Research Topics in SE & Distributed Systems

CS 475, Spring 2018
Concurrent & Distributed Systems
Broad Research Areas

Software Engineering
- ICSE
- FSE
- ISSTA
- ICPC
- ICST

Fault Reproduction
- Behavioral Clones
- Test Acceleration
- Logical Data Objects
- Flaky Tests
- Metamorphic Testing
- Blockchains
- Microservices
- Dataflow Analysis
- Microcheckpointing

Systems
- OSDI
- EuroSys

Program Analysis
- OOPSLA
- ECOOP
Accelerating Testing

With Darko Marinov (UIUC), Gail Kaiser (Columbia) and ElectricCloud, Inc
Automated Security Testing

With Luís Pina (GMU), Christian Hammer (Padderborn)
Simplified Software Lifecycle

- Make changes to code
- Build & test
- Commit

How long is too long of a build?

1 day?
6 hours?
10 minutes?
Testing Dominates Build Times

Projects taking > 1 hour to build on GitHub using Maven

- Testing: 90%
- Compiling: 8%
- Other: 2%
JUnit Test Execution

Overhead of restarting the JVM?

- Unit tests as fast as 3-5 ms
- JVM startup time is fairly constant (1.4 sec)

Up to 4,153%, avg 618%

*From our study of 20 popular FOSS apps
Accelerating Test Execution

Efficiently resetting state between tests reduces time by 60% on average!

“VMVM” transferred to industrial partner ElectricCloud
Flaky Tests

Test 1
Test 2
Test 3
Test 4
Flaky Tests

- Test might pass or fail given the SAME code
- Google: 16% of tests are “flaky” in some way
- How do you handle these flaky tests?
  - Typical fix: if you think something is flaky, run it again and again - outcome is only decided from the complete status
Flaky Tests

Test 3

Test 3

Test 3

“Test is OK!”

Test 3

Test 3

Test 3

“Test failed!”

Test 3

Test 3

Test 3

“Test outcome is unknown!”
Flaky Tests

• What happened to *accelerating* testing?
• Now tests need to be run three times!
• Can we identify with certainty that a test is a false alarm without re-running?
  • Previous approaches were slower than re-running!
DeFlaker: Automatically Detecting Flaky Tests

Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tiffany Yung and Darko Marinov
George Mason University, University of Illinois at Urbana-Champaign and Carnegie Mellon University

[To appear at ICSE 2018 in Gothenburg, May 31, 2018]
Flaky Tests

• Our key insight: there is lightweight information we can track while a test runs
• “Did this test run any code that changed?”
• Tracking coverage can be slow though! (40-50% overhead!)
  • …and we want to make things faster
DeFlaker’s Differential Coverage

DeFlaker tracks *differential coverage* — only tracking code that changed since the last execution of the tool.
Differential Coverage

Just syntactic diff (e.g. from git) is insufficient to notice coverage of all kinds of changes!

```java
public class SuperOld {
    public void magic() {
    }
}

public class SuperNew extends SuperOld {
    public void magic() {
        assert(false); // causes test to fail
    }
}

public class App extends SuperOld SuperNew {
}

public class TestApp {
    @Test public void testApp() {
        new App().magic(); // Now calls SuperNew.magic!
    }
}
```
DeFlaker

- Tracks line coverage of all changed statements (in both tests and SUT)
- Identifies non-statement changes in classes by parsing them, tracks with class-level coverage
- Detects flaky test failures “just-in-time” when they fail
- Implemented as a maven extension (3-line addition to pom.xml)
Evaluation

• What is the performance overhead of running DeFlaker?
• How many flaky tests does DeFlaker find in comparison to rerunning failed tests?
DeFlaker is Fast

Evaluation on 17 open source Java projects: average 5% overhead
DeFlaker Finds Flaky Tests

Flaky Detection Strategy:
- **Surefire**
- **Surefire + Fork**
- **Surefire + Fork + Reboot**
- **DeFlaker (NO reruns needed!)**
DeFlaker Findings

• HOW you re-run flaky tests matters much more than how many times you rerun them
• DeFlaker is extremely low overhead and can immediately identify flaky tests
• Also deployed shadowing live executions on TravisCI, found 87 new flaky tests and reported to developers, many now fixed
• Differential coverage may have many other useful applications as well
• Try it out! http://deflaker.org/
# Measuring Code Coverage

