# Automatic Runtime Recovery via Error Handler Synthesis

Tianxiao Gu[§] Chengnian Sun[*] Xiaoxing Ma[§] Jian Lü[§] Zhendong Su[*]
[§]Department of Computer Science and Technology, Nanjing University, China
[*]Department of Computer Science, University of California, Davis, USA
tianxiao.gu@gmail.com, cnsun@ucdavis.edu, xxm@nju.edu.cn, lj@nju.edu.cn, su@cs.ucdavis.edu

## ABSTRACT

Software systems are often subject to unexpected runtime errors. Automatic runtime recovery (ARR) techniques aim to recover them from erroneous states and maintain them functional in the field.

This paper proposes Ares, a novel, practical approach for ARR. Our key insight is leveraging a system's inherent error handling support to recover from unexpected errors. To this end, we synthesize error handlers in two ways: *error transformation* and *early return*. We also equip Ares with a lightweight in-vivo testing infrastructure to select the promising synthesis method and avoid potentially dangerous error handlers. Unlike existing ARR techniques with heavyweight mechanisms (*e.g.*, checkpoint-restart and runtime monitoring), our approach expands the intrinsic capability of runtime error resilience in software systems to handle unexpected errors. Ares's lightweight mechanism makes it practical and easy to be integrated into production environments.

We have implemented Ares on top of both the Java HotSpot VM and Android ART, and applied it to recover from 52 real-world bugs. The results are promising — Ares successfully recovers from 39 of them and incurs negligible overhead.

## CCS Concepts

•**Software and its engineering** → *Error handling and recovery;*

## Keywords

automatic runtime recovery, JVM, exception handling

## 1. INTRODUCTION

Deployed software systems are subject to runtime errors. Some of these errors can be ***anticipated*** and recovered by programmatically prepared error handlers. For example, reading a non-existing file is invalid and the programmer needs to explicitly tackle this case by following the usage description of certain APIs. The other errors refer to ***unanticipated*** errors that are usually related to bugs in programs (*e.g.*, divide-by-zero, invalid memory access). Ideally these errors (or bugs) should all be eliminated before release, but in reality some slip through the software testing phase and manifest after release and deployment. Different from anticipated errors, handlers usually do not exist for unanticipated errors. Thus, when triggered, they may lead to system failures and incur expensive losses, including security exploits, data corruptions and system unavailability.

***Automatic Runtime Recovery*** To mitigate this problem, a number of techniques [12, 25, 24, 22, 7, 27, 8, 18] have been proposed to make a deployed software system resilient to runtime errors. That is, once an unanticipated runtime error occurs in the field, these techniques try to recover the system from the faulty state to a good one in which the system can still function for subsequent usage. This process is usually referred to as *automatic runtime recovery* (ARR).

ARR techniques typically consist of two stages: amending runtime states to recover from an unanticipated error, and validating the correctness or feasibility of the recovery action. Generally, they can be categorized into two classes: heavyweight and lightweight. The former class relies on heavyweight mechanisms — *e.g.*, checkpoint creation and restoration, online validation by restart or re-execution, expensive instrumentation — to validate the recovery. For example, ARMOR [8] dynamically replaces a piece of problematic code snippet with another equivalent code snippet (provided by developers) and uses checkpoint to rule out invalid recovery solutions. Although in this class various recovery techniques [24, 13, 28, 20, 8] have been proposed, the significant runtime overhead is still a major challenge for heavyweight approaches to be practical.

Lightweight approaches [25, 18] are usually more efficient, but may be less effective than heavyweight approaches due to their aggressive nature and insufficient validation of a recovery. For example, FOC [25] simply discards any invalid memory write and synthesizes a type-specific default value for invalid memory read for server applications. Recently, a novel lightweight approach, RCV [18], has been demonstrated to be effective to some extent in practice. However, RCV has no proactive validation but passive error containment. As the error containment only takes account of parts of the data flow, persistent data may also be ruined by manufactured recovery indirectly. Besides, the control flow may be impacted by the error and leads to infinite recurrences of erroneous states.

***Ares*** In this paper, we propose a lightweight approach (referred to as Ares) for runtime error recovery. At the high



*ASE'16*, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970360

level, our approach expands the intrinsic capability of runtime error resilience existing in software systems to handle unanticipated errors. Ares is lightweight, barely incurring any overhead on normal program executions. Furthermore, instead of relying on heuristics to generate a *single* recovery action [25, 18], Ares synthesizes *multiple* error handler candidates at a time and uses a virtual testing technique to select the most promising one. Ares can successfully avoid infinite recurrences of erroneous states due to its adaptive and context-sensitive nature.

Concretely, once an unanticipated error occurs, we analyze the current call stack, and synthesize a set of recovery solutions (by collecting error handlers residing on the stack or synthesizing new error handlers). Next we determine which handler is the most viable in terms of minimizing the negative impact on the subsequent computation. We perform an in-vivo testing to analyze the impact of every handler. That is, we continue the execution by interpreting the buggy program after applying each recovery solution in a confined virtual execution environment (*viz.*, a sandbox). Note that this in-vivo testing is invoked in an on-demand manner only if an unanticipated error occurs, so it will not impact the performance of normal program executions. Finally, we choose the most promising recovery solution, apply it to and continue the host execution.

In this paper, we use the following two lightweight strategies to synthesize error handlers:

**Error Transformation** transforms an unanticipated error to an error of another type which has a proper handler on the method call stack.

**Early Return** simply ignores the unanticipated error and returns to the caller with a default value, *i.e.*, `null` for reference types.

We have realized the proposed technique on top of Java HotSpot VM in OpenJDK.[1] We embed Java PathFinder (JPF) into HotSpot VM and use it as the virtual execution environment for in-vivo testing. The system is evaluated on 43 bugs in large real-world programs, and can successfully recover from 31 of them.

We have also implemented a mobile version of Ares on top of Android ART.[2] It can successfully recover from 8 out of 9 real Android app bugs, avoiding crashing these apps. Compared to regular Java programs, recovering from these app bugs is equally important. A survey [2] on mobile apps shows that 79% of mobile users would not use a mobile app any more if it is unable to work at the first or second time.

In order to ensure reproducibility, we have made all the data used in this paper (*i.e.*, source code of Ares and the program subjects for evaluation) publicly available at http://lab.artemisprojects.org/groups/ares. These large real-world buggy programs will also benefit the research community by providing a benchmark suite for evaluating various bug detection, recovery and fixing techniques.

***Contributions*** Our main contributions are as follows:

- We propose a lightweight framework to perform ARR by exploiting and extending the intrinsic error resilience of software systems.
- We propose a sandbox approach to evaluating the effectiveness of multiple recovery solutions, and also a set of ranking strategies to choose the most viable one.
- Our implementation, Ares, is built on top of the widely deployed Java HotSpot VM and Android ART, making it easy to be integrated into production environments. Ares intercepts the internal exception handling mechanisms in the two VMs, so the runtime overhead is negligible during normal program executions.
- The evaluation of Ares is promising. We have applied Ares to 52 real world bugs from large software projects, and it can successfully recover from 39 of them. Because their software systems lack intrinsic error resilience, the 13 non-recoverable bugs by Ares require more advanced recovery strategies, which we leave as future work.

***Paper Organization*** The remainder of this paper is organized as follows. Section 2 introduces necessary background on exceptions in Java, and Section 3 demonstrates how Ares recovers from runtime exceptions via two illustrative examples. Section 4 describes the design and implementation of Ares. Section 5 presents our evaluation results. Section 6 discusses the limitation of Ares and other design decisions. Section 7 surveys related work and Section 8 concludes.

# 2. EXCEPTIONS IN JAVA

This section briefly introduces necessary background on Java exceptions to provide proper context for our work.

## 2.1 Exception Handling Mechanism

The Java programming language has a built-in exception handling mechanism. The basic construct is a *try-catch* block, composed of a *try* block and a *catch* block that catches exceptions thrown inside the *try* block. A *catch* block declares a catchable exception type $E_h$, that is, any exception of type $E_h$ or a subtype of $E_h$ is catchable by the *catch* block. We denote a *try-catch* block as

$$(l_s, l_e, l_c, E_h) \qquad \text{(Try-Catch Block)}$$

where $l_s$ and $l_e$ are the start (inclusive) and end (inclusive) of the *try* block; $l_c$ is the start (inclusive) of the *catch* block; $E_h$ is the catchable exception type. Note that multiple *try-catch* blocks can share the same *try* block. Without ambiguity we use exceptions and errors interchangeably in the remainder of this paper.

