CrashAutomata: An Approach for the Detection of Duplicate Crash Reports Based on Generalizable Automata

Neda Ebrahimi Koopaei
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montréal, QC, Canada
n_ebr@encs.concordia.ca

Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montréal, QC, Canada
abdelw@ece.concordia.ca

ABSTRACT

Crash reporting systems are useful tools that allow users to report system failures, subsequently contacting the appropriate support group for resolution. As a software system grows and becomes more versatile, the number of crashes increases. A large software company receives typically thousands of crashes a day, which make it difficult for software engineers to address these reports in a timely manner. Fortunately, not all reports are new; many of them are duplicates of previously reported crashes. Research has shown that early detection of duplicate reports can reduce the effort and time it takes to handle crash reports. In this paper, we propose a new approach for detecting duplicate crash reports, called CrashAutomata. CrashAutomata builds a model from historical crash reports (more precisely their stack traces) that is used to classify an incoming report. The model is based on varied-length n-grams and automata. Unlike existing techniques, CrashAutomata takes advantage of the generalization aspect of automata, making it possible to build a representative model of crash reports, reducing the number of false positives. When applied to crash reports of the Firefox system, CrashAutomata results in very high precision and recall. It also outperforms CrashGraph, a leading technique in the detection of duplicate crash reports.

Keywords

Mining bug repositories, Duplicate bug reports, Automata-based Modeling, Software maintenance.

1. INTRODUCTION

Most large software systems are equipped with a crash reporting tool that generates automatically a crash report when a field failure occurs. Examples of such tools include WER (Windows Error Reporting) [15], Apple Crash Reporter [3] and Mozilla Crash Reporter [17]. These tools can automatically collect a variety of data about the crash including stack traces, memory dumps, etc. This data is sent to a crash reporting server and processed by the developers of the system in order to provide fixes.

Analyzing crash reports, however, can be a tedious task. This is because of the large number of reports that are submitted every day, especially for software systems with a large user base. Fortunately, not all these bugs are unique. Studies have shown that many reported bugs are duplicates—the same fault causes different running instances of the same system to crash [4, 14, 21]. Consider, for example, a popular web browser such as Firefox. This system is used by millions of users. The same bug can trigger many of the running instances of Firefox to fail. If a crash report is submitted for each failure then the Firefox development team will be swamped with reports. Research has shown that detecting duplicate crash reports can significantly reduce the time and effort spent on triaging and handling crashes [9, 24, 29]. This led to many studies that aim to detect automatically duplicate crash (or bug) reports. These techniques treat the problem as a classification problem by building a model from historical bug reports, using machine learning techniques. The model is used to classify incoming bugs.

Existing techniques for detecting duplicate reports can be divided into two categories. The first category encompasses techniques that rely solely on the description of the reports [1, 9, 18, 24–27]. The problem with the description is that it is expressed in natural language and as such it tends to be informal and hence not quite reliable. To address this, researchers have turned to more formal crash report data such as stack traces [7, 8, 14, 29]. These techniques model information in historical stack traces and use the resulting model to classify incoming reports. Perhaps, one of the most effective approaches is CrashGraph. Introduce by Kim et al. [14], CrashGraph is used to detect duplicate crash reports in WER (Windows Error Reporting System). The approach aggregates multiple stack traces in the same group by constructing a graph where the nodes represent the stack trace functions and the edges represent the calling relationship. When applied to crashes of two Windows products CrashGraph achieves 71.5% precision and 62.4% recall [14].

The problem with CrashGraph and similar approaches is that they generate a training model that is too rigid, making them difficult to generalize to unseen cases, which explains the low recall. What we mean by generalization is the ability to model stack traces while considering possible calls that are not necessarily in the training set. Take for example the following fictive stack trace ABCDED used for training. A model that represents this trace should classify traces ABABCDED or AAABCDED as similar because they only differ from ABCDED due to contiguous repetition of AB and A respectively. Contiguous repetitions can be due to loops in the program. They did not appear in the stack trace used for training just because the loop was not exercised during the scenario that led to the crash. We will see in the rest of the paper that generalization goes beyond considering just contiguous repetitions.

In this paper, we propose an approach, called CrashAutomata that uses a combination of varied-length n-grams and automata to model stack traces. CrashAutomata is inspired by the work of Jiang et al. [10]. The authors developed an algorithm for anomaly detection that can be generalized to unseen cases by controlling a variable α. We adopted the algorithm to model stack traces and detect duplicate reports. We experimented with various values of α to determine the most suitable value that yields best detection accuracy. We also compared CrashAutomata with CrashGraph. The results show that our approach has a better recall than CrashCrash while keeping the same precision.

The remaining parts of this paper are organized as follows: In Section 2, we present background information on crash reports and the Mozilla crash reporting system, used in this study. In Section 3, we present CrashAutomata. In Section 4, we evaluate the effectiveness of CrashAutomata when applied to stack traces of the Firefox system. We also compare CrashAutomata to CrashGraph. Section 5 discusses threats to validity, following by related work. We conclude the paper and sketch future directions in Section 7.

2. CRASH REPORTS AND STACK TRACES IN MOZILLA

A typical crash report contains a crash signature, a description, a submission date, a product number, a product version, the operating system version, and a stack trace. WER (Windows Error Reporting) [15], Apple Crash Reporter [3] and Mozilla Crash Reporter [17] are good examples of widely deployed crash reporting systems. Figure 1 gives an overview of how a crash reporting system works. When a crash occurs in a software system, a user submits a crash report for troubleshooting (note that a crash report can also be generated automatically depending on the how the settings of the system). A crash reporting system receives the crash reports and groups them based on how similar they are. This way, when a new crash arrives, it can be assigned to the same developers who fixed similar crashes. The overall goal is to speed up the process of handling crashes.

The grouping technique varies from one crash reporting system to another. WER, for example, uses more than 500 proprietary heuristics for organizing crash reports into buckets [7, 13]. Mozilla, the crash reporting system used in this paper, groups crash reports based on stack trace information. More particularly, it uses the last function that was executed when the crash occurred as the main similarity criterion.

