Panel Presentation at the International Workshop on Program Debugging (IWPD 2017)

col·laced with ISSRE 2017

October 23, 2017

Artur Andrzejak

Heidelberg University

http://pvs.ifi.uni-heidelberg.de/
Panel Questions

A. What are the future and current niches of automated debugging?

B. When can we expect automated debugging being available for all programmers?
A. Future & Current Niches

• *Niches*? Why is automated debugging not *common*?

• Major practical problems:
  1. Insufficient precision for large code
  2. Limited support for bug understanding
  3. Still many, many projects without SW tests
    • ... and even forget about requirements!

What are the niches which tolerate 2 & 3?
Properties of Niches

1. No need for deep bug understanding
   – If the suspicion location tells you almost everything
   – I.e. it is relatively easy to verify and correct defect only from location information

2. Lack of SW tests is not a problem
   – If software tests are given implicitly
   – I.e. a test oracle is represented by an exception, (performance) malfunction, etc.
Potential Niches

• A. Configuration option bugs (triggering exceptions)
  – Pointing to fault-triggering config. option frequently shows the cause
    • E.g. path settings, memory size, window location, ..

• B. Memory leaks in Java
  – Identifying allocation site(s) which creates leaks allows to find obsolete references quickly
  – Test oracle: memory growth & failure

• C. Vulnerability faults
  – Test oracle: successful pen. tests or code crashes
  – Bug understanding: still needed
A. AUTOMATED DEBUGGING OF CONFIGURATION ERRORS

Results from:

Impact of Miconfigurations

• Causes of reported severe failures in a commercial storage app with 1000s of users (SOSP’11)¹

• Google incident in 2013²
  – “From 5:00 a.m. to 8:00 a.m. PT, some users received errors when trying to access Gmail, Drive, Talk, Google Sync, […] Groups, Sites, and Contacts.”
  – “A misconfiguration […] caused a fraction of the login requests to be […] concentrated on a relatively small number of servers.”

¹: An Empirical Study on Configuration Errors in Commercial and Open Source Systems by Zuoning Yin et al. (SOSP’11)
²: Google Apps Incident Report, April 17, 2013  http://goo.gl/5NPZeq
Approach Overview

- Output: a ranked list of suspicious options
- Static analysis
- No need to instrument, re-run or annotate code
- Inputs:
  1: Source code + positions of option read statements
  2: Error stack trace (only runtime data)
General Idea

- Can an execution path from A to B exist?
- Analyze for each option $O_k$, $k = 1, \ldots$, and rank options
- What is a “failure site”??
# Accuracy Evaluation

<table>
<thead>
<tr>
<th>App.</th>
<th>Id</th>
<th>Error Description</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>JChord</td>
<td>1</td>
<td>No main class is specified</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>No main method in the specified class</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Running a nonexistent analysis</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Invalid context-sensitive analysis time</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Printing nonexistent relations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Disassembling nonexistent classes</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Invalid type of reflection</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Wrong classpath</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>No testclass is specified</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Invalid type of output cases</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Value of alias-ration is out of bounds</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>No method list is specified</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>The tested method has missing args.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>Incorrect name of the tested method</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>Invalid symbols in name of output dir</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>File name contains invalid symbols</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>App.</th>
<th>Id</th>
<th>Error Description</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17</td>
<td>Carriage return at the end of URL</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>Old data dir after formatting namenode</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>Wrong host name of master node</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Usage of http instead of hdfs in URL</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>The storage dir of namenode not readable</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>Missing the &lt;property&gt; tags</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>Info port is in use by other process</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>Missing port in the URL</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>App.</th>
<th>Id</th>
<th>Error Description</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
<td>Wrong port of the rootdir URL</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>Wrong host name of the rootdir URL</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>No permission of the data directory</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>HMaster port is occupied</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>Wrong port of ZooKeeper</td>
<td>1</td>
</tr>
</tbody>
</table>

Comparable to state-of-the-art but without dynamic analysis
A Web Service

• ConfDebugger + automated finding of option read points => a fully automated diagnosis system

• In preparation: a public service

• Users need to submit stack trace and specify application

• Try it out:

https://pvs14.ifi.uni-heidelberg.de/
B. DETECTION AND ISOLATION OF MEMORY LEAKS

Work with

Diego Costa, M. Ghanavati, Felix Langner, (Univ. Heidelberg), Fumio Machida (NEC Labs, Japan), Rivalino Matias (Univ. Uberlandia, Brazil), Kishor Trivedi (Duke Univ., USA)
Memory Leaks

- **Memory leaks**: allocated memory objects which will be never used again (until process terminates)

**Leaks in C:**

```c
int *pointer;
pointer = malloc(4);
...
pointer = malloc(4);
free(pointer);
/* 1st memory block not released => leak */
```

**Leaks in Java:**

```java
private static List
    mylist = new ArrayList<>();

for (int i=1; i<10000; i++)
    mylist.add(i);
/* mylist not cleared => leak */
```
Artur Andrzejak

Research Interest

Papers/theses on memory leak diagnosis since 2003 (major venues)
Is this an Important Problem?

