ReCrash: Making Software Failures Reproducible by Preserving Object States

Shay Artzi Sunghun Kim Michael D. Ernst

MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA, 02139, USA
{artzi, hunkim, mernst}@csail.mit.edu

Abstract. It is very hard to fix a software failure without being able to reproduce it. However, reproducing a failure is often difficult and time-consuming. This paper proposes a novel technique, ReCrash, that generates multiple unit tests that reproduce a given program failure. During every execution of the target program, ReCrash stores partial copies of method arguments in memory. If the program fails (e.g., crashes), ReCrash uses the saved information to create unit tests reproducing the failure.

We present ReCrashJ, an implementation of ReCrash for Java. ReCrashJ reproduced real crashes from Javac, SVNKit, Eclipsec, 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 the crash occurs for a second time, at which point the test cases are generated.

Key words: Fault, bug, crash, failure, object, reproducing, capture, replay, test generation

1 Introduction

It is difficult to find and fix a software problem, and to verify the solution, without the ability to reproduce it. As an example, consider bug #30280 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 a ReCrash-enabled version of Eclipse. As soon as the Eclipse crash occurred, ReCrash would have generated 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.) The developer can use the same test to verify the bug fix.

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.

Fig. 1. An excerpt of comments (`https://bugs.eclipse.org/bugs/show_bug.cgi?id=30280`) between the user (U) who reported Eclipse bug #30280 and the developer (D) who fixed it.

Reproducing failures (in Java, these often 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 5.

Remote detection A problem that is discovered by someone other than the developer who can fix it may depend on local information such as 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 if a problem can be reproduced deterministically, the exposing execution might be complex, and the buggy method might be called multiple times before the bug is triggered. A simpler test case, such as a unit test, is often faster and easier to run and understand than the execution that triggered the failure.

We propose a technique, ReCrash, that addresses these problems in many cases. ReCrash automatically converts a failing program execution into a set of deterministic and self-contained unit tests. Each of the unit tests reproduces the problem that caused the program to fail. ReCrash has two phases: monitoring and test case generation.

monitoring During an execution of the target program, ReCrash maintains a shadow stack with an n-deep copy of the receiver and arguments to each method. From depth n on, these objects refer to the original objects on the heap (see Section 2.2.1). Thus, ReCrash exploits the object-oriented nature of the program by using objects as the unit of granularity for including or excluding values from the stored shadow stack. When the program fails (i.e., crashes), ReCrash serializes the shadow stack contents, including all heap objects referred to from the shadow stack. Heap data that was not copied to the shadow stack might have been side-effected between the

```
1  // Generated by reCrash
2  // Eclipse 2.1M4/JDK 1.4
3  //-- The original crash stack back trace for java.lang.NullPointerException:
4  //   org...QualifiedAllocationExpression.resolveType
5  //   org...Expression.resolve
6  //   ...
7  public class EclipseTest extends TestCase {
8    protected void setUp() throws Exception {
9      ShadowStackReader.readTrace("eclipse.trace");
10   }
11
12   public void test_resolveType() throws Throwable {
13     ShadowStackReader.setMethodStackItem(0);
14
15     // load receiver
16     // rec = i.new Y()
17     QualifiedAllocationExpression rec =
18       (QualifiedAllocationExpression)ShadowStackReader.readReceiver(0);
19
20     // load arguments
21     // arg_1 = Scope-locals: java.lang.String;
22     BlockScope arg_1 = (BlockScope)ShadowStackReader.readArgument(1);
23
24     // Method invocation
25     rec.resolveType(arg_1);
26  }
27
28   public void test_resolve() throws Throwable {
29     ShadowStackReader.setMethodStackItem(1);
30
31     // load receiver
32     // rec = i.new Y()
33     Expression rec = (Expression)ShadowStackReader.readReceiver(0);
34
35     // load arguments
36     // arg_1 = Scope-locals: java.lang.String;
37     BlockScope arg_1 = (BlockScope)ShadowStackReader.readArgument(1);
38
39     // Method invocation
40     rec.resolve(arg_1);
41   }
42 }
```

Fig. 2. Two test cases (out of eight) generated by ReCrash to reproduce Eclipse bug #30280. Each test case comes with a serialized representation of method arguments in file `eclipse.trace` (line 9). Lines 16, 21, 32, 36 show the `toString` representation of the object being read from the serialized trace.

point of the call and the point of the failure; this is a cause of imprecision for the ReCrash technique.

test generation ReCrash generates candidate tests by calling each method that was on the shadow call stack with the de-serialized receiver and arguments from the shadow stack. ReCrash outputs all candidate tests that reproduce the original failure. ReCrash outputs multiple tests because a test that calls only the method at top of the call stack may not provide enough context to find the error. A test calling a method closer to the bottom of the stack provides more context, but is less likely to reproduce the original failure.

ReCrash is effective, despite recording only partial information about the program state in-memory. For many failures, it is possible to reproduce them with *only* some of the information available on entry to the methods on the stack at the time of the failure. This may be due to the following characteristics of object-oriented programs.

- Many bugs are dependent on small parts of the heap (also the premise of unit testing).
- Good object-oriented style encapsulates important state nearby.
- Good object-oriented style avoids excessive use of globals. Furthermore, ReCrash has access to and will store any parts of the global state or environment that are passed as method arguments.

