Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Automatic Recovery from Resource Exhaustion Exceptions by Collecting Leaked Resources*

Zi-ying DAI†1, Xiao-guang MAO†‡1,2, Li-qian CHEN1, Yan LEI1,3

1College of Computer, National University of Defense Technology, Changsha 410073, China

2Laboratory of Science and Technology on Integrated Logistics Support, National University of Defense Technology, Changsha 410073, China

3Department of Computer Science, University of California, Davis, USA

†E-mail: ziyingdai@nudt.edu.cn; xgmao@nudt.edu.cn

Received Nov. X, 2013; Revision accepted XX, 2013; Crosschecked XX, 2013

Abstract: Despite the availability of garbage collectors, programmers must manually manage non-memory finite system resources such as file descriptors. Resource leaks can gradually consume all available resources and cause programs to raise resource exhaustion exceptions. However, programmers commonly provide no effective recovery for resource exhaustion exceptions, which often can cause programs to halt without completing their tasks. In this paper, we propose to automatically recover programs from resource exhaustion exceptions caused by resource leaks. We transform programs so that they can catch resource exhaustion exceptions, collect leaked resources, and then retry failed code. A resource collector is designed to identify leaked resources and safely release them. We implement our approach for Java programs. Experimental results show that our approach can successfully handle resource exhaustion exceptions caused by reported resource leaks and allow programs to continue to complete their tasks with an average execution time increase of 2.52% and negligible byte-code size increases.

Key words: Failure avoidance, Resource leaks, Resource collection, Exception handling, Reliability
doi:XX/jzus.C1000000 **Document code:** A **CLC number:**

1 Introduction

Automatic garbage collection has gained considerable success in many mainstream programming languages, such as Java and C#. A garbage collector relieves programmers from manual memory management and improves productivity and program reliability (Dybvig *et al.*, 1993). However, there are many other non-memory finite system resources (e.g. file descriptors and database connections) that programmers must manage manually. For programs written in Java-like languages, once acquired, a resource must be released by explicitly calling a cleanup method. A *resource leak* is a software bug that occurs when the cleanup method of the resource is not invoked after its last use. Resource leaks are common in Java programs (Torlak and Chandra, 2010). Growing resource leaks can degrade an application's performance and can even result in system crashes due to resource exhaustion.

The large majority of modern programs rely on exception handling constructs to notify abnormal situations and allow customized recoveries from exceptions. When a semantic error occurs or some exceptional situation is encountered, an exception is *thrown* (e.g. `throw` in Java). This exception causes the control flow to transfer from where the exception occurs to a point where the exception is *caught* (e.g. `try` and `catch` in Java). If an exception is not caught within the method where it occurs, it is implicitly propagated to the caller of this method. If all available resources are consumed (or leaked), a fur-

‡ Corresponding author

* Projects (Nos. 61379054 and 91318301) supported by the National Natural Science Foundation of China, and Projects (No. 2012AA011201) supported by National High Technology Research and Development Program of China (863 program)

©Zhejiang University and Springer-Verlag Berlin Heidelberg 2013

ther request for such resources will typically cause the program to throw a *resource exhaustion exception* (REE). For example, a Java program will throw a `FileNotFoundException` *saying "Too many open files"* when no more available file descriptors can be used to open a file.

Programmers can catch exceptions and provide their recovery code. However, the recovery code provided by the the programmer is often unsatisfactory. The Java code in Fig. 1a is such an example. This code snippet is from an old version of Ant[1] and is the running example used throughout this paper. The programmer opens a pattern file in line 5 but forgets to close it. If this method is repeatedly called in cases where there are many pattern files for a task, the available file descriptors can be exhausted. The file open (line 3) can fail with a thrown `FileNotFoundException` that is caught (line 6). The recovery code for this exception provided by the programmer is disappointing because the programmer just logs this exception, re-throws another exception and terminates the execution of this method (line 9), without any recoveries. The further the exception propagates from this method, the less likely the program can be successfully recovered from it. This usually causes the entire program to halt without completing the task. Instead of an exceptional case, this logging-rethrowing-terminating strategy is common for exception handling according to recent studies (Shah *et al.*, 2010; Cabral and Marques, 2007).

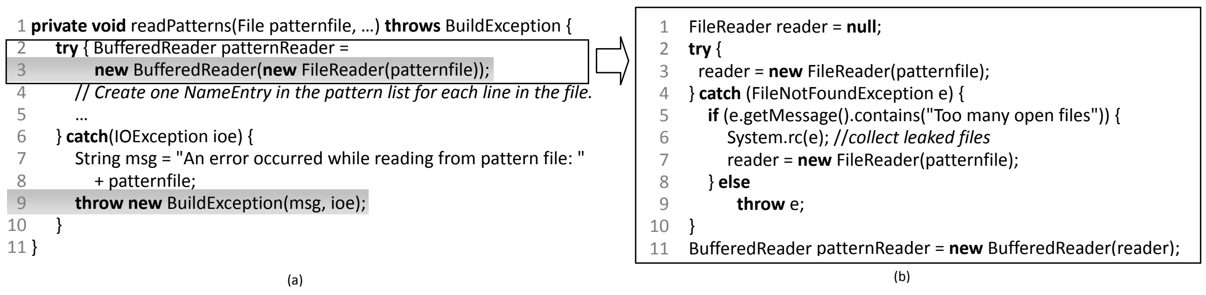
Designing effective recovery strategies for exceptions (i.e. recovering from exceptional states and continuing the execution of the program to complete its task) is difficult. When encountering a resource exhaustion exception, programmers typically do not know where to find available resources. Existing exception recovery approaches (Carzaniga *et al.*, 2013; Chang *et al.*, 2009) cannot avoid failures manifested as REEs. Even if they can fix the causal resource leak, there is no available resources to complete the task without collecting leaked resources. Considering the abundance of resource leaks and the poor quality of exceptional handling, REEs pose a great threat to the reliability of programs.

[1]In current version of Ant, the opened file is closed within a `finally` statement at the end of this method. However, the handling code for the `IOException` (superclass of `FileNotFoundException`) remains.

This paper presents an approach to automatically recover from REEs caused by resource leaks by collecting leaked resources and make the program execution able to proceed to complete its task. Our approach has two key components. The first component is the *program transformer* that analyzes the program, finds method calls where REEs can be thrown and transforms the program by adding recovery code for REEs. The recovery strategy consists of collecting leaked resources first and then retrying the exception-throwing method calls. We require that the REE-throwing method is failure atomic (Fetzer *et al.*, 2004) (i.e., the method leaves the program in a consistent state before exceptions are propagated to its caller). For example, the transformed result for the exception-throwing code in the lines 2 and 3 in Fig. 1a is presented in Fig. 1b[2]. The second component is the *resource collector* (called by *System.rc()* in the line 6 in Fig. 1b) that collects leaked resources. First, the resource collector identifies leaked resources as the unreleased and unreachable resources. For garbage collected languages, we adapt the garbage collector to retain leaked resources during garbage collections. Second, corresponding cleanup methods such as `close` of `BufferedReader` for the code in Fig. 1a are invoked to safely release these leaked resources in the right order. We ensure the safety of the resource collector by guaranteeing that when a resource is released there are no objects that depend on it and that have some actions (e.g., `close` and `finalize`) to perform in the future, and this resource does not refer to resources that may be manipulated later by the program.