<table>
<thead>
<tr>
<th>Project</th>
<th>Lang</th>
<th>Source Builds</th>
<th>LoC</th>
<th>Time range (months)</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache/commons-collections</td>
<td>java</td>
<td>25</td>
<td>12,765</td>
<td>40</td>
<td>84%</td>
</tr>
<tr>
<td>apache/commons-dbcp</td>
<td>java</td>
<td>25</td>
<td>5,662</td>
<td>19</td>
<td>48%</td>
</tr>
<tr>
<td>apache/commons-cs</td>
<td>java</td>
<td>5</td>
<td>971</td>
<td>65</td>
<td>67%</td>
</tr>
<tr>
<td>apache/commons-functor</td>
<td>java</td>
<td>25</td>
<td>2,693</td>
<td>69</td>
<td>83%</td>
</tr>
<tr>
<td>apache/commons-io</td>
<td>java</td>
<td>25</td>
<td>5,021</td>
<td>2</td>
<td>87%</td>
</tr>
<tr>
<td>apache/commons-joxpath</td>
<td>java</td>
<td>25</td>
<td>9,633</td>
<td>94</td>
<td>75%</td>
</tr>
<tr>
<td>apache/commons-math</td>
<td>java</td>
<td>25</td>
<td>45,084</td>
<td>16</td>
<td>90%</td>
</tr>
<tr>
<td>apache/commons-net</td>
<td>java</td>
<td>25</td>
<td>9,210</td>
<td>1</td>
<td>30%</td>
</tr>
<tr>
<td>apache/commons-validator</td>
<td>java</td>
<td>25</td>
<td>2,854</td>
<td>10</td>
<td>77%</td>
</tr>
<tr>
<td>apache/empire-db</td>
<td>java</td>
<td>25</td>
<td>21,238</td>
<td>60</td>
<td>14%</td>
</tr>
<tr>
<td>apache/hupcore</td>
<td>java</td>
<td>5</td>
<td>13,198</td>
<td>18</td>
<td>77%</td>
</tr>
<tr>
<td>ARM/Armed/Armed</td>
<td>python</td>
<td>C.IO</td>
<td>60</td>
<td>804</td>
<td>75%</td>
</tr>
<tr>
<td>bswap/times</td>
<td>elixir</td>
<td>C.IO</td>
<td>128</td>
<td>2,615</td>
<td>65%</td>
</tr>
<tr>
<td>brocd Минг/инверсия</td>
<td>Scala</td>
<td>C.IO</td>
<td>170</td>
<td>2,658</td>
<td>67%</td>
</tr>
<tr>
<td>containers/virtcontainers</td>
<td>go</td>
<td>C.IO</td>
<td>296</td>
<td>5,332</td>
<td>60%</td>
</tr>
<tr>
<td>coreslab/ingress-controller</td>
<td>go</td>
<td>C.IO</td>
<td>98</td>
<td>2,041</td>
<td>21%</td>
</tr>
<tr>
<td>demianm/epidemic/collector</td>
<td>java</td>
<td>17</td>
<td>248</td>
<td>794</td>
<td>92%</td>
</tr>
<tr>
<td>disk/Ans</td>
<td>python</td>
<td>C.IO</td>
<td>290</td>
<td>15,322</td>
<td>91%</td>
</tr>
<tr>
<td>dominoes/transactions</td>
<td>java</td>
<td>5</td>
<td>11,008</td>
<td>9</td>
<td>88%</td>
</tr>
<tr>
<td>dropwizard/dropwizard</td>
<td>java</td>
<td>5[18]</td>
<td>7,700</td>
<td>9</td>
<td>87%</td>
</tr>
<tr>
<td>eBates/ens-filter</td>
<td>java</td>
<td>17</td>
<td>204</td>
<td>280</td>
<td>99%</td>
</tr>
<tr>
<td>ISN/Networks/ISN-bigip-cfr</td>
<td>go</td>
<td>C.IO</td>
<td>103</td>
<td>5,621</td>
<td>83%</td>
</tr>
<tr>
<td>fastgospod</td>
<td>scala</td>
<td>17</td>
<td>233</td>
<td>640</td>
<td>45%</td>
</tr>
<tr>
<td>Gillespie/gvim-plugin-angular</td>
<td>node</td>