ReCrash's monitoring phase can be made efficient by reducing the amount of monitoring. For example, objects that are not changed by the code need not be monitored. As another example, ReCrash has an optimization mode called *second chance* in which the monitoring phase initially records only stack back-traces when the program fails. On subsequent runs, ReCrash monitors only the methods that were in the stack back-trace of failures. If the same problem reappears, it will be captured and might be reproduced. Second chance allows a vendor to use ReCrash in a similar way to an anti-virus program which frequently updates the virus profile data. When one client discovers a failure and sends the produced back-trace to the vendor, the vendor will send an update containing a list of additional methods to monitor, to all other clients. When ReCrash is used remotely (not by the developer who is debugging a failure), the test generation phase may be performed either remotely or by the developer.

ReCrashJ is an implementation of the ReCrash technique for Java. In a case study of real applications, ReCrashJ reproduced failures from Javac-jsr308, SVNKit, Eclipse Java Compiler, and BST with little performance overhead.

This paper makes the following contributions:

- The ReCrash technique efficiently captures and reproduces failures using in-memory storage of method arguments.
- Optimizations give the ReCrash technique low enough overhead to enable routine use, by monitoring only relevant parts of a program.
- The second chance mode operates with almost no overhead.
- ReCrashJ is a practical implementation of ReCrash for Java.
- Case studies show that ReCrashJ effectively and efficiently reproduces failures for several real applications.

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

2 ReCrash Technique

The ReCrash technique has two parts. Monitoring is done during program execution (Section 2.1), and test generation is done after the program fails (Section 2.3). Section 2.2 discusses several optimizations to the monitoring phase.

2.1 Monitoring phase

In the monitoring phase, ReCrash maintains in memory a shadow version of the call stack with copies of the receiver and arguments to each method. Figure 3 presents pseudo-code for the monitoring phase.

On entry to method m with arguments $args$, ReCrash generates a unique id id for the invocation of m, then pushes $\langle id, m, rec_{copy}, args_{copy}\rangle$ onto the ReCrash shadow stack, where rec_{copy} contains copies of the method's receiver and $args_{copy}$ contains a copy of the method's arguments.

ReCrash can use different copy strategies. It can make a deep copy of each argument, store only a reference, or use hybrid strategies (see Section 2.2.1). If m exits normally, ReCrash removes the method call data $\langle id, m, rec_{copy}, args_{copy}\rangle$ from the top of its shadow stack.

If an un-handled exception is thrown out of `main`, ReCrash outputs the un-handled exception and a deep copy of the current ReCrash shadow stack. The methods that ReCrash stores include at least the methods on the virtual machine call stack at the time of the failure. In cases where a method catches an exception and then throws a different uncaught exception, the ReCrash shadow stack contains additional methods (those that were on the shadow stack when the original exception was thrown). This additional information can improve the reproducibility of failures.

2.2 Optimizations to the monitoring phase

ReCrash's time and space overhead is dominated by the cost of copying the method arguments at method entry. We have considered two orthogonal ways to reduce overhead: reducing the depth of copied state for the method's arguments, and monitoring fewer methods. Recording less information might reduce the chances of reproducing a failure. Section 4 discusses tradeoffs between the performance overhead and the ability to reproduce failures.

2.2.1 Depth of copying arguments In order to recreate the state of the method's arguments at the time of the failure, ReCrash copies the arguments to the shadow stack on the method entry. However, a deep copy of the shadow stack is only stored when an exception is not caught by the program. Thus, any part of an argument's state that is not copied into the shadow stack may change between the method entry and the time of the failure. This change might prevent ReCrash from reproducing the failure.

This section considers different strategies for copying arguments (including the receiver) at the method entry. In each strategy a different amount of the argument state is copied to the shadow stack and the rest of the argument state uses references to the

Input: `copy` - a function that copies arguments (see Section 2.2.1)
Output: `file` - containing deep copy of the shadow stack

s : shadow stack of the current method on the call stack and their arguments

On program start:
begin
  $s \leftarrow$ new empty stack
end

On entry to method m in the call `rec.m(args)`:
begin
  $rec_{copy} \leftarrow$ `copy`(rec)
  $args_{copy} \leftarrow$ `copy`(args)
  $id \leftarrow$ generate unique id for current execution of m
  push tuple $\langle id, m, rec_{copy}, args_{copy} \rangle$ into s
end

On non-exceptional exit from method m:
begin
  $id \leftarrow$ lookup id for current execution of m
  pop from s until $\langle id, \_, \_, \_ \rangle$ is popped
end

On top-level (main) uncaught exception e:
begin
  file $\leftarrow$ new output file
  store e to *file*
  foreach $\langle id, m, rec_{copy}, args_{copy} \rangle$ in s **do**
    store $\langle id$, m, `deepCopy`(rec_{copy}), `deepCopy`($args_{copy}$)$\rangle$ to *file*
  end
end

Fig. 3. Pseudo-code for the ReCrash technique. If an exception is uncaught by the program (main), ReCrash stores the exception and a deep copy of the current shadow stack to the output file. The technique uses two auxiliary functions: `deepCopy` which copies all the reachable state of an object, and `copy` which is a parameter to this technique. The exact semantics of `copy` depends on the chosen copy strategy (see Section 2.2).

original objects (that might get side-effected). The different copying strategies, in order of increasing overhead, are:

reference (depth-0) Copying only the reference to the argument.

shallow (depth-1) Copying the argument itself. Each of the fields in the copy contains whatever the original argument did, a primitive or a reference. Shallow copying is more resilient to direct side effects on the top-level primitive fields of the arguments.

depth-*i* Copying an argument to a specified depth: all the state reachable with i or fewer dereferences.

deep copy Copy the entire state. This strategy gives ReCrash the best chance of reproducing the same method execution, since it preserves the object state at the time of the method entry.

