Automatically Analyzing Groups of Crashes for Finding Correlations

Marco Castelluccio
Mozilla
London, UK
University Federico II of Naples
Naples, Italy
marco.castelluccio@unina.it

Carlo Sansone
University Federico II of Naples
Naples, Italy
carlo.sansone@unina.it

Luisa Verdoliva
University Federico II of Naples
Naples, Italy
verdoliv@unina.it

Giovanni Poggi
University Federico II of Naples
Naples, Italy
poggi@unina.it

ABSTRACT

We devised an algorithm, inspired by contrast-set mining algorithms such as STUCCO, to automatically find statistically significant properties (correlations) in crash groups. Many earlier works focused on improving the clustering of crashes but, to the best of our knowledge, the problem of automatically describing properties of a cluster of crashes is so far unexplored. This means developers currently spend a fair amount of time analyzing the groups themselves, which in turn means that a) they are not spending their time actually developing a fix for the crash; and b) they might miss something in their exploration of the crash data (there is a large number of attributes in crash reports and it is hard and error-prone to manually analyze everything). Our algorithm helps developers and release managers understand crash reports more easily and in an automated way, helping in pinpointing the root cause of the crash. The tool implementing the algorithm has been deployed on Mozilla's crash reporting service.

CCS CONCEPTS

• **Software and its engineering → Software reliability**;

KEYWORDS

Crashes; Crash Reports; Crash Analysis.

ACM Reference format:
Marco Castelluccio, Carlo Sansone, Luisa Verdoliva, and Giovanni Poggi. 2017. Automatically Analyzing Groups of Crashes for Finding Correlations. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 10 pages.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106306

https://doi.org/10.1145/3106237.3106306

1 INTRODUCTION

Fixing crashes is one of the top priorities for software organizations, as they are one of the main pain points for users and might lead them to leave. Even a single crash can dramatically worsen how users perceive a software, especially if it causes the loss of important data. Acting quickly is thus really important to avoid losing users and keep a high quality software.

Several software organizations have deployed automated crash reporting systems, such as Mozilla's Socorro [1] and Windows Error Reporting [12], which are used to collect reports from users at the time of crash. A report received by Socorro comprises typically more than a hundred attribute-value fields. These reports are then analyzed by dedicated personnel to find out fixes and improve software quality. It should be realized, however, that these systems collect a huge number of crash reports daily, about three hundred thousand reports/day for Socorro, which cannot be processed on an individual basis. Therefore, the typical workflow consists of two key phases

(1) crash report clustering;
(2) cluster featuring and analysis.

The goal of clustering is to group together similar reports, as they are likely originated by multiple instances of the same software problem. Once the problem is fixed, all these reports can be discarded at once from further analysis. Moreover, clustering allows one to compute precious statistics on the cluster itself, enabling the second phase of the workflow. In fact, the typical features of interest in a cluster concern the frequency of occurrence of attribute-value pairs, which may provide useful hints for the solution of the problem. As an example, assume that a perfect clustering process succeeds in grouping together all crash reports originated by a given software bug, and assume also that all such reports are characterized by a distinctive feature which is never observed in reports of other clusters. While not conclusive, this observation would provide a strong clue for the analyst, and would probably allow a quick fix of the problem. This idealized process is summarized graphically in Figure 1.

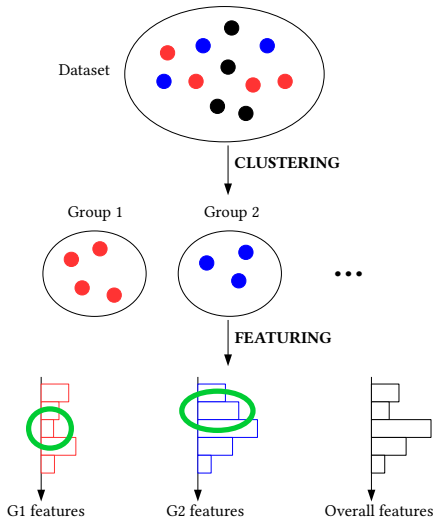


Figure 1: Idealized process: with perfect clustering, properties that define the groups are easily found.

