Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2007-054 November 20, 2007

ReCrash: Making Crashes Reproducible

Sunghun Kim , Shay Artzi, and Michael D. Ernst

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

ReCrash: Making Crashes Reproducible

Sunghun Kim Shay Artzi Michael D. Ernst
MIT Computer Science and Artificial Intelligence Laboratory
{hunkim, artzi, mernst}@csail.mit.edu

Abstract

It is difficult to fix a problem without being able to reproduce it. However, reproducing a problem is often difficult and time-consuming. This paper proposes a novel algorithm, ReCrash, that generates multiple unit tests that reproduce a given program crash. ReCrash dynamically tracks method calls during every execution of the target program. If the program crashes, ReCrash saves information about the relevant method calls and uses the saved information to create unit tests reproducing the crash.

We present reCrashJ, an implementation of ReCrash for Java. reCrashJ reproduced real crashes from javac, SVNKit, Eclipse JDT, and BST. reCrashJ is efficient, incurring 13%–64% performance overhead. If this overhead is unacceptable, then reCrashJ has another mode that has negligible overhead until a crash occurs and 0%-1.7% overhead until a second crash, at which point the test cases are generated.

1. Introduction

It is difficult to find and fix a problem, and to verify the solution, without the ability to reproduce it. As an example, consider bug #30280[1] from the Eclipse bug database (Figure 1).

A user found a crash and supplied a back-trace, but neither the developer nor the user could reproduce the problem. Two days after the bug report, the developer finally reproduced the problem; four minutes after reproducing the problem, the developer fixed it.

Our work aims to reduce the amount of time it takes a developer to reproduce a problem. Suppose that the user had been using ReCrash-enabled version of Eclipse. As soon as the Eclipse crash occurred, ReCrash would have created a set of unit tests (Figure 2), each of which reproduces the problem. The user could have sent these test cases with the initial bug report, eliminating the two-day delay for the developer to reproduce the problem.

Upon receiving the test cases, the developer could run them under a debugger to examine fields, step through execution, or otherwise investigate the cause of failure. (The readability of the test case is secondary to reproducibility; a test need not be readable, nor end-to-end, to be useful.) In this case, the developer would have

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=30280

Technical Report. MIT, Boston, USA.
Published online on November 19, 2007.
http://pag.csail.mit.edu/reCrash

.

2003-01-27 08:01	U:	I found crash (here is the back-trace)
2003-01-27 08:26	D:	Which build are you using?
		Do you have a test-case to reproduce?
2003-01-27 08:39	D:	Which JDK are you using?
2003-01-28 13:06	U:	I'm running eclipse 2.1, ...
		I was not able to reproduce the crash
2003-01-29 04:28	D:	Thanks for clarification ...
2003-01-29 04:33	D:	Reproduced
2003-01-29 04:37	D:	Fixed...

Figure 1: An excerpt of comments between the user (U) who reported Eclipse bug #30280 and the developer (D) who fixed it.

```
1  // Generated by reCrash
2  // Eclipse 2.1M4/JDK 1.4
3  //-- The original crash stack back trace
4  // for java.lang.NullPointerException:
5  //   org...QualifiedAllocationExpression.resolveType
6  //   org...Expression.resolve
7  //   ...
8  public class EclipseTest extends TestCase {
9   protected void setUp() throws Exception {
10   TraceReader.readTrace("eclipse.trace");
11  }
12
13  public void test_resolveType() throws Throwable {
14   TraceReader.setMethodTraceItem(0);
15   QualifiedAllocationExpression obj =
16   (QualifiedAllocationExpression)TraceReader.readObject(0);
17
18   // load arguments
19   // BlockScope a_1 = Scope-locals: java.lang.String;
20   BlockScope a_1 = (BlockScope)TraceReader.readObject(1);
21
22   // Method invocation
23   obj.resolveType(a_1);
24  }
25
26  public void test_resolve() throws Throwable {
27   TraceReader.setMethodTraceItem(1);
28   Expression obj = (Expression)TraceReader.readObject(0);
29
30   // load arguments
31   BlockScope a_1 = (BlockScope)TraceReader.readObject(1);
32
33   // Method invocation
34   obj.resolve(a_1);
35  }
36  ...
37 }
```

Figure 2: Two test cases generated by ReCrash to reproduce Eclipse bug #30280; see Figure 1. Line 19 shows the `toString` representation of the object being read from the serialized trace.

```
1 public class QualifiedAllocationExpression {
2 public TypeBinding resolveType(BlockScope scope) {
3   TypeBinding receiverType = null;
4   boolean hasError = false;
5   if (anonymousType == null) {
6     if ((enclosingInstanceType =
7           enclosingInstance.resolveType(scope)) == null){
8       hasError = true;
9   } else if (enclosingInstanceType.isArrayType()) {
10      ...
11      //hasError = true; *Missing and causing the error
12  } else if ((this.resolvedType =
13              receiverType = ...)) == null) {
14      hasError = true;
15  }
16  ...
17  // limit of fault-tolerance
18  if (hasError) return receiverType;
19  if (!receiverType.canBeInstantiated()) ...
20  ...
21 }
```

Figure 3: Buggy source code from the eclipse compiler causing bug #30280.

been led to the buggy code in Figure 3. The `NullPointerException` is thrown on line 19, but the problem is that the method did not return on line 18 because an earlier if statement in lines 9–11 failed to set the `receiverType` variable to a non-null value. The developer would fix this problem by adding `hasError=true` on line 11, and can use the same test to verify the solution.