ReCrash has an additional copying option, **used-fields**, applicable to all copying strategies except reference. When the used-fields option is selected, ReCrash performs deeper copying on fields that are used (read or written) in the method. For example, suppose that ReCrash is using shallow (depth-1) copying with used-fields, `x` is an argument to a method `m`, and `x.a` is used in `m`. Then ReCrash will perform shadow copying on `x` and `x.a`. The used fields are the fields that the method is likely to depend on, and therefore copying them increases the chance of reproducing the failure.

It is possible to use different strategies for different arguments. For instance, shallow copying the receiver and reference copying of all other arguments. ReCrash always uses the reference strategy for immutable parameters. A method's parameter p is immutable if the method never changes the state reachable from p. Therefore, an object passed to an immutable parameter will have the same state at the method entry and at the time of the failure.

2.2.2 Monitoring fewer methods ReCrash need not monitor methods that are unlikely to expose or reveal problems, or cannot be used in the generated tests. Those include empty methods, non-public methods, and simple methods such as getters and setters.

second-chance mode It would be most efficient to monitor only methods that will fail. However, it is impossible to compute this set of methods in advance. One way of approximating this set is to create a set of methods that already failed, updating the set each time a new method fails.

This is the underlying idea behind *second chance*, a mode of operating ReCrash that only monitors methods that have failed at least 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 (real) stack back-trace 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. Second chance mode has no impact on the reproducibility of a failure (the second time the failure appears).

2.2.3 Which optimizations to use When using ReCrash, a developer needs to decide which optimizations to use. Which copying strategy should ReCrash use for the arguments? Should ReCrash use the used-fields option on the arguments? Should ReCrash use the second-chance mode?

The answers depend on the developer's needs and the subject program. For example, if the developer doesn't mind missing the first time a failure happens, and the failure occurs relatively often, the second chance mode is a good fit. If one wants ReCrash to reproduce all possible failures and can suffer the performance hit, then deep copy might be the right mode. We found that using the copying strategy shallow copying (depth-1) with used-fields enabled ReCrash to reproduce most failures with acceptable performance overhead.

Input: *inFile*- containing deep copy of the shadow stack
Output: *outFile*- containing tests reproducing the failure
begin
    e : exception, the original exception resulting in the failure
    testSuite : Collection of test sources, each test reproducing the original failure
    outFile : output file for the generated tests
    testSuite ← empty test suite
    e ← load exception from *inFile*
    len ← get length of shadow stack from *inFile*
    for *i* =1 *to len* **do**
        t : test source
        t ← generateTest(*inFile*, *i*)
        if execution of *t* ends with *e* **then**
            add *t* to *testSuite*
        end
    end
    store *testSuite* into *outFile*
end

generateTest(File *file*, int *i*)
begin
    $\langle id, m, rec_{copy}, args_{copy} \rangle$ ← pop i^{th} tuple from *file*
    output("test *m.id*")
    output("rec = load i^{th} receiver from *file*")
    output("args = load i^{th} arguments from *file*")
    output("rec.m(args)");
end

Fig. 4. Pseudo-code for ReCrash test generation. ReCrash generates a test for every method on the original stack at the time of the failure, using the saved copy of the arguments. ReCrash outputs each test whose executions results in the same exception. The `generateTest` subroutine creates the source for a test. Each test loads the appropriate receiver and arguments from the file containing the shadow stack, and then calls the method on the receiver with the arguments.

2.3 Test generation phase

The test generation phase of ReCrash attempts to generate a unit test for each method invocation $\langle id, m, rec_{copy}, args_{copy} \rangle$ in the ReCrash shadow stack. The pseudo-code for the test generation phase is presented in Figure 4.

ReCrash generates a test for each of the method frames in the shadow stack (s). ReCrash restores the state of the arguments (modulo the shadow stack) 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 end with the same exception as the original failure are saved. Storing more than one test that ends with the same failure is useful because it is possible that some tests reproduce a failure, but would not help the developer understand, fix, or check her solution. See Section 3.2 for an example of such a case.

```
1 class RandomCrash {
2   public String hexAbs(int x) {
3     String result = null;
4     if (x > 0)
5       result = Integer.toHexString(x);
6     else if (x < 0)
7       result = Integer.toHexString(-x);
8     return toUpperCase(result);
9   }
10
11   public String toUpperCase(String s) {
12     return s.toUpperCase();
13   }
14
15   public static void main(String args[]) {
16     RandomCrash rCrash = new RandomCrash();
17     rCrash.hexAbs(random.nextInt());
18   }
19 }
```

Fig. 5. This program, taken from [30], will crash with a null pointer exception in the `toUpperCase` method, when the argument `x` to the method `hexAbs` is `0`. Since the value of `x` is randomly chosen (line 17), this crash is not deterministically repeatable. Figure 6 presents the instrumented version of the program.

3 Implementation

We have implemented the ReCrash technique for Java. This section describes the implementation, ReCrashJ, using as an example the program in Figure 5. ReCrashJ has two phases: monitoring (Section 3.1) and test generation (Section 3.2). Section 3.3 discusses implementation details of the optimizations of Section 2.2.

