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 Dec. 7, 2013; Revision accepted Mar. 7, 2014; Crosschecked July 16, 2014

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 approach for resource exhaustion exceptions, which often causes 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 to catch resource exhaustion exceptions, collect leaked resources, and then retry the failure 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 complete their tasks with an average execution time increase of 2.52% and negligible bytecode size increase.

Key words: Failure avoidance, Resource leaks, Resource collection, Exception handling, Reliability
doi:10.1631/jzus.C1300352 **Document code:** A **CLC number:** TP311

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 that programmers must manage manually, e.g., file descriptors and database connections. For programs written in Java-like languages, once acquired, a resource must be released by explicitly calling a cleanup method. Resource leak is a software bug that occurs when the resource cleanup method is not invoked after its last use. Resource leaks are common in Java programs (Torlak and Chandra, 2010). Growing resource leaks can degrade application performance and even result in system crashes due to resource exhaustion.

A 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 an exceptional situation is encountered, an exception is thrown (e.g., throw in Java). This exception causes the control flow to transfer from the point 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

‡ Corresponding author

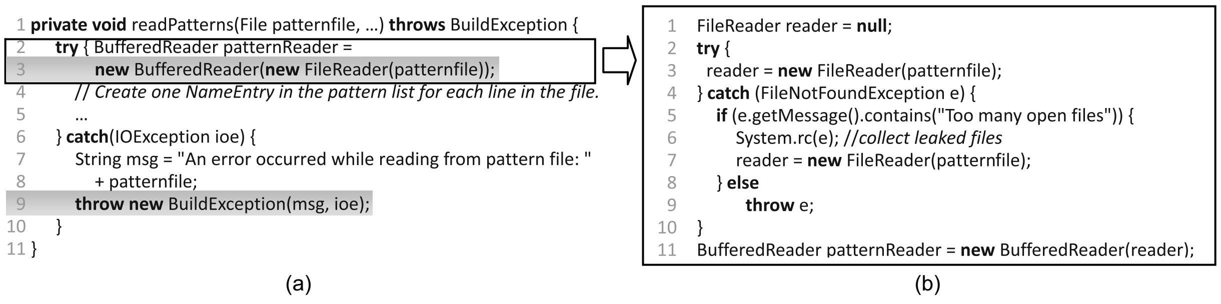
* Project supported by the National Natural Science Foundation of China (Nos. 61379054 and 91318301) and the National High-Tech R&D Program (863) of China (No. 2012AA011201)

is implicitly propagated to the caller of this method. If all available resources are consumed (or leaked), a further 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 programmer is often unsatisfactory. The Java code in Fig. 1a is such an example. This code snippet comes from an old version of Ant (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 IOException, superclass of FileNotFoundException, remains) 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) may 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, rethrows 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 (Cabral and Marques, 2007; Shah *et al.*, 2010).

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 (Chang *et al.*, 2009; Carzaniga *et al.*, 2013) cannot avoid failures being manifested as REEs. Even if they can fix the causal resource leak, there are 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.

This paper presents an approach for automatically recovering from REEs caused by resource leaks by collecting leaked resources and enabling the program execution to proceed to complete its task. Our approach has two key components: (1) 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 the REE-throwing method be failure atomic (Fetzer *et al.*, 2004); i.e., the program is left in a consistent state before exceptions are propagated to its caller. For example, the transformation result for the exception-throwing code in lines 2 and 3 in Fig. 1a is presented in Fig. 1b. Our approach actually transforms the method FileInputStream(File) that is called by FileReader(File). We transform calls only to the source methods of REEs. See Section 2 for details. The transformation in Fig. 1b is for illustration. Unless explicitly stated, classes used in this paper come from the Java system library. With no confusion we omit the package name for brevity. (2) The resource collector (called by System.rc() in line 6 in Fig. 1b) that collects leaked resources. First,



```
 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 (a) An example code snippet from Ant; (b) Transformation result for lines 2 and 3 from (a)

the resource collector identifies leaked resources as 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 right order. We ensure the safety of the resource collector by guaranteeing that when a resource is released no objects depend on it or 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 includes REEs and their corresponding resource specifications. We conduct a series of experiments to evaluate the effectiveness and overhead of our approach on standard benchmarks in the literature and reported resource leaks from real-world programs. Experimental results show that our approach can successfully recover from REEs caused by 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 bytecode size caused by the program transformer is negligible.