In JVM once an exception of type $E$ is thrown at a location $l$, JVM looks up an exception handler inside the current method on the top of the call stack. If there exists a *try-catch* block $(l_s, l_e, l_c, E_h)$ where $l_s \leq l \wedge l \leq l_e \wedge E <: E_h$,[3] the execution jumps to this handler $l_c$. Otherwise, the stack frame of this method is popped out and the exception is delivered down to the stack. The same procedure repeats for each method frame until an exception handler is located or the stack becomes empty.

## 2.2 Exception Hierarchy

Figure 1 shows the hierarchy of exception types in Java. Generally, there are two types of exceptions:

***Checked Exceptions*** Checked exceptions are used to indicate that certain anticipated errors may happen and *must* be noticed by programmers. Therefore, programmers should explicitly use a *try-catch* block to catch and handle these exceptions, or deliver them up to the caller. This is enforced by the type system of Java. For example, reading a non-existing

[1] http://openjdk.java.net/groups/hotspot/

[2] https://source.android.com/devices/tech/dalvik/

[3] The operator <: denotes subtyping relation.

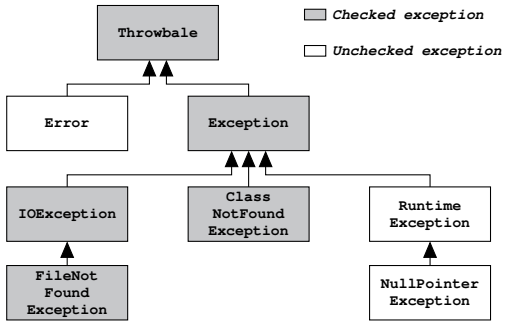


**Figure 1: Java exception type hierarchy.**

file will throw an exception of `FileNotFoundException`. Thus when reading a file, the programmer needs to either catch this exception or let the caller handle it.

***Unchecked Exceptions*** Unchecked exceptions do not require programmers to explicitly handle them, as they are often assumed not to occur during execution. Thereby, such an exception may crash the program unexpectedly. Unchecked exceptions in Java can be further classified into two categories `Error` (*i.e.*, `java.lang.Error`) and `RuntimeException`. The former category refers to errors related to Java virtual machines, such as `OutOfMemoryError` and `StackOverflowError`, which are usually not recoverable for programmers. On the other hand, `RuntimeException` and its subclasses are often symptoms of programmers' bugs, which are not expected to manifest at runtime, *e.g.*, `NullPointerException`. For simplicity, we will use unchecked exception to refer to the latter category.

In this paper, we focus on the recovery of the unchecked exceptions, namely, maintaining the system functional for subsequent usage by surviving unanticipated unchecked exceptions. When there is no ambiguity, we use error and unchecked exception interchangeably in the remainder of this paper.

# 3. ILLUSTRATIVE EXAMPLE

This section presents two examples to illustrate how Ares recovers a software system from unanticipated runtime exceptions with its two types of error handler synthesis strategies, *i.e.*, error transformation and early return.

## 3.1 Recovery via Error Transformation

We use a real security bug of Tomcat 7 to demonstrate how Ares recovers the system from the manifestation of this bug via error transformation. Figure 2a shows this information disclosure security bug (CVE-2013-2071) [1].

The method `fireOnComplete` on line 5 realizes the *Observer* design pattern. The listeners are registered to be called on some events, and may be implemented by third-party developers. A listener may be buggy and therefore it is possible that the method call on line 5 abnormally exists with an unchecked exception. However, the *catch* block on line 6 only handles `IOException`. This exception will be propagated along the stack, and results in the server failing to recycle the data of the current web request. Consequently the leaked data later becomes accessible for the next web request.

If Ares is deployed, this security exploit can be prevented. Error transformation will automatically convert the unchecked

```
1 for (AsyncListenerWrapper l : listenersCopy) {
2   try {
3     /* BUG: An unchecked exception may be
4        thrown in the call below. */
5     l.fireOnComplete(event);
6   } catch (IOException e) {
7   }}
```

**(a) Tomcat bug 54178 (CVE-2013-2017)**

```
1 for (AsyncListenerWrapper l : listenersCopy) {
2   try {
3     l.fireOnComplete(event);
4 - } catch (IOException ioe) {
5 + } catch (Throwable t) {
6 +   ExceptionUtils.handleThrowable(t);
7   }}
```

**(b) Patch**

**Figure 2: Tomcat bug 54178 (CVE-2013-2071) and its patch.**

```
1 /* BUG: getDigest(...) may return null */
2 Stringmd5a1 = getDigest(username, realm).toLowerCase
      (Locale.ENGLISH);
3 if (md5a1 == null)
4     return null;
```

**(a) Tomcat bug 54438**

```
1 String md5a1 = getDigest(username, realm);
2 /* FIX: check value of
3 getDigest(...) */
4 if (md5a1 == null)
5     return null;
6 md5a1 = md5a1.toLowerCase(Locale.ENGLISH);
```

**(b) Patch**

**Figure 3: Tomcat bug 54438 and its patch.**

exception thrown on line 5 into an `IOException` object which will be immediately processed by the handler on line 6, preventing information leakage. In fact, the official patch, shown in Figure 2b, is very similar to our recovery process. It just makes the *catch* block capable of handling every exception (*i.e.* `Throwable`).

The recovery strategy *error transformation* aims to exploit the error resilience existing in software systems (*i.e.*, existing exception handlers) to handle unanticipated unchecked exceptions. It is also similar to one of the common ways developers cope with unchecked exceptions, namely, catching the unchecked exception, converting it to another type and re-throwing the new exception.

## 3.2 Recovery via Early Return

This subsection shows another Tomcat bug, which can be recovered by the strategy *early return*. Figure 3a displays Tomcat Bug 54438. On line 2, the method call `getDigest` may return `null` if the `username` does not exist. This `null` further triggers a `NullPointerException` when we use the returned digest (*i.e.*, `null`) to invoke another method `toLowerCase`.

To recover from this exception, Ares intercepts the internal exception handling of JVM. It then ignores the exception and returns `null` for the call to `toLowerCase`. However from the perspective of overhead, Ares's synthesized error handler does not incur additional overhead on normal program executions. Figure 3b shows the developer's patch to this bug, which is equivalent to our synthesized error handler, although in

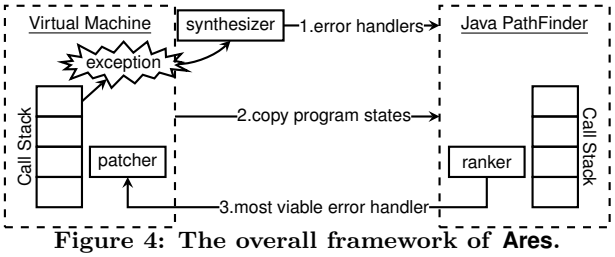

**Figure 4: The overall framework of Ares.**

different representations. It first checks whether the return value of `getDigest()` is `null`. If yes, then it returns from the current method.

Similar to error transformation, the strategy early return is inspired by another common way in which developers handle unchecked exceptions, that is, catching an unchecked exception and returning from the current method with a default value (*e.g.*, 0 for numerics types and null for reference types).

# 4. APPROACH

This section describes our approach for ARR. Figure 4 shows the overall framework of Ares.

**JVM** Our approach is built on a regular Java virtual machine (JVM). It is used to execute programs in normal mode. Ares only intercepts the exception handling of the underlying JVM. Once an error of interest occurs (*e.g.*, unanticipated unchecked exceptions), Ares stops JVM, takes over the execution, and starts the recovery process.

**Synthesizer** Based on the thrown exception and the content of the call stack in JVM, the synthesizer generates a set of candidate solutions for recovery (*i.e.*, error handlers), which will be tested in a "sandbox" to assess their capabilities of recovering from the exception.

**JPF** We use Java PathFinder (JPF) as the "sandbox", because it supports checkpoint and restoration, facilitating program state exploration and rollback. In detail, once we start the recovery process, a JPF instance is created and initialized with the program states in the host JVM. We apply a synthesized error handler in JPF each time until all handlers are tested.

**Ranker** After the testing of JPF, we propose an effective heuristic to rank these error handlers and return the most promising one.