3.1 Implementation of the monitoring phase

ReCrashJ instruments an existing program (in Java class file format, using the ASM instrumentation tool [23]) to perform the monitoring phase of the ReCrash technique (Section 2.1). The instrumented program can be deployed in the field instead of the original program. Figure 6 shows the instrumented version of the program in Figure 5.

The instrumentation has four parts, one for each of the tasks in Figure 3.

on program start The shadow stack is implemented using static fields in the ShadowStack class. When the program starts, the requested copy strategies for the receiver and the arguments is set (lines 33 and 34). The different copy strategy classes implement the strategies of Section 2.2.1.

on entry to method *m* The receiver and the arguments to the method are stored on the shadow stack at the beginning of each method. In addition, ReCrashJ generates an id for the invocation of *m*. The id is stored as a local variable (*id*) in the method. This is demonstrated by lines 3–5 and 18–20 of Figure 6. The specific behavior of the methods `addReceiver` and `addArgument` is determined by the type of the ShadowStack (see Section 3.3 for more details).

on non-exceptional exit from method *m* If the method successfully returns without a crash, ReCrashJ removes all the data (arguments and receiver) about the method

```
1 class RandomCrash {
2   public String hexAbs(int x) {
3     int _id = ShadowStack.push("hexAbs");
4     ShadowStack.addReceiver(this);
5     ShadowStack.addArgument(x);
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    ShadowStack.popUntil(_id);
14    return ret;
15  }
16
17  public String toUpperCase(String s) {
18    int _id = ShadowStack.push("toUpperCase");
19    ShadowStack.addReceiver(this);
20    ShadowStack.addArgument(s);
21
22    String ret = s.toUpperCase();
23    ShadowStack.popUntil(_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    ShadowStack.setReceiverCopyingStrategy(new ShallowCopy());
34    ShadowStack.setArgumentsCopyingStrategy(new ReferenceCopy());
35    try  __original_main__(args);
36      catch (Throwable e)  StackDataWriter.writeStackData(e);
37  }
38 }
```

Fig. 6. The instrumented program of Figure 5. Instrumentation code is bold.

execution from the shadow stack. ReCrashJ uses the unique identifier *id* to perform this cleanup (lines 13, 23).

on top-level uncaught exception In order to react to a thrown exception that is not caught by main, ReCrashJ replaces the original main by a new main as shown in lines 27–37 in Figure 6. The new main invokes the original main in a try-catch block and handles exceptions. When an exception is caught by the new try-catch block (line 36), ReCrashJ serializes the information on the shadow stack, and stores it to the output file (line 36).

3.1.1 Serialization

To serialize objects and store the serialized objects into a file, ReCrashJ uses 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 must be of `Serializable` types. XStream does not have this limitation. ReCrash should be similarly applicable to any language with a library for marshalling/unmarshalling data. It may be an advantage of Java that Java has libraries that

```
1 public void processList(List inputList) {
2     List minimizedList = minimizeList(inputList);
3
4     // minimizedList should not be null
5     if (minimizedList == null) {
6         StackDataWriter.writeStackData(); // record this state
7         minimizedList = inputList;
8     }
9
10     ...
11 }
```

Fig. 7. A manually written ReCrash annotation helps record a program state and reproduce errors that do not results in uncaught exceptions.

represent external resources, such as files, as Java objects. It may be that a language like Eiffel that uses opaque pointers would be at a relative disadvantage.

3.1.2 Alternatives to the shadow stack The instrumented program is deployable. No other program or configuration is needed in order to run it. 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 these tools, we would have to deploy a separate program (in addition to the instrumented program) to communicate with either JPDA or JVMTI.

3.1.3 Reproducing non-crashing failures A developer may wish to reproduce failures other than crashes, for example exceptions that are caught by an exception handler or errors that do not result in an exception. In this case, the vendor can add calls to `writeStackData` wherever the program becomes aware of a failure—for example, in a catch-all handler or where an invariant is found to be false in an invariant validation routine.

As an example, consider the method `processList` in Figure 7. This method processes large lists. It first tries to minimize the input list by removing duplicates before processing it (call in line 2). However, if due to a bug, the minimization method fails and returns `null`, it is possible to process the entire list without minimization. Thus, the processing method is able to recover from the bug in the minimization method and continue. However, the developer will probably be interested in debugging the problem in the minimization method. In this case, the developer can signal to ReCrash (using the annotation in Line 6) that it should reproduce the state of the method in this case.

3.2 Implementation of the test generation phase

ReCrashJ uses the stored shadow stack to generate a JUnit `testSuite`. Figure 8 shows the tests that ReCrashJ generated for the crash in Figure 5. Figure 2 shows two of the tests generated for the Eclipse compiler crash reported in Figure 1. Each test in the suite loads the receiver and the method arguments for one method from the serialized shadow stack, and then calls that method, which results in the same exception as the original

```
1 public void test_toUpperCase() {
2   ShadowStackReader.setMethodStackItem(2);
3 
4   // load receiver
5   RandomCrash rec = (RandomCrash)ShadowStackReader.readReceiver(0);
6 
7   // Method invocation
8   rec.toUpperCase(null);
9 }
10 
11 public void test_hexAbs() {
12   ShadowStackReader.setMethodStackItem(1);
13 
14   // load receiver
15   RandomCrash rec = (RandomCrash)ShadowStackReader.readReceiver(0);
16 
17   // Method invocation
18   rec.hexAbs(0);
19 }
```

Fig. 8. Tests generated by ReCrash for the program of Figure 5.

crash. To facilitate debugging, for each argument (including receiver) that has a custom `toString` method, ReCrashJ writes the argument's string representation as a comment (lines 16, 21, 32, 36 of Figure 2).