<td>21</td>
<td>1,213</td>
<td>19</td>
<td>100%</td>
</tr>
<tr>
<td>google/sketch/gsk-collections</td>
<td>java</td>
<td>25</td>
<td>38,242</td>
<td>16</td>
<td>93%</td>
</tr>
<tr>
<td>google/nerfs</td>
<td>java</td>
<td>5</td>
<td>3,401</td>
<td>45</td>
<td>92%</td>
</tr>
<tr>
<td>Hazel/Research/DeepDrive</td>
<td>Scala</td>
<td>C.IO</td>
<td>111</td>
<td>3,913</td>
<td>82%</td>
</tr>
<tr>
<td>Hector/Hello/Reader</td>
<td>java</td>
<td>5</td>
<td>137</td>
<td>5,939</td>
<td>6%</td>
</tr>
<tr>
<td>ikawa/kagome</td>
<td>go</td>
<td>C.IO</td>
<td>91</td>
<td>1,099</td>
<td>27</td>
</tr>
<tr>
<td>iloveip/Compiler</td>
<td>dotNet</td>
<td>C.IO</td>
<td>96</td>
<td>1,874</td>
<td>86%</td>
</tr>
<tr>
<td>iplay/soup</td>
<td>java</td>
<td>17</td>
<td>246</td>
<td>6,615</td>
<td>70%</td>
</tr>
<tr>
<td>iinvite/Tunnelbar</td>
<td>java</td>
<td>5</td>
<td>3,935</td>
<td>9</td>
<td>84%</td>
</tr>
<tr>
<td>JodaDate/joda-time</td>
<td>java</td>
<td>18[8]</td>
<td>248</td>
<td>14,780</td>
<td>90%</td>
</tr>
<tr>
<td>joel-costsgtio/jolattp-core</td>
<td>java</td>
<td>5[18]</td>
<td>241</td>
<td>11,104</td>
<td>91%</td>
</tr>
<tr>
<td>jmail/jkaila-pixy</td>
<td>go</td>
<td>C.IO</td>
<td>64</td>
<td>4,387</td>
<td>69%</td>
</tr>
<tr>
<td>jmail/Jkaila-pixy</td>
<td>node</td>
<td>C.IO</td>
<td>121</td>
<td>1,734</td>
<td>71%</td>
</tr>
<tr>
<td>MTTLibraries/topologies</td>
<td>Scala</td>
<td>C.IO</td>
<td>102</td>
<td>5,644</td>
<td>65%</td>
</tr>
<tr>
<td>Plume/Plume/Plume-plugin-quant</td>
<td>node</td>
<td>C.IO</td>
<td>62</td>
<td>6,028</td>
<td>22%</td>
</tr>
<tr>
<td>PragTohbeecee</td>
<td>elixir</td>
<td>C.IO</td>
<td>148</td>
<td>441</td>
<td>85%</td>
</tr>
<tr>
<td>raml-org/raml-java-parsers</td>
<td>java</td>
<td>17</td>
<td>248</td>
<td>6,455</td>
<td>16</td>
</tr>
<tr>
<td>Shlit/forward/apso</td>
<td>Scala</td>
<td>C.IO</td>
<td>92</td>
<td>1,629</td>
<td>59%</td>
</tr>
<tr>
<td>spath/modules/ormap</td>
<td>go</td>
<td>C.IO</td>
<td>55</td>
<td>5,983</td>
<td>81%</td>
</tr>
<tr>
<td>squre/oxendr</td>
<td>java</td>
<td>5</td>
<td>247</td>
<td>1,185</td>
<td>79%</td>
</tr>
<tr>
<td>square/retrofit</td>
<td>java</td>
<td>18</td>
<td>2,479</td>
<td>17</td>
<td>54%</td>
</tr>
<tr>
<td>SteamDatabase/ValveResourceFormat</td>
<td>dotNet</td>
<td>C.IO</td>
<td>179</td>
<td>2,794</td>
<td>73%</td>
</tr>
<tr>
<td>terasourceorg/terahome-fgw</td>
<td>java</td>
<td>C.IO</td>
<td>97</td>
<td>2,151</td>
<td>99%</td>
</tr>
<tr>
<td>undertow-undertow</td>
<td>java</td>
<td>5</td>
<td>238</td>
<td>5,008</td>
<td>60%</td>
</tr>
<tr>
<td>xing/xing</td>
<td>java</td>
<td>5</td>
<td>198</td>
<td>15,440</td>
<td>60%</td>
</tr>
</tbody>
</table>