**Patcher** The patcher will realize the error handler returned by the ranker in the host JVM on the fly. Then the program execution resumes.

## 4.1 Identifying Errors of Interest

Ares piggybacks on the internal exception handling mechanism of JVM. When an exception is thrown, Ares first checks whether the exception is of interest and then attempts to recover from it if yes. Ares only checks and recovers from the exceptions that satisfy the following two conditions,

1. The exception must be an unchecked exception, an object of `RuntimeException` or its subtype. We omit checked exceptions as they have been explicitly handled by programmers, which is enforced by the Java type system.

**Algorithm 1:** Force-Throwable Error Transformation

**Input:** $\langle m_1, \cdots, m_n \rangle$, the recovery context
**Input:** E, an unchecked exception
**Output:** an exception type to which E can be transformed

```
1 for i ← 2 to n do
2     H ← exception handlers in method m_i
3     l_{i-1} ← the location of the call to m_{i-1}
4     foreach handler (l_s, l_e, l_c, E_h) ∈ H do
5         if l_s ≤ l_{i-1} ∧ l_{i-1} ≤ l_e then
6             return E_h
7 return null
```

2. The unchecked exception should have no corresponding *effective* error handler. Apparently, an unchecked exception is of interest if it has no error handler in the whole call stack. We also consider another type of unchecked exceptions that are handled by a trivial overly-stated *catch* block, *e.g.*, unchecked exceptions caught by `catch( Throwable e){...}`. We refer to the first type as *uncaught exception* and the latter as *trivially handled exception*.

## 4.2 Error Handler Synthesis

We synthesize error handlers in two ways: *error transformation* and *early return*. Before detailing them, we introduce a notion *recovery context* to facilitate the description.

***Recovery Context*** Given a thrown exception $E$, let $H$ be the stack frame which has a *catch* block to handle $E$. Then the recovery context of $E$ is the sequence of frames in the stack from the top frame $T$ to the frame above $H$ (exclusive of $H$), *i.e.*, $[T, H)$. We also use $\langle m_1, \cdots, m_n \rangle$ ($m_1$ is the top frame) to denote a recovery context.

Take Figure 2 as an example. An unchecked exception is thrown in the method call `fireOnComplete(event)` on line 5. This exception cannot be handled by the existing handler for `IOException` on line 6, and is handled by a catch block for `Throwable` in a method frame *End* near the bottom of the call stack. Therefore, the recovery context for this exception is the stack frames from the top frame to the frame right above *End*.

For a trivially handled exception, we only try to recover from it before the last frame of its recovery context. If we fail to synthesize a proper error handler, we will honor the original semantics of the program and let the programmed error handler take charge of the execution.

### 4.2.1 Force-Throwable Error Transformation

The simplest way to reuse existing error handlers is to make every *catch* block catch `Throwable`. Thus, we propose the Force-Throwable Error Transformation (FTET) in Algorithm 1. FTET ignores the type of the raised exception and uses only the location information to search for error handlers. As Java is type-safe, we also need an error transformation that converts the unchecked exception to the exception that the *catch* block declares.

### 4.2.2 Stack-Based Error Transformation

A method should only throw a checked exception that is declared by the method. Therefore, we propose the Stack-Based Error Transformation (SBET) in Algorithm 2. In

**Algorithm 2:** Stack-Based Error Transformation

**Input:** $\langle m_1, \cdots, m_n \rangle$, the recovery context
**Input:** E, an unchecked exception
**Output:** a set R of checked exception types to which E can be transformed

```
1 R ← ∅
2 for i ← 2 to n do
3     H ← exception handlers in method m_i
4     S ← checked exceptions declared by method m_{i-1}
5     l_{i-1} ← the location of the call to m_{i-1}
6     foreach handler (l_s, l_e, l_c, E_h) ∈ H do
7         if E_h ∈ S ∧ l_s ≤ l_{i-1} ∧ l_{i-1} ≤ l_e then
8             R ← R ∪ {the type of E_h}
9 return R
```

```
1 public void printMultiLn(String s) {
2   int index = 0;
3   // look for hidden newlines inside strings
4   while ((index=s.indexOf('\n',index)) > -1 ) {
5     javaLine++;
6     index++;
7   }
8   writer.print(s);
9 }
```

**Figure 5: Tomcat bug 43758**

SBET, not any arbitrary checked exception can be the target exception of error transformation. First, the target exception should have a proper error handler in the recovery context. Second, the checked exception must be declared by one of the active methods in the recovery context.

In Algorithm 2, once an unchecked exception $E$ is thrown in the method $m_1$, we check whether its callers (from $m_2$ to $m_n$) have available exception handlers which we can leverage to handle $E$. Specifically, if a method $m_i$ declares to throw an exception $E_h$ (tested by $E_h \in S$ on line 7) and there is a handler for this exception, then we can transform $E$ to $E_h$.

#### 4.2.3 Early Return

This recovery strategy takes as input two parameters: a number $n$ ($n \geq 1$) of stack frames to pop out of the call stack and a value $v$ of the return type. Once an unanticipated error occurs, this strategy pops $n$ stack frames from the call stack and returns $v$ as the return value. If $n = 1$, we name this early return as **First Early Return** (**FER**).

These two parameters should be carefully chosen, otherwise early return will result in dense cascaded exceptions or introduce other unexpected program behavior after recovery. In particular, the major challenge of early return is how to fabricate a good return value. In FOC [25] and RCV [18], they use the default value of a type, *e.g.*, `0` for integers, `null` for object types. However, this may be problematic. Take Figure 5 (*i.e.*, Tomcat Bug 43758) as an example. The bug happens when the parameter `s` is `null`, leading to a `NullPointerException` when we call `indexOf` on line 4. If we choose $v = 0$ and return it as a default value, implemented in FOC and RCV, then the loop becomes infinite.

Different from FOC and RCV which propose a single recovery solution, Ares proposes a bounded number of parameters to perform early return recovery. We then use JPF to evaluate these parameters, which is able to weed out inappropriate parameters. More details are available in Section 4.3.

***Void-Only Early Return (VOER)*** Although it is difficult to choose a good value to return from the domain of a type, it is easy to choose a value for type `void`, which is `void` itself. We specialize early return by only returning from a method with `void` return type. Void-Only Early Return (VOER) just walks along the stack to locate the first method with void return type and then makes an early return there. There may not be such a method. Hence, VOER may fail to recover from some exceptions.

### 4.3 Evaluating Synthesized Handlers

As shown in Figure 4, after generating a set of error handlers for a runtime error, we invoke JPF to test the applicability of each handler. Specifically, Ares walks through the stack in the host JVM to collect necessary information to quickly instantiate an instance of JPF. Each time JPF applies one error handler, and checks whether it can recover the system from the bug. Thanks to the built-in support for state checking and restoration of JPF, to test a handler, we first save the current state in which the buggy thread is about to crash, and then apply the error handler. If the execution stops, JPF restores the state and starts over to test another error handler. After all error handlers are tested, JPF returns a set of viable error handlers, which are later ranked by the ranking strategy elaborated in Section 4.3.2.

#### 4.3.1 Evaluating an Error Handler

Given a runtime error thrown in the top stack frame $m_1$, let $\langle m_1, \cdots, m_n \rangle$ denote its recovery context. Before JPF executes any code of the synthesized error handler, a number $d$ of call stack frames need to be popped out. Take error transformation as an example, if the target error handler is in $m_i$, then stack frames $[m_1, \ldots, m_{i-1}]$ should be popped out first. The same procedure also applies to early return.

The JPF execution begins at the target error handler in $m_i$. The execution may call new methods, create new stack frames and increase the stack size. But these new stack frames do not belong to the recovery context. During the execution, there is a lowest stack frame $m_j$ in the recovery context. We use $[m_i, m_j)$ in measuring the length of testing, as they are related to the context in which the exception is thrown. The length $(j - i)$ is denoted as $c$. Besides, we also record the number of executed instructions as $s$. Intuitively, these two metrics are complementary in measuring the confidence of the JPF testing execution.

The handler evaluation may stop in one of the following scenarios:

**No Error** All methods in the recovery context complete their execution normally.

**Timeout** In order to maintain responsiveness of the system under recovery, we stop testing the current error handler if a maximum number of instructions (specified by the parameter TIMEOUT) has been executed.