Reproducing crashes (in Java, these result from un-handled exceptions) can be difficult for the following reasons.

Nondeterminism A problem that depends on timing (e.g., context switching), memory layout (e.g., hash codes), or random number generators will manifest itself only rarely. Reproducing the problem requires replacing the sources of nondeterminism with specific choices that reproduce previous behavior. As an example, consider the (somewhat contrived) RandomCrash program [30] of Figure 4.

Remote detection A problem that is discovered by someone other than the developer who will fix it may depend on user GUI actions, environment variables, the state of the file system, operating system behavior, and other explicit or implicit program inputs. Not all dependences may be apparent to the developer or the user; many users are not sophisticated enough to gather this information; some of the information may be confidential; and the effort of collecting it may be too burdensome for the user, the developer (during interactions with the user), or both.

Test case complexity Even once a problem can be reproduced deterministically, a simpler test case, such as a unit test, is often faster and easier to run and understand, than the execution that triggered the problem.

We propose an algorithm, ReCrash, that addresses these problems. ReCrash automatically converts a crashing program execution into a set of deterministic, self-contained unit tests. Each of the unit tests reproduces the problem from the original program execution.

ReCrash has two phases: monitoring and test case generation. A production version of the *monitoring phase* can have low enough overhead that the software vendor ships his programs with ReCrash support built in. (It is also easy for an end user to instrument a program with ReCrash.) During every execution of the target program, ReCrash monitors the arguments to each method call. When the program crashes, ReCrash records, for each method in the stack trace, the method signature and the arguments (in serialized form).

The *test generation* phase uses the stored signatures and arguments to create a unit test for each method in the stack back-trace. Reporting multiple test cases allows a developer to obtain more or less context. For example, in some cases it is enough to know that `null` can flow to a method, and more information would be extraneous. In other cases, it may be helpful to know why `null` was passed to a method, which may be clear from the calling context. When ReCrash is used remotely (not by the developer who is debugging a problem), the test generation phase may be performed either remotely or by the developer.

The ReCrash monitoring phase can be made even more efficient by reducing the amount of monitoring. For example, objects that are not changed by the code need not be monitored. As another example, in *second chance* mode the ReCrash monitoring phase does nothing but record stack back-traces when the program crashes. On subsequent runs, ReCrash monitors only the methods that were in the stack back-trace. If the same problem reappears it will be captured and reproduced.

reCrashJ is an implementation of the ReCrash algorithm for Java. In a case study of real applications, reCrashJ reproduced every crash to which we applied it, with little performance overhead. We examined crashes from from BST, Eclipse JDT, SVNKit, and javac/jsr308.

This paper makes the following contributions:

- The ReCrash algorithm efficiently captures and reproduces crashes by storing method arguments, and using them to generate unit test cases.
- Optimizations give the ReCrash technique low enough overhead to enable routine use, by monitoring only relevant parts of a program.
- reCrashJ is a practical implementation of ReCrash for Java.
- Case studies show that reCrashJ effectively and efficiently reproduces crashes for several real applications.

The remainder of this paper is organized as follows. Section 2 describes the ReCrash algorithm. Section 3 presents the reCrashJ implementation. Section 4 describes our experimental evaluation. Section 5 discusses some of ReCrash limitations. Section 6 surveys related work, and Section 7 concludes.

2. ReCrash Technique

This section provides a high-level description of the ReCrash technique. Section 3 gives implementation details of the reCrashJ system.

ReCrash is based on the idea that it is not necessary to replay the entire execution in order to reproduce a specific crash. For many crashes, it is possible to create a useful test suite with *only* the information available on entry to the methods on the stack at the time of the crash.

The ReCrash technique has two parts. Monitoring is done during program execution (Section 2.1), and test creation is done after the program crashes (Section 2.2). Section 2.3 discusses optimizations that can improve performance during the monitoring phase.

2.1 Monitoring phase

The monitoring phase of ReCrash keeps track of all the methods and arguments that are on the call stack at any given moment. On entry to method m with arguments *args*, ReCrash generates a

```
class RandomCrash {
  public String hexAbs(int x) {
    String result = null;
    if (x > 0)
      result = Integer.toHexString(x);
    else if (x < 0)
      result = Integer.toHexString(-x);
    return toUpperCase(result);
  }

  public String toUpperCase(String s) {
    return s.toUpperCase();
  }

  public static void main(String args) {
    RandomCrash rCrash = new RandomCrash();
    rCrash.hexAbs(random.nextInt());
  }
}
```

Figure 4: A nondeterministic crash that depends on a random number generator. Taken from [30].

unique id *id* for the invocation of *m*, then pushes *m*, *id*, and *args* onto the ReCrash stack. ReCrash can make a deep copy of each argument, store only a reference, or use hybrid strategies; see Section 2.3. If *m* exits cleanly, ReCrash removes method invocations from the top of its stack until, and including, *id*; this removes methods whose exceptional exits were handled by *m*.

If an un-handled exception is thrown by `main`, ReCrash stores the current ReCrash stack containing all the methods and arguments. These are exactly the methods on the stack backtrace at the time of the failure. ReCrash also stores the ReCrash stack if the program reaches any other point indicating failure, as designated by the vendor of the program; an example would be a general `catch` clause or a failure of a consistency checking routine, even if it does not terminate the program.

