ReCrash
Making crashes reproducible by preserving object states

Shay Artzi, Sunghun Kim*, Michael D. Ernst

MIT  * now at HKUST
Eclipse bug 30280:
2 days to reproduce, 4 minutes to fix

<table>
<thead>
<tr>
<th>Date/Time</th>
<th>User/Developer</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003-01-27 08:01</td>
<td>User</td>
<td>Eclipse crashed... I have no idea why... Here is the stack trace.</td>
</tr>
<tr>
<td>2003-01-27 08:26</td>
<td>Developer</td>
<td>What build are you using? Do you have a testcase to reproduce?</td>
</tr>
<tr>
<td>2003-01-27 08:39</td>
<td>Developer</td>
<td>Which JDK are you using?</td>
</tr>
<tr>
<td>2003-01-28 13:06</td>
<td>User</td>
<td>I’m running Eclipse 2.1, ... I was not able to reproduce the crash.</td>
</tr>
<tr>
<td>2003-01-29 04:33</td>
<td>Developer</td>
<td>Reproduced.</td>
</tr>
<tr>
<td>2003-01-29 04:37</td>
<td>Developer</td>
<td>Fixed.</td>
</tr>
</tbody>
</table>
Reproducing crashes

• If a crash can’t be reproduced:
  – Hard to fix
  – Hard to validate a solution
• Reproducing a crash is hard!
  – Nondeterminism
  – Configuration and system information
  – Steps to reproduce may be complex or long
  – In-field detection
  – Users rarely provide reproducible bug reports
Approach 1: Postmortem analysis

Examples: stack trace, core dump

Problems:
• Fault (bug) may be far from failure (exception)
  – Faulty method may not be in stack trace
• Too much information
  – Core dump: big; hard to interpret
• Not enough information
  – Shows effects (final values), not causes
  – Need initial values to reproduce the failure
Approach 2: Record & replay

- Logging: record interactions with environment
- Replay: use log to reproduce the execution
- Checkpoint: replay skips part of the execution

Problems:
- Unrealistic overhead
- Invasive changes to HW/OS/application
Record & replay for OO programs

• Object-oriented style uses only nearby state
  – Unit testing depends on this property
• ReCrash reproduces this nearby state
  – Does not replay an execution
  – Static and dynamic analyses reduce the size
• Lightweight: efficient, no harness, usable in-field
• Not guaranteed to reproduce every failure
ReCrash technique

Goal: Convert a crash into a set of unit tests

1. Monitoring: maintain a shadow stack
   – Contains a copy of each method argument
   – On program crash, write the shadow stack to a file

2. Test generation: create many unit tests
   – For each stack frame, create one unit test:
     • Invoke the method using arguments from the shadow stack
     • If the test does not reproduce the crash, discard the test
Maintaining the shadow stack

• On method \textit{entry}:
  – Push a new shadow stack frame
  – Copy the actual arguments to the shadow stack
• On non-exceptional method \textit{exit}:
  – Pop the shadow stack frame
• On program \textit{failure} (top-level exception):
  – Write the shadow stack to a file
    • Serializes all state referenced by the shadow stack
Create one JUnit test per stack frame

We expect the method to fail as it did at run time

```java
public void testResolveType() {
    AllocExpr rec = (AllocExpr) shadowStack.getArg(0);
    BlockScope arg = (BlockScope) shadowStack.getArg(1);

    rec.resolveType(arg);
}
```

Test case for Eclipse bug 30280

Read arguments from the saved shadow stack

Invoke the method from the stack frame
Evaluating unit tests

• Run each generated unit test
• Discard the test if it does not reproduce the run-time exception
How a developer uses the tests

- In a debugger, step through execution and examine fields
- Experiment by modifying the tests
- Verify a fix
- Create a regression test
  - Replace deserialized objects by real objects or mock objects
  - More readable and robust
Why create multiple tests?