Not every object is dynamically read from the stored shadow stack when a test is executed. ReCrashJ writes the values of primitives, strings, and null objects directly into the tests. For example, see lines 8 and 18 of Figure 8.

3.2.1 Generating multiple tests for each crash The tests in Figure 8 demonstrate a reason to create multiple tests that reproduce the crash, one for each method on the stack. The first test in Figure 8 is useless in detecting and solving the problem, because the developer is unable to understand the source of the null argument. This test would also continue to fail even when the problem is solved. On the other hand, the second test captures a value that is not handled correctly by the `hexAbs` method. This test is useful in determining and verifying a solution for the problem.

3.2.2 Extra information 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 failed.

3.3 Optimizations

In order to implement the different copying strategies of Section 2.2.1, ReCrashJ uses the Java Cloneable interface. ReCrashJ automatically adds the clone method to all classes that do not already implement it. The added clone method copies primitive fields and the references of non-primitive fields. The parameter immutability classification can be found statically [27] or by a combination of static and dynamic analysis [8]. ReCrashJ currently uses [8].

program	crash name	exception type	# of candidate tests	# of reproducible tests reference	used-fields	copy	serialized shadow stack size
Javac-jsr308	j1	null pointer	17	5	5	5	374
	j2	illegal argument	23	11	11	11	448
	j3	null pointer	8	1	1	1	435
	j4	index out of bounds	28	11	11	11	431
SVNKit	s1	index out of bounds	3	3	3	3	36
	s2	null pointer	2	2	2	2	34
	s3	null pointer	2	2	2	2	33
Eclipsec	e1	null pointer	13	0	1	8	62
BST	b1	class cast	3	3	3	3	5
	b2	class cast	3	3	3	3	5
	b3	unsupported encoding	3	3	3	3	25

Fig. 9. Subject programs and crashes used in our experimental study. For each crash, ReCrashJ generates multiple test cases that aim to reproduce the original crash. In the used-fields column ReCrashJ used the shallow (depth-1) copying strategy with the used-fields option. The serialized shadow stack size is in gzipped KB.

ReCrashJ approximates simple methods (i.e., getters and setters) as methods with no more than six opcodes. We use six opcodes as the bound since the standard getter (`getX() {return x}`) has 4 opcodes and the standard setter (`setX(int x) {this.x=x}`) has 6 opcodes without debug information.

4 Experimental Study

We evaluated ReCrashJ by performing experiments with crashes of real applications. We designed the experiments around the following research questions:

Q1 How reliably can ReCrashJ reproduce crashes?
Q2 What is the size of the stored deep copy of the shadow stack?
Q3 Are the tests generated by ReCrashJ useful for debugging?
Q4 What is the overhead (time and memory) of running ReCrashJ?

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 (serialized shadow stack) is small. We present the analysis of two real crashes in detail and show that the tests generated by ReCrashJ help to locate the source of the problem. In addition, the developers of one subject program found the generated test cases helpful. Overall, the experimental results indicate ReCrashJ is effective, scalable, and useful.

4.1 Subject systems and methodology

We use the following subject programs in our experiments:

- **Javac-jsr308**[1] is the OpenJDK Java compiler, extended with the implementation of JSR308 (“Annotations on Java Types”) [17]. We used four crashes that were provided to us by the developers. Javac-jsr308 version 0.1.0 has 5,017 methods in 86 kLOC.
- **SVNKit**[2] is a subversion[3] client. We used three crash examples for SVNKit bug reports #87 and #188. SVNKit version 0.8 has 2,819 methods in 22 kLOC.

  Eclipsec[4] is a Java compiler included in the Eclipse JDT. We used the crash from bug #30280 found in the Eclipse bug database. In version 2.1 of Eclipse, Eclipsec has 4,700 methods in 83 kLOC.
- **BST**[5] is a toy subject program used by Csallner in evaluating CnC [11, 12]. We used three BST crashes found by CnC. BST has 10 methods in 0.2 kLOC.

We used the following experimental procedure. We ran the ReCrashJ-instrumented versions of the subject programs on inputs that made the subject programs crash. We counted how many test cases reproduced each crash. We repeated this process for the different argument copying strategies introduced in Section 2.2.1, and with and without the second chance mode of Section 2.2.2. For the required parameter immutability classification (see Section 2.2.2) ReCrashJ runs PIDASA [8] one time for each subject program. It took less than half an hour to calculate the parameter immutability classification for each of the subject programs.

4.2 Reproducibility

Q1 How reliably can ReCrashJ reproduce crashes?

ReCrashJ was able to reproduce the crash in all cases (Figure 9). For some crashes (b1, b2, b3, s1, s2, and s3), every candidate test case reproduces the crash. For other crashes (e1, j1, j2, j3, and j4), only a subset of the generated test cases reproduces the crash.

In most cases, simply copying references is enough to reproduce crashes (‘reference’ column). However, in some cases an argument is side-effected, between the method entry and the point of the failure in the method, in such a way that will prevent the crash if the modified argument had been supplied on the method entry. In those cases (e.g., e1), using at least the shallow copying strategy with used-fields (Section 2.2.1) was necessary to reproduce the crash.

4.3 Stored deep copy of the shadow stack size

Q2 What is the size of the stored deep copy of the shadow stack?

