Finding Buggy Functions in Eclipse Crashes using CrashLocator Algorithm

Célia Cacciatore  
celia.cacciatore@gmail.com  
M2 IAGL

Jonathan Geoffroy  
geoffroy.jonathan@gmail.com  
M2 IAGL

December 1, 2014
1 Introduction

“If a program has not crashed yet, it is waiting for a critical moment before it crashes” as says one of the Murphy’s laws. It’s true that all software crash, so tracking and fixing bugs is an important part of software engineering. Big software companies, such as Microsoft[8] or Mozilla[1], have popularized crash reporting as a way to retrieve failures which occur in final releases.

But collecting these data is still not enough; there is also the localization of the buggy function that can be drowned in a large amount of crash traces and duplicated reports. This searching process is often done manually by developers, that’s why this task can be tedious and time-consuming. In order to help making this task more efficient, researchers have created a new algorithm called CrashLocator[9]; it proposes to score the suspiciousness of each function and return the function which is likely to be guilty.

Our goals are validating the CrashLocator results on another dataset, such as Eclipse’s[3], and checking that it can be executed in a reasonable time, that we’ll define later.
2 Crash Analysis

As we said, once developers have retrieved data about a failure in their program thanks to crash reports, there is a phase of crash analysis, where they extract interesting information about the bug before fixing the problem. This phase depends on data contained in crash reports, ways of extracting information and localization of the buggy function.

2.1 Crash Reports

A Crash Report contains information about a crash, which includes the crash trace. A crash trace is the list of called functions when a bug appears. Here is an example of crash trace:

```json
'traces': [
{
   'exceptionType': 'java.io.InterruptedIOException',
   'message': 'Read timed out',
   'elements': [
   {
      'method': 'java.net.SocketInputStream.socketRead',
      'source': 'Native Method'
   },
   {
      'method': 'java.net.SocketInputStream.read',
      'source': 'SocketInputStream.java (Compiled Code)'n   }
   ]
}
```

2.2 Bucketing

Failures in a program can be caused a large number of times, and each one can be transmitted to developers in a crash report. That’s why it’s very important to recognize
which crashes are caused by the same bug. A bucket gathers all crashes which present similarities, and are likely to be the same fault.

There are several ways to group these reports into buckets such as manual, crash point and top-K bucketing which we present now.

2.2.1 Manual Bucketing

The first way is to gather them manually. However, a manual bucketing is not only time-consuming but also an error-prone process. The reason is that it’s done by humans, so the bucketing process depends on their point of view.

The bucketing result can then be inconsistent: all duplicated crash traces aren’t necessarily gathered or two crash traces which linked to two different bugs can accidentally be put in the same bucket.

2.2.2 Crash Point Algorithm

CrashPoint algorithm only considers the crash point (i.e. the last called method before crash) in order to determine if two crash traces refers the same bug. Even this method is easy to develop and quickly executed, software can contains exception handler used to detect crashes. In this case, handlers could appear in many crash points. Their crash traces will be merged in the same bucket, but could refer to different bugs.

2.2.3 Top-K Algorithm

Top-K algorithm is similar to crash point algorithm, but use the k last methods of the crash trace to compare crash traces. In other words, two crash traces are considered as referring to the same bug if the k last methods are exactly the same. In particular, if k equals 1, top-k result is the same as crash point. If k is the size of the crash trace, then only two identical crash traces can be in the same bucket, even though a bug can be caused in many ways.

2.3 CrashLocator

CrashLocator is a method to help developers find the buggy function responsible for the crash by returning the most suspicious methods using the information in crash reports. The authors evaluated their algorithm on Mozilla dataset and found out that they can locate 63.7% of crashing faults by examining the 5 functions recommended by CrashLocator.

The suspiciousness score is obtained by multiplying four factors obtained in the crash reports:

Function Frequency (FF) measures the frequency of a function appearing in crash traces of a specific bucket.