• Not all tests may reproduce the failure
  – Due to state not captured on the shadow stack
    • Sockets, files, nondeterminism, distant program state
    • Does capture all values that are passed as arguments

• Some tests may not be useful for debugging
Not every test is useful

Stack trace:

NullPointerException
  at Class1.toString
  at Class2.myMethod
  ...

Tests:

```java
void test_toString() {
    Class1 receiver = null;
    receiver.toString();
}

void test_myMethod() {
    Class2 receiver = (Class2) shadowStack.getArg(0);
    receiver.myMethod();
}
```
Other features of ReCrash

• Non-crashing failures
  – Add a ReCrash annotation

• Caught exceptions that lead to later failures

• Adding extra information to test cases
  – Version number, configuration information

• Reducing the serialized stack
  – Size, privacy
Cost of monitoring

Key cost: **copying arguments** to shadow stack

Tradeoff: less information in shadow stack ⇒ lower chance of reproducing failures

1. **Depth** of copy
   - Deep, reference, or a hybrid

2. Save **less information** about each argument
   - Focus on important fields

3. Monitor **fewer methods**
   - Ignore methods not likely to crash or to be useful
Original program execution
Original program execution
Original program execution
Original program execution
Original program execution
Original program execution

Real stack

R:
A1:
A2:

R:
A1:

R:

18
1. Depth of copying
Deep copy
Deep copy
Deep copy

Real stack

Shadow stack
Deep copy
Deep copy
Deep copy

Multiple copies $\Rightarrow$ quadratic cost
Unusable in practice

Real stack

Shadow stack
Reference copy
Reference copy
Reference copy
Reference copy
Depth-1 copy

Real stack

Shadow stack
2. Ignoring some fields
Depth-1 copy

Real stack

Shadow stack
Depth1 + used fields
(= Depth2 − unused fields)
Pure methods

Analysis results

Real stack

Shadow stack

R:
A1:
A2:

R:
A1:

R:

R:

18

17
Pure methods
Immutable objects

Analysis results
Immutable objects
3. Ignoring some methods
Ignored methods

Analysis results

Real stack

A1:

R:

A2:

Shadow stack

R:

A1:

18

R:

A2:

17

18
Ignored methods
Methods that are unlikely to be useful

- Trivial methods
- Private methods
- Library methods
- Methods that are unlikely to crash
Second chance mode

Idea: monitor only methods that are likely to crash

• Initially, monitor no methods
• After a crash, add monitoring for methods in the stack trace
  – Can update all clients, not just the one that crashed

• Tradeoffs:
  + Very low overhead (no overhead until a crash)
  – Requires a failure to occur twice
Experimental study

1. Can ReCrash reproduce failures?
2. Are the ReCrash-generated tests useful?
3. How large are the test cases?
4. What is the overhead of running ReCrash?
Subject programs

Investigated 11 real crashes from:

- BST: .2 KLOC
- SVNKit: 22 KLOC
- Eclipse compiler: 83 KLOC
- Javac-jsr308: 86 KLOC
**Q1: Can ReCrash reproduce failures?**

<table>
<thead>
<tr>
<th>Program</th>
<th>Failure</th>
<th>Candidate tests</th>
<th>Reproducible tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>reference copy</td>
</tr>
<tr>
<td>BST</td>
<td>Class cast</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>BST</td>
<td>Class cast</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>BST</td>
<td>Unsupported</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>SVNKit</td>
<td>Index bounds</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>SVNKit</td>
<td>Null pointer</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SVNKit</td>
<td>Null pointer</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Eclipsec</td>
<td>Null pointer</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Javac-jsr308</td>
<td>Null pointer</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>Javac-jsr308</td>
<td>Illegal arg</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>Javac-jsr308</td>
<td>Null pointer</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Javac-jsr308</td>
<td>Index bounds</td>
<td>28</td>
<td>11</td>
</tr>
</tbody>
</table>
# Q1: Can ReCrash reproduce failures?