[1] `http://groups.csail.mit.edu/pag/jsr308/`
[2] `http://svnkit.com/`
[3] `http://subversion.tigris.org`
[4] `http://www.eclipse.org`
[5] `http://www.cc.gatech.edu/cnc/index.html`

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

Fig. 10. A code snippet that illustrates a Javac-jsr308 crash (j4). The crash (in line 9) happens when the parameters list is shorter then the arguments list.

If ReCrashJ is deployed in the field, and a crash happens, the user will need to send the serialized deep copy of the shadow stack to the program developers. For each crash, the last column of Figure 9 presents the size of the serialized deep copy of the shadow stack for each of the inspected crashes. The size can be further reduced if the tests are generated and evaluated locally. In this case ReCrash could trim the data from frames whose candidate test cases were discarded, or perform other minimization of the test cases.

4.4 Usability study

Q3 Are the tests generated by ReCrashJ useful for debugging?

To learn whether ReCrashJ's test cases help developers to find errors, we analyzed the generated test cases for each crash. We present a detailed analysis of two crashes, e1 and j4. We also present developers' comments about the utility of the generated tests for j1-4.

Eclipsec bug (e1): Figure 12 presents the bug resulting in crash e1. Eclipsec crashes in method `canBeInstantiated` (line 19) because an earlier if statement in lines 9–11 failed to set the `hasError` to `true`. Using the generated tests (Figure 2) the developer would have been led to the buggy code. The developer would fix this problem by adding `hasError=true` on line 11. Notice that the test case for method `canBeInstantiated` (not shown in the figure) will reproduce the crash, but is not helpful in understanding it as the state of the parameter is already corrupted. Also, note that just looking at the backtrace does make the problem obvious: stepping through will be more useful (the error is far removed from the crash, but is in a method on the stack). This is an example of why it is important to generate tests for multiple methods on the stack.

Javac-jsr308 bug (j4): Using Javac-jsr308 to compile source code containing an annotation with too many arguments results in an index-out-of-bounds exception. Figure 10 shows the erroneous source code. The compiler assumes that the `parameters` and `arguments` lists are of the same size (line 9), but they may not be.

ReCrashJ generates multiple test cases that reproduce the crash; one test is shown in Figure 11.

```java
public void test_visitMethodInvocation() throws Throwable {
  // load receiver
  // rec = ...
  SubtypeVisitor rec  = (SubtypeVisitor) ShadowStackReader.readReceiver(0);

  // load arguments
  // arg_1 = test("foo", "bar", "baz");
  MethodInvocationTree arg_1 = (MethodInvocationTree)
                                ShadowStackReader.readArgument(1);

   // Method invocation
  rec.visitMethodInvocation(arg_1, null);
}
```

Fig. 11. Test case generated for a Javac-jsr308 crash (j4).

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 entire, possibly proprietary, source code for debugging.

Developer testimonials We gave the tests for j1-4 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 find the fact that you wouldn't have to wait for the crash to occur again useful."

Developer 2: "One of the challenging things for me in debugging is that when I set a break point, 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."

4.5 Performance overhead

Q4 What is the runtime overhead (time and memory) of ReCrashJ?

We compared the time and memory usage of the original and instrumented versions of the subject programs, while performing the following tasks (the three Java classes were taken from the /samples/nio/server directory in JDK 1.7):

SVNKit checkout:	Checking out a project[6], 880 files, 44Mb
SVNKit update:	Updating a project, 880 files, 44Mb
Eclipsec Content:	Compiling `Content.java` (48 LOC)
Eclipsec String:	Compiling `StringContent.java` (99 LOC)
Eclipsec Channel:	Compiling `ChannelIOSecure.java` (642 LOC)
Eclipsec JLex:	Compiling JLex version 1.2.4 (7,841 LOC)[7]

[6] `http://amock.googlecode.com/svn/trunk`

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

```
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  }
22 }
```

Fig. 12. Buggy source code from Eclipsec (Eclipse Java compiler) causing bug #30280. The program crashes with a `NullPointerException` in the `canBeInstantiated` method (line 19). This case happens when the positive path of the `if` in line 9 is taken, in which case `hasError` is not set to false (line 11).

task	execution time			
	original	**reference**	**shallow w/used-fields**	**deep copy**
SVNKit checkout	1.17	1.62 (38%)	1.75 (50%)	1657 (142,000%)
SVNKit update	0.56	0.62 (11%)	0.63 (13%)	657 (118,000%)
Eclipsec Content	0.95	1.08 (13%)	1.12 (15%)	114 (12,000%)
Eclipsec String	1.07	1.36 (27%)	1.39 (31%)	1656 (155,000%)
Eclipsec Channel	1.27	1.72 (34%)	1.74 (37%)	8101 (638,000%)
Eclipsec JLex	3.45	4.93 (42%)	5.51 (60%)	> 2 days

task	second-chance execution time			
	original	**reference**	**shallow w/used-fields**	**deep copy**
SVNKit checkout	1.17	1.17 (0.0%)	1.18 (0.8%)	1.42 (21%)
SVNKit update	0.56	0.56 (0.0%)	0.56 (0.3%)	0.56 (0.8%)
Eclipsec Content	0.95	0.97 (1.5%)	0.96 (0.9%)	3.98 (317%)
Eclipsec String	1.07	1.09 (1.7%)	1.08 (0.8%)	8.99 (742%)
Eclipsec Channel	1.27	1.27 (0.1%)	1.27 (0.0%)	16.6 (1,210%)
Eclipsec JLex	3.45	3.47 (0.7%)	3.48 (1.1%)	1637 (47,000%)

Fig. 13. Execution times of the original and instrumented programs in seconds. Slowdowns from the baseline appear in parentheses. The columns are described in Section 2.2.1.