2.2 Test creation phase

The test creation phase of ReCrash attempts to create a unit test for each method invocation *m*, *id*, *args* in the ReCrash stack. The generated test uses the trace to restore the state of the arguments that were passed to *m* in execution *id*, and then invokes *m* the same way it was invoked in the original execution. Only tests that expose the original crash are retained in the test suite that ReCrash outputs.

Figure 2 shows two tests created for the eclipse compiler when it crashes on line 19 of Figure 3. The first test, `test_resolveType`, was created for the method that was on the top of the stack when the exception was thrown. The second test, `test_resolve`, was created for the second method on the stack.

Creating a test for each method invocation on the stack is useful because it is possible that some tests reproduce a failure, but would not help the developer understand, fix, or check her solution. For example, the program in Figure 4 [30] will crash with a null pointer exception in the `toUpperCase` method, when the parameter `x` to the method `hexAbs` is `0`. The tests reCrashJ creates for this program are shown in Figure 5. The first test is useless in detecting and solving the problem, because the developer is unable to to understand the source of the null parameter. This test would also continue to fail even if the problem is solved. On the other hand, the second test captures a value not handled correctly by the method `hexAbs`. This test is useful in determining and and testing the solution.

Section 4 discusses the tradeoffs between the performance overhead and the ability to reproduce failures for different alternatives.

2.3 Optimizations

```
public void test_toUpperCase() {
  StackDataReader.setMethodStackItem(2);
  RandomCrash thisObject
    = (RandomCrash) StackDataReader.readObject(0);
  thisObject.toUpperCase(null);
}

public void test_hexAbs() {
  StackDataReader.setMethodStackItem(1);
  RandomCrash thisObject
    = (RandomCrash) StackDataReader.readObject(0);
  thisObject.hexAbs(0);
}
```

Figure 5: Tests created by ReCrash for the program of Figure 4.

ReCrash's time and space overhead is mostly determined by the cost of copying method arguments during the monitoring phase (Section 2.1). We have not highly optimized our implementation, but we have considered two general ways to reduce overhead: monitoring less information about each argument, or monitoring information about fewer methods.

2.3.1 *Monitoring Partial Arguments*

Here are the several alternatives for monitoring arguments:

reference Copying the reference has the smallest possible performance overhead. However, if between the method entry and the point of the failure, the argument state (all fields of recursively reachable objects) is modified, then when a failure is detected, ReCrash will be storing the modified arguments. The failure might not be reproducible because the execution might not be driven to the same failure.

shallow Performing a shallow copy is more expensive than copying the reference, but is resilient to direct side effects on the argument (reassignment of fields of the parameter), so ReCrash will be able to reproduce more failures. ReCrash will still be unable to reproduce failures that depend on deeper objects — fields that do not get copied and happen to be modified between the method entry and the point of failure.

used fields Shallow copy can be extended by copying not just the argument, but all of its fields that are used (read or written) in the method. These are the fields the method is most likely to depend on, and so copying them is mostly likely to be advantageous in increasing the reproducibility of tests. ReCrash uses static pointer analysis to determine the set of field actually used in the method for each argument.

depth *i* The above approaches can copy an argument to a specified depth: all the state reachable with *i* or fewer dereferences, possibly augmented by deeper copying of used fields.

copy Making a deep copy is the most expensive, but gives ReCrash the best chance of reproducing the same method execution.

It is possible to apply different alternatives to different arguments—for instance, shallow copying the receiver and reference copying of all other parameters.

As an optimization, ReCrash can avoid copying immutable parameters or fields. A method's parameter *p* is immutable, if the method never changes the state reachable from *p*. Parameter immutability information can be found statically [25, 27] or by a combination of static and dynamic analysis [8]. If a parameter is immutable, ReCrash can always copy the reference since the state of

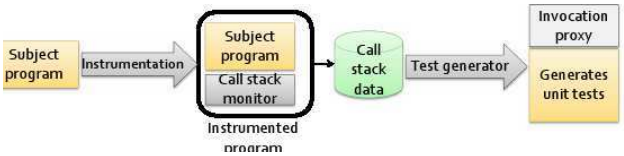

Figure 6: reCrashJ architecture overview.

the argument at the method entry will be the same as at the time of the failure.

2.3.2 *Ignoring Methods*

ReCrash need not monitor methods that are unlikely to expose or reveal problems. Those include empty methods, non-public methods, and simple getters and setters (which our implementation defines as those with less than seven byte code instructions).

ReCrash has one more optimization mode, *second chance*. Even a failure that cannot be reproduced at will is likely to appear more than once. In second chance mode, ReCrash initially monitors no method calls. Each time a failure occurs, ReCrash enables method argument monitoring for all methods found on the stack backtrace at the time of the failure. This mode is efficient, but requires a failure to be repeated twice (possibly with other failures in between) before ReCrash can reproduce it.

3. Implementation

We have implemented the ReCrash algorithm for Java. This section describes the implementation, reCrashJ.

Figure 6 shows the reCrashJ architecture. First, reCrashJ instruments an existing program (in Java class file format, using the ASM instrumentation tool [22]) to dynamically monitor method invocations on the stack. Figure 7 shows an instrumented Java program. The instrumented program (IP) can be deployed instead of the original program.