2 Proposed approach

We aim to recover from REEs and then retry the failure code to make the program able to continue its execution. Our approach is fully automatic by transforming programs. The transformation is source-to-source/bytecode-to-bytecode, requiring no user annotations. The architecture of this approach is presented in Fig. 2. There are two working stages. The first stage is 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 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 some types of resources have been exhausted during runtime and the corresponding exception is thrown by the program, the REE will be caught by the transformation program. Then, the resource collector begins to collect leaked resources and the method call that failed is retried. If the recovery succeeds, the program continues to execute. Otherwise, the REE will be thrown again and propagated to the caller as in the original program.

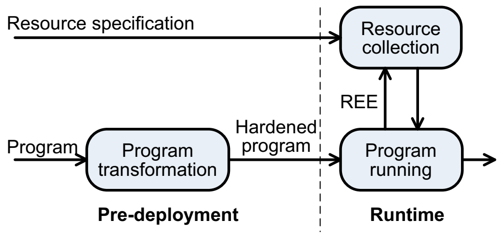


Fig. 2 Overview of our approach (REE: resource exhaustion exception)

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 do some cleanup work before its associated 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 by its several design decisions. Refer to Section 7.2 for details.

The input to our approach includes program and resource specifications. 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 being specified, including the type of 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(), $\cdots$}⟩. The resource management API method pairs for acquiring and releasing resources are sometimes called resource-releasing specifications (Wu *et al.*, 2011). How to gain resource-releasing specifications has been well studied (Weimer and Necula, 2005; Wu *et al.*, 2011) and we consider it orthogonal to our work. Converting such resource-releasing specifications to the resource specifications for our approach is straightforward. For an 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 obtain the resource specification $\langle e,M \rangle$.

Besides common system-level resources, there are other application-specific resources that have a limited amount available to programs for their own purposes. We expect that our approach can manage not only common system resources but also application-specific ones. Catching REEs directly by the runtime system is an alternative, but here we choose to transform application code and/or libraries, which enables our approach to easily 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 an REE and this exception propagates to the calling method (our approach can guarantee this), recovery of 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 exception, the more likely the recovery succeeds. We require that REE-throwing methods should be failure atomic (Fetzer *et al.*, 2004). If the recovery code is far from the source of the exception, side effects produced by failure code become nontrivial because the program may have performed many actions and state reversion to keep failure atomicity be 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 REE. A method m is the source of an 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 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 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 an REE class returns an 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 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 at 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: (1) 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. (2) 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 a '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 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 multiple REEs are 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 a normal execution of the program. During runtime when an REE occurs, a 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 an REE occurs, we first identify corresponding leaked resources and then release these resources by invoking releasing methods provided in the resource specifications.

We identify 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 the program should not be left in inconsistent states such as those that may crash the program. The second requirement is that the resource collector should be able to release all leaked resources. 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) no objects depend on r or have some actions (e.g., close or finalize) to perform in the future, and (2) resource-releasing methods for 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 by REEs caused by exhaustion of non-memory system resources. Consider cases in which some leaked resources have not yet been released by the resource collector. If the garbage collector begins to work, the objects of these leaked resources will be destroyed and their occupied resources permanently leaked; that is, resource collections in the future cannot release them. To avoid this, we adapt the garbage collector to retain leaked resources during garbage collections. The set $\mathcal{R} = \{m.\text{this} \mid m \in M \wedge \exists e.(\langle e, M\rangle \in \mathcal{S})\}$ 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 should have 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 for the resource, we insert this statement 'closed = true;' ('closed = false;').

3.2 Identifying leaked resources

The issue of determining whether an object is live or not is undecidable in general. Garbage collectors and most resource leak detection approaches conservatively consider leaked resources as unreachable ones (Martin *et al.*, 2005; Weimer and Necula, 2008; Torlak and Chandra, 2010). We 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 a common tracing garbage collector. The output of the first pass consists of three sets:

1. R_{u} as the set of unreleased and unreachable resources whose exhaustion causes the thrown REE;

2. R_{r} as the set of reachable resources whose exhaustion causes the thrown REE;

3. 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 ($R_{\mathrm{u}} \cap F_{\mathrm{u}} = \varnothing$).