Inverse Bucket Frequency (IBF) measures the inverse frequency of a function appearing in crash traces of other buckets.
Inverse Average Distance to Crash Point (IAD) measures the distance between the function and the crash point in crash traces of a bucket.

Function’s Lines of Code (FLOC) measures the number of lines of code of the function in the source code.

In summary, CrashLocator gives higher scores to functions that appear more frequently in crash traces in a bucket, less frequently in crash traces of other buckets, closer to the crash point and larger in terms of lines of code.
3 Methodology

3.1 NodeCrashLocator

Our application NodeCrashLocator is a NodeJS implementation of CrashLocator which computes the suspiciousness of each method in each crash trace of the dataset and returns the 5 most suspicious methods which are likely to have caused the failure. We check that our percentage of correctness is the same as in the CrashLocator paper (63.7%).

Moreover, we test NodeCrashLocator accuracy on different types of bucketing. In the CrashLocator article, the authors wonder if a better bucketing algorithm could improve crashing fault localization performance, but they only try on manual bucketing on Mozilla dataset. In addition to manual bucketing, we use the crash point and top-K bucketing algorithms on Eclipse dataset and check whether there is a difference or not.

3.2 Used Dataset

We try our NodeCrashLocator on the Eclipse dataset. The Eclipse dataset is composed of 21,915 reports, and manually bucketed by Eclipse developers. This bucketing creates 17,953 different buckets, in which 1,815 contains more than one crash trace. Note that it differs from the Mozilla dataset used in the CrashLocator paper.

3.3 Performance Measurement

We also measure the performances of NodeCrashLocator as it’s designed to be executed on large datasets. Our NodeJS implementation must return the result in a few minutes as it can typically be used to find buggy functions of a dataset at the beginning of a bug correction developement phase. However, for a bigger dataset, it can take more time.

We make the most of the opportunity to compare how bucketing affects NodeCrashLocator performances. We measure the computing time for the three types of bucketing we chose to implement.

We make all the measurements on the same computer so we can easily compare our results. Here is a recap chart describing the machine we use:

<table>
<thead>
<tr>
<th>OS</th>
<th>Ubuntu 14.04.1 LTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Core(TM) i5 CPU M 480 2.67GHz</td>
</tr>
<tr>
<td>CPU cores</td>
<td>2</td>
</tr>
<tr>
<td>CPU cache size</td>
<td>3072 KB</td>
</tr>
<tr>
<td>RAM</td>
<td>4096 MB</td>
</tr>
<tr>
<td>SWAP</td>
<td>8192 MB</td>
</tr>
</tbody>
</table>
4 Results

4.1 Counting Lines of Code

We have written a little program in Java using Spoon[7] to count the number of lines of code for a method, and make it work as a pre-processing part of our program. We first download the Eclipse source code[4], then find our method in Spoon AST and count the number of lines. Our result is the medium length for all methods, but it equals 3 because of a great number of getter and setter methods, so it won’t be a discriminative factor.

4.2 Bucketings

We first have written three programs whose roles are to divide our Eclipse into buckets using manual bucketing, Crash Point and Top-K algorithms. Our Crash Point algorithm is the same as Top-K where k = 1.

Here you can find a table summing up what we obtained with all kinds of bucketing:

<table>
<thead>
<tr>
<th>Bucketing algorithm</th>
<th>Number of buckets</th>
<th>Number of buckets with more than one crash trace</th>
<th>Medium number of crash traces per bucket</th>
<th>Timing (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual bucketing</td>
<td>17,953</td>
<td>1,815</td>
<td>2</td>
<td>3,370</td>
</tr>
<tr>
<td>Crash Point</td>
<td>8,337</td>
<td>2,487</td>
<td>6</td>
<td>2,948</td>
</tr>
<tr>
<td>Top-K (k = 5)</td>
<td>13,962</td>
<td>2,767</td>
<td>4</td>
<td>3,015</td>
</tr>
<tr>
<td>Top-K (k = 10)</td>
<td>16,551</td>
<td>2,504</td>
<td>3</td>
<td>3,056</td>
</tr>
</tbody>
</table>

As we can immediately see, the number of buckets varies a lot and can double depending on the bucketing algorithm. But when it comes to buckets with more than one crash trace, which are the ones we are interested in, we get approximately the same amount of buckets, except for the manual bucket which is a little lower. But even if this amount is similar, there are approximately three times more crash traces in the Crash Point buckets than in Eclipse developers’ buckets. So the number of buckets we take into account is quite the same, but their content differs greatly.