The IP keeps track of the arguments to every method on the stack. When the program crashes, IP stores all the relevant stack call data. From the stored data, reCrashJ generates multiple test cases that reproduce the original crash. Figures 2 and 5 show examples of generated test cases.

If a test case uses a non-public class, reCrashJ uses a method invocation proxy that uses reflection in order to use non-public classes. The following sections describe each module in detail.

3.1 Stack Monitor

The reCrashJ stack monitor in the instrumented program performs two functions: call stack data *capturing*, and *storing* call stack data to a file when a crash happens.

ReCrash captures method information at the beginning of each method as shown in line 3 of Figure 7. The stack monitor saves the receiver object and each of the other arguments on a per-thread stack (in memory) using one of the techniques described in Section 2.3.1 (lines 4–5).

If there is a crash, the stored method call information in the ReCrash stack will be serialized and written to a file (line 34). If the method successfully returns without a crash, the method information will be removed from the ReCrash stack (line 13).

We used the XStream framework [6] rather than Java serialization. Java serialization is limited to classes implementing the `java.io.Serializable` interface, and in which all fields are typically also from Serializable types (to avoid run time errors). XStream

```
1 class RandomCrash {
2   public String hexAbs(int x) {
3     int _id = Monitor.youMayCrash("hexAbs");
4     Monitor.addObject(this, "RandomCrash");
5     Monitor.addArgument(x, "int");
6     String result = null;
7     if (x > 0)
8        result = Integer.toHexString(x);
9     else if (x < 0)
10       result = Integer.toHexString(-x);
11
12    String ret = toUpperCase(result);
13    Monitor.youAreOK(_id);
14    return ret;
15  }
16
17  public String toUpperCase(String s) {
18    int _id = Monitor.youMayCrash("toUpperCase");
19    Monitor.addObject(this, "RandomCrash");
20    Monitor.addArgument(s, "String");
21
22    String ret = s.toUpperCase();
23    Monitor.youAreOK(_id);
24    return ret;
25  }
26
27  public static void __original__main__(String args[])
28    RandomCrash rCrash = new RandomCrash();
29    rCrash.hexAbs(random.nextInt());
30  }
31
32  public static void main(String args[]) {
33    try  __original__main__(args);
34      catch (Throwable e)  StackDataWriter.writeStackData(e);
35  }
36 }
```

Figure 7: The instrumented random crash program of Figure 4. Instrumented lines are bold.

does not have this limitation and can serialize any instance of any class.

Both the Java Platform Debugger Architecture (JPDA) [2] and the Java Virtual Machine Tool Interface (JVMTI) [3] provide features to access Java objects in the stack with low overhead. However, in order to use those tools, ReCrash would have to be deployed with a separate program that communicates with either JVM or JPDA. This is why we have chosen to instrument a given subject program and turns it in to a ReCrash-enabled program which can then be deployed. No other program or configuration is needed in order to run the ReCrash-enabled version. Pre-instrumentation also permits the use of static analysis to determine which object can be referenced or copied. Doing this analysis dynamically could be prohibitively expensive.

3.2 Instrumentation

By default, reCrashJ captures failures that result in an exception thrown by the main method. reCrashJ replaces the original main and adds a new main as shown in lines 27 and 32 in Figure 7. The new main invokes the original main in a try/catch block and handles exceptions.

A vendor may wish to reproduce other violations, for example exceptions that are caught by an exception handler or violations that do not result in an exception. In this case, the vendor can add a call to `writeStackData` wherever the program becomes aware of a violation–for example, in a catch-all handler or at a sanity check.

Adding such calls is easy. One of the authors who had never seen the source code found the appropriate point at which to add the `writeStackData` method for SVNKit and javac/jsr308 in 10 minutes.

When instrumenting a subject program, developers can embed

an identifier, such as a version number, in the subject program. This identifier will appear in the generated test cases as shown in line 2 of Figure 2. This identifier can help the developers to identify which version of their program crashed.

3.3 Test Generator

Using the method call data stored when the program crashes, reCrashJ generates a JUnit test suite with multiple JUnit tests, one for each of the methods on the stack at the time of the failure. Figure 5 shows a generated test case for the program in Figure 4.

Each test loads the receiver and the other arguments from the serialized representation of the stack back-trace. Then the test invokes the corresponding method on the receiver using the arguments.

4. Experimental Study

We experimentally evaluated the usefulness of the reCrashJ tool, by performing experiments with sample crashes in real applications. We designed the experiments around the following research questions:

Q1 How reliably can reCrashJ reproduce crashes?

Q2 Are the tests generated by reCrashJ useful for debugging?

Q3 What is the overhead (time and memory) of running reCrashJ?

Q4 What is the size of the recorded method call data?

Our results indicate that reCrashJ can reproduce many crashes, that it generates useful tests, that it, incurs low overhead, and that the size of the stored data is small. We analyzed three real bugs in detail and show that the tests generated by reCrashJ help debugging the source of the problem. We also asked the original program developers about the usefulness of tests generated by reCrashJ and the developers found the generated test cases helpful in debugging. Overall, the experimental results indicate reCrashJ is effective, scalable and useful.

4.1 Subject Systems

Our experiments used the following subject programs:

- **BST** (0.2 kLOC, 10 methods)[2] is a toy subject program used by Csallner in evaluating CnC [11, 12]. In our experiments, we used three BST crashes found by CnC.