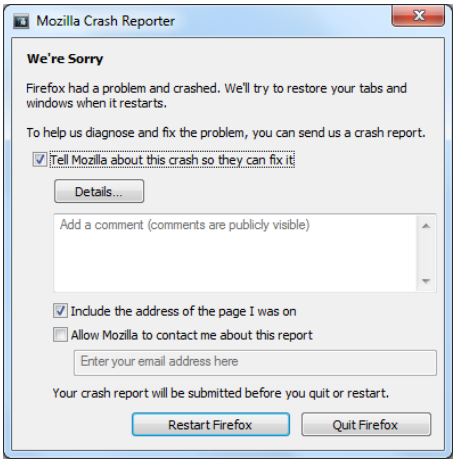


Figure 2: Dialog window presented to the users when they experience a crash.

Needless to say, real-world operations are very far from this simplistic case. On/off features rarely occur, and the analyst must focus on minor variations in the frequencies of occurrence of attribute-value pairs across groups. Moreover, the most distinctive features concern usually *joint* occurrences. If the number of elementary features is already large, the number of features concerning more complex behaviors, possibly involving tuples of attribute-value pairs, makes brute force analysis simply infeasible. It requires very skilled analysts to navigate effectively through these data and extract useful clues. To further complicate things, the preliminary clustering of crashes is itself far from perfect, which may strongly affect the results of subsequent analyses. When a cluster includes reports that have no relation with one another, the resulting features are averaged together and hardly distinctive anymore. On the contrary, when there are too small groups, since reports for the same crash are divided in multiple clusters, features become unstable, leading to erroneous conclusions.

The above discussion underlines the need of effective automated tools that support the analyst's work in both phases on the process to *i)* perform a reliable clustering of crash reports, and *ii)* single out the most meaningful features. Many previous studies in the literature have focused on the first problem, namely, proposing a number of competing solutions to best cluster crashes in groups. Section 6 contains a more detailed explanation of some of them. In this paper, instead, we focus on the second problem, and propose an automated tool to support group understanding after the clustering has already taken place. The proposed tool finds statistically significant properties in crash groups, sorts them by decreasing importance, and submits them to the analyst. Developers are therefore freed from this tedious preliminary analysis, and can focus on fixing the crash. It should be also underlined that the manual analysis, given the large number of attributes in crash reports, is not only tedious but also error-prone (also due to the effects of fatigue). The proposed tool may happen to find interesting properties that the analyst could miss. Automatically finding properties of crash groups also allows release managers to quickly act with temporary workarounds, for example by blocking updates to a crashy version for a particular set of users.

Specifically, our approach is based on a data mining technique, contrast-set learning [24], applied successfully to a number of other problems in software engineering [27] and beyond (e.g. [16]). The approach we present in our study can also help with the triage of crash groups, in fact release managers can decide on their importance, after understanding the possible causes and properties of a crash. We evaluate the system using crash data collected from the Mozilla crash reporting system and bug tracking system. Although a systematic analysis of performance in not feasible for practical reasons, we collected significant evidence that the system may actually help understanding a group of crashes and reduce the time needed to solve the problem.

The remainder of this paper is organized as follows. Section 2 provides background information about Socorro, the Mozilla crash reporting system used in our study. Section 3 describes the proposed algorithm. Section 4 presents the validation of the results of our algorithm, applied to real world cases for Mozilla Firefox crashes. Section 5 discusses threats to the validity of this study. Section 6 summarizes related works and Section 7 concludes the paper.

2 SOCORRO AND CRASH REPORTS

Mozilla's applications are shipped with a built-in automatic crash reporting tool [1]. When end users encounter a crash, they are presented with a dialog window that asks them to submit a report (see Figure 2).

Crash reports include stack traces of the threads that were running at the time of the crash and other information about the user's environment (e.g. operating system, memory-related information, modules loaded in the process, etc.). A subset of the fields contained in a crash report is depicted in Table 1. The reader may refer to [21] for an up-to-date JSON schema of a crash report. Some of the information contained in a crash report might be sensitive, which

Table 1: A subset of the attributes present in a crash report.