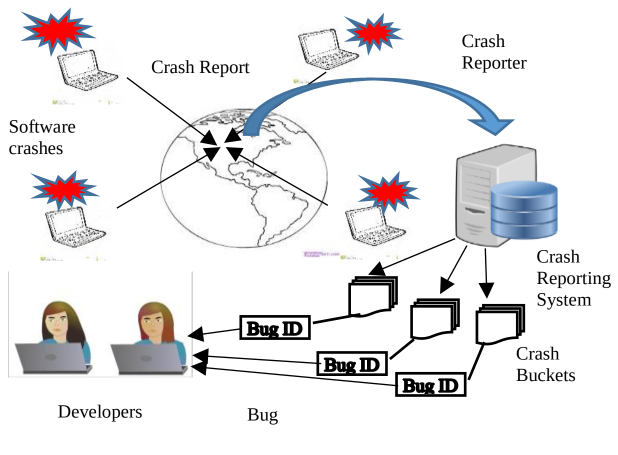


Fig. 1. An overview of how a crash reporting system works

Table 1 shows an excerpt of a stack trace in one of Mozilla's crash reports. The stack trace contains five frames; each contains information about the executed function. A frame signature is defined as the method signature (composed of the function name and the module name). In this example, the program crashed at Frame 0 (function signature mozglue.dll/mozalloc_abort). More formally, a stack trace can be defined as an ordered set of frames, F={f_0, f_1,…, f_n}, where n is the number of functions in the stack trace and f_0 is the top frame of the stack trace.

Crash reports are examined by a team of triagers who decide whether the crash is valid or not. Triagers turn valid crashes into bug reports, assign to them a priority (severity) level, and assign them to the appropriate developers. In the rest of the paper, we use the terms bug reports and crash reports interchangeably.

It should be noted that Mozilla's bucketing method may result in many duplicate reports being spread over multiple buckets, just because the top frames of their stack traces are different. Even worse, many unrelated crashes may be grouped together, which defeats the purpose of having a bucketing system in the first place. This awkward bucketing can cause further problems for developers when attempting to understand the fault by looking at crash traces that may in fact be unrelated.

Table 1. An example of a bucket in Mozilla crash reports

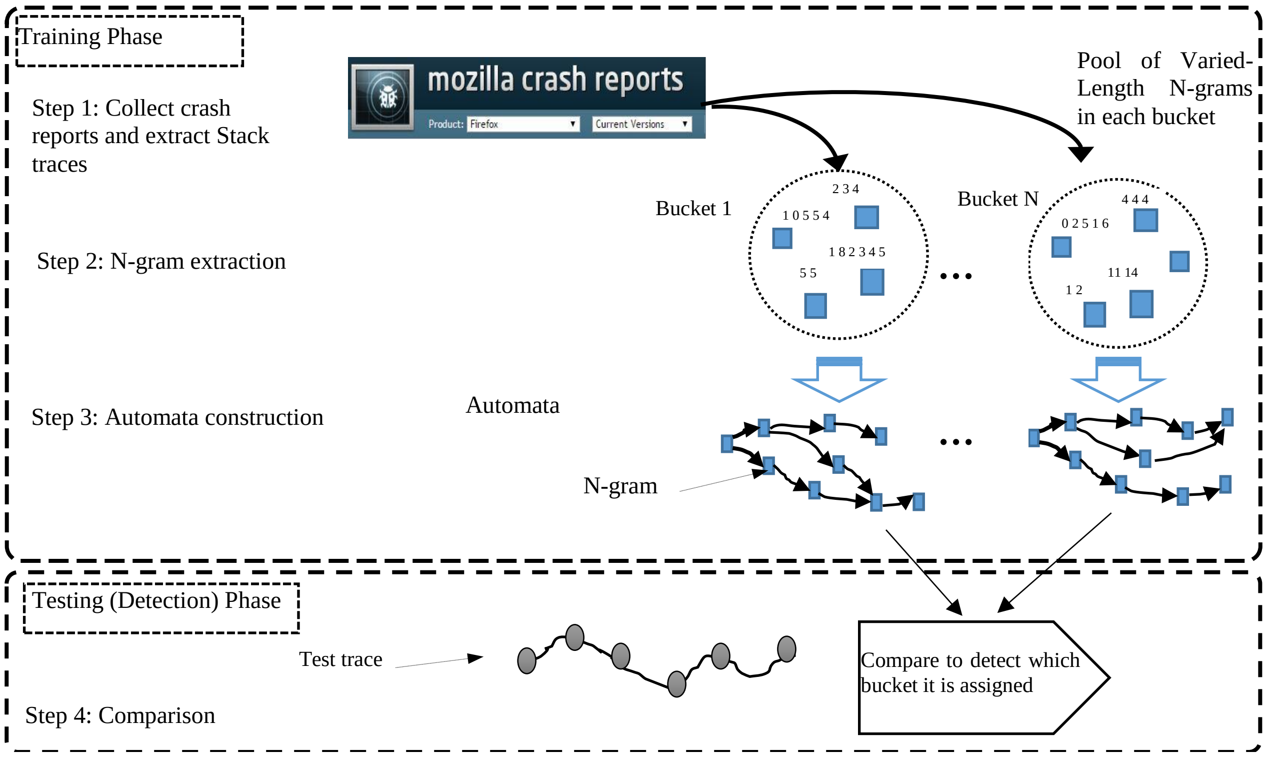
Frame	Module	Signature
0	mozglue.dll	mozalloc_abort(char const* const)
1	mozglue.dll	mozalloc_handle_oom(unsigned int)
2	mozglue.dll	moz_xmalloc
3	xul.dll	mozilla::net::CacheFileMetadata::WriteMetadata(unsigned int, mozilla::net::CacheFileMetadataListener*)
4	xul.dll	mozilla::net::CacheFile::WriteMetadataIfNeededLocked(bool)
5	xul.dll	mozilla::net::CacheFile::DeactivateChunk(mozilla::net::CacheFileChunk*)

3. CRASHAUTOMATA APPROACH

Figure 2 shows an overview of our approach. The approach is divided into two phases: training and testing (detection) phases. In the training phase, we use historical crash reports to build a model that characterizes the information contained in stack traces of duplicate bugs. This phase is further divided into multiple steps. The first step consists of collecting crash traces from the Mozilla Crash Reporter and restructuring them to form valid grouping of duplicate bugs. In the second step, we extract varied-length n-grams from stack traces of each bucket. These n-grams will be used to construct an automaton for each bucket. To control the level of generalization of the automaton, we introduce a variable α, which regulates the number of n-grams that are extracted. We will describe this process in Section 3.2.1. The testing phase consists of assessing the effectiveness of the model in classifying unseen crash reports.

3.1 Training Phase: Collecting Stack Traces and Creating Buckets

As discussed earlier, Mozilla crashes are grouped based on the top frame signature (the last invoked function) of their corresponding stack traces. As a result, many duplicate crashes may end up in different buckets. This said, duplicate bug reports that are generated from these crash reports may also be in different buckets. To test the effectiveness of our approach, we need to have stack traces of all duplicate bugs in the same bucket. To achieve this, we simply reorganize Mozilla's buckets by bringing together stack traces that correspond to the same bugs into one bucket.