**Uninterpretable Behavior** The JPF execution encounters a VM behavior that it cannot interpret, *e.g.*, a call to uninterpretable native code.

**Cascaded Error** The applied error handler in JPF execution triggers another runtime error.

Finally, the testing result of an error handler is represented as a tuple $(t, r, d, s, c)$, where $t$ is the type of the error handler (either error transformation or early return), $r$ is the stop scenario of the JPF execution, $d$ is the number of discarded stack frames in the recovery context, $s$ is the number of

executed instructions, $c$ is the number of stack frames in the recovery context that have completed their execution.

#### 4.3.2 Ranking Error Handlers

The ranking heuristic is designed based on observation that the most promising error handler usually outperforms others in two aspects: fewer discarded stack frames which is measured by $d$, and longer JPF testing which is measured by $s$ and $c$.

First, we classify testing results as either *benign* or *malignant*. *No Error* and *Timeout* are straightforwardly treated as benign. For *Uninterpretable Behavior* and *Cascaded Error*, their executed instructions $s$ must exceed a threshold STEPS. For *Cascaded Error*, despite STEPS, their exercised stack frames $c$ must further exceed another threshold FRAMES. Any other result is treated as malignant.

For benign error handlers, their online testing reflects high confidence. Thus, we prefer the one with fewer discarded stack frames $d$. If two testing results have the same $d$, we choose the one with more executed instructions. Malignant error handlers are only used when there is no benign one. For these error handlers, their online testing brings poor confidence. Thus, we simply prefer the one with more executed instructions.

### 4.4 Implementation

We have implemented Ares on two popular platforms: Java HotSpot VM and Android ART. This design decision enables our approach to be a drop-in substitute for standard VMs, and easy to deploy in production environments without complex configurations.

#### 4.4.1 Ares on Java HotSpot VM

The Java HotSpot VM is an open-source industrial-strength JVM. It has been distributed as the default JVM vendor for popular Linux distributions (*e.g.*, Ubuntu and Fedora). We intercept the standard exception handling mechanism of HotSpot to identify errors of interest and perform recovery. In this way, our recovery system has little impact on the performance of software systems in normal execution (*i.e.*, when no unanticipated errors occur). There is even no overhead when no anticipated or unanticipated exceptions occur, which is usually the case for majority of execution time.

We embed JPF in the host JVM to test synthesized error handlers and invoke it on demand when recovery is required. When a thread is about to crash due to an unanticipated error of interest, we intercept the error handling process, and start a JPF instance to continue the execution by applying an error handler in the same thread. Specifically, we first create a call stack for JPF by duplicating the stack frames of the recovery context (the top frames in the call stack of the about-to-crash thread) in the host JVM. Then the program execution is altered with an error handler and resumed in this JPF instance. This process is repeated, with each iteration applying a different error handler. Lastly, we rank these error handlers and choose the most promising one.

As JPF and JVM have different object models, when JPF needs to access an object in the host JVM at runtime, Ares converts that object to the representation of JPF. In order to reduce runtime overhead, we only convert JVM objects on demand, that is, we only convert the minimally sufficient objects when they are requested. After the testing for a handler completes, Ares resets all values of objects for the next testing. Interactions with external resources (*e.g.*, files, databases) are uninterpretable behaviors that are not supported by JPF. These behaviors should indeed be forbidden as they may induce side effects on the host execution during online testing, which are difficult to revert when JPF tests another error handler.

If a synthesized error handler is applied for recovery, Ares does not persist it in the buggy method for future program execution. This is mainly because the error handler is synthesized based on the context (*e.g.*, method call stack) in which the error manifests. Next time the method encounters the same error, Ares will synthesize another error handler based on the context, which may be different from the previous handler as the context may differ.

#### 4.4.2 Ares on Android ART

The mobile version of Ares is implemented on top the new Android ART released in Android 5.0.1. We modify the ART runtime and deploy it in a Nexus 5 mobile phone. However, currently we only implement a conservative strategy in ART instead of embedding JPF. Android ART uses a different byte code representation (*i.e.*, dalvik) and a different layout of stack frames, which is not supported by JPF.

Our conservative strategy in Android ART first attempts to apply SBET. If SBET is not applicable, it then attempts to apply VOER. If both fail, we just abort the recovery process. Our evaluation on nine bugs in real-world Android apps shows that this strategy is always able to find an appropriate error handler.

**Table 1: Programs used in our experiments.**

| | Bugs | Recovered | |
|---|---|---|---|
| Tomcat | 19 | 14 | web application server |
| Jetty | 12 | 8 | web application server |
| JMeter | 4 | 4 | GUI application |
| GanttProject | 8 | 5 | GUI application |
| Android | 9 | 8 | various mobile apps |
| Total | 52 | 39 | |

## 5. EXPERIMENTS

To demonstrate the effectiveness of Ares, this section presents our extensive evaluation of Ares on widely-used web servers, desktop GUI applications and mobile apps. The experiments on server and desktop applications were conducted on a Linux machine with Intel Quad-Core i7 3.4GHz CPU and 12 GB memory; those on mobile apps were done on a Nexus 5 smart phone.

We have evaluated Ares on *all 52 exception-related bugs* that we were able to reproduce from several widely-adopted projects, and Ares successfully recovered from 39 of them, *e.g.*, the program can continue running to serve new user requests. The 13 non-recoverable bugs are mainly due to lack of intrinsic error resilience in these software systems. Tackling them requires more advanced recovery strategies besides error transformation and early return, which we leave as future work.

***Program Subjects*** Table 1 lists the details of the program subjects used in our evaluation. Both Tomcat and Jetty are popular Java web servers that have been under active development for over ten years, and widely deployed in production environments. JMeter is a web testing tool. GanttProject

**Table 2: Acronyms of exception names**

| Exception Name | Acronym |
|---|---|
| ArrayIndexOutOfBoundsException | AIO |
| BufferOverflowException | BOE |
| ConcurrentModificationException | CME |
| ClientAbortException | CAE |
| DeploymentException | DE |
| Exception | E |
| EofException | EE |
| FileNotFoundException | FNF |
| IllegalArgumentException | IAE |
| IllegalAccessException | ICE |
| IllegalJidException | IJE |
| IllegalStateException | ISE |
| IllegalUserActionException | IUA |
| InteruptedException | IE |
| IOException | IOE |
| JasperException | JE |
| MalformedURLException | MUE |
| MalformedCachePatternException | MCP |
| MPXJException | MPX |
| NumberFormatError | NFE |
| NullPointerException | NPE |
| RuntimeException | RE |
| ServletException | SE |
| StringIndexOutOfBoundsException | SIO |
| SQLException | SQE |
| Throwable | T |
| UnavailableException | UE |
| UnsupportedEncodingException | UEE |
| UnsupportedOperationException | UOE |
| XNIException | XNI |

is a project planning tool. The Android apps include web browsers, instant messengers and productivity tools.

***Collection of Bugs*** In order to collect these bugs, we first searched the bug repositories, revision logs and release notes with the keywords "exception" or "NPE". This step yielded 244 bugs. Then we attempted to reproduce all these bugs according to the instructions recorded in their bug reports. At last, we obtained 52 bugs that were reproducible in our testing environment. Besides, almost all bugs were reproduced in a standalone server with a deployed web application or by manually exercising a GUI application. For the other bugs, we directly used unit tests provided in bug reports.

To save space and facilitate description, we use acronyms of exceptions in the rest of this section, as shown in Table 2.

## 5.1 Evaluation on Java HotSpot VM

We evaluate Ares with 43 real-world bugs on the Java HotSpot VM. In order to better understand the effectiveness of Ares, we also evaluate four basic strategies with the same set of bugs, *i.e.* FTET, SBET, FER and VOER. In each basic strategy evaluation, given a buggy program, we apply the strategy to recover the program from not only the unanticipated exception triggered by the bug, but also the cascaded exceptions that are of interest for recovery. In contrast, on each exception of interest, Ares will adaptively select the most promising error handler based on the context of the exception rather than sticking to a single strategy.

The parameters of Ares are configured as follows, $\mathsf{TIMEOUT} = 1000, \mathsf{STEP} = 100, \mathsf{FRAMES} = 1$. As Algorithm 2 shows, SBET returns a set of exception types to transform to. In the evaluation of the basic strategy SBET, we only use the first exception type in the nearest call stack frame as the transformation target. Similarly, FER and VOER also propose a list of call stack frames to return from, as described in Section 4.2.3; and we select the top frame as the target in their evaluations.