Total 48 projects, 389,238 LOC

Average 165 8,110 29 75% 76%
Coverage of Changes

Percent of builds satisfying patch coverage % at level indicated by color:
- 0
- (0–25)
- (25–50]
- (50–75]
- (75–100)
- 100%
Changes of Coverage

Patches with changes to code files:
- Increase
- No impact
- Decrease

Patches with no changes to code files:
- Increase
- No impact
- Decrease
Testing Infrastructure

- **Projects**
- **Projects * Versions**

**Projects (VMVM, ICSE)**: ~5 CPU-Years of build/test

**Projects (ElectricTest, FSE)**: ~10 CPU-Years of build/test

**Projects (DeFlaker, ICSE)**: ~10 CPU-Years of build/test

**Today**: ~10 CPU-Years of build/test
Testing Infrastructure

• How do we do 15,000 build+test cycles quickly and cheaply?
• An embarrassingly parallel problem
• Each commit of each project can be built with no dependency on any other build
• Hence, can trivially scale to do as many at once as makes sense…
Backend Infrastructure

• Master/slave architecture
• Failure assumptions: master can not fail, slaves may fail by crashing
• In practice, we've seen worker nodes fail by crashing from underlying hardware problems
• Slave nodes boot up, bring up a shared NFS mount, run an "on boot" script that is provided by the master to determine how to run builds…
Testing Infrastructure

- **web interface**
- **Master Build Machine**
  - **Git Clones**
  - **Build Logs**
- **Build DB**
  - **MySQL**
  - **NFS Mounts**
- **Build Cluster**

**Exactly 1 (on EC2, usually r3.large)**

**Up to 250+ (on EC2, m3.medium $0.0096/hr or c3.large $0.02/hr)**
Example Interface: Flaky Tests

- Can select an experiment to visualize/export
- View results of re-running same build several times
“Live” Infrastructure
"Live" Infrastructure

Developer

Pushes code

GitHub

Polls for updates to developer’s project

Travis CI

Triggers build

Our server

Merges to our shadow repo, syncs

Cloud Database

All of the results of our instrumentation...
Dynamic Data Flow Analysis: Taint Tracking

Inputs

Flagged (“Tainted”) Input

Application

Outputs

Output that is derived from tainted input
Taint Tracking: Applications

End-user privacy testing: Does this application send my personal data to remote servers?
Taint Tracking: Applications

Testing: Are my test cases overly specified?

```java
@Test
public void testEnrolled() throws Exception {
    Student s = new Student();
    s.id = 5;
    s.name = "Bob";
    s.setEnrolled();
    assertTrue(s.isEnrolled);
}
```

No assertion depends on these values
Problem: No Tool Support
“Normal” Taint Tracking

• Associate tags with data, then propagate the tags

• Approaches:
  • Operating System modifications [Vandebogart ’07], [Zeldovich ’06]
  • Language interpreter modifications [Chandra ’07], [Enck ’10], [Nair ’07], [Son ’13]
  • Source code modifications [Lam ’06], [Xu ’06]
  • Binary instrumentation of applications [Clause ’07], [Cheng ’06], [Kemerlis ’12]

Not portable

Hard to be sound, precise, and performant
Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs

Jonathan Bell and Gail Kaiser
Columbia University

[OOPSLA 2014; Artifact Evaluated & Approved]
Phosphor

• Leverages benefits of interpreter-based approaches (information about variables) but fully portably