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 R_{rf}, as the set of objects in R_{u} that are reachable from objects in F_{u}.

The third pass is performed as in Algorithms 1 and 2. The output of Algorithm 1 includes S as the set of leaked resources that can be safely released, and $S_{\mathrm{b}} \subseteq S$ which contains 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 $R_{\mathrm{u}} - R_{\mathrm{rf}}$. 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 depth-first search to traverse the heap by following references between objects. It first performs initialization (lines 1 to 7 in Algorithm 1). Then, it traverses the heap from objects in $R_{\mathrm{u}} - R_{\mathrm{rf}}$ (lines 8 to 12). Finally, it identifies objects from $R_{\mathrm{u}} - R_{\mathrm{rf}}$ that belong to S and/or S_{b} (lines 13 to 20). An object $o \in S$ is safe to be released because (1) it is not reachable from the program ($o \in R_{\mathrm{u}}$), (2) it is independent of objects with actions to perform in the future ($o \notin R_{\mathrm{rf}}$), and (3) o 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 independent of objects in $R_{\mathrm{u}} - R_{\mathrm{rf}}$. The procedure DFT($o$) (Algorithm 2) performs depth-first search from o. Algorithm 2 is a variant of the classical depth-first search algorithm. Its complexity is the same as that of the classical depth-first search algorithm. To illustrate Algorithm 1, Fig. 3 presents an example heap. Assume $R_{\mathrm{u}} = \{r_1,r_2,r_3,r_4,r_5\}$, $o_{\mathrm{f}} \in F_{\mathrm{u}}$ and $r_0 \in R_{\mathrm{r}}$. r_0 is reachable from a local variable. If taking this heap as input, Algorithm 1 will produce $S = \{r_4, r_5\}$ and $S_{\mathrm{b}} = \{r_4\}$. Note that $r_1 \in R_{\mathrm{rf}}$ and refer(r_2) = refer(r_3) = true.

Algorithm 1 Identifying leaked resources

```
Require: $H, R_u, R_{rf}, R_r$
Ensure: $S, S_b$
 1: $S \leftarrow \varnothing$
 2: $S_b \leftarrow \varnothing$
 3: for $o \in H$ do
 4:     visited($o$) ← false
 5:     reached($o$) ← false
 6:     refer($o$) ← 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
```

Algorithm 2 Depth-first traversal of the heap

```
Require: $o$, the current object
 1: visited($o$) ← true
 2: for $o'$ referred to by $o$ do
 3:     if $o' \in R_r$ then
 4:         refer($o$) ← true
 5:         continue
 6:     end if
 7:     if refer($o$) = true then
 8:         refer($o'$) ← 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($o$) ← true
16:     end if
17: end for
```

3.3 Releasing leaked resources

Assume that resource-releasing methods just perform the work related to releasing the occupied

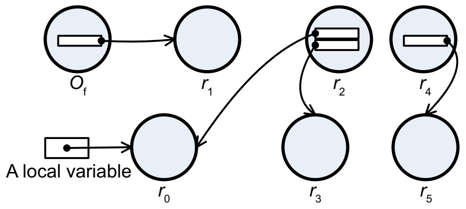


Fig. 3 Example heap reference graph. Circles represent objects, and arrows represent references which originate from reference objects (variables) and point to the referents

resources and do nothing else. For example, a resource-releasing method typically nullifies a field referring to a resource, and there are rarely cases in which a resource-releasing method assigns a resource to fields of other 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 do 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 for o to release it. The function c records the number of references from objects in S 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 9 to 16). After a leaked resource o is released (line 11), decrease $c(o')$ by 1 for each leaked resource o' in S that o directly refers to (lines 12 to 15). These objects referred to by released resources are candidates for the next iteration of resource collections (lines 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 12 to 15 and 18 to 22) is no more than twice the number of references from the released resource to objects in

Algorithm 3 Collecting leaked resources

```
Require: $\mathcal{H} = \langle V, E\rangle, S, S_\text{b}$
 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: $O \leftarrow \varnothing$
 8: while $S_\text{b} \neq \varnothing$ do
 9:     for $o \in S_\text{b}$ 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:     $S_\text{b} \leftarrow \varnothing$
18:     for $o \in O$ do
19:         if $c(o) = 0$ then
20:             $S_\text{b} \leftarrow S_\text{b} \cup \{o\}$
21:         end if
22:     end for
23: end while
```

S. If the reference graph of leaked resources in S is denoted as $\langle S,E'\rangle$, then the complexity of Algorithm 3 is $O(|S| \cdot O(\text{Release}) + 2|E'|)$. The algorithm avoids traversing the heap multiple times and obtains low complexity linear with the scale of the reference graph of leaked resources.