**Table 3: Summary of Recovery Results on Java HotSpot VM**

| Result | FTET | SBET | FER | VOER | Ares |
|---|---|---|---|---|---|
| N.A. | 2 | 13 | 0 | 3 | 0 |
| Failure | 33 | 24 | 13 | 16 | 12 |
| Repair | 5 | 4 | 10 | 3 | 10 |
| Plausible | 3 | 2 | 20 | 21 | 21 |
| Recovery | 8 | 6 | 30 | 24 | 31 |
| Repair Rate | 12.2% | 13.3% | 23.3% | 7.5% | 23.3% |
| Plausible Rate | 7.3% | 6.7% | 46.5% | 52.5% | 48.8% |
| Success Rate | 19.5% | 20.0% | 69.8% | 60.0% | 72.1% |

The result of a recovery is analyzed in two steps:

**Step One** We manually check whether the recovery makes software functional for later use. Take GanttProject as an example. After recovery, if we still can make planning, we then classify this recovery as effective; if the application crashes or we cannot make planning, we then classify the recovery as ineffective.

**Step Two** If the recovery is effective, we further compare the synthesized error handler with the developer's patch. If they are semantically equivalent, then we classify the error handler as a *repair*, otherwise as a *plausible* recovery.

Table 3 summarizes the recovery results. The row `N.A.` represents the number of cases where the corresponding strategy is not applicable (*e.g.*, no existing exception handler for FTET and SBET); the row `Failure` represents the number of cases where the corresponding strategy fails to recover from the bugs. The `Recovery` row is the sum of `Repair` and `Plausible`, the `Repair Rate` is computed as `Repair`/(`Recovery` + `Failure`), the `Plausible Rate` is computed as `Plausible`/(`Recovery`+`Failure`), and the `Success Rate` is computed as `Recovery`/(`Recovery` + `Failure`).

Among the four basic strategies, FER is the most effective one. This observation contributes much to our ranking algorithm. However, FER may result in a catastrophic infinite loop (*e.g.*, the bug in Figure 5). An infinite loop should be prevented in advance, as it may propagate bad effects rapidly. VOER is also effective. However, it results in fewer repairs but more plausible recoveries. Although FTET has more applicable scenarios, its repair rate and success rate are slightly lower than SBET.

Ares has the best overall recovery result and also results in fewer cascaded exceptions, especially compared to FER (will be discussed in the following section together with Table 4). Although our ranking heuristic of Ares overall works well in the evaluation, it rejects two repairs (*i.e.*, Bug 29 and 30 in Table 4), and accepts two plausible recovery handlers with fewer cascaded errors than the repairs. We believe that with an enhanced ranking mechanism, the evaluation results will be further improved, which we leave as future work.

### 5.1.1 Details of Evaluation

Table 4 shows the details of our evaluation on the 44 bugs. The first column shows the unique IDs of these bugs in order to conveniently refer to them in this paper, and the second column shows the real bug IDs in their corresponding bug repositories. The third column lists the types of exceptions thrown when these bugs manifest themselves.

The multi-column *Basic Strategy* shows the statistics of recoveries with the four basic strategies. We only show the