• Instruments all byte code that runs in the JVM (including the JRE API) to track taint tags
  • Add a variable for each variable
  • Adds propagation logic
Key contribution:
How do we efficiently store metadata for every variable without modifying the JVM itself?
Phosphor: Instrumentation Strategy

double pie = 3.14;
double more = 1;
double more_pie = pie + more;
int ret = callSomeMethod(pie);

double pie = 3.14;
int pie_tag = 0;
double more = 1;
int more_tag = 0;
double more_pie = pie + more;
int more_pie_tag = pie_tag | more_tag;
TaintedInt tmp = callSomeMethod(pie_tag, pie);
int ret = tmp.val;
int ret_tag = tmp.tag;

(Of course, we do this all at byte code, not source code)
Segmentation Fault

It’s not so easy.
(Yes: You can seg-fault the JVM trivially)
JVM Type Organization

• Primitive Types
  • int, long, char, byte, etc.

• Reference Types
  • Arrays, instances of classes
  • All reference types are assignable to java.lang.Object
## Phosphor's Taint Tag Storage

<table>
<thead>
<tr>
<th></th>
<th>Local variable</th>
<th>Method argument</th>
<th>Return value</th>
<th>Operand stack</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stored as a field of the object</td>
</tr>
<tr>
<td><strong>Object array</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stored as a field of each object</td>
</tr>
<tr>
<td><strong>Primitive</strong></td>
<td>Shadow variable</td>
<td>Shadow argument</td>
<td>&quot;Boxed&quot;</td>
<td>Below the value on stack</td>
<td>Shadow field</td>
</tr>
<tr>
<td><strong>Primitive array</strong></td>
<td>Shadow array variable</td>
<td>Shadow array argument</td>
<td>&quot;Boxed&quot;</td>
<td>Array below value on stack</td>
<td>Shadow array field</td>
</tr>
</tbody>
</table>
Taint Propagation

• Modify all byte code instructions to be taint-aware by adding extra instructions

• Examples:
  • Arithmetic -> combine tags of inputs
  • Load variable to stack -> Also load taint tag to stack
  • Modify method calls to pass taint tags
Complications

- Primitives on the stack
- Non-modifiable classes (e.g. Object, StackTraceElement, Byte, some others)
- Arrays of primitive values
  - And multi-dimensional arrays, and upcasting arrays
- Native methods
Phosphor Doubles Stack Size

Code Snippet

```java
void foo(int i, int j)
{
    int k = i + j;
}
```

Bytecode

```
ILOAD 1
ILOAD 2
IADD
ISTORE 3
```

Code Snippet (instrumented)

```java
void foo(int i_tag, int i,
         int j_tag, int j)
{
    int k = i + j;
    int k_tag = i_tag | j_tag;
}
```

Instrumented Bytecode

```
ILOAD 1
ILOAD 2
ILOAD 3
ILOAD 4
DUP2_X1
POP2
IADD
SWAP
IOR
ISTORE 6
ISTORE 5
```

Methods can get long, but avoids need for expensive shadow stack
Non-Modifiable Classes (and Arrays)

• Candidate approach:
  • Use a HashMap, with each untrackable object as key
  • Very, very slow (need to access a globally-locked HashMap for EVERY operation you do involving an object)

• Phosphor:
  • Special case everything
  • Primitive arrays: get their own shadow array, tracked through upcasting
  • Non-modifiable classes: HashMap
Challenge 1: Upcasting

Solution 1: Box taint tag with array when upcasting

```java
byte[] array = new byte[5];
Object ret = array;
int[] array_tag = new int[5];
byte[] array = new byte[5];
Object ret = new TaintedByteArray(array_tag, array);
byte[] foo = (byte[]) ret;
int[] foo_tag = ((TaintedByteArray) ret).tag;
byte[] foo_tag = ((TaintedByteArray) ret).val;
```
Challenge 1: Upcasting

Also needed for multi-dimension primitive arrays

```java
byte[] array = new byte[5];
byte[][] ret = new byte[][]{array};

int[] array_tag = new int[5];
byte[] array = new byte[5];

TaintedByteArray[] ret = new TaintedByteArray[1];
ret[0] = new TaintedByteArray(array_tag, array);

byte[] foo = ret[0];
int[] foo_tag = ret[0].tag;
byte[] foo_tag = ret[0].val;
```
Challenge: Native Code