We implement our approach based on Soot (Vallée-Rai *et al.*, 1999) and Jikes RVM (Arnold *et al.*, 2000) for Java programs. The input to our approach is REEs as well as their corresponding resource specifications. We conduct a series of experiments to evaluate the effectiveness and overhead of our approach on standard benchmarks in literature and reported resource leaks from real-world programs. The experimental results show that our approach can successfully recover from REEs caused by

[2]Our approach actually transforms the method `FileInputStream(File)` that is called by `FileReader(File)`. We only transform calls to source methods of REEs. See Section 2 for details. The transformation in Fig. 1b is for illustration. All classes in this paper are from the Java system library unless explicitly stated otherwise. We omit the package name for brevity without confusions.



```
1 private void readPatterns(File patternfile, …) throws BuildException {
2     try { BufferedReader patternReader =
3             new BufferedReader(new FileReader(patternfile));
4         // Create one NameEntry in the pattern list for each line in the file.
5         …
6     } catch(IOException ioe) {
7         String msg = "An error occurred while reading from pattern file: "
8             + patternfile;
9         throw new BuildException(msg, ioe);
10    }
11 }
```

(a)

```
1  FileReader reader = null;
2  try {
3    reader = new FileReader(patternfile);
4  } catch (FileNotFoundException e) {
5      if (e.getMessage().contains("Too many open files")) {
6          System.rc(e); //collect leaked files
7          reader = new FileReader(patternfile);
8      } else
9          throw e;
10 }
11 BufferedReader patternReader = new BufferedReader(reader);
```

(b)

Fig. 1 The left part (a) is an example code snippet from Ant, and the right part (b) is the transformed result for the lines 2 and 3 from the left part

these reported resource leaks and make the programs able to continue to complete their tasks. The runtime overhead for benchmark programs is very low, around 3%, and the average execution time increase is 2.56%. The increase of byte-code size caused by the program transformer is negligible.

The rest of this paper is organized as follows. Section 2 presents the proposed approach and the program transformer. In Section 3, we present the design of the resource collector. We discuss the soundness and completeness of our approach in Section 4. The implementation of our approach for Java programs is presented in Section 5. Section 6 presents the experimental evaluation. Related work is presented in Section 7. Finally, we conclude in Section 8.

2 Proposed Approach

We aim to recover from REEs and then retry the failed code to make the program able to continue its execution. Our approach is fully automatic by transforming programs. The transformation is source-to-source/byte-code-to-byte-code, without any user annotations required. The architecture of this approach is presented in Fig. 2. There are two working stages. The first stage is the pre-deployment transformation. In this stage, we transform the program to add the recovery code for method calls possibly throwing REEs. The second stage is the runtime recovery by collecting leaked resources. A *resource collector* is developed and deployed into the underlying virtual machine/execution system, on top of which the hardened program from the first stage runs. If there are resources that have been exhausted during runtime and the corresponding exception is thrown by the program, the REE will be caught by

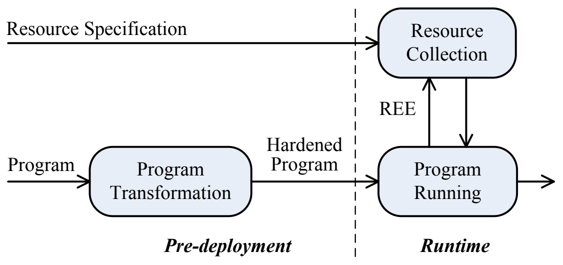


Fig. 2 Overview of our approach

the transformed program. Then, the resource collector begins to collect leaked resources and the failed method call is retried. If the recovery succeeds, the program continues to execute. Otherwise, the REE will be thrown again and propagated to the caller in the same way as the original program does.

Most garbage collectors adopt the finalization mechanism that allows a `finalize` method to be associated with an object. The garbage collector invokes the `finalize` method to perform some cleanup work before its associating object is garbage collected. Our resource collector and the finalization mechanism both aim at reclaiming leak resources. However, finalization is unqualified to perform resource collections for various reasons. In contrast, our resource collector improves the situation based on its several design decisions. Please refer to Section 7.2 for details.

The input to our approach includes the program and resource specifications. A resource specification $\langle e, M\rangle$ is a tuple, where e is the REE and M is the set of methods that should be called to release the exhausted resources. A method $m \in M$ is fully-qualified with all its parameters specified, including the type of the receiver *this* which we consider as a special parameter for object-oriented programs. We use *m.this* to denote the type of the receiver of m and use $\mathcal{S}$ to denote the set of all input resource specifications. The resource collector calls the methods in M to release leaked resources in response to the exception e. An example resource specification for Java programs is ⟨`FileNotFoundException` *saying "Too many open files"*, {`BufferedReader: void close()`, `FileReader: void close()`, ...}⟩. The resource management API method pairs for acquiring and releasing resources are sometimes called resource-releasing specifications (Wu *et al.*, 2011). The problem of how to gain resource-releasing specifications has be well-studied (Wu *et al.*, 2011; Weimer and Necula, 2005) and we consider it orthogonal to our work. Converting such resource-releasing specifications to the resource specifications for our approach is straightforward. For a REE e, we find all resources R whose exhaustion can cause programs to throw e. We use M to denote the set of resource-releasing methods of each $r \in R$ in resource-releasing specifications. Then we get the resource specification $\langle e, M \rangle$.

Besides common system-level resources, there are also other application-specific resources that have a limited amount available to programs for their own purposes. We expect that our approach can not only manage common system resources but also application-specific ones. It is an alternative to catch REEs directly by the runtime system, but here we choose to transform application code and/or libraries, which makes it easy for our approach to scale to new application-specific resources without modifications to the underlying runtime system. We analyze and transform the program by adding recovery code for REEs. It consists of two steps. The first step is to identify the REE-throwing method calls. The second step is to augment the program with recovery code for these REE-throwing method calls.

2.1 Finding REE-Throwing Method Calls

To identify REE-throwing method calls, there are three aspects to be considered. First, every REE raised during runtime must be handled. Second, a thrown REE should not be handled more than once. Consider the fact that an exception can propagate across multiple methods along the stack up and exceptions thrown by different method calls can be the same one. If the program is not recovered from a REE and this exception propagates to the calling method (our approach can guarantee this), recovery for calls to the calling method typically does not succeed. Such a second recovery should not be performed to avoid extra overhead. Third, the closer the recovery code is from the source of the exception, the more likely the recovery succeeds. We require that REE-throwing methods are *failure atomic* (Fetzer *et al.*, 2004). If the recovery code is far from the source of the exception, side effects produced by failed code become nontrivial because the program may have performed many actions and the state reversion to maintain the failure atomicity becomes costly. It is desirable to recover REEs at program points as close as possible to the source of REEs.

We first introduce the concept of *source methods* of a REE. A method m is the *source* of a REE if this REE can propagate to m's caller and this REE is created by m (i.e., it is not propagated from m's callees). For the example code in Fig. 1a, `readPatterns` is the source method of the `BuildException`. It is desirable to handle REEs at the points of calls to the source methods of these REEs.

We identify all source methods by analyzing every method of the program. We analyze each REE that is created by this method and decide whether they can escape from (not caught by) the method. If there is one such REE, this method is the source of the REE. We use an intraprocedural points-to analysis to determine the *may* aliases of an exception. Within the body of a method, an invocation of the constructor of a REE class returns a REE and we do not need to process invocations of other methods. Our analysis forwardly propagates information along the control flow edges. At control flow join points, we merge the incoming sets of REEs for each variable. For an assignment statement "$v_1 = v_2$", the set of REEs to which v_1 can point is updated to the set of REEs to which v_2 can point. This analysis to identify the source methods is sound. However, it can produce false positives. After identifying the source methods of REEs, we scan the program to find all the calls to these source methods. These calls are targets of our transformation.