The procedure Release(o) calls releasing methods for o specified in the resource specifications. Dynamically calling a method is not easy in general. The most difficult task is to decide the parameter values used in the method. Fortunately, we observe that in practice resource-releasing methods are simple in terms of the way of their invocations. 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 introduced since Java 1.5 to release resources. This interface includes only one method 'close' without parameters. Socket, ServerSocket, and Connection have a similar releasing method 'close'. The interface AutoClosable (http://jdk7.java.net/), newly introduced into Java 1.7, 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.

4 Discussions

If the original program would not raise REEs, our transformation guarantees that the transformation program behaves exactly in the same way as the original program. If an 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 non-local control transfers, the transformation 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 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 cannot provide a general solution for all such cases, we believe that it is worthy to perform 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 REEs being raised that cannot 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 Martin *et al.* (2005), Weimer and Necula (2008), and Torlak and Chandra (2010). Second, in resource collection (Algorithm 3) some leaked resources may not be released. The release of a leaked resource o may destroy references indirectly reachable from o besides these references directly reachable from o, and thus the algorithm may omit 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 works well in practice with low complexity.

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 memory management. We mainly use the full-heap tracing functionality of MMTK to decide unreachable resources. We employ 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, we use the Mark-Sweep garbage collector and straightforwardly adapt 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

We have conducted several experiments to evaluate our approach. The main issues include (1) the effectiveness of our approach in recovering 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 10 trials. We conduct all experiments on a PC with 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

Our approach successfully recovers two real-world programs, Ant (http://ant.apache.org/) and BIRT (http://www.eclipse.org/birt/phoenix/), 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 the REEs raised 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 during the whole run.

The first example is from Apache Ant, which is a famous Java project building 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 10 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 256 MB memory is used to run programs here. The per-process limit of the file descriptor is 1024, which is the default value on our experimental machine. The details to transform programs to handle FileNotFoundException are presented in Section 6.2.

We first run the original Ant under the unmodified Jikes RVM. Ant raises the FileNotFoundException saying "Too many open 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 five 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, which is an open source Eclipse-based reporting system. There is a database connection leak numbered 237 190 in BIRT v2.3.1 in the BIRT bug database. The connection leak occurs when there is more than one data source in the report design. The method dataEngineShutdown of the class DataSource$ShutdownListener in the package org.eclipse.birt.data.engine.executor closes only 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 (http://tomcat.apache.org/) v5.5.26. MySQL (http://www.mysql.com/) v5.0.67 is used as the database. We use the default configuration of Tomcat. We configure the maximum number of concurrent connections of MySQL to be 100, which 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 Firefox (http://www.getfirefox.net/) 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 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 101st iteration halts abnormally with the exception message "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 nonatomic. 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 of the first 1001 iterations is presented in Fig. 4. The 'Base' series represents 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 the maximum number of connections is violated and the resource collector is triggered. The performance of our approach is stable. For the total 1001 iterations, the resource collector has an 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 bytecode 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

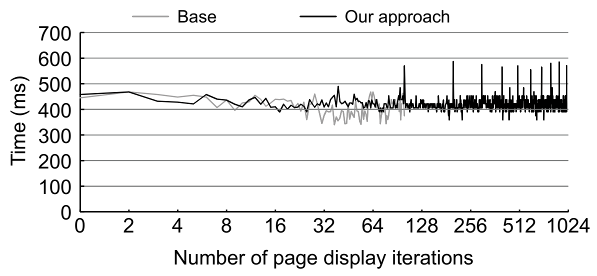


Fig. 4 Time for each page display of the first 1001 iterations of the BIRT example (the x-axis is logarithmic)

suite (Blackburn *et al.*, 2006) of both version 2006-10-MR2 and version 9.12-bach, and SPECjvm98. We run each benchmark program with available memory fixed at twice the minimum with which it can execute. The default workload is used for the DaCapo benchmark suite. Programs from SPECjvm98 are run with a 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 reader/writer, and sockets in the Java system library. Specifications for these resources are simple and we refer mainly to Java API documentation and the source code if necessary to create these specifications. The per-process limit of the file descriptor is 1024, which is the default value on our experimental machine. There are four source methods for 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 four calls of these source methods and these four calls are all in the Java system library. The effect of our transformer on the size of class files is negligible.