Note that the time our programs take to perform the bucketing is quite the same for all algorithms.
4.3 NodeCrashLocator

We now have to apply the CrashLocator algorithm to compute the suspiciousness score of a method. Once we have divided our dataset into buckets, we just need to collect the needed data, use them to compute the different factors, multiply them and obtain the awaited score. Our program could be summarized as follows:

```plaintext
// Table which memorize all data about each method
table information;

// Getting all needed data
for each bucket in dataset, do:
    for each crashTrace in bucket, do:
        for each method in crashTrace, do:
            information[bucket,method].data = getData(bucket, method, crashTrace)
        done
    done
done

// Compute each factor from data and obtain the final score
for each bucket in buckets, do:
    for each method in methodsInformation, do:
        FF = scoreFF(data)
        IBF = scoreIBF(data)
        IAD = scoreIAD(data)
        FLOC = getFLOC() // we got it in pre-processing when counting the lines of code
        information[bucket, method].score = FF * IBF * IAD * FLOC
    done
done

// Return the score of each method
for each method in methodsInformation, do:
    display(methodsInformation[method].score);
done
```

We are here parsing our files in dataset only once to get all the data we need and then use them to compute each factor, which is much more efficient than searching for information for each factor. We also take advantage of NodeJS abilities to parse easily JSON files and to parallelize each processing we needed, which means each loop is performed asynchronously in order to improve our performances.
### 4.4 Accuracy measurement

NodeCrashLocator returns for each bucket the 5 most suspicious methods. Our results show that in most buckets, the most suspicious method has a far higher score than the others. For example, if we take the bucket number 170 determined by Eclipse developers, we get the following results:

```plaintext
{'170':
    {'java.net.InetAddress.getAllByName': 41.613182808403714,
     'java.net.SocketInputStream.socketRead': 2.600823925525232,
     'java.net.Socket.<init>': 2.1800509742296446,
     'java.net.InetAddress.getByName': 2.080659140420186,
     'java.net.PlainSocketImpl.socketConnect': 1.9275660382590345}
}
```

Next step is to check the accuracy of our results, which should be around 63.7% according to the paper, by looking at the fixes that correct the problem. For a given bucket, we first get the bugId mentioned in the crash traces and look for it on Eclipse official bug referencing site[2]. We then find the methods which were modified in the corresponding commit and check if they are the same as the ones NodeCrashLocator returned. The problem is that in a commit, many methods were corrected, method names were changed and sometimes even entire classes were modified, which makes it quite difficult to say whether our results and Eclipse modifications concern the same methods or not. Plus, as it is manual checking, we can never be sure even if we find the same name of method in our results and in Eclipse commit that it was really the one whose corrections made the bug disappear. That’s why we can’t conclude about NodeCrashLocator’s percentage of accuracy.

We also wondered about how bucketing could impact the suspiciousness score of methods. In fact, our results show that from one bucketing to another, the scores vary a lot. For example, if we take the Crash Point bucketing and search for the most suspicious method we found with manual bucketing, we get these results:

```plaintext
{'979e25b1e203b9c72d816499991d3bba3129d247e':
    {'java.net.InetAddress.getAllByName': 390.231382739731,
     'java.net.InetAddress.getByName': 4.434447531133308,
     'java.net.Socket.<init>': 1.9616577995141846,
     'java.net.InetAddress.getAllByName': 1.5485723309417,
```