2.2 Exception Handling Transformation

The transformation is performed on the bytecode. However, we discuss the approach here on the source code level for convenience. The calls to source methods of REEs are targets for which we augment the recovery code. There are two cases. The first case is that the call to the source method is a separate statement. We simply surround this call with the exception handling statement (`try` and `catch` in Java). The handling code (within the `catch` statement in Java) consists of the call to the resource collector with the REE thrown by the source method as the parameter, and then the call to the source method. The second case is that the called source method has a return and the call is involved within some expression (e.g., in Fig. 1a, `new FileReader(patternfile)` is involved in the expression `new BufferedReader(new FileReader(patternfile))`). The program transformer first adds a few lines of code just before the statement involving the source method call. The first line of code added is the introduction of a local variable of the return type of the source method with the `null` initial value (e.g. line 1 in Fig. 1b). The second line of code added is assigning the call to the source method (exactly the copy of its call in the original program) to the local variable (e.g. line 3 in Fig. 1b). This added line of code is augmented with the recovery code in the same way as that in the first case. Finally, the call to the source method within the expression is replaced with the local variable (e.g. line 11 in Fig. 1b). Each REE is handled separately if there are multiple REEs that may be raised by a source method call.

3 Resource Collector

We recover the program from REEs by collecting leaked resources. We assume that the execution environment has been reasonably configured to provide adequate resources for the normal execution of the program. During runtime when the REE occurs, the typical cause is that the activated resource leak bugs of the program lead to too many leaked resources. To collect leaked resources, we adapt the garbage collector if any to leave leaked resources alone during garbage collections. When the REE occurs, we first identify corresponding leaked resources and then release these resources by invoking releasing methods provided in the resource specifications.

We identity two requirements of the resource collector. *The first and most important is safety.* The aim of the resource collector is to recover the program from exceptions. Hence, it is obligated to cause no unexpected side effects and should not leave the program in inconsistent states such as states that can crash the program. *The second requirement is that it should release all leaked resources if possible.* More collected leaked resources means higher likelihood that the recovery succeeds. To coordinate these two conflicting requirements, we design the following strategy. A leaked resource r is released by the resource collector if and only if (1) there are no objects that depend on r and that have some actions (e.g. `close` or `finalize`) to perform in the future, and (2) resource-releasing methods of r do not have access to resources that may be manipulated later by the program.

3.1 Retaining Leaked Resources during Garbage Collections

Managed languages such as Java are often equipped with garbage collectors. For such languages, the garbage collector and our resource collector coexist. The garbage collector is triggered by large memory consumption and the resource collector is triggered by REEs caused by exhaustion of non-memory system resources. Consider cases when there are some leaked resources that have not yet been released by the resource collector. If the garbage collector begins to work then, the objects of these leaked resources will be destroyed and their occupied resources will be permanently leaked, that is, resource collections in the future can not release them. To avoid this, we adapt the garbage collector to retain leaked resources during garbage collections. The set $\mathcal{R} = \{m.this \mid m \in M \wedge \exists e.(\langle e, M\rangle \in \mathcal{S})\}$ denotes all types of interesting resources whose exhaustion will trigger the resource collector. Before destroying a garbage (typically unreachable) object whose type belongs to $\mathcal{R}$, the garbage collector first checks whether this resource has been released. If so, the garbage collector retains it.

We require that a resource has a field that indicates whether it has been released, such as the boolean field *closed* in `Socket`. If there is no such field, we can easily instrument the code of the resource to add one. The instrumentation is as follows. We first add a boolean field *closed* with the initial value as *true* to the code of the resource. Then, at each exit point of each releasing (acquiring) method of the resource, we insert this statement "*`closed = true;`*" ("*`closed = false;`*").

3.2 Identifying Leaked Resources

The problem of determining whether an object is live (will be used later) or not is undecidable in general. Garbage collectors and most resource leak detection approaches conservatively consider leaked resources as unreachable ones (Torlak and Chandra, 2010; Weimer and Necula, 2008; Martin *et al.*, 2005). We also employ the same idea and identify leaked resources as those unreachable ones among all unreleased resources. To identify leaked resources that can be safely released, we traverse the heap three times. The first pass is to determine whether objects in the heap are reachable from the root objects of the program by tracing the references between objects in the heap, just like the common tracing garbage collector. The output of the first pass consists of three sets:

- R_u as the set of unreleased and unreachable resources whose exhaustion causes the thrown REE;
- R_r as the set of reachable resources whose exhaustion causes the thrown REE;
- F_u as the set of unreachable objects with actions to perform in the future such as finalization-ready objects. These actions are required by *other* mechanisms such as the *finalization* of the garbage collector ($R_u \cap F_u = \emptyset$).

The second pass is to determine whether objects in R_u are reachable from objects in F_u. If F_u is empty, this pass is not necessary. The output of the second pass is the set:

- R_{rf} as the set of objects in R_u that are reachable from objects in F_u.

The algorithm to perform the third pass is presented in Algorithm 1 and Algorithm 2. The output of this algorithm includes S as the set of leaked resources that can be safely released, and $S_b \subseteq S$ that includes leaked resources that can be released immediately. The function *visited* records whether an object has been visited during the traverse. The function *reached* records whether an object can be reached from root objects in $R_u - R_{rf}$. The function $refer$ records whether an object or its reference objects (objects directly or indirectly referring to it) can directly or indirectly refer to an object in R_r. The algorithm performs the depth first

Algorithm 1 Algorithm to identify leaked resources

```
Require:
    $H, R_u, R_{rf}, R_r$
Ensure:
    $S, S_b$
 1: $S \leftarrow \emptyset$
 2: $S_b \leftarrow \emptyset$
 3: for $o \in H$ do
 4:     $visited(o) \leftarrow false$
 5:     $reached(o) \leftarrow false$
 6:     $refer(o) \leftarrow false$
 7: end for
 8: for $o \in R_u - R_{rf}$ do
 9:     if $visited(o) = false$ then
10:         DFT($o$)
11:     end if
12: end for
13: for $o \in R_u - R_{rf}$ do
14:     if $refer(o) = false$ then
15:         $S \leftarrow S \cup \{o\}$
16:         if $reached(o) = false$ then
17:             $S_b = S_b \cup \{o\}$
18:         end if
19:     end if
20: end for
```

search to traverse the heap by following references between objects. It first performs the initialization (lines from 1 to 7). Then, it traverses the heap from objects in $R_u - R_{rf}$ (lines from 8 to 12). Finally, it identifies objects from $R_u - R_{rf}$ that belongs to S and/or S_b (lines from 13 to 20). An object $o \in S$ is safe to be released because (1) it is not reachable from the program ($o \in R_u$), (2) it is not depended on by objects with actions to perform in the future ($o \notin R_{rf}$), and (3) it and its reference objects do not refer to any reachable resources ($refer(o) = false$). An object in S_b can be safely released immediately because it is not depended on (referred to) by objects in $R_u - R_{rf}$. The procedure DFT(o) (Algorithm 2) performs the depth first search from o. Algorithm 1 is a variant of the classic depth first search algorithm. Its complexity is the same as that of the classic depth first search algorithm. To illustrate Algorithm 1, Fig. 3 presents an example heap. We assume $R_u = \{r_1, r_2, r_3, r_4, r_5\}$, $o_f \in F_u$ and $r_0 \in R_r$. r_0 is reachable from a local variable. If taking this heap as input, the Algorithm 1 will produce the output $S = \{r_4, r_5\}$ and $S_b = \{r_4\}$. Please note that $r_1 \in R_{rf}$ and $refer(r_2) = refer(r_3) = true$.