• Among 2500+ surveyed Java applications, 47% contained a memory leak\(^1\)

• Firefox browser: 40+ memory leaks reported in first quarter of 2014\(^2\)

• Amazon AWS (“cloud”): In 2012, an incomplete DNS update “triggered a latent memory leak bug in the reporting agent on the storage servers [...] of EBS”\(^3\)
  – 100’s of commercial customers experienced degraded performance for hours

---

Exploiting Version Differences

- The main idea is a **differential approach**: compare consecutive development versions
  - Essentially anomaly detection vs. previous version(s)
Isolating Causes of Memory Leaks

- We instrument all allocation sites, i.e. `obj = new Class()`
- We monitor object allocations and destructions via an agent called Live Object Monitor (LOM)
- ... Grouped into allocation families, each corresponding to a single allocation site
Comparing Snapshots between Versions

Idea: compare resource allocation between SW versions

Software version $i$ ↔ Software version $i+1$

- $new_1$
  - allocated
  - released
- $\ldots$
- $new_k$
  - allocated
  - released
- $\ldots$
- $new_n$
  - allocated
  - released

Problem!
Artur Andrzejak

Sample Results on Leak Isolation

Output: ranked list of suspicious allocation sites

<table>
<thead>
<tr>
<th>Application</th>
<th>Issue</th>
<th>Non-Leaky Version</th>
<th>Leaky Version</th>
<th>Rank of Defect Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>8632</td>
<td>0.20.0</td>
<td>2.0.0a</td>
<td>1</td>
</tr>
<tr>
<td>HDFS</td>
<td>5671</td>
<td>2.0.6</td>
<td>2.2</td>
<td>157</td>
</tr>
<tr>
<td>Yarn</td>
<td>1382</td>
<td>0.23.11</td>
<td>2.2</td>
<td>125</td>
</tr>
<tr>
<td>Thrift</td>
<td>1468</td>
<td>0.4.0</td>
<td>0.50</td>
<td>Not listed</td>
</tr>
<tr>
<td>Snappy</td>
<td>91</td>
<td>1.1.1.3</td>
<td>1.1.1.4</td>
<td>1</td>
</tr>
<tr>
<td>Solr</td>
<td>1042</td>
<td>1.2.1</td>
<td>1.3</td>
<td>4</td>
</tr>
<tr>
<td>Nutch</td>
<td>925</td>
<td>0.8</td>
<td>1.2</td>
<td>9</td>
</tr>
</tbody>
</table>

- Only few unit tests have responsible allocation site
- No unit test has triggered “root cause” site

Observation: Some projects have marginal test coverage of potential leak-inducing allocation sites
Panel Question B

When can we expect automated debugging being available for all programmers?

Automated Debugging

Slicing-based Debugging
- Static Slicing (81)
- Dynamic slicing (88, 93)
- Whyline (08)
- Holmes (08)

Statistical Debugging
- Tarantula (01)
- CBI (03)
- Wong: crosstab analysis-based method (08)

Delta Debugging
- DD (99,01)
- HDD (05)
- Wong (10)

Spectrum-based Debugging
- Renieris (03)

Machine Learning-based Methods
- Brun: for Latent Code Errors (04)
- Briand: Decision tree algorithm (07)
- Wong: back-propagation neural network (09)

Formula-based Debugging
- Darwin (09)
- BugAssist (11)

Data-Mining-based Methods
- Cellier: FCA (08)
- Nessa: N-grams (09)

Model-based Methods
- DeMillo: modeling failure (97)
- Wotawa: dependency model (02)

Sources:
... Or: when will automated debugging be *accepted* by some programmers?

Remember roadblocks for practical usage:

1. Most projects without *SW tests or requirements*
2. Limited support for bug understanding
Some Thoughts About Solutions

• Software test creation
  – Target **middle ground** in the spectrum between “fully automated” and “fully manual”
    • Reduce the manual overhead of test creation instead of automated generation of (largely useless) tests

• Bug understanding
  – Merge with techniques “outside the box”
  – Example: Time Travel Debuggers (TTD)
Automated Fault Localisation and TTD

- Recently, TTD is becoming realistic and really used
  - Java: Chronon, C/C++: undoDB (used at SAP)

- Compare TTD and automated fault localisation (AFL)
  - TTD: Developer points to a potentially suspicious location, and DB helps to quickly understand the execution and find root causes
  - AFL: Shows you some code locations such that there might be something wrong
Complementarity of AFL and TTD

• TTD: Developer points to a *potentially suspicious* location, and DB helps to *quickly* find the root causes
  – Which locations are suspicious? => AFL needed!

• AFL: Shows you *some* code locations such that there might be *something* wrong
  – *Some*: which is most likely, where should I start?
  – *Might be*: lot of thinking needed to confirm or reject
  – *Something*: which variables/values are wrong and why?
  – Even then, the root cause might still need to be found

• => A lot of code comprehension needed for AFL!
Automated Debugging and TTD

• Some technical ideas
  – Suggests breakpoints automatically by AFL
    • ...Then a TTD takes over
  – Instrument only statements of the suspicious code AND updates of all variables seen there
    • => Faster TTD recording
  – Ask users during debugging „is this variable value OK“?
    • => implicit tests; filtering of suspicious locations
Thank you!

QUESTIONS ARE WELCOME