- **SVNKit 0.8** (22 kLOC, 2,819 methods)[3] is a Java implementation of a Subversion[4] client. We found three crash examples by studying SVNKit bug reports #87, #188, and a new bug we found by using the application.

  Eclipse Compiler 2.1 (83 kLOC, 4,700 methods)[5] is a Java compiler included with the Eclipse IDE.

- **javac/jsr308 0.1.0** (86 kLOC, 5,017 methods)[6] is the OpenJDK Java compiler, extended with the reference implementation of JSR308("Annotations on Java Types") [16]. The compiler developers provided us with four crashes.

[2] `http://www.cc.gatech.edu/cnc/index.html`
[3] `http://sourceforge.net/projects/tinysql`
[4] `http://subversion.tigris.org`
[5] `http://www.eclipse.org`
[6] `http://groups.csail.mit.edu/pag/jsr308/`

```
1 public class SVNCommandLine {
2   [...]
3   private List myURLs;
4
5   public String getURL(int index) {
6     return (String) myURLs.get(index);
7   }
8
9   protected void init(String[] arguments)
10     throws SVNException {
11     [...]
12     myURLs = new ArrayList();
13     [...]
14     for (int i = 0; i < arguments.length; i++)
15       [...]
16
17  }
18 }
```

Figure 9: A code snapshot that illustrates a SVNKit crash (s1).

4.2 Reproducibility

Q1 How reliably can reCrashJ reproduce crashes?

To measure the effectiveness of reCrashJ in reproducing crashes, we used the following experimental procedure. We run the reCrashJ-instrumented versions of the subject programs on inputs that made the subject programs crash. When the instrumented version of program crashes, reCrashJ generates multiple test cases—one per stack frame at the moment of the crash. reCrashJ runs each generated test case and outputs those that reproduce the original crash by terminating with the same exception thrown from the same location as the original crash. We verified that the reCrashJ did reproduce the crash, then counted their number. We repeated this process for three of the argument copying strategies introduced in Section 2.3.1, and with and without the second chance mode of Section 2.3.2.

The results of the experiment (Figure 8) are promising—reCrashJ was able to reproduce the crash in all cases. For some crashes (b1, b2, b3, s1, s2, and s3), every internally-generated test case reproduces the crash. For some other crashes (e1, j1, j2, j3, and j4), only a subset of the generated test cases reproduce the crash.

In most cases, simply monitoring references is enough to reproduce crashes ('reference' column). However, this is not enough if the object is side-effected, between the method entry and the point of the failure in the method, in such a way that will prevent the failure if the object had the new state on method entry. In those cases (e.g., e1), using uses-fields (Section 2.3), is necessary to reproduce the crash.

4.3 Usability Study

Q2 Are the tests generated by reCrashJ useful for debugging?

To learn whether reCrashJ 's output helps developers to find errors, we analyzed a crash in SVNKit, and crash in the eclipse compiler, and we asked the JSR308 developers to analyze a crash in Javac/JSR308.

SVNKit bug (s1): SVNKit bug, #87 in the SVNKit database, causes the program to throw an index-out-of-bounds exception if a user omits the URL from the checkout command. Figure 9 shows the source code responsible for this bug: When no arguments are supplied, the `myURLs` list remains empty. The crash occurs when the `getURL` method (line 5) attempts to access a non-existent array element.

Figure 10 shows a test case that reCrashJ generates for this crash. The `test_1` method calls `getURL` with argument 0, which causes

program	crash name	crash type	# of generated tests	# of reproducible tests			*method call data size
				reference	used fields	copy	(zipped in KB)
BST	b1	class cast exception	3	3	3	3	5
	b2	class cast exception	3	3	3	3	5
	b3	unsupported encoding	3	3	3	3	25
Eclipse JDT	e1	null pointer exception	13	0	1	8	62
SVNKit	s1	index out of bounds	3	3	3	3	36
	s2	null pointer exception	2	2	2	2	34
	s3	null pointer exception	2	2	2	2	33
javac/jsr308	j1	null pointer exception	17	5	5	5	374
	j2	illegal argument exception	23	11	11	11	448
	j3	null pointer exception	8	1	1	1	435
	j4	index out of bounds	28	11	11	11	431

Figure 8: Subject programs and crashes used for our experimental study. For each crash, reCrashJ generates multiple test cases that aim to reproduce the original crash. The size of stack data file for each crash is noted. *The stack data file size is from used-fields mode.

```
1
2 public void test_getURL() {
3   SVNCommandLine thisObject =
4         (SVNCommandLine) StackDataReader.readObject(0);
5
6   // load arguments
7   int arg_1 = 0;
8
9   // Method invocation
10  thisObject.getURL(arg_1);
11 }
```

Figure 10: A generated test case to reproduce the bug in Figure 9.

```
1 public Void visitMethodInvocation
2      (MethodInvocationTree node, Void p) {
3   [...]
4   List<AnnotatedClassType> parameters
5     = method.getAnnotatedParameterTypes();
6
7   [...]
8   List<AnnotatedClassType> arguments =
9             new LinkedList<AnnotatedClassType>();
10  for (ExpressionTree arg : node.getArguments())
11      arguments.add(factory.getClass(arg));
12
13  for (int i = 0; i < arguments.size(); i++) {
14    if (!checker.isSubtype(arguments.get(i),
15                           parameters.get(i)))
16      [...]
17  }
18  [...]
19 }
```