Algorithm 2 DFT algorithm to perform the depth-first traversal of the heap

```
Require:
    $o$ : the current object
 1: $visited(o) \leftarrow true$
 2: for $o'$referred by $o$ do
 3:     if $o' \in R_r$ then
 4:         $refer(o) \leftarrow true$
 5:         continue
 6:     end if
 7:     if $refer(o) = ture$ then
 8:         $refer(o') \leftarrow true$
 9:     end if
10:     $reached(o') = true$
11:     if $visited(o') = false$ then
12:         DFS($o'$)
13:     end if
14:     if $refer(o') = true$ then
15:         $refer(o) \leftarrow true$
16:     end if
17: end for
```

3.3 Releasing Leaked Resources

We make the assumption that resource-releasing methods just perform the work related to release the occupied resources and do nothing else. For example, a resource-releasing method typically nullifies a field referring to a resource but we rarely observe cases that a resource-releasing method assigns a resource to fields of another accessible resources. This assumption is reasonable for existing resources and we believe that it should be obeyed when designing new resources considering that low coupling is one of the key principles of software engineering.

Under such an assumption, we can deduce that resource releasing methods *destruct* existing references among resources but not *construct* new references among them. Leaked resources are not guaranteed to be independent. The ordering of releasing of leaked resources is important. The general rule is that reference resources should be released before their referent resources. The algorithm to decide the ordering of resource releases and then release leaked resources in order is presented in Algorithm 3. The input includes the reference graph $\mathcal{H}$ of the heap whose edge $\langle o, o' \rangle$ represents that o directly refers to o', and two sets S and S_b that are outputs of Algorithm 1. The procedure RELEASE(o) calls releasing methods of o to release it. The function c records the number of references from objects in S

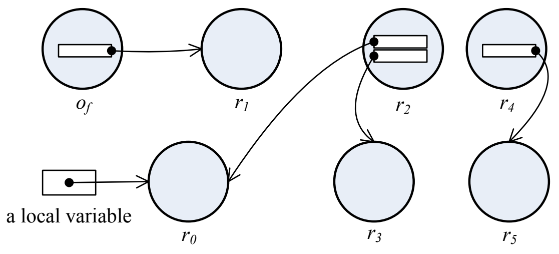


Fig. 3 Example heap reference graph. Circles represent objects. Arrows represent references, originating from reference objects (variables) and pointing to the referents

to an object in S. The main idea of this algorithm is to release a leaked resource in S when there is no reference to it from leaked resources in S. Leaked resources in the input S_b can be released immediately (lines from 9 to 16). After a leaked resource o is released (line 11), we decrement $c(o')$ by 1 for each leaked resource o' in S that o directly refers to (lines from 12 to 15). These objects referred to by a released resources are candidates for the next iteration of resource collections (lines from 18 to 22). For each released resource, the cost of the algorithm to update c and choose leaked resources for the next iteration of resource collection (lines from 12 to 15 and from 18 to 22) is no more than two times the number of references from the released resource to objects in S. If the reference graph of leaked resources in S is denoted as $\langle S, E' \rangle$, then the complexity of Algorithm 3 is $O(|S| \times O(\text{RELEASE}) + 2|E'|)$. The algorithm avoids traversing the heap multiple times and gets the low complexity linear to the scale of the reference graph of leaked resources.

The procedure RELEASE(o) calls releasing methods of o specified in the resource specifications. To dynamically call a method is not easy in general. The most difficult task is to decide values for parameters of the method. Fortunately, we observe that resource-releasing methods are simple in terms of the way of their invocations in practice. Formally, *we make the following assumption on resource-releasing methods: there is only one releasing method for each resource type and this releasing method has no parameters.* This assumption holds for all non-memory resources in the Java system library and we believe that programmers should obey it when designing new resources. For example, all resources in the `java.io` package and some other resources implement the interface `Closeable` that is introduced since Java 1.5 to release resources. This interface includes only one method `close` without parameters. `Socket`, `ServerSocket` and `Connection` also have a similar releasing method `close`. The interface `AutoClosable`[3] is newly introduced into Java 1.7 and also meets the assumption. Under this assumption, the procedure RELEASE(o) is as simple as calling the single releasing method on o. To avoid possible dead locks, we employ a separate thread to perform RELEASE(o). This thread is given the privilege to run immediately. All application threads are blocked until this thread terminates or it is blocked by some locks.

Algorithm 3 Algorithm to collect leaked resources

```
Require:
    $\mathcal{H} = \langle V, E \rangle, S, S_b$
 1: for $o \in S$ do
 2:     $c(o) \leftarrow 0$
 3: end for
 4: for $\langle o, o' \rangle \in E$ s.t. $o \in S \wedge o' \in S$ do
 5:     $c(o') = c(o') + 1$
 6: end for
 7: $O \leftarrow \emptyset$
 8: while $S_b \neq \emptyset$ do
 9:     for $o \in S_b$ do
10:         $S \leftarrow S - \{o\}$
11:         RELEASE($o$)
12:         for $\langle o, o' \rangle \in E \wedge o' \in S$ do
13:             $O \leftarrow O \cup \{o'\}$
14:             $c(o') = c(o') - 1$
15:         end for
16:     end for
17:     $S_b \leftarrow \emptyset$
18:     for $o \in O$ do
19:         if $c(o) = 0$ then
20:             $S_b \leftarrow S_b \cup \{o\}$
21:         end if
22:     end for
23: end while
```

4 Discussions

If the original program would not raise REEs, our transformation guarantees that the transformed program behaves exactly in the same way as the original program. If a REE would be thrown, our recovery code first collects leaked resources and then retries to execute the REE-throwing method call. If the REE is raised not because of resource exhaustion but for some intended reasons such as no-local control transfers, the transformed program retains this intended behavior, that is, the retrial of the execution of the REE-throwing method call should raise the REE as before, provided that the resource collector does not cause unexpected side effects. This kind of special use of REEs is rare. We did not observe such cases in our experiments. Other exception handling approaches Dobolyi and Weimer (2008) explicitly assume that programs do not employ exceptions for such special purposes.

It can be seen that the soundness of our approach depends on the safety of the resource collector. The resource collector is safe provided that the assumptions made above hold. In practice, these assumptions are reasonable. However, we admit that there may be some exceptional resources in poorly designed programs that contradict these assumptions. In such cases, we can manually refactor resource releasing methods such that they just release resources and do nothing else. Although we can not provide a general solution for all such cases, we believe that it is worthy to perform the recovery because otherwise the task will be inevitably aborted and the entire program may possibly halt or even crash. If a resource releasing method does not meet the assumption in Section 3.3 that is intended to simplify its invocation, the resource collector does not release corresponding leaked resources. In our current implementation, we do not consider reflection when finding REE source methods. This may lead to some raised REEs that can not be handled by our approach. In future work, we plan to dynamically capture calls of these methods.

In addition, we do not claim that our approach can collect all leaked resources. There are two reasons for the incompleteness. First, there may be some leaked resources that are still reachable. Our resource collector cannot release such leaked resources. This is a limitation to all existing leak detection approaches that approximate the liveness of resources by their reachability, such as Torlak and Chandra (2010); Weimer and Necula (2008); Martin *et al.* (2005). Second, the resource collection Algorithm 3 may omit to release some leaked resources. The release of a leaked resource o may destroy references indirectly reachable from o besides these references directly from o and thus the algorithm may

[3]http://jdk7.java.net/

omit to release some leaked resources. To release all leaked resources in such cases, we have to traverse the heap once more after each leaked resource has been released. Our algorithm traverses the heap only once and we prefer its low complexity because it works well in practice.