Figure 13 compares the execution time of the original subject programs and the instrumented ones, measured using the UNIX `time` command. Because of variability in network, disk, and CPU usage, there is some noise in the measurements, but the trend is clear. The deep copy version is completely unusable, except possibly in second chance

task	maximum memory usage (MB)		
	original	**shallow w/used-fields**	**overhead%**
SVNKit checkout	8.7	9.3	6.9
SVNKit update	7.6	7.8	2.6
Eclipsec Content	4.8	8.4	75.0
Eclipsec String	5.3	9.5	79.2
Eclipsec Channel	5.2	9.9	90.3
Eclipsec JLex	9.1	11.1	21.9

Fig. 14. Memory use overhead of the instrumented programs, as measured by JProfiler.

mode, where it might be usable for in-house testing. A more optimized implementation could be more efficient, but will probably still be impractical.

Even copying only the references can be expensive, 11%–42% run-time overhead. The shallow copying strategy with used-fields has a very similar overhead, 13%–60% overhead. These values are probably usable for in-house testing.

Second chance mode, however, is where our system shines. It reduces the overhead to a 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. This simple idea is sufficient, effective, and efficient.

For the memory usage comparison, we used JProfiler[8] to measure the maximum memory used in performing each task. Figure 14 shows the memory usage — in our experiments, ReCrashJ adds 0.2M–4.7M memory overhead for the effective shallow copying strategy with the used-fields option. The memory overhead for the same copying strategy in the second-chance mode was negligible.

5 Discussion

This section discusses limitations of our technique and threats to the validity of our experiments.

5.1 Limitation in reproducing failures

ReCrash cannot necessarily reproduce all failures. The following list contains some cases that might prevent ReCrash from reproducing a failure:

failures dependent on timing This category of failures includes concurrency-related failures such as failures resulting from a race condition in a multi-threaded application. Monitoring concurrency timing dependent failures and reproducing them is future work.

failures dependent explicitly on external resources ReCrash may be unable to reproduce a failure that depends on external resources such as file system, network, hardware devices, etc. For example, a failure might depend on loading a file that no

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

longer exists. ReCrash could be combined with tracing tools for calls that access external resources. This would be less expensive than performing full tracing.

failures dependent implicitly on external resources Failures may depend implicitly on external resources. For instance, an argument such as a socket or GUI object, cannot be serialized.

failures dependent on global state, or side-effected argument state Failures may depend on global state that is not serialized. Or a failure might depend on a part of the argument state that is not stored to the shadow stack (due to the copy strategy) and is side-effected between the method entrance and the time of the failure.
We believe that not storing global state is not a great limitation for Java, as noted in Section 1 and validated by our experiments. Evaluating the effect of storing global state on reproducibility is future work.

ReCrash might still be able to reproduce a failure that depends on one of these cases. This may occur if the method call that ReCrash cannot reproduce results in a legal state that triggers a failure later in the execution. In this case the developer might be able to find, understand, and fix the failure even without understanding the exact condition that led to the state exposing it.

5.2 Buggy methods not in the stack at time of the failure

It is possible that a failure is caused by a method that is not on the call stack at the time of the failure. For example, consider the code in Figure 15. This figure contains the source of crash s1 (SVNKit bug #87). An index-out-of-bounds exception is thrown if the user omits the URL from the check out command. When no URL is supplied, the method `init` should set a default URL or throw an exception about a missing URL, but it does neither.

The program in Figure 15 will crash on Line 5, and at the time of the crash the call stack will contain the method `getURL` but will not contain the method `init`. The test case ReCrash creates for method `getURL` calls the method `getURL` with 0 as a parameter, and the exception is thrown. This test does not help the developer to find the reason for the illegal state of `myURLs`. However, ReCrash generates test cases for each frame in the call stack. Thus, ReCrash will generate a test for the `run` method (not shown), which calls `init`. Debugging using the test for the `run` method will expose the real bug.

5.3 Reusing ReCrash-generated tests

The ReCrash-generated tests are not intended to be added as is to the program test suite. The reason is that the tests de-serialize objects. If the structure of a class changes, the de-serialization will stop working. However, the generated test cases provide the skeleton of tests, and developers can replace the de-serialization with normal code.

Developers are not intended to examine the JUnit code of the generated tests to find a bug. Rather, a developer can use a debugger to examine the test's execution, revealing the cause of the failure.

```
public class SVNCommandLine {
  private list myURLs;

  public String getURL(int index) {
    return (String) myURLs.get(index);
  }

  protected void init(String[] arguments) throws SVNException {
    myURLs = new ArrayList();
    for(int i = 0; i < arguments.length; i++) {
      ...
    }
  }
}
```

Fig. 15. A code snippet from SVNKit illustrating crash s1. The method causing the crash (`init`) will not be on the stack at the time of the crash.

5.4 Privacy

One of the problems with debugging deployed applications is that users may be reluctant to send data to the developers for 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.