Training Phase
Step 1: Collect crash reports and extract Stack traces
mozilla crash reports
Product: Firefox
Current Versions
Pool of Varied-Length N-grams in each bucket
Bucket 1
2 3 4
1 0 5 5 4
1 8 2 3 4 5
5 5
Bucket N
4 4 4
0 2 5 1 6
11 14
1 2
Step 2: N-gram extraction
Step 3: Automata construction
Automata
N-gram
Testing (Detection) Phase
Test trace
Compare to detect which bucket it is assigned
Step 4: Comparison

Fig. 2. Overview of CrashAutomata

3.2 Training Phase: Automata Construction

Once the buckets are reorganized, the next step is to build an automaton from the stack traces of each bucket. One way to achieve this is to simply consider each frame signature as a state in the automaton. A transition from one state to another occurs between two consecutive frame signatures.

This method, however, suffers from two limitations. First, it may result in large automata, which may impact the scalability of the approach. The second and perhaps most important limitation is that it tends to be too rigid, meaning that the resulting automata cannot easily generalize to unseen cases. To make this clear, take for example the stack trace ABCDE, where A, B, C, D, and E are function calls (frame signatures in a Mozilla stack trace). It is reasonable to assume that since AB was invoked once it may also be invoked twice or more due to the presence of loops in the code. Therefore, an incoming report with a stack trace ABABABCDE should be deemed similar. We address both limitations in CrashAutomata by building automata using varied-length n-gram extracted from stack trace sequences. We also introduce a variable α that controls the level of generalization of the automaton. These points are further discussed in the next subsections.

3.2.1 Varied-length n-gram extraction

We address the scalability issue by using an n-gram extraction technique that identifies the frequent common sub-sequences or patterns in a sequence, where the length of the patterns can vary from one to *n* (the number of frame signatures in a trace). To this end, we adopt the algorithm presented by Jiang et al. in [10], used to detect anomalies in large datasets. This algorithm analyzes the training stack trace sequences, and extracts from them frequent patterns as n-grams according to a certain threshold *α*. The threshold is used to control the level of generalization of the resulting automaton.

To illustrate the steps of the algorithm (see Algorithm 1), we use the following three sequences (taken from [10]): T1: ABCDE, T2: CDEA and T3: CDEBA. These sequences represent, in our case, three stack traces of the same bucket, where A, B, C, D, and E are frame signatures. At the beginning, the algorithm extracts all unique frame signatures from the stack traces and labels them as 1-gram.

```
Algorithm 1

Input: the set of unique traces
Output: the sets of varied-length n-grams.

$C_1 = \{$the set of single components $c_1^i$ with $f(c_1^i) > 0\}$.
$k = 1$
do
  for each two elements $c_k^i, c_k^j$ from the set $C_k$,
    if the last $k-1$ component sequence of $c_k^i$ equals
      the first $k-1$ component sequence of $c_k^j$,
      then generate a new sequence
          $s = c_k^i$ plus the last component of $c_k^j$;
          count $f(s)$, the number of times that $s$ appears in
          the trace data;
          if $f(s) > \alpha \cdot min(f(c_k^i), f(c_k^j))$,
            then put $s$ into the set $C_{k+1}$.
  $k = k + 1$.
while $C_k$ is not empty
return all $C_j$, for $1 \leq j \leq k-1$
```

Algorithm 1. Algorithm for varied-length n-gram extraction (from [10])

In the consecutive steps, two n-grams of length k (C_k^i and C_k^j) are combined to make an n-gram of length k+1. The new pattern, we refer to it as p_{k+1}, is retained in the list of final n-grams if the frequency of p_{k+1} is greater than α multiplied by the minimum frequency of C_k^i and C_k^j. Otherwise, it is pruned. From the previous example, take α = 0.6. If we combine the two valid 1-grams A and B, we obtain AB. However, the frequency of AB in all traces is 1 (it only appears in T1), which is less than α (= 0.6) * minimum frequency of A and B (= 1). Therefore, AB will be pruned from the final list of n-grams that will form the automaton. The pattern CD, on the other hand, which is a composition of two valid 1-grams C and D, is retained because its frequency (which is 3) is greater than α (= 0.6) * minimum frequency of C and D (= 3). The process of constructing n-grams continues this way until there are no n-grams to construct. In our case, the 3-gram CDE is the last n-gram to be constructed. The final list of n-grams output by the algorithm when using traces T1, T2, and T3 is shown in Figure 4.

K1	K2	K3
A (3) ✓	AB (1) ✗	CDE (3) ✓
B (2) ✓	BA (1) ✗	
C (3) ✓	BC (1) ✗	
D (3) ✓	CD (3) ✓	
E (3) ✓	DE (3) ✓	
	EA (1) ✗	
	EB (1) ✗	

Fig. 4. The n-grams extracted using Algorithm 1 applied to T1, T2, and T3

Note that the value of α varies from 0 to 1. A smaller α constructs a more generalized model, whereas when α is closer to 1, the model becomes more rigid. If α = 1, the longest n-gram is the trace length itself, whereas when α = 1 the n-grams are all 1-grams in the sequence. The challenge is to find an appropriate α that yields best accuracy when classifying incoming reports. An automaton that is too general (α converges to 0) will lead to many false negatives. On the other hand, an automaton that is too strict (α converges to 1) will result in many false positives. In the case study, we experiment with α varying from 0 to 1 in order to determine α that leads to best accuracy.

3.2.2 Automata Construction

To build the automaton from the list of the varied-length n-grams extracted in the previous step, we adopt Jiang et al.'s algorithm in [10] (see Algorithm 2).

The output of the algorithm is a state transition matrix E where the rows and columns represent the n-grams extracted from the previous step. The algorithm starts with the n-gram set that has the longest length k (in the previous example, k would be 3). Within each set of k-grams, it processes the k-grams in the descending order of their frequency (i.e., the k-gram that has the highest frequency is processed first). These rules aim to minimize the final number of n-grams and edges in the final automaton. The next steps of the algorithm are straightforward. For each element C_k^i of a set of k-grams, we search in the trace if it exists, and if so, it is replaced by a state number (state numbers can be saved in a table along with the elements they represent).