Figure 11: A code snapshot that illustrates a javac/jsr308 crash (j4).

the crash. Running the generated test case reveals that `myURLs` is not in the correct state at the time of the call to `getURL`. This alone, however, does not explain *why* `myURLs` is in an incorrect state. Here's where the multiple test cases created by reCrashJ are useful—the test case generated for the next level (i.e., 2 call away in the stack) contains a call to the `run` method. That method calls `init`—which results in `myURLs` being incorrectly initialized. This second-level test case clearly exposes the bug. At this point, fixing the bug is trivial.

javac/jsr308 bug (j4): Due to a bug, using javac/jsr308 to compile source code with an annotation with multiple arguments, results in an index-out-of-bounds exception. Figure 11 shows the erroneous source code. The problem is that, during parsing, the compiler assumes that the `parameters` and `arguments` lists are of the same size (line 15), while, in fact, they may not be.

```
1 public void test_2() throws Throwable {
2   // load the SubtypeVisitor instance
3   SubtypeVisitor thisObject  =
4       (SubtypeVisitor) StackDataReader.readObject(0);
5
6   // load arguments
7   // MethodInvocationTree arg_1 =
8   //  test("foo", "bar", "baz");
9   MethodInvocationTree arg_1 =
10      (MethodInvocationTree)StackDataReader.readObject(1);
11  Void arg_2 = null;
12
13  // Method invocation
14  thisObject.visitMethodInvocation(arg_1, arg_2);
15 }
```

Figure 12: Test case generated for a javac/jsr308 crash (j4).

reCrashJ generates multiple test cases that reproduce the crash; one is shown in Figure 12. test_2 loads the receiver from serialized data (line 3). Then, the test loads the argument (line 9). To facilitate debugging, reCrashJ writes the arguments' string representation as a comment (line 8). Finally, the test calls `visitMethodInvocation` (line 14), which reproduces the original crash. By running this test case, a developer of javac/jsr308 can identify the cause of the crash.

Note that the generated test does not require the whole source code and encodes only the necessary minimum to reproduce the crash. This makes reCrashJ especially useful in scenarios where the compiler crash happens in the field, and the user cannot provide the developers with the whole source code to reproduce the crash.

Developer's testimonial We gave the generated tests cases to two javac/jsr308 developers and asked for comments about the tests' usefulness. We received positive responses from both developers.

Developer 1: "I often have to climb back up through a stack trace when debugging. ReCrash seems to generate a test method for multiple levels of the stack, which would make it useful"

"I like the fact that you wouldn't have to wait for the crash to occur again is useful."

Developer 2: "One of the challenging things for me in debugging is that when I set a breakpoint, the break point maybe be executed multiple times before the actual instance where the error is cased, [...] Using ReCrash, I was able to jump (almost directly) to the necessary breakpoint."

4.4 Performance Overhead

Q3 What is the runtime overhead (time and memory) of reCrashJ?

task	Execution Time				
	original program	**reference**	**used fields**	**used fields(Imm)**	**deep copy**
SVNKit checkout	1.17	1.62 (38%)	1.75 (50%)	1.75 (50%)	1657 (142,000%)
SVNKit update	0.556	0.617 (11%)	0.632 (13%)	0.628 (13%)	657 (118,000%)
Eclipse Content	0.954	1.08 (13%)	1.1 (15%)	1.1(15%)	114 (12,000%)
Eclipse String	1.07	1.36 (27%)	1.41 (32%)	1.39 (31%)	1656 (155,000%)
Eclipse Channel	1.27	1.7 (34%)	1.77 (40%)	1.74 (37%)	8101 (638,000%)
Eclipse JLex	3.45	4.9 (42%)	5.63 (64%)	5.51 (60%)	> 2 days
task	Second Chance-Execution Time				
	original program	**reference**	**used fields**	**used fields(Imm)**	**deep copy**
SVNKit checkout	1.17	1.17 (0%)	1.17 (0%)	1.18 (0.8%)	1.42 (21%)
SVNKit update	0.556	0.556 (0%)	0.56 (0%)	0.558 (0.3%)	0.56 (0.8%)
Eclipse Content	0.954	0.969(1.5%)	0.97 (1.4%)	0.963 (0.9%)	3.98 (317%)
Eclipse String	1.07	1.09 (1.7%)	1.08 (0.8%)	1.08 (1%)	8.99 (742%)
Eclipse Channel	1.27	1.27 (0.1%)	1.27 (0%)	1.27 (0%)	16.6 (1,210%)
Eclipse JLex	3.45	3.47 (0.7%)	3.47 (0.9%)	3.48 (1.1%)	1637 (47,000%)

Figure 13: Execution times of the original and instrumented programs. Slowdowns from the baseline appear in parentheses. The columns are described in Section 2.3.1. (IMM) stands for the immutability optimization.

reCrashJ will be most useful if is runtime overhead is minimal. We measured the user time and memory usage while performing the following tasks using the original and instrumented versions of the subject programs.

- **SVNKit checkout.** We checked out files from a SVN repository.
- **SVNKit update** We performed SVNKit update of the checked out source code.
- **Eclipse Content** We compiled `Content.java` (48 LOC) from JDK 1.7 `samples/nio/server`.
- **Eclipse String** We compiled `StringContent.java` (99 LOC) from JDK 1.7 `samples/nio/server`.
- **Eclipse ChannelIOSecure** We compiled ChannelIOSecure.java (642 LOC) from JDK 1.7 `samples/nio/server`.
- **Eclipse JLex** We compiled JLex (7,841 LOC)[7].