We can’t instrument everything!
Native Code

```java
public native int hashCode();
```

What caller expects

```java
public TaintedInt hashCode();
```


```java
public TaintedInt hashCode$$wrapper() {
   return new TaintedInt(0, hashCode());
}
```
Wrappers work both ways: native code can still call a method with the old signature

```java
public int[] someMethod(byte in)
{
    return someMethod$$wrapper(0, in).val;
}

public TaintedIntArray someMethod$$wrapper(int in_tag, byte in)
{
    //The original method "someMethod", but with taint tracking
}
```
Native Code

• The main design limitation
• Return value’s tag becomes combination of all parameters (heuristic); not found to be a problem in our evaluation
• Note: reflection is NOT a limitation, is easiest of all challenges to work around (runtime wrappers)
Configuration Options

• Tag propagation modes:
  • Data flow  
  • Control flow

• Tag format:
  • Integer (bit vectors)
  • Object (maintain relationships sets)

• Automatic Tagging and Checking

```c
int c = a + b;
if (a == 0) c = 0;
```
Implicit vs. Explicit Data Flow

```java
String secretStr = "secret";
String leakedStr = "";
switch(secretStr.charAt(i))
{
    case 'a':
        leakedStr += 'a';
        break;
    case 'b':
        leakedStr += 'b';
        ...
}

No “explicit” data flow!
```
Phosphor: API

Getting and setting tags on objects

Interface TaintedWithObjTag.class
public Taint getPHOSPHOR_TAG();
public void setPHOSPHOR_TAG(Object o);

Getting and setting tags on primitives

MultiTainter.class
public static Taint getTaint(<Primitive Type> c);
public static float tainted<Primitive Type>(<Primitive Type> f, Object tag);

Getting relationships between tags

Class Taint.class
public LinkedList<Taint> getDependencies();
public Object getLabel();
Evaluation

• Soundness & Precision
• Performance
• Portability
Soundness & Precision

• DroidBench - series of unit tests for Java taint tracking
  • Passed all except for implicit flows (intended behavior)
Macrobenchmarks

Phosphor Relative Runtime Overhead (Hotspot 7)

Average now: 34% (from 53%)
## Portability

<table>
<thead>
<tr>
<th>JVM</th>
<th>Version(s)</th>
<th>Success?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle (Hotspot)</td>
<td>1.7.0_45, 1.8.0_0</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenJDK</td>
<td>1.7.0_45, 1.8.0_0</td>
<td>Yes</td>
</tr>
<tr>
<td>Android Dalvik</td>
<td>4.3.1</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Harmony</td>
<td>6.0M3</td>
<td>Yes</td>
</tr>
<tr>
<td>Kaffe VM</td>
<td>1.1.9</td>
<td>Yes</td>
</tr>
<tr>
<td>Jikes RVM</td>
<td>3.1.3</td>
<td>No, but may be possible with more work</td>
</tr>
</tbody>
</table>
Phosphor

- MIT License on GitHub
- Already adopted as a tool by 3+ independent research groups across the world (UW, UCLA, U Lisbon)
- Actively maintained beyond OOPSLA publication
  - Array handling as presented now is much simpler than in 2014
  - Tags can be anything (not just ints)
  - Preliminary support for implicit flows
Data Flow Analysis for Data Security Testing
Data Flow Analysis for Data Security

Inputs

Sensitive Data From Device
Ex. GPS, IMEI
Building a Dynamic Analysis Toolkit
Checkpoint/Rollback

• What is it?
  • Checkpoint an app captures its state
  • Rollback an app returns it to that same state

• Why?
  • Fault Tolerance
  • Input generation/state exploration
  • Speculative Execution
Checkpoint/Rollback

```java
public class Sample {
    static int i;

    public static void main(String[] args) {
        int k = 10;
        foo(k);
        Checkpointer.rollback();
    }

    public static void foo(int in) {
        Sample.bar(in, i);
    }

    public static void bar(int in) {
        i = in;
        Checkpointer.checkpoint();
    }

    public static void Sample.main args, k
    Sample.foo in
    Sample.bar in, i
    Checkpointer.checkpoint()
}
```
Checkpoint/Rollback