When applied to traces T1, T2, and T3, the resulting E matrix with $\alpha = 0.6$ is shown is Table 1. Figure 5 shows the automaton extracted from this matrix.

```
Algorithm 2. Automaton Construction
for $\alpha\ from$ 0 $to$ 1 step 0.1
Input: the set of unique traces and the sets of n-grams
Output: the automaton $E$

set $E[m][n] = 0$ for any two n-grams $m$, $n$
for each trace $T$
    set $k = L$ and $l = T's$ length
  do
        for each $k$-gram $C_k^i$ selected from $C_k$ according to the
                sorted order (with the most frequent one first),

            search and replace all $C_k^i$ in $T$ with the assigned state
            number;

            if the length of the replaced part equals $l$,
                then break from the inner loop.

            $k = k - 1$.
    while the length of the replaced part $\neq l$ and $k \geq 1$.

    from left to right, set $E[m][n]$ = 1 if an n-gram $n$ follows another
    n-gram $m$ contiguously in the trace $T$

remove the unused n-grams/states from $E$
return the matrix $E$
```

Algorithm 2. Automaton construction algorithm used in CrashAutomata (taken from [10]**)**

Table 2. The E matrix constructed with CrashAutomata from traces T1, T2, and T3

n-gram	A	B	C	D	E	CD	DE	CDE
A	0	1	0	0	0	0	0	0
B	1	0	0	0	0	0	0	1
C	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0
CD	0	0	0	0	0	0	0	0
CDE	1	1	0	0	0	0	0	0

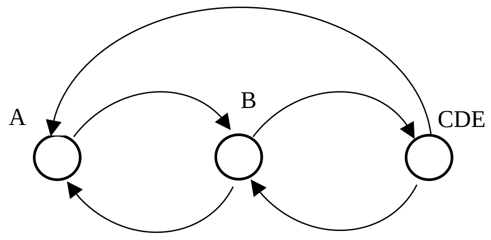


Fig. 5. The automaton that is extracted from T1, T2, and T3 with $\alpha = 0.6$.

As we can see, this automaton generalizes to sequences that are not in T1, T2, and T3. For example, the sequence ABCDEBCDE would be considered a valid sequence.

3.3 Testing (Detection) Phase

Once we construct the automata for the buckets of the dataset, we use them to classify incoming crash reports (more precisely, by using their stack traces). For this, we need to change the sequences of an incoming stack trace into the extracted n-grams identified in the previous step. If the trace contains non-defined n-grams (due for example to new functions that were not encountered during the training phase), we simply assign to them a new ID. We compare the sequence of n-grams in the new trace with the ones in the corresponding automaton. We introduce a new threshold, τ, beyond which we deem that an incoming trace is supported by the automaton.

We measure the effectiveness of CrashAutomata using precision and recall and the F-measure, which are defined using true positive (TP), false positive (FP), and false negative (FN) (see [20] for more details on these measures). For a bucket B_i, we measure TP, FP, and FN as follows:

- TP_{Bi} = The number of traces that are correctly classified
- FP_{Bi} = The number of traces of the other buckets that are classified as bucket B_i
- FN_{Bi} = The number of traces of bucket B_i that were classified as belonging to other buckets other than B_i

We derive precision and recall for each Bucket, *B_i*, as follows. Note that a high FP will reduce precision, whereas a high FN will reduce recall:

$$Precision\ (B_i) = \frac{TP_{Bi}}{TP_{Bi} + FP_{Bi}}$$

$$Recall\ (B_i) = \frac{TP_{Bi}}{TP_{Bi} + FN_{Bi}}$$

$$F_{measure}(B_i) = 2 * \frac{Precision(B_i) * Recall(B_i)}{Precision(B_i) + Recall(B_i)}$$

4. CASE STUDY

The objective of the case study is to assess the accuracy of CrashAutomata using the F-measure. We also determine the most suitable α and τ values that yield best accuracy. In addition, we compare CrashAutomata to CrashGraph. We chose CrashGraph because (a) it relies only on stack traces, and (b) it provides the best accuracy so far compared to other techniques.

4.1 Dataset

We chose in this study to use crash reports of the Firefox system, which are managed using the Socorro server [23], used for collecting, processing and reporting crashes in Mozilla Crash Reporter. We restructured the buckets of Firefox crash reports as discussed in Section 3.1. To achieve this, we, first, selected Firefox bug reports with more than one linked signature. Then for each signature (bucket), we randomly downloaded 200 stack traces of its crash reports from Mozilla Crash Reporter. For those buckets with less than 200 crash reports, we downloaded all of their stack traces. Stack traces can be downloaded from the Mozilla system using a simple script that fetches each crash report.

We generated our new buckets according to the stack traces that are related to duplicate bugs. In total, we downloaded 5,706 stack traces and created nine (9) new buckets in which their stack traces were grouped according to their related bug IDs. In the next sections, we refer to these newly constructed buckets as B1, B2,…, B9.

Table 1 shows the properties of the newly constructed buckets of Firefox. Bucket 1 contains the largest number of stack traces. We downloaded these stack traces from 29 different signatures in Mozilla, which are related to the same bug.

Table 2. Properties of the dataset.

Bucket ID	*Total # of Traces*
B1	***3,221***
B2	***293***
B3	***524***
B4	***534***
B5	***300***
B6	***172***
B7	***178***
B8	***243***
B9	***241***

4.2 Results of Applying CrashAutomata

For each bucket, we chose 70% of the stack traces for training and used the remaining 30% for testing (this is a typical practice in machine learning [30]). We construct an automaton for each bucket. We run CrashAutomata using different values of α in order to determine the most suitable α that yields best classification. We also examine different values of the threshold τ. Table 3 shows an example of the result of CrashAutomata when used with α=1 and τ=0.95. The row and columns show the buckets. A cell M_{ij} in the table shows the number of stack traces in Bucket B_i that are assigned to Bucket B_j by CrashAutomata. For example, Cell M_{11} shows the number of stack traces of Bucket B1 that are correctly classified. Cell M_{18} shows that 2 traces of Bucket B_1 are misclassified as belonging to Bucket B_8. The column 'U' (Unspecified) refers to traces that were not classified in any of the buckets.

Table 3. An example of a classification result of CrashAutomata

Bucket	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**U**
B1	852	0	0	0	0	0	1	2	40	72
B2	0	88	0	0	0	0	0	0	0	0
B3	1	0	157	0	0	0	0	0	0	0
B4	6	0	0	148	0	0	0	0	0	7
B5	2	0	0	0	63	0	7	0	0	19
B6	0	0	1	0	0	49	0	0	1	1
B7	12	0	0	0	0	0	39	0	0	3
B8	12	0	0	0	0	0	0	61	0	0
B9	1	0	0	0	0	0	0	0	69	3

We calculate the F-measure for all buckets and the average F-measure. Figure 6 shows the average F-measure obtained for all buckets by varying α from 0 to 1 with a 0.1 step and τ = 90%. We believe that 90% similarity is a strong indication that the incoming stack trace should indeed belong to the bucket.