We find that the resource collector has never been 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 has been evaluated in Section 6.1 against two known resource leak bugs. The runtime overhead of benchmark programs is presented in Fig. 5. Because many programs

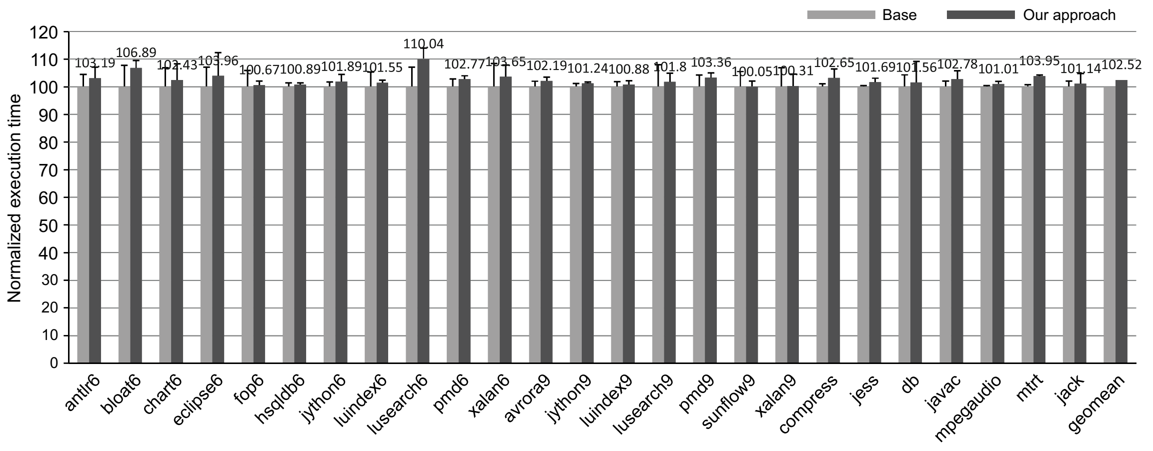


Fig. 5 Runtime overhead on the DaCapo and SPECjvm98 benchmarks. The time is normalized so that the time of running untransformed benchmarks on the unmodified Jikes RVM (Base series) is 100. The thin error bars represent the ranges of the 10 trials. To avoid name collisions, we append names of benchmark programs from DaCapo 2006-10-MR with 6 and append names of benchmark programs from DaCapo 9.12-bach with 9

from DaCapo 9.12-bach cannot run under Jikes RVM v3.3.1, we present only results of those that can be successfully executed (Fig. 5). The runtime overhead of our approach is 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 much work that addresses memory leaks (Guyer *et al.*, 2006; Bond and McKinley, 2008). General automatic approaches to localizing bugs (Lei *et al.*, 2012) and fixing bugs (Qi *et al.*, 2012) have also been proposed. The work closely related to our approach falls into two categories: recovery from exceptions and resource leaks.

7.1 Recovery from exceptions

Carzaniga *et al.* (2013) recovered from runtime exceptions in Java programs by automatically applying workarounds. Chang *et al.* (2009) proposed 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. These two approaches may fix the resource leak that causes the REE, but they cannot successfully recover from the REE because all resources have been exhausted. Dobolyi and Weimer (2008) transformed Java programs to insert 'null' checks and recovery actions guarding every dereference that is potentially 'null'. Friedrich *et al.* (2010) proposed 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) presented an approach to locating and repairing faults in the form of incorrect assignments in Java programs. Such a fault manifests as a flow of incorrect values, which finally leads to an exception. Exceptions originating from these types of faults typically exclude REEs.