Name	Description
Platform	The name of the Operating System.
Platform Version	The detailed version of the Operating System (e.g. *uname -a* on Linux).
Addons	A list of the addons, with their version, installed in the Firefox profile.
Modules	A list of the modules (DLL files on Windows, SO files on Linux, dylib files on Mac), with their version, loaded in the application's process.
User Comment	A (usually brief) comment left by the user at the time of crashing.
CPU Info	Detailed information (vendor, family, model, stepping, number of cores) about the CPU of the user.
Adapter Vendor ID	The vendor of the graphics card on the user's machine. There are other related attributes such as Adapter Device ID, Adapter Driver Version, etc.
Safe Mode	A boolean variable that indicates whether Firefox was running in safe mode.
User Agent Locale	The language of the user.
...	...

is why the submission of crash reports is not silent, but requires the user to accept a prompt.

As can be seen from Figure 2, the user has a chance to enter a short comment at the time of crash. This allows users to specify details about their crash report. For example, what they were doing right before they experienced the crash. Crash reports are then sent to the Socorro server [23], which:

(1) assigns a unique ID to each report;
(2) performs some post-processing on the reports;
(3) groups the reports together using an extremely fast, but not very reliable, algorithm, described below.

See Figure 3 for an overview of the Socorro architecture.

The reports are clustered based on the top method signature of the stack trace of the crashing thread (or another thread, if the crash is due to the application willingly terminating itself after a hang). Table 2 shows an example of a stack trace, with the group name it was assigned by the Socorro algorithm.

There are several rules that allow to skip some methods if they are deemed to be useless for grouping purposes (e.g. a very generic function, a function from an external driver, etc.). Some of the rules are general purpose (e.g. C++ standard library functions), some are really specific to the Mozilla applications (e.g. XPCOM [22] functions). This large set of rules has been built over time, manually, by developers.

This algorithm is sometimes ineffective, as two crashes that happen in the same function might be completely different from

Table 2: Example stack trace. The group name is in bold.

Frame	Module	Signature
0	xul.dll	**mozilla::storage::Service::getSingleton()**
1	xul.dll	mozilla::storage::ServiceConstructor
2	xul.dll	nsComponentManagerImpl::CreateInstanceByContractID(char const*, nsISupports*, nsID const&, void**)
3	xul.dll	nsComponentManagerImpl::GetServiceByContractID(char const*, nsID const&, void**)
4	xul.dll	nsCOMPtr_base::assign_from_gs_contractid(nsGetServiceByContractID, nsID const&)
5	xul.dll	nsCOMPtr<mozIStorageService>::nsCOMPtr<mozIStorageService>(nsGetServiceByContractID)
6	xul.dll	nsPermissionManager::OpenDatabase(nsIFile*)
7	xul.dll	nsPermissionManager::InitDB(bool)
8	xul.dll	nsPermissionManager::Init()
9	xul.dll	nsPermissionManager::GetXPCOMSingleton()
10	xul.dll	nsIPermissionManagerConstructor
11	xul.dll	nsComponentManagerImpl::CreateInstanceByContractID(char const*, nsISupports*, nsID const&, void**)
12	xul.dll	nsComponentManagerImpl::GetServiceByContractID(char const*, nsID const&, void**)
13	xul.dll	nsCOMPtr_base::assign_from_gs_contractid(nsGetServiceByContractID, nsID const&)
14	xul.dll	nsCOMPtr<nsIPermissionManager>::nsCOMPtr<nsIPermissionManager>(nsGetServiceByContractID)
15	xul.dll	mozilla::services::GetPermissionManager()
16	xul.dll	mozilla::dom::NotificationTelemetryService::RecordPermissions()
17	xul.dll	NotificationTelemetryServiceConstructor
18	xul.dll	nsComponentManagerImpl::CreateInstanceByContractID(char const*, nsISupports*, nsID const&, void**)
19	xul.dll	nsComponentManagerImpl::GetServiceByContractID(char const*, nsID const&, void**)
20	xul.dll	nsCOMPtr_base::assign_from_gs_contractid(nsGetServiceByContractID, nsID const&)
21	xul.dll	nsCOMPtr<nsISupports>::nsCOMPtr<nsISupports>(nsGetServiceByContractID)
22	xul.dll	NS_CreateServicesFromCategory(char const*, nsISupports*, char const*, char16_t const*)
23	xul.dll	nsXREDirProvider::DoStartup()
24	xul.dll	XREMain::XRE_mainRun()
25	xul.dll	XREMain::XRE_main(int, char** const, nsXREAppData const*)
26	xul.dll	XRE_main
27	firefox.exe	do_main
28	firefox.exe	wmain
29	firefox.exe	__scrt_common_main_seh
30	kernel32.dll	BaseThreadInitThunk
31	ntdll.dll	__RtlUserThreadStart
32	ntdll.dll	_RtlUserThreadStart