5 Implementation

We employ the Soot (Vallée-Rai *et al.*, 1999) program analysis framework to implement a prototype tool for Java programs to find REE source methods and transform the original program to add recovery code. This tool statically analyzes and transforms a standard intermediate representation of Java bytecode and no source code of the target program is needed. We implement the resource collector on Jikes RVM v3.1.1, a production-level, open-source Java-in-Java virtual machine (Arnold *et al.*, 2000). The resource collector is based on the MMTK memory management toolkit (Blackburn *et al.*, 2004) that Jikes RVM employs to perform its memory management. We mainly utilize the full-heap tracing functionality of MMTK to decide unreachable resources. We employ the Java's reflection utility to dynamically call resource-releasing methods to collect leaked resources. The design and implementation of the resource collector are independent of the garbage collector, so the resource collector can work with any garbage collectors of Jikes RVM. Currently, our implementation uses the Mark-Sweep garbage collector and straightforwardly adapts it to retain leaked resources during its collections. The interface of the resource collector is a method `rc` added to the class `System` with the caught REE as the parameter. The resource specifications are provided to the resource collector through a configuration file.

6 Experimental Results

In this section, we conduct several experiments to evaluate our approach. The main issues include (1) the effectiveness of our approach to recover real-world programs from REEs, and (2) the overhead of our approach in terms of running time and the size of class files. In the experiments, we use the default configuration of Jikes RVM. This configuration has the highest performance. Each running time given here is the geometric mean of results of ten trials. We conducted all experiments on a machine of the 3.0 GHz Intel Core i5-2320 CPU and 4 GB RAM, running Linux 2.6.38.6.

6.1 Examples of Recoveries from REEs

In this section, we present two examples that our approach successfully recovers real-world programs (Ant[4] and BIRT[5]) from REEs. We find that resource leaks are common in bug repositories and forums. However, there are few resource leaks that have attached reproducible test cases to cause corresponding REEs to be thrown. We analyze resource leaks and write by ourselves reproducible test cases, which is very time-consuming, or we use the attached test cases if they can reliably reproduce the leak and trigger corresponding REEs. We then transform the program and run it under the modified Jikes RVM with the resource collector. We guarantee that raised REEs are successfully recovered. In these two examples, we try to evaluate the overhead of the resource collector. The overhead is computed as the ratio of the time spent on identifying and collecting leaked resources to the time spent on the whole run.

The first example is from the Apache Ant that is a famous Java project build tool. There is a file descriptor leak numbered 4008 in Ant v1.4 in the bug database of Ant. The code snippet of this bug is presented in Fig. 1a. The `readPatterns` method opens a file, reads its content but does not close it at the end. Each call of this method will leak one file descriptor. Because there is no attached reproducible test case in the bug report, we analyze the leak and then write one by ourselves. We have ten copies of the 515 files in the *src* directory of the Ant v1.4 source distribution. The test case is an Ant task that copies all these 5150 files to another directory. To trigger the `FileNotFoundException`, we use one pattern file for each of the 5150 files. The median 256M memory is used to run programs here. The per-process limit of the file descriptor is 1024 that is the default value on our experimental machine. The details to transform programs to handle `FileNotFoundException` are presented in the next Section (Section 6.2).

We first run the original Ant under the unmodified Jikes RVM. Ant raises the `FileNotFoundException` saying "Too many open

[4]http://ant.apache.org/
[5]http://www.eclipse.org/birt/phoenix/.

files" and none of the 5150 files is copied. We then transform Ant and run it under the modified Jikes RVM with the resource collector. Ant successfully copies all the 5150 files this time and normally stops. During this run, the resource collector is triggered 5 times and it releases in total 5120 leaked file descriptors. The overhead of the resource collector is 5.37%.

The second example is from BIRT that is an open source Eclipse-based reporting system. There is a database connection leak numbered 237190 in BIRT v2.3.1 in the BIRT bug database. The connection leak occurs when there are more than one data sources in the report design. The method `dataEngineShutdown` of the class `DataSource$ShutdownListener` in the package `org.eclipse.birt.data.engine.executor` only closes connections of the current data source, which will lead to serious resource leaks. The experimental environment is set up by deploying BIRT v2.3.1 into Tomcat[6] v5.5.26. MySQL[7] v5.0.67 is used as the database. We use the default configuration of Tomcat. We configure the maximal concurrent connections of MySQL to be 100 that is also the default value. The reproducible test case used here is the one provided by the bug reporters. This test case is a report design that contains a single JDBC data source and a single scripted data source. The JDBC data source selects a single column from a simple table. The scripted data source simply prints "Hello world.". We use the Firefox[8] to display the report on the local machine. To reproduce the bug, we write a Firefox plugin to repeatedly open the same web page, that is, iteratively run the test report. The plugin also records the time spent on each page loading to evaluate the overhead of the resource collector.

It is shown that each iteration of the page display leaks one database connection. We first run the original programs under the unmodified Jikes RVM. The first 100 iterations all complete successfully. However, the 101th iteration halts abnormally with the exception message as "Cannot open the connection for the driver ... Too many connections." The "Hello World." is not displayed. Then we transform the programs. We confine the transformation to BIRT. The REE is the `JDBCException` in the package `org.eclipse.birt.report.data.oda.jdbc` *saying "Too many connections"*. A few of the source methods of this REE are failure non-atomic. We ensure their failure atomicity by simply adding several lines of code to revert values of several variables before the REE is thrown. We run the hardened programs under the modified Jikes RVM with the resource collector. This time we successfully run the report for more than half an hour until we terminate it. The report is repeated for about 4000 iterations. Each iteration completes its task and correctly prints "Hello World.". The time spent on page display for each one of the first 1001 iterations is presented in Fig. 4. The *Base* series represent times of runs of the original programs under the unmodified Jikes RVM. The resource collector is triggered 10 times. It releases in total 1000 leaked connections. It can be seen that our approach has little overhead in an iteration except when the limit of maximal connections is violated and the resource collector is triggered. The performance of our approach remains stable in the long term. For the total 1001 iterations, the resource collector has the overhead of 0.92%. For single iterations, the resource collector poses an average time increase of 32.45% on iterations triggering the resource collector over other iterations. The byte-code size increase in this experiment is negligible.

6.2 Performance and Overhead

Our approach modifies both the program and the Java Virtual Machine (JVM). To validate the usefulness of our approach, we must evaluate its impact on the execution time and the size of class files. We use programs from the DaCapo benchmark suite Blackburn *et al.* (2006) of both version 2006-10-MR2 and version 9.12-bach, and SPECjvm98 Corporation (1999). We run each benchmark program with available memory as fixed at two times the minimum with which it can execute. The default workload is used for the DaCapo benchmark suite. For programs from SPECjvm98, we run them with the large input size (-s100).

In these experiments, we consider the system resource *file descriptor* and the corresponding REE is the `FileNotFoundException` *saying "Too many open files"*. Resources whose exhaustion can throw this REE include file input/output streams, file read-

[6]http://tomcat.apache.org/
[7]http://www.mysql.com/
[8]http://www.getfirefox.net/.