Based on their survey in Cabral and Marques (2007), Cabral and Marques (2008) claimed that there is something wrong with current exception handling models, and proposed 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 applies only 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) introduced the concept of 'failure atomicity'. A method is failure atomic if its failed executions due to occurring 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) implemented failure atomicity using checkpointing. Cabral and Marques (2008) implemented 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 do some cleanup work before its associated 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 manner (Boehm, 2003), which makes it a known fact that finalization is unqualified to perform resource collections. There are two main reasons: (1) The 'finalize' methods are bound to the 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 from performance degradation due to huge resource consumption while there is still a large amount of memory available. (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 by three design decisions. First, our approach separates non-memory resource collections from memory collections. Resource collections are triggered in response to REEs, independent 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) no objects depend on r or have some actions (e.g., 'close' and 'finalize') to perform in the future, and (2) resource-releasing methods for 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 no matter whether they are used locally or globally.

To cope with resource leaks, Weimer and Necula (2008) proposed a language extension called 'compensation stack', which allows programmers to annotate resource-acquiring methods with compensations such as resource-releasing method invocations. These compensations are put into stacks that guarantee compensations included to be executed in 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 cannot 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 deterministic order. Resource trees can be closed automatically when the thread that uses it dies. Similar to compensation stacks (Weimer and Necula, 2008), Furm cannot guarantee 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 or leak resources.

7.3 Dynamic resource leak detection and collection

Our previous work (Dai *et al.*, 2013) presents the Resco tool 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. In this study, the improvement over Resco 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 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 releases only leaked resources that can be safely released immediately (S_{b}), while our approach tries to release all leaked resources in S (details are given 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.

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 in finding 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 is then 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 for detecting 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, object staleness cannot be easily applied to non-memory resource collections.

8 Conclusions

This paper presents an approach to automatically recovering 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 transformation needed. In response to REEs, the recovery code first triggers the resource collector to safely collect leaked resources and then retries the method call that failed. 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, C., Avgustinov, P., Christensen, A.S., *et al.*, 2005. Adding trace matching with free variables to Aspectj. *ACM SIGPLAN Not.*, **40**(10):345-364. [doi:10.1145/1103845.1094839]

Arnold, M., Fink, S., Grove, D., *et al.*, 2000. Adaptive optimization in the Jalapeno JVM. Proc. 15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, p.47-65. [doi:10.1145/353171.353175]

Arnold, M., Vechev, M., Yahav, E., 2011. QVM: an efficient runtime for detecting defects in deployed systems. *ACM Trans. Softw. Eng. Methodol.*, **21**(1):2:1-2:35. [doi:10.1145/2063239.2063241]

Blackburn, S.M., Cheng, P., McKinley, K.S., 2004. Oil and water? High performance garbage collection in Java with MMTK. Proc. 26th Int. Conf. on Software Engineering, p.137-146. [doi:10.1109/ICSE.2004.1317436]

Blackburn, S.M., Garner, R., Hoffmann, C., *et al.*, 2006. The DaCapo benchmarks: Java benchmarking development and analysis. Proc. 21st Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications, p.169-190. [doi:10.1145/1167473.1167488]

Boehm, H.J., 2003. Destructors, finalizers, and synchronization. Proc. 30th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, p.262-272. [doi:10.1145/604131.604153]

Bond, M.D., McKinley, K.S., 2008. Tolerating memory leaks. Proc. 23rd ACM SIGPLAN Conf. on Object-Oriented Programming Systems Languages and Applications, p.109-126. [doi:10.1145/1449764.1449774]

Cabral, B., Marques, P., 2007. Exception handling: a field study in Java and .NET. *LNCS*, **4609**:151-175. [doi:10.1007/978-3-540-73589-2_8]

Cabral, B., Marques, P., 2008. A case for automatic exception handling. IEEE/ACM Int. Conf. on Automated Software Engineering, p.403-406.

Carzaniga, A., Gorla, A., Mattavelli, A., *et al.*, 2013. Automatic recovery from runtime failures. Proc. Int. Conf. on Software Engineering, p.782-791.