**Table 4: Recovery of bugs on the Java HotSpot VM.**

| # | Bug ID | Error | Basic Strategy[α] | | | | | | | | | | | | Ares | | | | | | | | | | | | | | | | | | | | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | FTET | | | SBET | | | FER | | | VOER | | | FTET | | | | SBET | | | | FER | | | | VOER | | | | Final | | | | | | | (ms) |
| | | | $t$ | ℂ | ℝ | $t$ | ℂ | ℝ | $t$ | ℂ | ℝ | $t$ | ℂ | ℝ | $r$ | $d$ | $s$ | $c$ | $r$ | $d$ | $s$ | $c$ | $r$ | $d$ | $s$ | $c$ | $r$ | $d$ | $s$ | $c$ | $t$ | $r$ | $d$ | $s$ | $c$ | ℂ | ℝ | |
| 1 | TC 43338 | IAE | E | 0 | F | ICE | 0 | F | void | 0 | P | 1 | 0 | P | C | 4 | 211 | 0 | C | 4 | 211 | 0 | C | 1 | 583 | 1 | C | 1 | 583 | 3 | $void^1$ | C | 1 | 583 | 3 | 0 | P | 723 |
| 2 | TC 43758 | NPE | E | 0 | F | JE | 0 | F | 0 | ∞ | F | 2 | 1 | R | U | 14 | 21 | 0 | U | 14 | 21 | 0 | C | 1 | 16 | 0 | U | 2 | 120 | 0 | $void^2$ | U | 2 | 120 | 0 | 1 | R | 887 |
| 3 | TC 46298 | NPE | SQE | 0 | R | SQE | 0 | R | void | 0 | P | 1 | 0 | P | T | 2 | — | 0 | T | 2 | — | 0 | U | 1 | 454 | 5 | U | 1 | 454 | 5 | $void^1$ | U | 1 | 454 | 5 | 0 | P | 862 |
| 4 | TC 49184 | AIO | T | 0 | F | N.A. | | | void | 2 | P | 1 | 1 | F | T | 3 | — | 0 | N.A. | | | | N | 1 | 50 | 3 | N | 1 | 50 | 3 | $void^1$ | N | 1 | 50 | 3 | 2 | P | 766 |
| 5 | TC 49883 | UOE | LE | 0 | F | LE | 0 | F | null | 7 | P | 6 | 4 | P | T | 11 | — | 0 | T | 11 | — | 0 | T | 1 | — | 1 | T | 6 | — | 1 | $null^1$ | T | 1 | — | 1 | 7 | P | 249 |
| 6 | TC 51401 | IAE | E | 5 | F | N.A. | | | void | 5 | P | 1 | 5 | P | C | 1 | 211 | 0 | N.A. | | | | U | 1 | 160 | 0 | U | 1 | 160 | 0 | $void^1$ | U | 1 | 160 | 0 | 5 | P | 803 |
| 7 | TC 51403 | NPE | E | 10 | F | N.A. | | | null | 22 | F | N.A. | | | C | 1 | 113 | 0 | N.A. | | | | C | 1 | 6 | 0 | N.A. | | | | E | C | 1 | 113 | 0 | 10 | F | 679 |
| 8 | TC 51550 | ISE | IOE | 0 | F | IOE | 0 | F | null | 45 | F | 5 | 0 | F | N | 8 | 683 | 1 | N | 8 | 683 | 1 | T | 1 | — | 7 | T | 5 | — | 3 | $void^1$ | T | 1 | — | 7 | 5 | F | 969 |
| 9 | TC 51910 | NPE | T | 0 | F | IOE | 0 | F | void | 1 | R | 1 | 0 | R | C | 7 | 394 | 0 | N | 7 | 761 | 1 | N | 1 | 507 | 7 | N | 1 | 507 | 7 | $void^1$ | N | 1 | 567 | 7 | 0 | R | 313 |
| 10 | TC 53677 | ISE | CAE | 0 | F | IOE | 0 | F | void | 14 | F | 1 | 14 | F | N | 2 | 410 | 1 | T | 2 | — | 0 | N | 1 | 93 | 2 | N | 1 | 93 | 2 | $void^1$ | N | 1 | 93 | 2 | 4 | F | 869 |
| 11 | TC 54178 | RE | IOE | 0 | R | IOE | 0 | R | void | 0 | P | 1 | 0 | P | C | 2 | 239 | 0 | C | 2 | 239 | 0 | T | 1 | — | 4 | T | 1 | — | 4 | $void^1$ | T | 1 | — | 4 | 0 | P | 1277 |
| 12 | TC 54438 | NPE | T | 0 | F | IOE | 0 | F | null | 0 | R | 5 | 0 | F | N | 5 | 995 | 1 | T | 5 | — | 0 | T | 1 | — | 2 | N | 5 | 294 | 1 | $null^1$ | T | 1 | — | 2 | 0 | R | 766 |
| 13 | TC 54703 | NPE | IOE | 0 | P | IOE | 0 | P | null | 0 | P | 3 | 0 | P | C | 1 | 823 | 3 | C | 1 | 823 | 3 | C | 1 | 824 | 3 | C | 3 | 806 | 1 | $null^1$ | C | 1 | 824 | 3 | 0 | P | 836 |
| 14 | TC 55454 | NPE | E | 1 | P | IOE | 0 | P | null | 4 | P | 4 | 0 | P | N | 1 | 6 | 1 | N.A. | | | | C | 1 | 1 | 0 | N.A. | | | | E | N | 1 | 6 | 1 | 1 | P | 854 |
| 15 | TC 56010 | IAE | T | 0 | F | IOE | 0 | F | void | 0 | F | 1 | 0 | F | U | 3 | 3 | 0 | U | 3 | 3 | 0 | U | 1 | 33 | 0 | U | 1 | 33 | 0 | $void^2$ | N | 2 | 4 | 2 | 1 | F | 635 |
| 16 | TC 56246 | NPE | N.A. | | | N.A. | | | null | 0 | R | N.A. | | | N.A. | | | | N.A. | | | | T | 1 | — | 0 | N.A. | | | | $null^1$ | T | 1 | — | 0 | 0 | R | 591 |
| 17 | TC 56736 | ISE | IOE | 0 | F | N.A. | | | 0 | 0 | R | 2 | 0 | P | C | 4 | 703 | 0 | N.A. | | | | T | 1 | — | 3 | T | 2 | — | 2 | $0^1$ | T | 1 | — | 3 | 0 | R | 1131 |
| 18 | TC 58232 | NPE | DE | 1 | F | N.A. | | | null | 4 | F | 2 | 2 | F | C | 3 | 48 | 0 | N.A. | | | | C | 1 | 1 | 0 | T | 1 | — | 1 | $void^2$ | T | 1 | — | 1 | 1 | F | 1043 |
| 19 | TC 58490 | NPE | MUE | 0 | F | N.A. | | | null | 39 | P | 2 | 0 | F | U | 2 | 1 | 0 | N.A. | | | | C | 1 | 85 | 0 | U | 2 | 1 | 0 | $null^1$ | C | 1 | 85 | 0 | 7 | P | 532 |
| 20 | JT 335500 | NPE | EE | 0 | F | IOE | 0 | F | void | 16 | P | 1 | 2 | F | T | 6 | — | 0 | T | 6 | — | 0 | U | 1 | 11 | 0 | U | 1 | 11 | 0 | $void^4$ | T | 4 | — | 2 | 0 | P | 797 |
| 21 | JT 358027 | NPE | EE | 0 | F | IOE | 0 | F | null | 6 | F | 2 | 0 | P | C | 4 | 196 | 0 | T | 4 | — | 0 | C | 1 | 10 | 0 | C | 2 | 187 | 2 | $void^2$ | C | 2 | 187 | 2 | 0 | P | 775 |
| 22 | JT 375490 | NPE | IOE | 0 | F | FNF | 0 | F | null | 0 | F | 2 | 0 | F | N | 1 | 35 | 0 | N | 1 | 35 | 0 | U | 1 | 88 | 0 | N | 2 | 4 | 2 | IOE | N | 1 | 35 | 0 | 0 | F | 620 |
| 23 | JT 393158 | ISE | T | 3 | F | IOE | 0 | F | 0 | 2 | P | 2 | 2 | P | T | 2 | — | 2 | T | 4 | — | 0 | T | 1 | — | 3 | T | 2 | — | 2 | $0^1$ | T | 1 | — | 3 | 1 | P | 1082 |
| 24 | JT 395794 | NPE | IAE | 0 | F | IOE | 0 | F | null | 0 | R | 2 | 0 | P | T | 4 | — | 0 | C | 8 | 767 | 0 | U | 1 | 190 | 1 | U | 2 | 112 | 1 | $null^1$ | U | 1 | 190 | 1 | 0 | R | 1065 |
| 25 | JT 401531 | SIO | UE | 0 | F | JE | 0 | F | 0 | 0 | F | 7 | 0 | F | T | 7 | — | 0 | C | 8 | 8 | 0 | U | 1 | 11 | 0 | U | 7 | 172 | 0 | $null^1$ | T | 2 | — | 2 | 2 | F | 763 |
| 26 | JT 402106 | BOE | E | 0 | F | N.A. | | | null | 0 | P | N.A. | | | T | 1 | — | 0 | N.A. | | | | U | 1 | 5 | 0 | N.A. | | | | E | T | 1 | — | 0 | 1 | F | 630 |
| 27 | JT 404283 | NPE | IOE | 0 | R | N.A. | | | void | 0 | R | 1 | 0 | R | T | 1 | — | 0 | N.A. | | | | U | 1 | 974 | 2 | U | 1 | 974 | 2 | IOE | T | 1 | — | 0 | 0 | R | 714 |
| 28 | JT 411844 | AIO | N.A. | | | N.A. | | | void | 0 | P | 1 | 0 | P | N.A. | | | | N.A. | | | | T | 1 | — | 0 | T | 1 | — | 0 | $void^1$ | T | 1 | — | 0 | 0 | P | 500 |
| 29 | JT 424051 | NPE | T | 0 | F | IOE | 0 | F | null | 5 | R | 3 | 0 | F | U | 2 | 93 | 0 | N.A. | | | | C | 1 | 12 | 1 | N.A. | | | | T | U | 2 | 93 | 0 | 1 | P | 518 |
| 30 | JT 446107 | NPE | UE | 0 | F | SE | 0 | F | null | 1 | R | 3 | 1 | P | U | 4 | 907 | 0 | U | 4 | 920 | 0 | U | 1 | 2 | 1 | N | 3 | 6 | 2 | $null^2$ | U | 2 | 145 | 0 | 0 | P | 765 |
| 31 | JT 465700 | NPE | E | 0 | F | IOE | 0 | F | 0 | 0 | F | 2 | 0 | F | C | 8 | 354 | 0 | U | 8 | 752 | 0 | T | 1 | — | 0 | U | 2 | 579 | 6 | $0^1$ | T | 1 | — | 0 | 0 | F | 900 |
| 32 | JM 39599 | CME | IUA | 0 | F | IUA | 0 | F | null | 2 | P | 2 | 0 | P | T | 8 | — | 0 | T | 8 | — | 0 | U | 1 | 547 | 7 | U | 2 | 924 | 6 | $null^1$ | U | 1 | 547 | 7 | 2 | P | 579 |
| 33 | JM 51869 | IAE | IUA | 1 | F | IUA | 1 | F | null | 1 | R | 3 | 3 | F | T | 8 | — | 0 | T | 8 | — | 0 | T | 1 | — | 3 | U | 3 | 548 | 3 | $null^1$ | T | 1 | — | 3 | 3 | R | 588 |
| 34 | JM 53874 | NPE | UEE | 0 | R | UEE | 0 | R | null | 0 | R | 3 | 0 | P | U | 1 | 301 | 0 | U | 1 | 301 | 0 | T | 1 | — | 1 | U | 3 | 225 | 1 | $null^1$ | T | 1 | — | 1 | 0 | R | 745 |
| 35 | JM 55694 | NPE | MCP | 0 | P | N.A. | | | void | 1 | P | 1 | 1 | P | U | 3 | 302 | 0 | N.A. | | | | C | 1 | 182 | 0 | C | 1 | 182 | 0 | $void^2$ | T | 2 | — | 3 | 0 | R | 494 |
| 36 | GP 461 | NPE | E | 0 | F | IOE | 0 | F | null | 36 | P | 2 | 9 | P | C | 6 | 127 | 0 | C | 6 | 127 | 0 | T | 1 | — | 0 | T | 2 | — | 0 | $null^1$ | T | 1 | — | 0 | 36 | P | 725 |
| 37 | GP 465 | AIO | IE | 0 | F | N.A. | | | null | 2 | F | 3 | 0 | F | N.A. | | | | N.A. | | | | U | 1 | 1 | 0 | N.A. | | | | $0^6$ | T | 6 | — | 2 | 0 | F | 381 |
| 38 | GP 523 | NPE | IOE | 0 | F | IOE | 0 | F | 0 | 5 | F | 5 | 2 | F | C | 9 | 130 | 0 | C | 9 | 130 | 0 | C | 1 | 7 | 0 | U | 5 | 923 | 0 | $null^3$ | U | 2 | 993 | 3 | 2 | F | 433 |
| 39 | GP 577 | RE | IOE | 1 | F | IOE | 0 | F | null | 20 | P | 2 | 1 | P | C | 7 | 130 | 0 | C | 7 | 130 | 0 | C | 1 | 47 | 0 | C | 2 | 225 | 0 | $void^3$ | C | 3 | 487 | 0 | 1 | P | 361 |
| 40 | GP 607 | NPE | XNI | 1 | R | XNI | 1 | R | void | 0 | F | 1 | 0 | F | C | 2 | 190 | 4 | C | 2 | 190 | 4 | T | 1 | — | 5 | T | 1 | — | 5 | $void^1$ | T | 1 | — | 5 | 0 | F | 703 |
| 41 | GP 708 | NPE | MPX | 0 | F | MPX | 0 | F | void | 1 | P | 1 | 1 | P | C | 6 | 128 | 0 | C | 6 | 128 | 0 | C | 1 | 257 | 6 | C | 1 | 257 | 6 | $void^1$ | C | 1 | 257 | 6 | 1 | P | 502 |
| 42 | GP 817 | NPE | IE | 0 | F | N.A. | | | void | 0 | P | 1 | 0 | P | N.A. | | | | N.A. | | | | T | 1 | — | 2 | T | 1 | — | 2 | $void^1$ | T | 1 | — | 2 | 0 | P | 590 |
| 43 | GP 830 | NPE | E | 0 | F | IOE | 0 | F | void | 2 | P | 1 | 1 | P | N.A. | | | | T | 11 | — | 4 | C | 1 | 65 | 2 | C | 1 | 65 | 2 | $void^4$ | T | 4 | — | 5 | 0 | P | 417 |