<table>
<thead>
<tr>
<th>Program</th>
<th>Failure</th>
<th>Candidate tests</th>
<th>Reproducible tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>reference copy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>depth 1 + used-fields</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>deep copy</td>
</tr>
<tr>
<td>BST</td>
<td>Class cast</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Class cast</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Unsupported</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>SVNKit</td>
<td>Index bounds</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Null pointer</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Null pointer</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Eclipssec</td>
<td>Null pointer</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Javac-jsr308</td>
<td>Null pointer</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Illegal arg</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Null pointer</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Index bounds</td>
<td>28</td>
<td>11</td>
</tr>
</tbody>
</table>
Q2: Are the ReCrash tests useful?

• Developers found the tests useful
  – Developer 1: “You don’t have to wait for the crash to occur again”; also liked multiple tests
  – Developer 2: “Using ReCrash, I was able to jump (almost directly) to the necessary breakpoint”

• Developers found the stack trace insufficient
  – Unable to reproduce
  – The failure may be far removed from the fault
Q3: How large are the test cases?

- The JUnit test suite uses the shadow stack
- Serializes all reachable parts of the heap

<table>
<thead>
<tr>
<th>Program</th>
<th>Average shadow stack size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BST</td>
<td>12</td>
</tr>
<tr>
<td>SVNKit</td>
<td>34</td>
</tr>
<tr>
<td>Eclipse</td>
<td>62</td>
</tr>
<tr>
<td>Javac-jsr308</td>
<td>422</td>
</tr>
</tbody>
</table>
Q4: Time overhead of ReCrash

Overhead of instrumented program in the field
Q4: Memory overhead of ReCrash

Absolute memory overhead: \(0.2 \text{M} - 4.7 \text{M}\)
Generating unit tests from system runs

• Test factoring [Saff 2005, Elbaum 2006]
  – Developer selects a portion of the program
  – System logs interactions with the environment
  – Unit test replays execution in a test harness

• Contract-driven development [Leitner 2007]
  – Reference copying, intended for durable tests

• Backward-in-time debuggers [Lienhard 2008]
  – Heavier-weight logging and checkpoints
Future work

• Capture more state
  – Concurrency, timing, external resources

• Other implementation tradeoffs
  – Copy-on-write
  – Existing VM hooks
  – Logging/debugging techniques
  – These are probably orthogonal to ReCrash
ReCrash converts failures into tests

- ReCrash effectively reproduces failures
  - Replicates program states
  - Generates multiple unit tests
- The unit tests are useful
- Low overhead
  - Records only relevant parts of an execution
  - 4 program analyses; second chance mode
  - Can deploy instrumented programs in the field
- Download: http://pag.csail.mit.edu/ReCrash/
ReCrash converts failures into tests

- ReCrash effectively reproduces failures
  - Replicates program states
  - Generates multiple unit tests
- The unit tests are useful
- Low overhead
  - Records only relevant parts of an execution
  - 4 program analyses; second chance mode
  - Can deploy instrumented programs in the field
- Download: http://pag.csail.mit.edu/ReCrash/
Maintaining the shadow stack
Maintaining the shadow stack

On method entry

Real stack

R:
A1:
A2:

Shadow stack

R:
A1:

18

R:

17

18
Maintaining the shadow stack

On method entry:
1. Push a new shadow stack frame
Maintaining the shadow stack

On method entry:
1. Push a new shadow stack frame
2. Copy the actual arguments to the shadow stack
Maintaining the shadow stack

On method exit
Maintaining the shadow stack

On method exit:
1. Pop shadow stack frame
Maintaining the shadow stack

On program failure (top-level exception):
Maintaining the shadow stack

On program failure (top-level exception):
1. Write the shadow stack to a file
Maintaining the shadow stack

On program failure (top-level exception):
1. Write the shadow stack to a file
   Serializes all referenced state