Figure 13 compares user-mode time of the original and instrumented programs, measured using the UNIX `time` command. Because of variability in network, disk, and CPU usage, these is some noise in the measurements (occasionally an optimization slightly slowed down the subject program), but the trend is clear.

Even storing only references is surprisingly expensive: 11%–42% run-time overhead. Together with immutability analysis to avoid storing unchanged data, copying "one and a half" levels of data structure (the first level plus any fields on the second level that are used in the method) has very little additional overhead: a total of 13%–60%. These values are higher than we would prefer, but are probably usable for truly critical bugs.

Our (unoptimized, serializing) deep copy version is completely unusable, except possibly in second chance mode, where it might be usable for in-house testing. A better implementation could be more efficient, but would probably still be impractical.

Second chance mode, however, is where our system shines. It reduces overheads to the barely noticeable 0%–1.7% — and that is *after* a crash has already been observed, before which the overhead is negligible (essentially 0%). Second chance mode obtains these benefits by monitoring only a very small subset of all the methods in the program. The observation that this simple idea is sufficient, effective, and efficient is one of the main contributions of our work.

[7] `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

task	Memory usage (MB)	
	original program	**used-fields**
SVNKit checkout	1.8	2.3
Eclipse JLex	4.6	4.8

Figure 14: Memory use overhead of the instrumented programs.

For the memory usage comparison, we measured the maximum memory use using JProfiler [8]. We performed the SVNKit checkout and Eclipse JLex tasks using the original and instrumented programs. Figure 14 shows the memory usage—reCrashJ adds only 0.2–0.5M memory overhead.

4.5 Recorded Data Size

Q4 What is the size of the recorded method call data?

If ReCrash is deployed in the field, and a crash happens, ReCrash sends the stored method call data back to the program developers. For each crash, Figure 8 shows the sizes of the stored method call data. The sizes are small—around 35KB for SVNKit, and around 450KB for javac/jsr308. Data this small can easily be sent to the developers.

5. Discussion

This section discusses threats to validity and a limitation of our technique.

5.1 Threats to Validity

We identify the following key threats to validity.

Systems and crashes examined might not be representative. We might have a system selection bias. We examined only 11 crashes from four subject systems. For each crash we improved our system until it could reproduce that crash. It is time-consuming to find a real bug (by studying bug reports), download an older version of the software, compile it, and reproduce the bug. We may have accidentally chosen systems or crashes that have better (or worse) than average ReCrash reproducibility. For example, ReCrash may not be effective for crashes that depend on data in static fields.

Monitors only Runtime Exceptions. Our experiments consider any violations that manifest as as runtime exceptions such as null

[8] `http://www.ej-technologies.com/products/jprofiler/overview.html`

pointer or index out of bounds exceptions. Future experiments should consider other types of violations.

5.2 Privacy

ReCrash stores the serialized stack data in XML files. An XML file can be read by numerous applications and thus our serialized data might even be useful for future tasks of program understanding. On the other hand, one of the problem we set to solve is debugging deployed application when the client is unable to send his data to the developer in fear of exposing proprietary data. For instance, a user who discovers a bug in eclipse might be unable to send proprietary source files to the developer.

Sending ReCrash stack data to the developer should be seen as an improvement to existing solutions, since it only contains parts of processed data that were accessible to the arguments on the time of the crash. However, this is not the perfect solution as some proprietary information might still escape. As a possible partial solution ReCrash might provide a stack data reader, so that the client can review the encoded data and decide how much of it is safe to send it to the developers.

6. Related work

There are several topics related to our work. This section presents the most related, divided into categories.

6.1 Record and Replay

Many existing record and replay techniques for postmortem analysis and debugging [10, 14, 18, 9, 20, 17, 31, 15] are based on three components: checkpoints, log, and replay. The checkpoint provides a snapshot of the full state of the program at specific times, while the log records events between snapshots. The replay component uses the log to replay the events between checkpoints. By contrast, ReCrash performs a checkpoint at each method entry and has no log of events, only an in-memory record of stack elements. ReCrash do not replay the entire execution, instead ReCrash allows the developer to observe the system in several states before the failure, then run the original program until the failure is reproduced. ReCrash logging simplicity allow it to be deployed remotely without significant overhead to the clients. Most of the previous techniques are designed for in-house testing or debugging, and have unreasonable overhead for deployed applications.

BugNet [20], ReVirt [15], and FlashBack [28] requires changes to the host operating system while FDR [31] uses a proprietary hardware recorder. ReCrash, on the other hand, can be deployed in any environment, and its log file can be used to reproduce a recorded failure in different environments.

Choi et al. [9], Instant Replay [18], BugNet [21], and many others emphasize the ability to deterministically replay an execution of a non-deterministic program. They are able to reproduce race conditions and other non-deterministic failures. In order to achieve this goal, these techniques either impose a large space and time overhead [9, 18]; or they only allow replaying a narrow window of time [21]. Similar to BugNet [21], ReCrash only allow replaying a small part of the execution before the failure. ReCrash is only able to deterministically reproduce the a non-deterministic failure if one of the generated tests captures the state after the non-determinism.

jRapture [29], test factoring [26], and ADDA [10] capture the interactions between the program and the environment to create smaller test cases or enable reproducing failures. These techniques capture a trace, and then run the subject code in a special harness, such as a mock object representing all interactions with the rest of the system, that reads values out of the trace whenever the subject code would have interacted with its environment. Test factoring does this at the level of method calls. ADDA does it at the level of file operations and C standard library functions. By contrast, our approach does not record a trace; it sets up the system in a particular start state, and then the system runs unassisted.