[α] $t$ indicates the type of error handler. Specifically, $t$ is the target exception type for SBET and FTET, the return type for FER, the discarded stack frames for VOER, and a pair of the return type and the number of discarded stack frames for generic early return. ℂ indicates the number of the actual cascaded errors in the host JVM. ℝ indicates the recovery result. For recovery result, we use F for *failure*, R for *repair*, and P for *plausible recovery*. $(t, r, d, s, c)$ is the testing result of the JPF execution described in Section 4.3.1. For the stop scenario $r$, we use N for *No Error*, T for *Timeout*, U for *Uninterpretable Behavior*, and C for *Cascaded Error*.

type of the first error that is triggered by the bug. For each basic strategy, we list a tuple including the *type* of recovery solutions, the number of cascaded errors and the recovery result.

For Ares, we first list the testing result $(r, d, s, c)$ for each basic strategy. Note that SBET and VOER may fail first, *e.g.*, Bug 14. At last, we use a 7-tuple for the most promising error handler determined by the ranking, including the testing result $(t, r, d, s, c)$, the actual cascaded errors in the host JVM after applying the error handler, and the recovery result. As JPF has its own implementation of a small set of library classes, Ares may fail to apply all basic strategies in JPF (*e.g.*, Bug 42 and 43) and also result in false cascaded errors that disappear in the host JVM.

***Ares versus FER*** The major advantage of Ares over FER is the significantly reduced number of cascaded exceptions induced by the recovery handler. Table 5 shows the statistics

**Table 5: Statistics of Cascaded Exceptions of FER and Ares**

| | MIN | MAX | StdDev | Mean | Median |
|---|---|---|---|---|---|
| FER | 0 | 45 | 11.0 | 5.8 | 1 |
| Ares | 0 | 36 | 5.8 | 2.2 | 0 |

of the cascaded exceptions by the two strategies. We also conducted Wilcoxon Signed-Rank Test. The $p$ value for the 1-tail test is 0.0008, showing that the improvement is statistical significant. Note that the results in Table 5 are computed by excluding Bug 2, in which FER causes an infinite loop triggering infinite number of exceptions. This bug further demonstrates the advantage of Ares over basic strategies.

***Overhead*** Ares performs recovery on demand, and requires no instrumentation into programs. Thus it imposes no overhead on normal program executions. The overhead intro-

```
1  /* method processMatches */
2  List matches = new ArrayList();
3  if (isScopeVariable()){
4    String inputString=vars.get(getVariableName());
5    /* BUG: inputString may be null */
6  + if(inputString == null) {
7  +   log.warn("...");
8  +   return Collections.emptyList();
9  + }
10   matchStrings(..., matches, ..., inputString);
11 } else {...}
12 return matches;
```

**Figure 6: JMeter bug 55694 and its patch.**

```
1  try {
2    setFormatter((Formatter) cl.loadClass(
         formatterName).newInstance());
3  } catch (Exception e) {
4  - // Ignore
5  + // Ignore and fallback to defaults
6  + setFormatter(new SimpleFormatter());
7  }
```

**Figure 7: Tomcat bug 51403 and its patch.**

duced by Ares only occurs when an exception is thrown and the recovery is performed. As Table 4 shows, the recovery pausing time of all basic strategies is less than one millisecond; and that of Ares ranges from 249 to 1277 milliseconds, which is mainly used to bootstrap JPF.

### 5.1.2 *Analysis of Bug Samples*

In addition to the bugs discussed in previous sections, we further discuss three bugs in detail as follows.

***JMeter 55694*** As shown in Figure 6, `processMatches` calls `matchStrings` (on line 10) to find substrings that match `inputString` (on line 4), and saves the results in `matches`, which is allocated as an empty list (on line 1). `matchStrings` first uses `inputString` to create a matcher object. If the value of `inputString` is null, the constructor results in an NPE. FER and VOER return from the constructor abnormally and results in a cascaded error in `matchStrings`. Ares detects the cascaded error and returns from `matchStrings` directly. Finally, the execution returns from the `processMatches` on line 12 with an empty list. Thus, it has the same behavior as the patch except the log (on line 7).

***GanttProject 830*** This bug makes GanttProject not responsive to any of the bug reporter's requests. The patch to this bug is two-fold: It first fixes the broken logic to calculate the correct value for a date range and then adds a catch-and-ignore error handler surrounding buggy methods, as the calculation of date range is non-trivial and there may still be bugs.

Although Ares cannot calculate the correct date range, it lets users continue editing and saving their work by using early return. Once the edited file is opened with a fixed version, helpful warnings will lead users to manually refill the nullified broken date ranges. Thus, we claim that Ares produces a plausible recovery.

***Tomcat 51403*** This bug cannot be fixed by any default error handlers in this paper. As shown in Figure 7, the patch assigns a non-default value to the formatter. Currently, Ares cannot allocate an object instead of `null`. Using a new-allocated object requires to update all related references, which has been well studied in [21].

## 5.2 Evaluation on Android ART

We only evaluate Android bugs using the conservative strategy as described in Section 4.4.2. As shown in Table 6, Ares can recover 8 of 9. Two bugs are recovered by SBET and 6 are recovered by VOER. The only unrecoverable one incurs an "Application Not Responding" (ANR).

***Recovery with VOER*** VOER can discard up to 61 frames to recover from an error in this experiment. In Firefox bug 1136157, a `NullPointerException` occurs during recursively destroying a set of GUI widgets when closing a page. There are no reusable error handlers and no methods with void return type in the first 60 frames. VOER results in destroying the page, which is just the desired behavior.

**Table 6: Recovery of Android apps bugs**

| App | Bug ID | Failure | Type | Result |
|---|---|---|---|---|
| Swiftp | 22 | IAE | **IOE** | FTP 550 |
| AardDict | 68 | NPE | **32** | Ignored |
| MobileOrg | 192 | ISE | 2 | ANR |
| MobileOrg | 344 | NPE | **13** | Ignored |
| My Expense | 136 | NFE | **4** | Finished |
| OI Notepad | 3 | NPE | **2** | Ignored |
| Conversation | 839 | NPE | **IJE** | Ignored |
| Firefox | 1136157 | NPE | **61** | Finished |
| Firefox | 1114499 | NPE | **13** | Ignored |

***Recovery with SBET*** While VOER exhibits significant recovery ability, SBET can also successfully recover from two errors. Conversation bug 839 occurs when the app fails to parse the user id into a string by scanning a QR-Code. A later fetch of this id attempts to build a `Jid` from the empty string, which results in a `NullPointerException`. Ares converts this error into an `InvalidJidException` to recover the app from a crash. Although the user fails to load the id, we believe that it is better than the crash of the whole app.