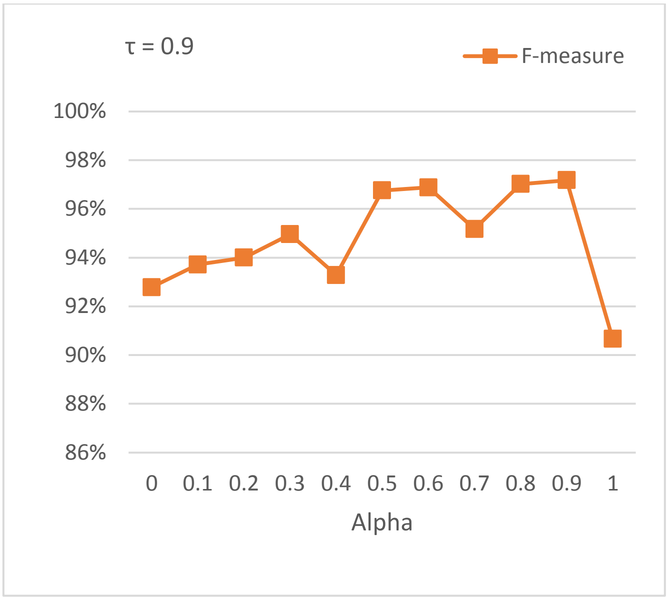


Fig. 6. Average F-measure by varying α and τ= 0.9.

As we can see from Figure 6, the best accuracy (97% accuracy) is obtained when α = 0.9. Figure 7 shows the F-measure for each bucket with α = 0.9 and τ = 90%. The results show that the accuracy of CrashAutomata is more than 96% for all buckets, except for Bucket B5 (the accuracy is 88%). By analyzing the stack traces of Bucket B5, we realized that these traces contain many signatures where only the function names are indicated, without specifying the module names. We believe that this may have caused many traces in the testing phase to be classified as unspecified.

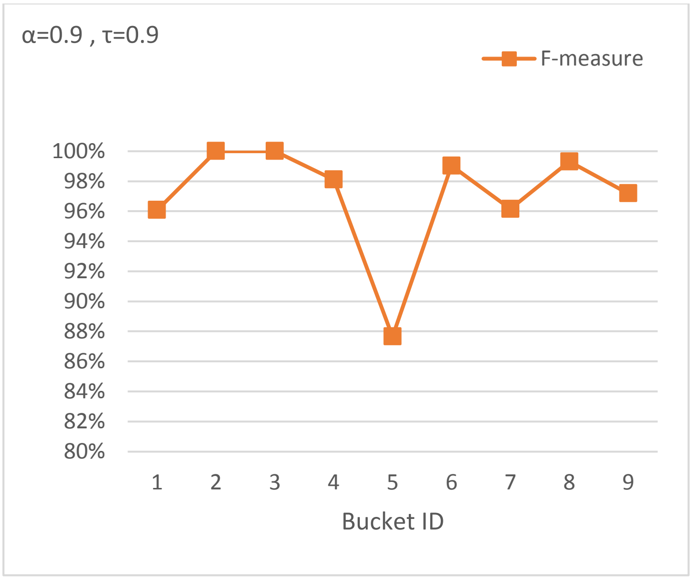


Fig. 7. F-measure for each bucket, τ=0.9 and α=0.9.

We also examined whether a higher threshold τ can lead to better results. We experimented with τ = 95%. Figures 8 and 9 show the results. As we can see, we obtained similar results as for τ = 90%.

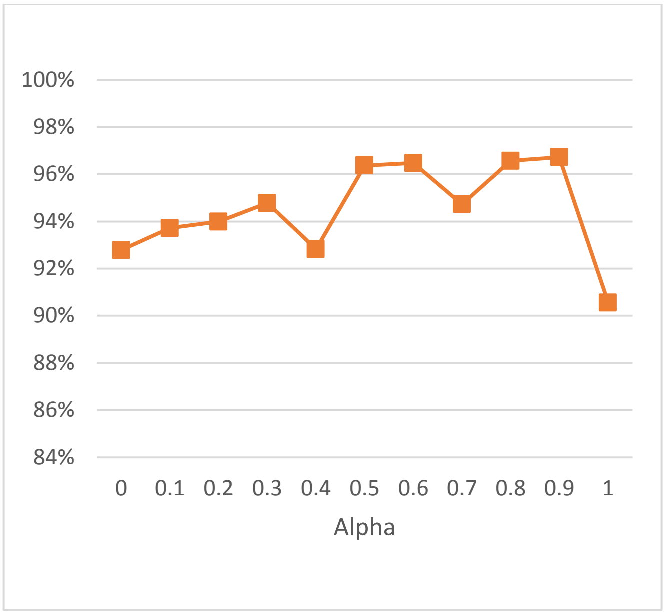


Fig. 8. Average F-measure by varying α and τ= 0.95.

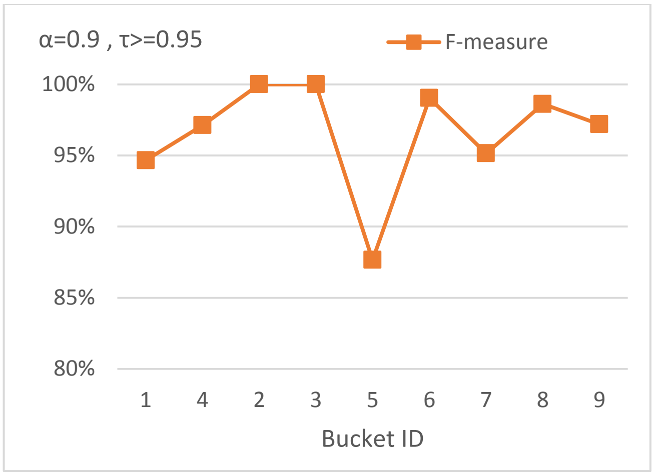


Fig. 9. F-measure for each bucket, τ=0.95 and α=0.9.

4.3 Comparison with CrashGraph

Kim et al. [14] introduced CrashGraph to detect duplicate crash reports in WER (Windows Reporting System). The approach aggregates the view of multiple crash traces in the same bucket by constructing a graph where the nodes are the frame signatures and the edges represent the calling relationship. CraphGraph uses a similarity threshold, just like the one we use in CrashAutomata, τ, when measuring the similarity between an incoming stack trace and the constructed graph.

We implemented CrashGraph and applied it to our dataset with the objective of comparing its accuracy with CrashAutomata. The results obtained by CrashGraph using a similarity metric of τ=0.9 and τ=0.95 are shown in Figures 10 and 11 respectively.