- Often implemented with page fault handlers to lazily copy an app's code, heap, data, and files.

```java
public class Sample {
  static int i;
  public static void main(String[] args) {
    int k = 10;
    foo(k);
    Checkpointer.rollback();
  }
  public static void foo(int in) {
    bar(in);
  }
  public static void bar(int in) {
    i = in;
    Checkpointer.checkpoint();
  }
}
```

Active Stack Frame:
- Sample.main
- Sample.foo
- Sample.bar

Checkpoint
public class Sample
{
    static int i;
    public static void main(String[] args)
    {
        int k = 10;
        foo(k);
        Checkpointer.rollback();
    }
    public static void foo(int in)
    {
        bar(in);
    }
    public static void bar(int in)
    {
        i = in;
        Checkpointer.checkpoint();
    }
}
How to implement Checkpoint/Rollback?

• Expensive to capture all of this data!
• Lazy approach: use page fault handlers - fork the process, trap whenever process tries to write a page, make a copy of the page so you can restore later
• Still expensive!
• Annoying to do in complex runtimes (e.g. JVM)…
Checkpoint/Rollback + JVM

- Prior work hooked in between a modified JVM’s garbage collector and some system functionality for page faults
- Not portable, still slow (what if objects are sparsely populated amongst pages)
Checkpoint/Rollback + JVM

• Alternative idea: why not copy individual variables?
• Avoids copying JVM state (which we can assume we can reproduce other ways)
• How do you decide to copy an individual variable though?
Checkpoint/Rollback + JVM

```java
int field;
public void foo(){
    int x = field;
}
```

Add this before every field access?

- **Slow!**

Traverse the whole heap to set shouldCopy?

- **Slow!**

```java
int field;
boolean shouldCopy;
boolean shouldRestore;
public void foo(){
    if(this.shouldCopy)
        makeCopyOfThis();
    else if(this.shouldRestore)
        restoreThisFromCopy();
    int x = field;
}
```
CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs

Jonathan Bell and Luís Pina
George Mason University

ECOOP 2018
Key Insight

Enabling insight:

Class Foo

Class Bar

o1  o2  o3  o4

o5  o6  o7  o8
CROCHET

• Uses this key insight (that we can change the type of objects in the JVM on-the cheap) to perform a **lazy heap traversal** to visit all objects, only as they are accessed by the app

• Much like page faults

• Adds a method, `$$\text{access}()$` to all objects, calls it before any read or write to the object

• `$$\text{access}()$` is an empty method, and gets optimized away in common case

• As it reaches new objects, changes type of the frontier of objects to define `$$\text{access}()$` to self-propagate
Lazy Heap Traversal

Starting state: no objects traversed

Step 1: Find all roots, change them into a \( \text{proxy} \)

Step 2: Once you dereference a \( \text{proxy} \), it turns everything it can directly reference into a \( \text{proxy} \) and disables itself
Lazy Heap Traversal

- Guarantees propagation just before any edge of the graph is traversed
- As we transition each object, we copy its fields too (checkpoint) or restore them (rollback)
- Patch around reflection, Unsafe, static fields
- And it works for stack frames too!
Java Stack Smashing

- Enter: JVMTI - the JVM Tooling Interface
- Used to implement debuggers
- Can use to fetch stack trace, pop stack frames, get/set local variables
Java Stack Smashing

Checkpoint code:

```java
//Original Code
void someFunc(int i, int[] ar)
{
    int j = i + 1;
    ar[i] = ar[i] - j;
    j--;
    otherFunc(j, ar); //Checkpoint is called by otherFunc
    ar[i] = 10;
}

//Checkpoint code
void someFunc(int i, int[] ar)
{
    boolean captureStack = false;
    int j = i + 1;
    ar[i] = ar[i] - j;
    j--;
    otherFunc(j, ar);
    if (captureStack)
        Checkpointer.captureStack();
    ar[i] = 10;
}
```
Java Stack Smashing

//Original Code
void someFunc(int i, int[] ar)
{
    int j = i + 1;
    ar[i] = ar[i] - j;
    j--;
    otherFunc(j, ar);  //Checkpoint is called by otherFunc
    ar[i] = 10;
}