Swiftp can be used for sharing files between different mobile phones. Swiftp bug 22 is caused by a misuse of an Android API, which is used for creating named temporary files. If the name of the temporary file has fewer than three characters, the API throws an `IllegalArgumentException`. This error makes the thread terminate abnormally. Ares converts this error into an `IOException`, which has an error handler that sends back an FTP 550 error together with a message indicating the fail of the rename. Obviously, this recovery is not a repair but better than no recovery, which results in a connection loss.

# 6. DISCUSSION

In this section, we generally discuss the correctness guarantee of Ares. Similar to most of the ARR techniques [12, 25, 22, 7, 27, 18], Ares does not guarantee the correctness of the recovered buggy software systems. However, the use of JPF as a sandbox execution environment makes Ares capable of eliminating a number of dangerous or infeasible recovery plans at the early stage (*e.g.*, avoid applying FER for Bug 2 in Table 4), and selecting the promising/optimal recovery plan to execute. This step greatly reduces the risk of the recovery plan's causing worse consequence than the buggy program's default behavior of handling the unanticipated bug. Besides, we plan to develop new strategies, particularly for bugs that currently Ares and FER cannot handle. By combining sandbox testing and new ranking algorithms, we would avoid using FER to recover bugs that it cannot handle.

In terms of persistent data safety, Ares's in-vivo testing phase forbids writing data to persistent storage. Any of such attempts will abort the recovery process and resume the execution in the host VM. For the writes to persistent storage in the host VM after a recovery handler is applied, we can leverage the monitoring technique (orthogonal to Ares) proposed by Long *et al.* in [18], that is, tracking whether data that are affected by the recovery process can flow to the persistent storage.

Compared to correctness-critical software (*e.g.*, databases, compilers), ARR techniques are suitable for recovering *interactive* programs that have the following property: These programs usually have multiple features, and the malfunction of one feature (although it can crash the entire program) has little or no impact on the operation of other features. In this case, if we can recover the program from this failure, other features can still run correctly in the current interaction or the following ones. For example, in the Tomcat bug in Figure 2, although the failure of a buggy third-party listener can lead Tomcat to malfunction by leaking private information (similar to crashing), it does not (and should not) affect the execution of other features. Our 52 real-world bugs cover web servers, GUI applications, and mobile apps, most of which have this property . The evaluation of Ares on them shows that Ares does not introduce worse consequences than program subjects' default error handlers.

# 7. RELATED WORK

This section surveys three lines of related work to Ares.

## 7.1 Automatic Runtime Recovery

Generally, the recovery ability of ARR techniques comes from either redundancy [24, 8] or just default behavior [25, 18]. The validation of the recovery can be either testing by re-execution [24, 8] or suppressing catastrophic operations [18, 20]. We mainly discuss these approaches in this paper. Other approaches [12, 22, 7, 27] that focus on specific error detection and recovery are not discussed in detail.

***Checkpoints and Re-execution*** Checkpoint has been extensively studied in software recovery [14, 4]. Lots of ARR techniques [24, 28, 8] piggyback checkpoints to facilitate validation of their recoveries. ASSURE [28] also reuses existing error handler. However, it requires profiling runs to collect reusable error handlers before deployment. Ares has no need to make checkpoints and can dynamically collect reusable error handlers. It adopts a lightweight testing infrastructure to rule out potential dangerous recovery.

***Memory Error Suppression*** Memory error suppression based approaches [25, 20, 18] continue the execution in a recovery mode until the recovery ends up. Failure-Oblivious Computing (FOC) [25] discards invalid memory *writes* (*e.g.*, out of bounds writes, null dereference) and manufactures default values for invalid memory *reads*. RCV [18] extends FOC with *recovery shepherding* to prevent manufactured values from ruining persistent data. However, the shepherding may remain for a long time if there are infected long-live variables. Besides, both FOC and RCV can only synthesize a single recovery and may result in infinite loops. APPEND [13] instruments programs to recover potential null dereferences by default or user-provided error handlers. NPEFix [10] uses 9 different strategies to handle NPE but cannot determine the final strategy and also incurs very high overhead. Ares are not limited in handling memory errors and requires no additional code, data and instrumentation. Thus, it incurs negligible overhead in the host JVM.

## 7.2 Improving Exception Handling

Exception handling is usually used in programmer participated error recovery [15] but in fact is not well treated by programmers [26, 5, 30]. Many approaches attempt to automate exception handling by either using specific model [6], predefined strategies [9] or default exception handling [11]. Ares can adaptively synthesize a number of error handlers and select the most promising one. Besides, we implement Ares on a modern production platform and evaluate it with real applications used in industry. Azim *et al.* [3] analyze the log of Android to detect errors and also handle them by synthesized error handlers that either ignore errors or reload GUI. Ares is built on the ART runtime and also supports error transformation.

## 7.3 Automatic Program Repair

Recently, a number of approaches on automatic program repair have been proposed [29, 16]. These approaches usually require costly computation to find a plausibly correct repair [17]. In contrast, ARR usually has a strict timing requirement and aims to seek results better than immediate failures rather than a repair. Ares can also synthesize many repairs. In fact, a certain number of repairs generated by existing work simply delete functionality [23, 19].

# 8. CONCLUSION

This paper has presented Ares, an automatic runtime recovery system implemented on top of the industry-strength Java HotSpot VM and the Android ART VM. Ares is lightweight (as it requires only minimal modifications to the runtime), and efficient (as it intercepts only the exception handling mechanism, incurring no overhead on normal program execution). Thus, it can be seamlessly integrated into production environments. Ares is also effective as demonstrated by our evaluation on 52 real-world bugs — it is able to successfully recover from 39 of the bugs. To ensure reproducibility, we have released all the source code and data used in this paper, and more details about our evaluation can be found at http://lab.artemisprojects.org/groups/ares.

# 9. ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their comments. This research was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), National Natural Science Foundation (Grant Nos. 61472177, 91318301, 61321491) of China, the Collaborative Innovation Center of Novel Software Technology and Industrialization, the United States National Science Foundation (NSF) Grants 1117603, 1319187, 1349528 and 1528133, and a Google Faculty Research Award. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

# 10. REFERENCES

[1] CVE-2013-2071. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2071, accessed: 2015-07-22.

[2] Mobile Apps: What Consumers Really Need and Want. https://info.dynatrace.com/APM_13_WP_Mobile_

App_Survey_Report_Registration.html, accessed: 2015-07-22.

[3] T. Azim, I. Neamtiu, and L. M. Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 2014 International Conference on Automated Software Engineering (ASE)*, pages 623–628, 2014.

[4] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent advances in checkpoint/recovery systems. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[5] B. Cabral and P. Marques. Exception handling: A field study in Java and .NET. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, pages 151–175, 2007.

[6] B. Cabral and P. Marques. A transactional model for automatic exception handling. *Computer Languages, Systems & Structures*, 37(1):43–61, Apr. 2011.

[7] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP)*, pages 609–633, 2011.

[8] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 782–791, 2013.

[9] H. Chang, L. Mariani, and M. Pezzè. Exception Handlers for Healing Component-based Systems. *ACM Trans. Softw. Eng. Methodol.*, 22(4):30:1–30:40, 2013.

[10] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus. Npefix: Automatic runtime repair of null pointer exceptions in java. *CoRR*, abs/1512.07423, 2015.

[11] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, 31(6):531–540, June 1982.

[12] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 78–95, 2003.

[13] K. Dobolyi and W. Weimer. Changing Java's semantics for handling null pointer exceptions. In *19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 47–56, 2008.

[14] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34(3):375–408, Sept. 2002.

[15] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[16] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 802–811, 2013.

[17] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 3–13, 2012.

[18] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 227–238, 2014.

[19] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.

[20] V. Nagarajan, D. Jeffrey, and R. Gupta. Self-recovery in server programs. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 49–58, 2009.

[21] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force-executing binary programs for security applications. In *Proceedings of the USENIX Security Symposium*, pages 829–844, 2014.

[22] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 87–102, 2009.

[23] Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36, 2015.

[24] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 235–248, 2005.

[25] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the Symposium on Opearting Systems Design & Implementation (OSDI)*, pages 21–21, 2004.

[26] P. Sacramento, B. Cabral, and P. Marques. Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions? In *Proceedings of the International Conference on Innovative Views of .NET Technologies*, 2006.

[27] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 277–287, 2012.

[28] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: Automatic software self-healing using rescue points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, 2009.

[29] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International*

*Conference on Software Engineering (ICSE)*, pages 364–374, 2009.

[30] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–265, 2014.