While we have not solved this problem, using ReCrash might be seen as an intermediate solution. Sending only the parts of the shadow stack that are used in selected tests is less likely to contain proprietary data. In addition, it should be possible for a client to prevent ReCrash from storing sensitive data by marking it (i.e., via program annotations). Another possible solution is to provide a shadow stack reader to the client, so that the client can review the encoded data and decide which parts of it (or which tests) are safe to send to the developers.

5.5 Threats to validity

We identify the following key threats to validity.

Systems and crashes examined might not be representative. We might have a system or crash selection bias. Our experiments use every crash we considered. However, we examined only 11 crashes from 4 subject systems. 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.

All failures are runtime exceptions. Our experiments use failures that manifest as runtime exceptions, such as null pointer or index out of bounds exceptions. However, ReCrashJ can monitor user-annotated exceptions or errors as discussed in Section 3.1.3. Evaluating the usefulness of the ReCrash annotation requires a user study. Future experiments and user studies should consider other types of failures including user-annotated non-exception points.

6 Related Work

6.1 Record and replay

Many existing record and replay techniques for postmortem analysis and debugging [10, 14, 19, 9, 21, 18, 31, 15] are based on three components: checkpoints, logging, 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 does 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's logging simplicity allows it to be deployed remotely with relatively low overhead. Most of the previous techniques are designed for in-house testing or debugging, and have unreasonable overhead for deployed applications.

BugNet [21], ReVirt [15], and FlashBack [28] require 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 can be used to reproduce a recorded failure in different environments.

Choi et al. [9], Instant Replay [19], BugNet [22], 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, 19], or they only allow replaying a narrow window of time [22]. Similar to BugNet [22], ReCrash only allows replaying a part of the execution before the failure. ReCrash is only able to deterministically reproduce a non-deterministic failure if one of the generated tests captures the state after the non-determinism. ReCrash could be combined with other monitoring tools (storing only some of the environment dependencies) to create an efficient technique that can reproduce non-deterministic failures.

jRapture [29], test factoring [26], test carving [16], 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. However, ReCrash can be viewed as writing mock objects in the method call data. The objects recorded are a faithful representation of the actual objects down to a given depth. At lower depths their fields contain values that are possibly incorrect, but that in practice are effective in reproducing program behavior.

6.2 Test generation

Contract Driven Development [20] generates test cases using contracts (pre- and post-conditions) written by developers. If a contract is violated during the execution of an application in debug mode, CDD captures the transitive state of the arguments to one method (often the failing method). CDD then attempts to use the captured state to generate a test case that reproduces the violation. Since the arguments' state is captured after the violation, it is equal to ReCrash's reference copying strategy. ReCrash might generate more useful tests and reproduce more failures because ReCrash can store arguments' state before a violation occurs, and because ReCrash generates multiple tests, one for each method on the shadow stack at the time of the failure. CDD is designed to be used in the development process, whereas ReCrash can be used either in-house or in-field. ReCrash monitors the stack and generates a test case without the need for special IDE support.

CnC [12], JCrasher [11], Eclat [24], Randoop [25], and DSDCrasher [13] use 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 attempt to generates tests based on values observed during an actual program execution.

6.3 Remote data collection

Crash reporting 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 vendor uses the stack back-trace to correlate the report to known problems. If several reports share similar characteristics, the vendor may try to reproduce the original failure. However, reproducing the original failure requires non-trivial human effort. The core-dump provides one snapshot of the program state at the end of time (after the crash). ReCrash provides several partial snapshots and enables execution of each of them, using a test. ReCrash stored data is smaller than a core-dump and thus is easier to send to the vendor.

7 Conclusion

We have introduced the ReCrash technique for generating unit tests that reproduce a software failure. The tests utilize partial snapshots of the program state that ReCrash captures on each method execution and records in the case of a failure. ReCrash is simple to implement, it is scalable, and it generates simple, helpful test cases that effectively reproduce failures. Our ReCrashJ tool implements the technique for Java.

We have evaluated ReCrashJ with real crashes from Javac-jsr308, SVNKit, Eclipsec, and BST. ReCrashJ created tests that reproduced every crash we have inspected, and developers found the generated tests useful for debugging. The performance overhead of programs instrumented by ReCrashJ was 13%–64% for the shallow copying strategy with used-fields and 0%–1.7% for all copying strategies in the second-chance mode. In our experiments, ReCrashJ increases memory usage when using the effective shallow copying strategy with used-fields, by 0.2M–4.7M. The size of the stored snapshots, the serialized shadow stack, is manageable: 0.5k—448k. ReCrashJ is usable in real software deployment.

The ReCrashJ tool described in this paper is publicly available for download from `http://pag.csail.mit.edu/ReCrash`.

Acknowledgements We thank Matthias Hauswirth and the anonymous reviewers for their comments.

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. JVMTI 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. Technical Report MIT-CSAIL-TR-2006-056, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, September 5, 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. S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the ACM SIGSOFT 14th Symposium on the Foundations of Software Engineering (FSE 2006)*, pages 253–264, Portland, OR, USA, November 7–9, 2006.
17. M. D. Ernst. Annotations on Java types: JSR 308 working document. http://pag.csail.mit.edu/jsr308/, November 12, 2007.
18. 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.
19. T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.
20. A. Leitner, I. Ciupa, M. Oriol, B. Meyer, 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.
21. 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.
22. S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.
23. ObjectWeb Consortium. ASM - Home Page, 2007. http://asm.objectweb.org/.
24. 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.
25. 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.
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. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA 2007, Proceedings of the 2007 International Symposium on Software Testing and Analysis*, London, UK, July 10–12, 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.