//Rollback code
void someFunc(int i, int[] ar)
{
    int j;
    boolean captureStack = false;
    if(Rollbacker.doRollback())
    {
        i = Rollbacker.localInt();
        ar = Rollbacker.localIntArray();
        j = Rollbacker.localInt();
        Rollbacker.removeRollbackCode();
    }
    else
    {
        //Original Code up-to otherFunc()
    }
    ar[i] = 10;
}
Java Stack Smashing

• We can use this to **completely** reproduce the stack trace and state of a thread
• Not shown: we grab everything sitting on the operand stack too!
• Not shown: we grab all of the monitors that each thread holds too!
Overhead on DaCapo, no checkpoint

Average: 6%!
Time to checkpoint

<table>
<thead>
<tr>
<th>Relative Checkpoint time</th>
<th>CHashMap100</th>
<th>HashMap100</th>
<th>LHashMap100</th>
<th>TreeMap100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crochet</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Crochet, Checkpoint</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Serialize</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

4 Microbenchmarks, each traversing a map with 100,000 entries
Future Work

• Short term:
  • Hybrid Static-Dynamic Dataflow Analysis
  • Using Phosphor for other analyses (e.g. tracking Path Constraints)
  • Abusing JVM for checkpoint-rollback WITH Phosphor
Tools and Approaches to Make Reliable Software Easier
Dependency Analysis for Debugging and Behavioral Clones

```java
static void setUpFullSourceWorkspace(boolean large) throws Exception {
    // Get wksp info
    IWorkspace workspace = ResourcesPlugin.getWorkspace();
    final IWorkspaceRoot workspaceRoot = workspace.getWorkspaceRoot();

    String targetWorkspace = workspaceRoot.getName();
    // Modify resources workspace
    // running them
    IfclipsePreferences projectTable = workspace.getPreferences()
        .getProjectTable(Collections.<K,V> synchronizedMap otra);

    File wkspDir = new File(targetWorkspace);
    FullSourceProjectsFilter filter = new FullSourceProjectsFilter()
        .directories = wkspDir.listFiles(filter);
    int dirLength = directories == null ? 0 : directories.length;
```
Runtime Monitoring of Deployed Software

![Problem Report for Skype](image)

Skype quit unexpectedly.

Click Report to restart the application again. The report will be sent to Apple automatically.

- **Comments**
  
  Provide any steps necessary to reproduce the problem.

**Problem Details and System Configuration**

- **Process:** Skype.exe
- **Path:** /Applications/Skype.app/Contents/MacOS/Skype
- **Version:** 5.7.0.206
- **Time:** 2012-03-22 12:14:40.988
- **User ID:** 302
- **Report:** 100

**Crashed Threads:**

- 1. Dispatch queue: com.apple.NSThreadPool
- 2. Dispatch queue: com.apple.NSThreadPool

**Stack Trace:**

1. Thread 1 (Dispatch Queue: com.apple.NSThreadPool)
   
2. Thread 2 (Dispatch Queue: com.apple.NSThreadPool)
   
3. Thread 3 (Dispatch Queue: com.apple.NSThreadPool)
   
4. Thread 4 (Dispatch Queue: com.apple.NSThreadPool)
   
5. Thread 5 (Dispatch Queue: com.apple.NSThreadPool)
   
6. Thread 6 (Dispatch Queue: com.apple.NSThreadPool)
   
7. Thread 7 (Dispatch Queue: com.apple.NSThreadPool)

**Expand Details**
My Collaborators

Faculty and PhD students at:
- Columbia
- UIUC
- CMU
- Penn
- UCLA
- UVA
- Saarland
- Padderbor
- Potsdam
- ElectricCloud, Inc

Plus: Masters Students
- Zhimao Liu
- Namo Lu
- Mandy Wang
- Winnie Narang
- Nikhil Sarda
- Ethan Hann
- Jason Halpern
- Evgeny Fedetov
- John Murphy

Plus: Undergraduate Students
- Sravya Kalva
- Pakeezha Arfany
- Monica Jeyasan
- Jonathan Barrios
- David Rincon-Cruz
- Emilia Pakulski
- Alana Ramjit
- Jennifer Lam
- Xingzhou Derek He
- Sidharth Shanker
- Miriam Melnick
- Alison Yang