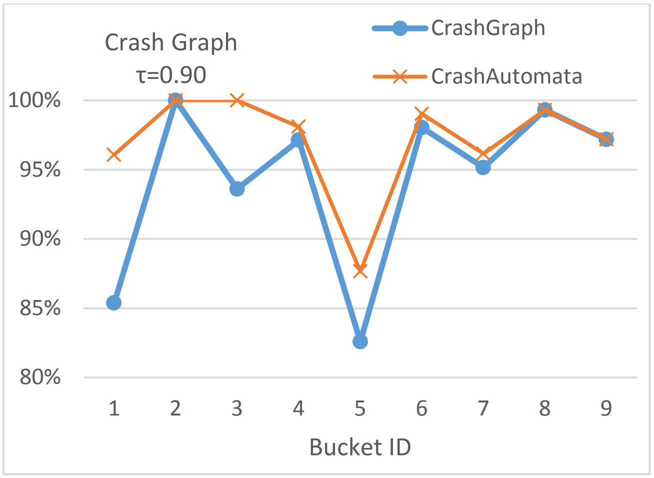


Fig. 10. CrashGraph vs. CrashAutomata F-measure for each bucket with τ= 0.90

As we can see, in both cases, CrashGraph achieves between 83% and 100% accuracy depending on the bucket. It is interesting to see that worse accuracy is obtained for Bucket 5, just like in CrashAutomata. CrashAutomata performs the same or better than CrashGraph for all buckets when τ= 0.90 (Figure 10). By increasing the threshold to τ=0.95, CrashGraph performs slightly better than CrashAutomata for buckets 2, 7, and 8.

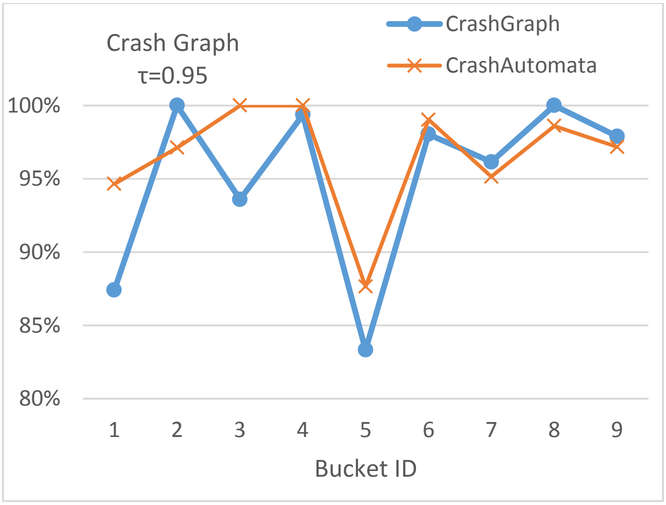


Fig. 11. CrashGraph vs. CrashAutomata F-measure for each bucket with τ= 0.95.

We examined the average precision, recall, and F-measure of both approach with τ= 0.95 (we used this because CrashGraph performs well using this threshold). The result is shown in Figure 12. As we can see, both approaches have high accuracy (100%), but the recall of CrashAutomata is 4% in average higher than CrashGraph. We attribute this to the generalization ability of CrashAutomata.

Furthermore, we studied the number of false negatives of both approaches to pinpoint the buckets that caused the low recall. It should be noted that false negatives include two groups of stack traces. The first type is stack traces that CrashAutomata assigns wrongly to other buckets (referred to in Figure 13 as Misclassified). The second type consists of stack traces that are labelled as Unspecified, i.e., they were not classified in any bucket. Figure 13 (and Table 4) compares the percentage of both types in both CrashAutomata and CrashGraph. The results show that in almost all buckets, CrashAutomata performs better than CrashGraph. In CrashGraph many stack traces are unspecified. The highest number of unspecified stack traces belongs to Bucket B5 in which CrashGraph has around 10% more stack traces than our approach. For Bucket B1, CrashAutomata shows 10% unspecified stack traces while in CrashGraph, this value reaches to 25%, which is a noticeable difference between the two methods. The other significant difference is related to Bucket B3, where the number of falsely assigned stack traces in CrashAutomata is around zero whereas this percentage in CrashGraph is 12%. This concludes that the generalization aspect of CrashAutomata reduces significantly the number of false negatives, which explains the better recall.

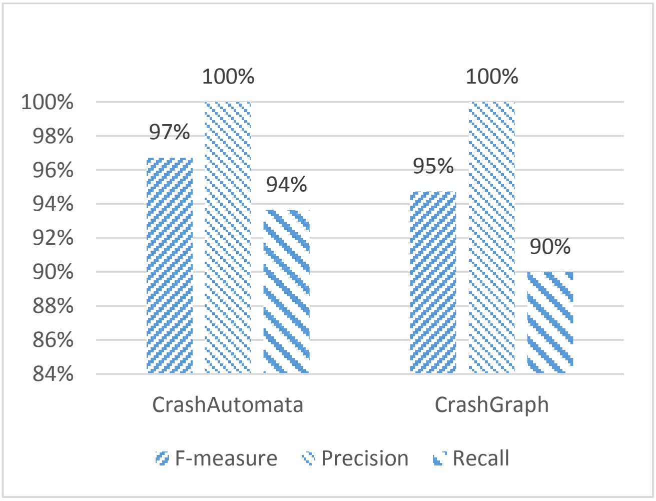


Fig. 12. Comparison between CrashAutomata (α=0.9, τ=0.95) and CrashGraph (τ=0.95).

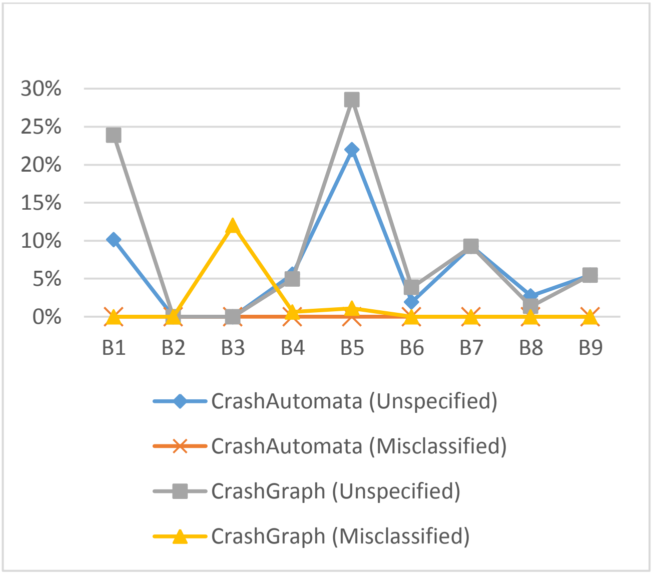


Fig. 13. Comparing false negatives in CrashAutomata and CrashGraph

Table 4. Comparing false negatives in CrashAutomata and CrashGraph

	CrashAutomata		CrashGraph	
Bucket	Unspecified	Misclassified	Unspecified	Misclassified
B1	10%	0%	24%	0%
B2	0%	0%	0%	0%
B3	0%	0%	0%	12%
B4	6%	0%	5%	1%
B5	22%	0%	29%	1%
B6	2%	0%	4%	0%
B7	9%	0%	9%	0%
B8	3%	0%	1%	0%
B9	5%	0%	5%	0%

4.4 Discussion

The case study shows promising results. CrashAutomata was able to achieve 100% precision and between 86% and 100% recall. We showed that it has a better recall than CrashGraph, which does not support any generalization of the trained model. In what follows, we discuss two aspects of CrashAutomata that may need further research.