6.2 Test Generation

Contract Driven Development [19] generates test cases using contracts (pre and post conditions) defined by developers. When run in debug mode, CDD captures debug information when a contract is violated. CDD uses the debug information to generate test cases. ReCrash similarly generates test cases from information saved when a failure was detected. However, CDD is designed to be used in the development process rather than in deployment. ReCrash instruments a target program and the target program is deployed in-field. The instrumented program monitors the stack and generates a test suite without the need for contracts or special IDE support.

CnC [12],JCrasher [11], Eclat [23], Randoop [24], and DSD-Crasher [13] are using random inputs to find program crash points. Their generated tests may not be representative of actual use, whereas ReCrash generates tests that reproduce real crashes. Palulu's [7] model-based test generation similarly attempts to generates tests based on values observed during an actual program execution.

6.3 Remote data collection

Crash report systems such as Dr. Watson [4], Apple's Crash Reporter [1], and talkback [5] send a stack back-trace or program core-dump to the vendor. The stack back-trace helps the vendor to correlate between problems. However, reproducing the original failure requires non-trivial human effort. The core-dump enables to examination of the program state at the time of the crash, it requires debugging tools, and deep knowledge of the target software. ReCrash automatically generates stack method data smaller than a core-dump that can be similarly sent to the vendor. The developer can use this data to create a test that reproduces the original failure.

7. Conclusion

We have introduced the ReCrash algorithm, which captures crashes and generates unit tests that reproduce them. ReCrash is simple to implement, it is scalable, and it generates simple, helpful test cases that effectively reproduce crashes. The ReCrash performance overhead is acceptable, and it creates method call data of manageable size. Our reCrashJ tool implements the algorithm for Java.

We have evaluated reCrashJ with real crashes from SVNKit, Eclipse JDT, and javac/jsr308. reCrashJ was able to reproduce all the crashes. The performance overhead of instrumented programs by reCrashJ has low user time overhead: 0%-1.7% for the second chance mode and 13%–64% for the **use field** mode. reCrashJ increase memory use by 0.2–0.5M. The size of stored stack method data is manageable (0.5k–448k). Considering the overhead and the stack data size, reCrashJ is usable in real software deployment.

The generated test cases are useful for debugging. We showed real bug examples and how the generated tests can locate the bugs. Two developers' testimonials indicate that the generated test cases help to find bug locations or reduce debugging work.

References

[1] Apple Crash Reporter, 2007. http://developer.apple.com/technotes/tn2004/tn2123.html.

[2] Java Platform Debugger Architecture, 2007. http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

[3] JVMTM Tool Interface (JVM TI), 2007. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html.

[4] Microsoft Online Crash Analysis, 2007. http://oca.microsoft.com.

[5] Talkback Reports, 2007. http://talkback-public.mozilla.org.

[6] XStream Project Homepage, 2007. http://xstream.codehaus.org/.

[7] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *M-TOOS 2006: 1st Workshop on Model-Based Testing and Object-Oriented Systems*, Portland, OR, USA, October 23, 2006.

[8] S. Artzi, A. Kieżun, D. Glasser, and M. D. Ernst. Combined static and dynamic mutability analysis. In *ASE 2007: Proceedings of the 22nd Annual International Conference on Automated Software Engineering*, Atlanta, GA, USA, November 7–9, 2007.

[9] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, 1998.

[10] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 261–270, Minneapolis, MN, USA, May 23–25, 2007.

[11] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, September 2004.

[12] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE'05, Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, St. Louis, MO, USA, May 18–20, 2005.

[13] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 245–254, Portland, ME, USA, July 18–20, 2006.

[14] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 66–71, 2006.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[16] M. D. Ernst and D. Coward. JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308/`, October 17, 2006.

[17] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX-ATC'06: Proceedings of the Annual Technical Conference on USENIX'06 Annual Technical Conference*, pages 27–27, Boston, MA, 2006.

[18] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[19] A. Leitner, I. Ciupa, and A. Fiva. Contract Driven Development = Test Driven Development – Writing Test Cases. In *Proc. of the 12th European Software Engineering Conference held jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 425–434, September 2007.

[20] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 284–295, 2005.

[21] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.

[22] ObjectWeb Consortium. ASM - Home Page, 2007. http://asm.objectweb.org/.

[23] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 23–25, 2007.

[25] A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM 2004, Proceedings of the International Conference on Software Maintenance*, pages 82–91, Chicago, Illinois, September 12–14, 2004.

[26] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, November 9–11, 2005.

[27] A. Sălcianu and M. C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI'05, Sixth International Conference on Verification, Model Checking and Abstract Interpretation*, pages 199–215, Paris, France, January 17–19, 2005.

[28] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 3–3, Boston, MA, 2004.

[29] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 158–167, 2000.

[30] A. Tomb, G. P. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA*, pages 97–107, 2007.

[31] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 122–135, 2003.