Chang, H., Mariani, L., Pezze, M., 2009. In-field healing of integration problems with COTS components. Proc. 31st Int. Conf. on Software Engineering, p.166-176. [doi:10.1109/ICSE.2009.5070518]

Chen, F., Roşu, G., 2007. MOP: an efficient and generic runtime verification framework. Proc. 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems and Applications, p.569-588. [doi:10.1145/1297027.1297069]

Dai, Z., Mao, X., Lei, L., *et al.*, 2013. Resco: automatic collection of leaked resources. *IEICE Trans. Inform. Syst.*, **E96-D**(1):28-39.

DeLine, R., Fähndrich, M., 2001. Enforcing high-level protocols in low-level software. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, p.59-69. [doi:10.1145/378795.378811]

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

Dybvig, R.K., Bruggeman, C., Eby, D., 1993. Guardians in a generation-based garbage collector. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, p.207-216. [doi:10.1145/155090.155110]

Fetzer, C., Felber, P., Hogstedt, K., 2004. Automatic detection and masking of nonatomic exception handling. *IEEE Trans. Softw. Eng.*, **30**(8):547-560. [doi:10.1109/TSE.2004.35]

Friedrich, G., Fugini, M., Mussi, E., *et al.*, 2010. Exception handling for repair in service-based processes. *IEEE Trans. Softw. Eng.*, **36**(2):198-215. [doi:10.1109/TSE.2010.8]

Guyer, S.Z., McKinley, K.S., Frampton, D., 2006. Free-me: a static analysis for automatic individual object reclamation. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, p.364-375. [doi:10.1145/1133981.1134024]

Hejlsberg, A., Golde, P., Wiltamuth, S., 2003. C# Language Specification. Addison Wesley.

Herlihy, M., Luchangco, V., Moir, M., 2006. A flexible framework for implementing software transactional memory. *ACM SIGPLAN Not.*, **41**(10):253-262. [doi:10.1145/1167515.1167495]

Lei, Y., Mao, X.G., Dai, Z.Y., *et al.*, 2012. Effective fault localization approach using feedback. *IEICE Trans. Inform. Syst.*, **E95.D**(9):2247-2257. [doi:10.1587/transinf.E95.D.2247]

Martin, M., Livshits, B., Lam, M.S., 2005. Finding application errors and security flaws using PQL: a program query language. Proc. 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, p.365-383. [doi:10.1145/1094811.1094840]

Park, D.A., Rice, S.V., 2006. A framework for unified resource management in Java. Proc. 4th Int. Symp. on Principles and Practice of Programming in Java, p.113-122. [doi:10.1145/1168054.1168070]

Qi, Y.H., Mao, X.G., Wen, Y.J., *et al.*, 2012. More efficient automatic repair of large-scale programs using weak recompilation. *Sci. China Inform. Sci.*, **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. *IEEE Trans. Softw. Eng.*, **36**(2):150-161. [doi:10.1109/TSE.2010.7]

Sinha, S., Shah, H., Görg, C., *et al.*, 2009. Fault localization and repair for Java runtime exceptions. Proc. 18th Int. Symp. on Software Testing and Analysis, p.153-164. [doi:10.1145/1572272.1572291]

Torlak, E., Chandra, S., 2010. Effective interprocedural resource leak detection. Proc. 32nd ACM/IEEE Int. Conf. on Software Engineering, p.535-544. [doi:10.1145/1806799.1806876]

Vallée-Rai, R., Co, P., Gagnon, E., *et al.*, 1999. SOOT—a Java bytecode optimization framework. Proc. Conf. Centre for Advanced Studies on Collaborative Research, p.13.

Walsh, J.D., Bordeleau, F., Selic, B., 2007. Domain analysis of dynamic system reconfiguration. *Softw. Syst. Model*, **6**(4):355-380. [doi:10.1007/s10270-006-0038-4]

Weimer, W., Necula, G.C., 2005. Mining temporal specifications for error detection. *LNCS*, **3440**:461-476. [doi:10.1007/978-3-540-31980-1_30]

Weimer, W., Necula, G.C., 2008. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, **30**(2):8:1-8:51. [doi:10.1145/1330017.1330019]

Wu, Q., Liang, G.T., Wang, Q.X., *et al.*, 2011. Iterative mining of resource-releasing specifications. Proc. 26th IEEE/ACM Int. Conf. on Automated Software Engineering, p.233-242. [doi:10.1109/ASE.2011.6100058]