First, as buckets have not the same name, we can’t be sure they refer to the same bug, but we can still suppose so as they are the only ones corresponding in each bucketing. Second, we can see that the same method is the most suspicious, which seems
to be coherent. But this time, the score is approximatively 10 times what we obtained before! If this method is really the buggy one, then the Crash Point bucketing tends to be more efficient in locating faulty functions. This observation can also be made with top-k algorithm:

```
'62e66cd15249f32ef3046e98ec7178fba5117980':
  {'java.net.InetAddress.getAllByName':
    120.32637480731799,
    'java.net.InetAddress.getByName': 4.2973705288327855,
    'java.net.Socket.<init>': 2.551859708039284,
    'org.eclipse.vcm.internal.core.ccvs.client.PServerConnection.createSocket':
    2.313968746294577,
    'org.eclipse.vcm.internal.core.ccvs.client.PServerConnection.open':
    1.8800996063643436}
```

In fact, for each bucket we can identify as the same, Crash Point has the higher scores and manual bucketing has the lowest ones. Crash Point bucketing is also the one with larger buckets, which can have influenced the suspiciousness score. So we can conclude that bucketing helps determine the buggy function.

4.5 Performance Measurement

We explained before that NodeCrashLocator should return results in a reasonable time. We estimated that few minutes are acceptable as it can concern a lot of crash reports. We chose NodeJS as it’s a language designed to interact with large amounts of data, so we expect our performances to be under the limit we set.

Here is a table summarizing the times we obtained with each type of bucketing:

<table>
<thead>
<tr>
<th>Bucketing algorithm</th>
<th>Computing time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual bucketing</td>
<td>3,370</td>
</tr>
<tr>
<td>Crash Point</td>
<td>2,948</td>
</tr>
<tr>
<td>Top-K (k = 5)</td>
<td>3,015</td>
</tr>
<tr>
<td>Top-K (k = 10)</td>
<td>3,056</td>
</tr>
</tbody>
</table>

First, we can notice that computing is made in a few seconds, which is far beyond what we hoped to obtain with Eclipse dataset. Even if we add the bucketing phase, we are still under the limit we set. NodeCrashLocator is so efficient enough to be used without wasting time waiting for results.

Second, we can see little difference between the times we got for the three bucketing algorithms, so the number and length of buckets have barely any impact.
5 Limits and Perspectives

We wanted to check CrashLocator’s efficiency on another dataset than the one mentioned in the paper, but we couldn’t make sure we obtained the same results as the fixes weren’t easy to analyze. It could be interesting to try again the algorithm on another dataset and collaborate with developers to check the accuracy of CrashLocator in locating the buggy functions. Moreover, it would be great if it was possible to automatically check these results, for example by associating NodeCrashLocator with bug reporting systems.

We also counted the number of lines of functions in a pre-processing program, which means this wasn’t taken into account when measuring the time of our algorithm. It would be better to find a way to locate a method in the source code and count its number of lines directly inside our program. But we still would have to make sure we only count instructions, as that’s currently the case in our Java and Spoon processor.

We could try to measure the performances of NodeCrashLocator on bigger datasets too.

Finally, there are some tracks to improve CrashLocator that are mentioned in the article, such as expanding the stack traces which are often incomplete. This could also be an improvement of NodeCrashLocator.
6 Conclusion

We based our program on CrashLocator, a method developed by researchers whose goal is to locate the function which is likely to be responsible for a crash using information in crash reports. We chose to focus on computing of suspiciousness score of methods in a bucket. We made experiments with three types of bucketing to check whether it could affect NodeCrashLocator results or not.

Our goal was to check CrashLocator accuracy and measure its performances on another dataset than the one the authors used. We obtained the 5 most suspicious methods for each bucket but couldn’t make sure they were the faulty function which were modified to correct the bug, so we can’t make sure the algorithm can find 63.7% of buggy methods on other datasets. However, we were able to get very good performances in parsing, bucketing and computing scores with our NodeJS version of CrashLocator, so it seems this method could be efficiently used by developers in real life.

It would still be a good idea the check the results of CrashLocator on other datasets, potentially bigger ones, and collaborate with developers to see if the results we obtain are really workable.
References