Generalization of the automata: The goal of CrashAutomata is to detect duplicate reports early in the crash handling process to save software developers time and effort. Unlike existing techniques, CrashAutomata is built with generalization in mind by modeling stack traces in a way that unseen traces can be easily classified. Since not every normal trace is seen and collected in the training data, a certain capacity of generalization is desirable to reduce false positives and false negatives in detection. The question is how much generalization the automata should have in order to obtain good detection accuracy. In this paper, we experimented with various values of α to find the most suitable one. We expect that α changes from one dataset to another. The danger with generalization is that it may lead to automata that are too loose, which may affect the true positives (true duplicates may end up classified as non-duplicates). It is therefore recommended to keep a tight representation of the automata to guarantee an adequate true positive rate.

Unspecified stack traces: During the experiments, the traces that were misclassified by CrashAutomata were all labelled as 'Unspecified', i.e., they were not classified as belonging to any other buckets. Unspecified traces may be an indication that new buckets are needed. These traces were wrongly assigned to existing buckets (again because of the way the Mozilla crash reporter assigns crash reports to buckets, which is based on the top frame signature). This said, we can use the actual representation of buckets to design a new bucketing system that relies on the automata representation to classify incoming crashes. The new system starts with a reliable set of buckets (just like the ones we constructed from the Mozilla crash reporting system) and classifies incoming crash reports by measuring the similarity between the stack traces and the automata representation. Traces that show a high degree of dissimilarity with all existing buckets should lead to the creation of new buckets.

5. THREATS TO VALIDITY

The selection of the dataset is one of the common threats to validity for a classification approach in machine learning. It is possible that the crash reports of the selected system (Firefox) may be biased by sharing common properties that we are not aware of and therefore, invalidate our results. However, Firefox is used in many similar studies so we believe that it is a representative system for this research. This said, we acknowledge that we need to apply our approach to other datasets.

Another threat to validity lies in the way we have selected the stack traces in this study. We selected the stack traces randomly to avoid any bias. One may argue that a better approach would be to select stack traces based on other criteria such as the size of the traces or the number of distinct functions they contain, etc. We believe that longer and more complex traces may perhaps have an impact on the running time of the approach, but we are not convinced that the accuracy of our approach depends on the complexity or the stack traces. Besides, having 200 traces in each bucket, as it is the case in our approach, should provide good coverage of the running system.

In addition, we see a threat to validity that stems from the fact that we implemented CrashGraph based on the description of the approach in the paper [14]. Unfortunately, we were not able to have access to the implementation of the authors. CrashGraph is a very simple approach, which consists of building a dynamic graph from multiple traces. The algorithms and the implementation are straightforward. We tested our implementation on many examples to make sure it works properly.

The use of the threshold, τ, may be a threat to validity since a different threshold may lead to different results. We mitigated this threat by testing with τ = 90% and τ = 95%. A lower threshold may result in less accuracy.

The real link between crash reports and their bug in not defined in Mozilla and Bugzilla [6]. Therefore, we assumed that the selected crash reports were related to the corresponding bug. This could be a threat to the validity of the study. However, the results of the study (100% precision) seem to suggest that this is a valid assumption. A similar assumption was made by other researchers such as [8, 28].

Finally, we see a threat to validity that stems from the fact that we only used crashes from the Mozilla web site, which is an open source repository. The results may not be generalizable to industrial systems. Unfortunately, we do not have access to industrial systems to experiment with and mitigate this threat.

6. RELATED WORK

Detecting duplicate crash reports and grouping them effectively improve the duplicate bug detection as well. To detect duplicates, methods use information inside crash reports such as stack traces or information from bug reports,e.g., comments and descriptions.

Schröter et al. [22] proposed a study on the usage of stack traces by developers from the Eclipse project. They showed the usefulness of stack traces in fixing bugs by only examining key patterns in the stack traces.

Jalbert et al. [9] introduced an approach to detect duplicate bug reports using a classifier. The classifier combines features of reports and uses textual similarity metrics and graph clustering algorithms. Similar to Jalbert et al. [9], Bartz et al. [13] trained a classifier on WER failure reports that predicts the similarity between two failure reports considering the call stack as the key feature. Another method by Sureka et al. [26] uses character n-gram-based modes for duplicate bug detection. Instead of using words in feature selection, they use characters. The method searches for top-N bug reports and visualizes them with numerical scores to the triagers.

Wang et al. [28], improved bug localization by detecting different crash types related to a same bug. If the occurrence of one bug causes the other bug to occur, this bug is referred to as correlated. The authors proposed an algorithm to improve the bug localization using crash correlation groups.

The method proposed by Wang et al. [29] detects duplicate bug reports by combining both natural language information and execution information of existing bug reports in the Firefox bug reporting. They used natural language processing techniques together with stack trace similarity measurements to identify the duplicate bug reports from the non-duplicate ones.

There are many other techniques [1, 2, 5, 11, 12, 16, 19] which use either the contextual parts of bug reports or stack traces from passing and failing execution traces. They usually dependent on instrumentation, predicates, and coverage reports of successful traces. These methods are not applicable to crash reports, since only failing crash reports are available.

Dhaliwal et al. [8] applied two level grouping on Mozilla Firefox buckets. The first level is a grouping done according to the crash signature. The second level is to subgroup stack traces based on their similarities. Like our approach they generate a representative stack trace for each subgroup, however, the representative trace contains the frequency of each module in each frame.

Although WER uses strong heuristics in generating its buckets, sometimes crashes caused by the same bug are put in different buckets. In addition WER may generate many buckets that contain only one or a few number of crash reports. To improve the accuracy of bucketing in WER, Rebucket [7] was proposed by Dang et al. for clustering crash reports based on call stack similarity. Rebucket measures the similarity between call stacks in WER and assigns crash reports to the buckets according to similarity values.