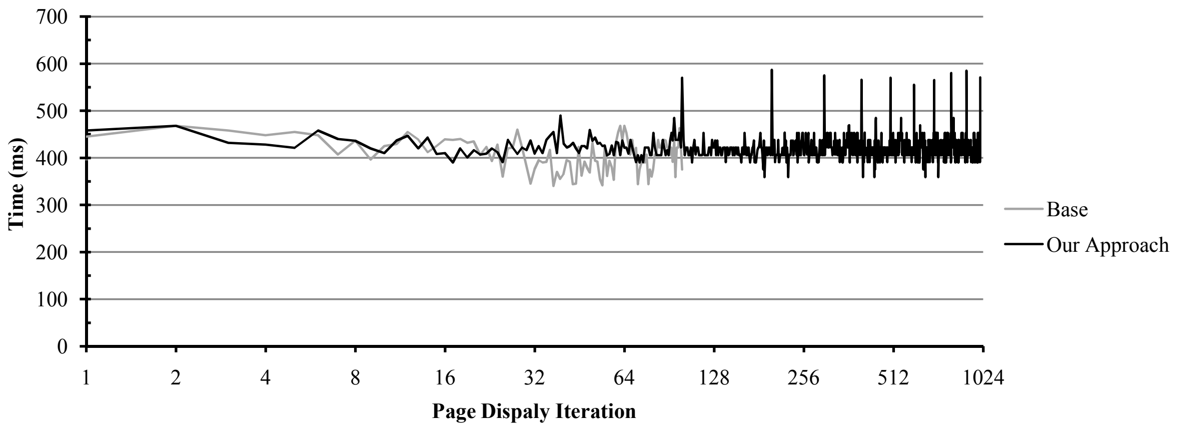


Fig. 4 Time for each page display of the first 1001 iterations of the BIRT example. The x-axis is logarithmic

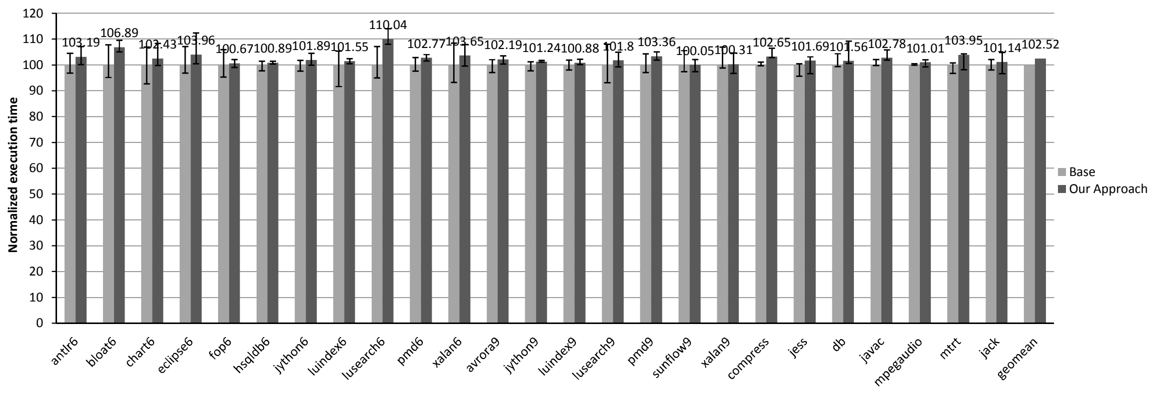


Fig. 5 Runtime overhead on the DaCapo and SPECjvm98 benchmarks. The time is normalized so that the time of running untransformed benchmarks on the unmodified Jikes RVM (*Base* series) is 100. The thin error bars represent the ranges of the ten trials

er/writer, and sockets in the Java system library. Specifications for this resources are simple and we mainly refer to the Java API documentation and the source code if necessary to create these specifications. The per-process limit of the file descriptor is 1024 that is the default value on our experimental machine. There are four source methods of this exception: `private native void open(String name)` of `FileInputStream`, `private native void open(String name)` and `private native void openAppend(String name)` of `FileOutputStream`, and `private native void open(String name, int mode)` of `RandomAccessFile`. They are all failure atomic. There are only 4 calls of these source methods and these 4 calls are all in Java system library. The effect of our transformer on the size of class files is negligible.

We found that the resource collector was never triggered in these experiments. However, many of these benchmarks leak some file descriptors during runtime. To evaluate our approach, We write a callback to intentionally run the resource collector once for each benchmark just before it exits. The runtime overhead of the resource collector is also evaluated in the above section (Section 6.1) against two known resource leak bugs. The runtime overhead of benchmark programs is presented in Fig. 5. To avoid name collisions between benchmark programs from DaCapo 2006-10-MR and DaCapo 9.12-bach, we append names of benchmark programs from DaCapo 2006-10-MR with 6 and append names of benchmark programs from DaCapo 9.12-bach with 9. Because there are many programs from DaCapo 9.12-bach that can not run under Jikes RVM v3.3.1, we only present results of those ones that can successfully execute in Fig. 5. It can be seen that the runtime overhead of our approach is very low, typically around 3%. The geometric mean of overhead for all programs is 2.52%. Two large runtime increases come from *bloat* with 6.89% and *lusearch* with 10.04% from DaCapo 2006-10-MR2. Time increases for all other programs are below 4%.

7 Related Work

Our approach targets REEs caused by leaks of non-memory system resources such as file descriptors and database connections. There is a great deal of work that addresses memory leaks (Bond and McKinley, 2008; Guyer *et al.*, 2006). General automatic approaches to localize bugs (Lei *et al.*, 2012) and fix bugs (Qi *et al.*, 2012) are also proposed. Our approach is closely related to the work that falls into two categories: recovery from exceptions and resource leaks.

7.1 Recovery from Exceptions

Carzaniga *et al.* (2013) propose to recover from runtime exceptions in Java programs by automatically applying workarounds. Chang *et al.* (2009) propose a self-healing approach to mask manifestation of faults derived from the integration of COTS components into applications. The healing connectors derived from already experienced integration faults are injected into applications to respond to exceptions. Those two approaches may fix the resource leak that causes the REE, but they cannot successfully recover from the REE because all available resources have been exhausted. Dobolyi and Weimer (2008) propose to transform Java programs to insert `null` checks and recovery actions guarding every dereference that is potentially `null`. Friedrich *et al.* (2010) propose to automatically handle exceptions in service-based processes in a self-healing manner and to repair errors through a model-based approach. Sinha *et al.* (2009) present an approach for locating and repairing faults in the form of incorrect assignments in Java programs. Such a fault manifests as a flow of an incorrect value that finally leads to an exception. Exceptions originating from these types of faults typically exclude REEs.

Based on their survey (Cabral and Marques, 2007), Cabral and Marques (2008) claim that there is something wrong with current exception handling models and propose the *automatic exception handling* model. Benign recovery actions are predefined for platform-level exceptions and shipped directly with the runtime system. When an exception occurs inside a *try* block, the system will execute one or more corresponding recovery actions, and then the code inside the *try* block is retried. This approach only applies to platform-level exceptions while ours transforms application programs and has no such limitations. It has more reflexibility and can handle both platform-level and application-level REEs.

Fetzer *et al.* (2004) introduce the concept of *failure atomicity*. A method is failure atomic if its failed executions due to occurred exceptions leave the program in a consistent state. This state consistency can be guaranteed through reverting all modifications performed by the method before the exception propagates to its calling method. Failure atomicity is necessary for all retry based recoveries to succeed. Fetzer *et al.* (2004) implement the failure atomicity by using checkpointing. Cabral and Marques (2008) implement the failure atomicity through Software Transactional Memory (STM) Herlihy *et al.* (2006). These techniques can be used to implement failure atomicity for our approach.

