leak and is thus reported to the user for inspection. Work in [8] defines a reachability problem on the program's guarded value flow graph, and detects leaks by identifying value flows from the source (malloc) to the sink (free). Saturn [36] reduces the problem of leak detection to a boolean satisfiability problem, and uses a SAT-solver to identify potential bugs. Shape analysis based on 3-valued logic [11] has been proposed to assert the absence of leaks in list manipulation functions. Hackett and Rugina [17] identify leaks with a shape analysis that tracks individual heap cells. Orlovich and Rugina [26] use backward dataflow analysis to disprove the feasibility of potential leak errors. Clouseau [19] employs pointer ownership to describe the responsibilities for freeing heap memory, and formulates leak analysis as an ownership constraint system. Work in [20] proposes a type system to describe object ownership for polymorphic containers, and uses type inference to detect constraint violations.

These prior efforts target C and C++whereas we are interested in a garbage-collected language. A reachability formulation cannot find leaks for managed languages, because object deallocation is done automatically by GC. In contrast, LeakChecker exploits developer insight to identify leaking objects at a high, semantic level. For memory leaks in Java programs, work in [29] presents a static live region analysis to detect array-related memory leaks. The problem of detecting liveness regions of arrays is formulated using a constraint graph that models linear inequalities over variables. The approach from [10] uses separation logic and shape analysis to find unused objects in Java applications. However, these two analyses can be prohibitively expensive and they have not been applied on large-scale applications.

Dynamic analyses for memory leak detection Heap analysis tools such as [12] take heap snapshots and visualize the object graph to help users find unnecessary references. However, they do not provide the ability to automatically pinpoint the cause of a memory leak. Work done in the research community uses either growing types [24, 21] (i.e., types with growing number of run-time instances) or object staleness [18, 6, 38] (i.e., the elapsed time since the last use of an object) to identify suspicious data structures that may be related to a memory leak. Work from [37] leverages higher-level application semantics to detect leaks related to transactional code structures. All these existing dynamic analyses require appropriate executable programs and test inputs, and can detect problems only when leaks are triggered in a particular test execution. It may be very difficult to meet these requirements, especially during development and in-house testing. In addition, dynamic approaches cannot work for partial programs such as components, plugins, and mobile apps. The proposed static approach does not have these limitations.

Static object liveness analysis Escape analyses [9, 5, 35, 15] are designed to identify objects whose lifetimes are within the lifetime of the stack frame of the method that allocates the objects. These objects can be stack-allocated for increased performance. Work by Ruggieri and Murtagh [28] uses static dataflow analysis to approximate object lifetimes in order to enable various optimizations on object allocation and deallocation. Gheorghioiu *et al.* propose a static analysis [16] to identify *unitary* allocation sites whose instances are completely disjoint so that these instances can be pre-allocated and reused. Work from [3] presents *recency abstraction*, a technique that distinguishes most-recently-allocated-object (MRAO) and non-MRAO for each allocation site in order to enable strong updates for a points-to analysis. Naik and Aiken [25] propose an iteration count abstraction and a must-not-alias analysis to rule out false data races. Later work [4, 39] detects objects that create disjoint instances in order to identify reusable data structures. All these techniques approximate object liveness using reachability, and thus cannot detect objects that are reachable but no longer used. Instead of computing liveness, LeakChecker takes a loop-centric approach, which detects leaks based on the unique insight that an object that escapes a loop iteration but never flows back in is very likely to be a leak.

7. CONCLUSIONS

This paper presents LeakChecker, the first practical static memory leak detector for Java. Leak detection is based on the observation that an event loop is often the place where severe leaks occur, and these leaks are commonly caused by objects outside the loop keeping unnecessary references to objects created inside the loop. Such a loop often iterates a large number of times, causing these references to accumulate and degrade program performance. LeakChecker uses a novel static analysis to identify such unnecessary references and reports leaks with sufficient information that can quickly help the developer find the root causes and come up with the necessary fixes. We have implemented the analysis and evaluated it on eight large programs with leaks. The experimental results show that LeakChecker successfully finds leaks in each of them and the false positive rate is reasonably low. These promising initial results strongly suggest that the proposed technique can be used in practice to help programmers find and fix memory leaks during development. Future work can investigate algorithmic refinements to achieve higher precision (e.g., through modeling of destructive updates). Approaches to identify suspicious loops to be checked—for example, using structural information extracted from the code, or frequency information from run-time execution—are also of significant interest.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This material is based upon work supported by the U.S. National Science Foundation under grants CCF-0546040, CCF-1017204, and CNS-1321179.

8. REFERENCES

[1] Apache Derby. `db.apache.org/derby`.

[2] Apache log4j. `logging.apache.org/log4j`.

[3] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis Symposium (SAS)*, pages 221–239, 2006.

[4] S. Bhattacharya, M. Nanda, K. Gopinath, and M. Gupta. Reuse, recycle to de-bloat software. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 408–432, 2011.

[5] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 20–34, 1999.

[6] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–72, 2006.

[7] M. D. Bond and K. S. McKinley. Leak pruning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009.

[8] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 480–491, 2007.

[9] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[10] D. Distefano and I. Filipović. Memory leaks detection in Java by bi-abductive inference. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 278–292, 2010.

[11] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *Static Analysis Symposium (SAS)*, pages 115–134, 2000.

[12] Eclipse Memory Analyzer Tool. `www.eclipse.org/mat`.

[13] Eclipse project. `www.eclipse.org`.

[14] FindBugs. `findbugs.sourceforge.net`.

[15] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC)*, pages 82–93, 2000.

[16] O. Gheorghioiu, A. Salcianu, and M. Rinard. Interprocedural compatibility analysis for static object preallocation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284, 2003.

[17] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 310–323, 2005.

[18] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 156–164, 2004.

[19] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, 2003.

[20] D. L. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. In *International Conference on Software Engineering (ICSE)*, pages 252–261, 2006.

[21] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007.

[22] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *International Symposium on Memory Management (ISMM)*, pages 131–140, 2008.

[23] Mckoi SQL database. `www.mckoi.com`.

[24] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 351–377, 2003.

[25] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 327–338, 2007.

[26] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Static Analysis Symposium (SAS)*, pages 405–424, 2006.

[27] D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *International Conference on Automated Software Engineering (ASE)*, pages 194–203, 2007.

[28] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 285–293, 1988.

[29] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in Java. In *International Conference on Compiler Construction (CC)*, pages 50–66, 2000.

[30] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[31] Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138, 2005.

[32] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 254–264, 2012.

[33] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *The USENIX Annual Technical Conference (USENIX)*, pages 307–320, 2008.

[34] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction (CC)*, pages 18–34, 2000.

[35] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 187–206, 1999.

[36] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 115–125, 2005.

[37] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282, 2011.

[38] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[39] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 738–763, 2012.