The most similar work to our work as discussed before is CrashGraph [14]. Although CrashGraph could achieve better results than previous methods in WER, we showed that in Mozilla the results obtained by our approach outperform the results by CrashGraph. In comparison with WER, our CrashAutomata is a simpler and independent from developers' investigations.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented CrashAutomata, a technique for detecting duplicate crash (bug) reports using stack traces. Unlike other techniques, CrashAutomata is built with generalization in mind. Stack traces are first processed to extract varied-length n-grams, used to form automata. The extract algorithm relies on a variable α that controls the level of generalization of the automaton. The idea is to have a model that can be general enough to classify similar traces that were unseen during training. Once the automata are built, every time a new stack trace (crash report) arrives, instead of comparing crashes one by one to detect the duplicates, crashes could be assigned to a bucket with high accuracy of being duplicate of bugs related to that bucket. This approach can facilitate the triaging process and other crash handling tasks.

We experimented with CrashAutomata on crash reports of the Firefox system (downloaded from the Mozilla crash reporting system). The F-measure of our approach is in average 97%. We showed that CrashAutomata outperforms CrashGraph. It results in better recall than CrashGraph while keeping the same precision. We attributed this to the generalization power of CrashAutomata.

In future, we will experiment with CrashAutomata on other systems. We will also investigate the variables that can lead to a generalizable model that has high true positive rate while reducing false positives and negatives.

ACKNOWLEDGMENT

This study was partly supported by a grant from NSERC (the Canadian Natural Sciences and Engineering Council).

8. REFERENCES

[1] Aggarwal, K., Rutgers, T., Timbers, F., Hindle, A., Greiner, R. and Stroulia, E. 2015. Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge. (2015), 211–220.

[2] Alipour, A., Hindle, A. and Stroulia, E. 2013. A contextual approach towards more accurate duplicate bug report detection. *IEEE International Working Conference on Mining Software Repositories*. (2013), 183–192.

[3] Apple Crash Reporter: *https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AnalyzingCrashReports/AnalyzingCrashReports.html*.

[4] Bettenburg, N., Premraj, R., Zimmermann, T. and Kim, S. 2008. Duplicate bug reports considered harmful... Really? *IEEE International Conference on Software Maintenance, ICSM*. (2008), 337–345.

[5] Brodie, M., Ma, S., Rachevsky, L. and Champlin, J. 2005. Automated problem determination using call-stack matching. *Journal of Network and Systems Management*. 13, 2 (2005), 219–236.

[6] Bugzilla: *https://bugzilla.mozilla.org/*.

[7] Dang, Y., Wu, R., Zhang, H., Zhang, D. and Nobel, P. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. *Proceedings - International Conference on Software Engineering*. (2012), 1084–1093.

[8] Dhaliwal, T., Khomh, F. and Zou, Y. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. *IEEE International Conference on Software Maintenance, ICSM*. November 2009 (2011), 333–342.

[9] Jalbert, N. and Weimer, W. 2008. Automated duplicate detection for bug tracking systems. *Proceedings of the International Conference on Dependable Systems and Networks*. (2008), 52–61.

[10] Jiang, G., Chen, H., Ungureanu, C. and Yoshihira, K. 2007. *Trace analysis for fault detection for application server*. CRC Press.

[11] Jones, J. a., Harrold, M.J. and Stasko, J. 2002. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. (2002), 467–477.

[12] Jones, J. a. J. a and Harrold, M.J.M.J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. *Automated Software Engineering*. (2005), 282–292.

[13] Kevin Bartz, Jack W. Stokes, John C. Platt,Ryan Kivett, David Grant, Silviu Calinoiu, G.L. 2008. Finding Similar Failures Using Callstack Similarity. *SysML*. (2008).

[14] Kim, S., Zimmermann, T. and Nagappan, N. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. *Proceedings of the International Conference on Dependable Systems and Networks*. (2011), 486–493.

[15] Kinshumann, K., Glerum, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G. and Hunt, G. 2011. Debugging in the (very) large. *Communications of the ACM*. 54, 7 (2011), 111.

[16] Liblit, B., Naik, M., Zheng, A.X., Aiken, A. and Jordan, M.I. 2005. Scalable statistical bug isolation. *ACM SIGPLAN Notices*. 40, 6 (2005), 15.

[17] Mozilla Crash Reporter: *https://crash-stats.mozilla.com/home/products/Firefox*.

[18] Nguyen, A.T., Nguyen, T.T.T.N., Lo, D. and Sun, C. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. (2012), 70.

[19] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J.S.J. and Wang, B.W. Bin 2003. Automated support for classifying software failure reports. *25th International Conference on Software Engineering, 2003. Proceedings.* (2003), 465–475.

[20] Powers, D.M.. 2011. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies*. 2, 1 (2011), 37–63.

[21] Runeson, P., Alexandersson, M. and Nyholm, O. 2007. Detection of duplicate defect reports using natural language processing. *Proceedings - International Conference on Software Engineering*. (2007), 499–508.

[22] Schröter, A., Bettenburg, N. and Premraj, R. 2010. Do stack traces help developers fix bugs? *Proceedings - International Conference on Software Engineering*. (2010), 118–121.

[23] Socorro: *http://socorro.readthedocs.org/en/latest/*.

[24] Sun, C., Lo, D., Khoo, S.C. and Jiang, J. 2011. Towards more accurate retrieval of duplicate bug reports. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*. (2011), 253–262.

[25] Sun, C., Lo, D., Wang, X., Jiang, J. and Khoo, S.-C. 2010. A discriminative model approach for accurate duplicate bug report retrieval. *2010 ACM/IEEE 32nd International Conference on Software Engineering*. 1, (2010), 45–54.

[26] Sureka, A. and Jalote, P. 2010. Detecting duplicate bug report using character N-gram-based features. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. (2010), 366–374.

[27] Tian, Y., Sun, C. and Lo, D. 2012. Improved duplicate bug report identification. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. (2012), 385–390.

[28] Wang, S., Khomh, F. and Zou, Y. 2013. Improving bug localization using correlations in crash reports. *IEEE International Working Conference on Mining Software Repositories*. (2013), 247–256.

[29] Wang, X.W.X., Zhang, L.Z.L., Xie, T.X.T., Anvik, J. and Sun, J.S.J. 2008. An approach to detecting duplicate bug reports using natural language and execution information. *2008 ACM/IEEE 30th International Conference on Software Engineering*. (2008).

[30] Witten, I.H., Eibe, F. and Hall, M.A. 2011. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.