7.2 Language Features to Facilitate Resource Management

Most garbage collectors allow a `finalize` method to be associated with an object. The `finalize` method is intended to perform some cleanup work before its associating object is garbage collected. Our approach is analogous to the finalization mechanism since both aim at reclaiming unreachable resources. However, the execution of `finalize` methods may be arbitrarily delayed in an indeterminate way (Boehm, 2003), which makes it a known fact that finalization is unqualified to perform resource collections. There are two main reasons: (1) `finalize` methods are bound to garbage collector that may not run until the application is about to run out of memory. However, the application may already exhaust some non-memory resources or may suffer performance degradation due to huge resource consumption while there is still a large amount of available memory, and (2) various finalization implementations do not always execute `finalize` methods immediately when they are ready to be called (Boehm, 2003). Asynchronous finalization is a necessary feature for the correct implementation, but the situation becomes worse because of delayed invocations of ready `finalize` methods. Besides this delayed execution, another main drawback of Java's finalization is that the ordering of invocations of different finalize methods cannot be guaranteed. As dependencies between resources are common, Java's finalization is not safe.

Our approach improves the situation based on three design decisions. First, our approach separates non-memory resource collections from memory collections. Resource collections are triggered in response to REEs independently of memory usage. Second, the separate thread to release leaked resources is given the privilege to run immediately. Third, while we release as many leaked resources as possible we guarantee the ordering of leaked resource releases and the safety of the resource collector by the design strategy that a leaked resource r is released by the resource collector if and only if (1) there are no objects that depend on r and that have some actions (e.g., `close` and `finalize`) to perform in the future, and (2) resource releasing methods of r do not have access to resources that may be manipulated later by the program.

Many languages provide the mechanism of automatic releases of scoped resources. When a resource is out of its lexical scope, its releasing method is automatically invoked. Examples include destructors of C ++ and the `using` statement of C# (Hejlsberg *et al.*, 2003). Java 7 introduces the `try-with-resource` statement called Automatic Resource Management (ARM). Resources declared in this statement will be automatically closed once the program runs out of the `try` block. The declared resource should implement the `java.lang.AutoCloseable` interface. When resources are used in the local scope, these mechanisms can automatically release resources in time. However, there are situations in which resources are not confined to a convenient lexical scope. Our approach can collect leaked resources whether they are used "locally" or "globally".

To cope with resource leaks, Weimer and Necula (2008) propose a language extension called compensation stack that allows programmers to annotate resource-acquiring methods with compensations such as resource-releasing method invocations. These compensations are put into stacks that guarantee included compensations to execute in the last-in-first-out order. If compensations are within a heap-allocated stack, they will be executed automatically when the stack is finalized. In such cases, this approach can not guarantee the timely releases of leaked resources. The Furm (Park and Rice, 2006) groups resources into a resource tree in which a single `release` call can close all these resources in a deterministic order. Resource trees can be closed automatically when the thread that uses it dies. Similarly to compensation stacks (Weimer and Necula, 2008), Furm can not guarantee the timely releases of leaked resources to avoid REEs. The type system of the Vault programming language (DeLine and Fähndrich, 2001) allows function post-conditions to be specified to guarantee that annotated functions cannot allocate and leak resources.

7.3 Dynamic Resource Leak Detection and Collection

Our previous work presents the Resco tool (Dai *et al.*, 2013) to collect leaked non-memory resources. Resco counts the consumption of resources and ensures that the limits of resources are not violated. When the limit of resources is about to be reached (i.e., when 90% of available resources are consumed), Resco identifies unreachable resources and then releases them. The improvement over Resco of the work presented in this paper mainly lies in two aspects.

First, our approach aims to recover from REEs. Even if all available resources are consumed, it is not necessarily obligatory to collect leaked resources considering cases in which the program does not acquire such resources any more. Our approach collects leaked resources in response to REEs to avoid failures caused by resource leaks and meanwhile it does not perform unnecessary collections to avoid unnecessary overhead. In addition, Resco's requirement of counting resource consumptions compromises its applicability. To perform such resource consumption counting, the specific quantities of resources acquired by resource-acquiring methods and released by resource-releasing methods must be specified in the resource collection configuration. Limits of resources must also be specified in resource monitors before the program deployment. However, in dynamically reconfigurable systems (Walsh *et al.*, 2007), resource limits may not be fixed but change as the program runs.

Second, our approach can release more leaked sources that are omitted by Resco. For Resco, objects of leaked resources may be destroyed by the garbage collector and their occupied resources will be permanently leaked. In contrast, our approach retains leaked resources during garbage collections and can release them later by the resource collector if necessary. In addition, Resco only releases leaked resources that can be safely released immediately (S_b) while our approach tries to release all leaked resources in S (details are in Section 3.3). As would be expected, our approach imposes more runtime overhead than Resco. However, this overhead is low enough to be acceptable.

The QVM (Arnold *et al.*, 2011) is based on a Java Virtual Machine that detects and helps diagnose defects as violations of specified correctness properties. The PQL (Martin *et al.*, 2005) approach is shown to be effective to find mismatched method pairs which typically include resource leaks. As mismatched method pairs are liveness queries that depend on the absence of the second method call, pattern matches are found at the end of an execution. Performing resource releasing then is too late and makes no sense. Other approaches based on aspects (e.g., Allan *et al.* (2005) and Chen and Roşu (2007)) cannot precisely capture object death due to the lack of direct support from garbage collectors. So, they are not suitable to detect resource leaks. There are several techniques that explore the staleness of objects to aggressively collect leaked memory (Bond and McKinley, 2008). However, as cleanup of non-memory resources is not reversible, the object staleness cannot be easily applied to non-memory resource collections.

8 Conclusion

This paper presents an approach to automatically recover programs from REEs caused by leaks of non-memory system resources. We transform the program to add recovery code only for calls to source methods of REEs. This avoids handling many methods that are possibly failure nonatomic and significantly reduces the amount of needed transformation. In response to REEs, the recovery code first triggers the resource collector to safely collect leaked resources and then retries the failed method call. We design a linear algorithm to try to collect all leaked resources. Meanwhile, it avoids traversing the heap multiple times. Our approach improves the resilience of the program to resource leaks and its reliability by enabling it to continue to complete its task after REEs occur.

References

Allan, Chris, Avgustinov, Pavel, Christensen, Aske Simon, Hendren, Laurie, Kuzins, Sascha, Lhoták, Ondřej, de Moor, Oege, Sereni, Damien, Sittampalam, Ganesh, Tibble, Julian, 2005. adding trace matching with free variables to aspectj. *SIGPLAN Not.*, **40**(10):345–364. Available from http://doi.acm.org/10.1145/1103845.1094839. [doi:10.1145/1103845.1094839]

Arnold, Matthew, Fink, Stephen, Grove, David, Hind, Michael, Sweeney, Peter F., 2000. Adaptive optimization in the jalapeno jvm. Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, New York, NY, USA, p.47–65, Available from http://doi.acm.org/10.1145/353171.353175. [doi:10.1145/353171.353175]

Arnold, Matthew, Vechev, Martin, Yahav, Eran, 2011. qvm: an efficient runtime for detecting defects in deployed systems. *ACM Trans. Softw. Eng. Methodol.*, **21**(1):2:1–2:35. Available from http://doi.acm.org/10.1145/2063239.2063241. [doi:10.1145/2063239.2063241]

Blackburn, Stephen M., Cheng, Perry, McKinley, Kathryn S., 2004. Oil and water? high performance garbage collection in java with mmtk. Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, p.137–146, Available from http://dl.acm.org/citation.cfm?id=998675.999420

Blackburn, Stephen M., Garner, Robin, Hoffmann, Chris, Khang, Asjad M., McKinley, Kathryn S., Bentzur, Rotem, Diwan, Amer, Feinberg, Daniel, Frampton, Daniel, Guyer, Samuel Z., Hirzel, Martin, Hosking, Antony, Jump, Maria, Lee, Han, Moss, J. Eliot B., Phansalkar, Aashish, Stefanović, Darko, VanDrunen, Thomas, von Dincklage, Daniel, Wiedermann, Ben, 2006. The dacapo benchmarks: Java benchmarking development and analysis. Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, New York, NY, USA, p.169–190, Available from http://doi.acm.org/10.1145/1167473.1167488. [doi:10.1145/1167473.1167488]

Boehm, Hans-J., 2003. Destructors, finalizers, and synchronization. Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, p.262–272, Available from http://doi.acm.org/10.1145/604131.604153. [doi:10.1145/604131.604153]

Bond, Michael D., McKinley, Kathryn S., 2008. Tolerating memory leaks. Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, New York, NY, USA, p.109–126, Available from http://doi.acm.org/10.1145/1449764.1449774. [doi:10.1145/1449764.1449774]

Cabral, B., Marques, P., 2008. A case for automatic exception handling. Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, p.403–406, Available from http://dx.doi.org/10.1109/ASE.2008.59. [doi:10.1109/ASE.2008.59]

Cabral, Bruno, Marques, Paulo, 2007. Exception handling: A field study in java and .net. ECOOP 2007 European Conference on Object-Oriented Programming, **4609**:151-175. [doi:10.1007/978-3-540-73589-2]

Carzaniga, Antonio, Gorla, Alessandra, Mattavelli, Andrea, Perino, Nicolò, Pezzè, Mauro, 2013. Automatic recovery from runtime failures. Proceedings of the 2013 International Conference on Software Engineering, Piscataway, NJ, USA, p.782–791, Available from http://dl.acm.org/citation.cfm?id=2486788.2486891

Chang, Herve, Mariani, Leonardo, Pezze, Mauro, 2009. In-field healing of integration problems with cots components. Proceedings of the 31st International Conference on Software Engineering, Washington, DC, USA, p.166–176, Available from http://dx.doi.org/10.1109/ICSE.2009.5070518. [doi:10.1109/ICSE.2009.5070518]

Chen, Feng, Roşu, Grigore, 2007. Mop: An efficient and generic runtime verification framework. Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, New York, NY, USA, p.569–588, Available from http://doi.acm.org/10.1145/1297027.1297069. [doi:10.1145/1297027.1297069]

Corporation, Standard Performance Evaluation, 1999. specjvm98 documentation. *release 1.03 edition*, .

Dai, Z., Mao, X., Lei, L., Wan, X., Ben, K., 2013. resco: automatic collection of leaked resources. *IEICE Transactions on Information and Systems*, **96**(1):28-39.

DeLine, Robert, Fähndrich, Manuel, 2001. Enforcing high-level protocols in low-level software. Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, New York, NY, USA, p.59–69, Available from http://doi.acm.org/10.1145/378795.378811. [doi:10.1145/378795.378811]

Dobolyi, K., Weimer, W., 2008. Changing java's semantics for handling null pointer exceptions. Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on, p.47-56. [doi:10.1109/ISSRE.2008.59]

Dybvig, R. Kent, Bruggeman, Carl, Eby, David, 1993. Guardians in a generation-based garbage collector. Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, New York, NY, USA, p.207–216, Available from http://doi.acm.org/10.1145/155090.155110. [doi:10.1145/155090.155110]

Fetzer, C., Felber, P., Hogstedt, K., 2004. automatic detection and masking of nonatomic exception handling. *Software Engineering, IEEE Transactions on*, **30**(8):547-560. [doi:10.1109/TSE.2004.35]

Friedrich, G., Fugini, M., Mussi, E., Pernici, B., Tagni, G., 2010. exception handling for repair in service-based processes. *Software Engineering, IEEE Transactions on*, **36**(2):198-215. [doi:10.1109/TSE.2010.8]

Guyer, Samuel Z., McKinley, Kathryn S., Frampton, Daniel, 2006. Free-me: A static analysis for automatic individual object reclamation. Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, p.364–375, Available from http://doi.acm.org/10.1145/1133981.1134024. [doi:10.1145/1133981.1134024]

Hejlsberg, A., Golde, P., Wiltamuth, S., 2003. c# language specification. *Addison Wesley*, .

Herlihy, Maurice, Luchangco, Victor, Moir, Mark, 2006. a flexible framework for implementing software transactional memory. *SIGPLAN Not.*, **41**(10):253–262. Available from http://doi.acm.org/10.1145/1167515.1167495. [doi:10.1145/1167515.1167495]

Lei, Yan, Mao, Xiaoguang, Dai, Ziying, Wei, Dengping, 2012. effective fault localization approach using feedback. *IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS*, **95**(9):2247–2257. [doi:10.1587/transinf.E95.D.2247]

Martin, Michael, Livshits, Benjamin, Lam, Monica S., 2005. Finding application errors and security flaws using pql: A program query language. Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, New York, NY, USA, p.365–383, Available from http://doi.acm.org/10.1145/1094811.1094840. [doi:10.1145/1094811.1094840]

Park, Derek A., Rice, Stephen V., 2006. A framework for unified resource management in java. Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java, New York, NY, USA, p.113–122, Available from http://doi.acm.org/10.1145/1168054.1168070. [doi:10.1145/1168054.1168070]

Qi, YuHua, Mao, XiaoGuang, Wen, YanJun, Dai, ZiYing, Gu, Bin, 2012. more efficient automatic repair of large-scale programs using weak recompilation. *SCIENCE CHINA-INFORMATION SCIENCES*, **55**(12):2785–2799. [doi:10.1007/s11432-012-4741-1]

Shah, H.B., Gorg, C., Harrold, M.J., 2010. understanding exception handling: viewpoints of novices and experts. *Software Engineering, IEEE Transactions on*, **36**(2):150-161. [doi:10.1109/TSE.2010.7]

Sinha, Saurabh, Shah, Hina, Görg, Carsten, Jiang, Shujuan, Kim, Mijung, Harrold, Mary Jean, 2009. Fault localization and repair for java runtime exceptions. Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, New York, NY, USA, p.153–164, Available from http://doi.acm.org/10.1145/1572272.1572291. [doi:10.1145/1572272.1572291]

Torlak, Emina, Chandra, Satish, 2010. Effective interprocedural resource leak detection. Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, New York, NY, USA, p.535–544, Available from http://doi.acm.org/10.1145/1806799.1806876. [doi:10.1145/1806799.1806876]

Vallée-Rai, Raja, Co, Phong, Gagnon, Etienne, Hendren, Laurie, Lam, Patrick, Sundaresan, Vijay, 1999. Soot -

a java bytecode optimization framework. Proceedings of the 1999 Conference of the Centre for Advanced S-tudies on Collaborative Research, p.13–, Available from http://dl.acm.org/citation.cfm?id=781995.782008

Walsh, J. DąŕArcy, Bordeleau, F., Selic, B., 2007. domain analysis of dynamic system reconfiguration. *Softw Syst Model*, **6**:355–380.

Weimer, Westley, Necula, George C., 2008. exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, **30**(2):8:1–8:51. Available from http://doi.acm.org/10.1145/1330017.1330019. [doi:10.1145/1330017.1330019]

Weimer, Westley, Necula, GeorgeC., 2005. Mining temporal specifications for error detection. Tools and Algorithms for the Construction and Analysis of Systems, **3440**:461-476. [doi:10.1007/978-3-540-31980-1]

Wu, Qian, Liang, Guangtai, Wang, Qianxiang, Xie, Tao, Mei, Hong, 2011. Iterative mining of resource-releasing specifications. Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, p.233–242, Available from http://dx.doi.org/10.1109/ASE.2011.6100058. [doi:10.1109/ASE.2011.6100058]