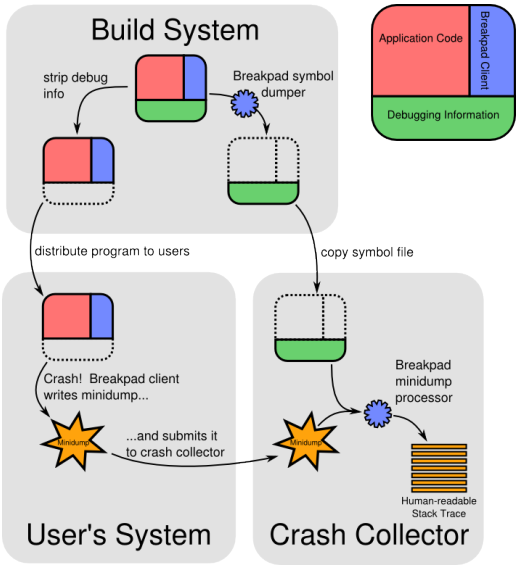


Figure 3: Overview of the crash reporting system

each other. This is particularly noticeable with crashes related to the JavaScript JIT compiler. However, processing speed is deemed more important than accuracy in this context and new clustering methods should be also very fast to qualify as a viable alternative.

3 AUTOMATIC ANALYSIS OF CRASH GROUPS

The analysis method adopted here is a slightly modified version of the contrast set mining algorithm STUCCO (Searching and Testing for Understandable Consistent COntrasts) proposed originally by Bay and Pazzani [3, 4]. To illustrate the method we will refer to a toy example, with the dataset partitioned in two clusters, or groups, with cardinalities $|G_1| = 700$ and $|G_2| = 300$, and reports including only $n = 2$ attributes, platform (p), and graphics card (g), which can

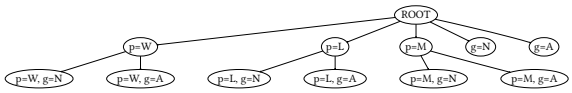


Figure 4: Root and all possible specializations

take three and two values respectively, $p \in \{W, L, M\}$ (for Windows, Linux, and Mac) and $g \in \{N, A\}$ (for NVIDIA, and AMD).

3.1 The Contrast Set Mining Problem

In the contrast set mining framework, the dataset is a set of n-dimensional vectors, whose components are discrete values. The vectors are partitioned beforehand in mutually exclusive groups, $G_1, G_2, \ldots$, according to external criteria.

A contrast-set is defined as a set of attribute-value pairs. For example, $cset1 = \{p = W\}$ is a contrast set concerning a single attribute-value pair, while $cset2 = \{p = W, g = N\}$ concerns a couple of attribute-value pairs, and is actually a *specialization* of the former. The support of a contrast-set in a group, $S(cset, G)$, is the percentage of vectors in the group for which the contrast-set is true. Contrast-set supports are the features used to characterize groups. So, for example, having $S(cset1, G_1) = 0.7$ and $S(cset1, G_2) = 0.3$, means that, in Group 1, 70% of crashes occurred on a Windows platform, while in Group 2 the percentage was 30%. Such a large difference seems to indicate that the platform is not irrelevant for these crashes. Accordingly, the goal of contrast-set mining is to find contrast-sets, also called *deviations*, whose support differs meaningfully across groups.

More formally, for a contrast set to be declared a deviation, it must be both *large* and *significant*. The first condition is expressed as

$$\max_{ij} \left| S(cset, G_i) - S(cset, G_j) \right| \geq \delta \tag{1}$$

where δ is a constant (minimum support difference) defined by the user. Significance, instead, is declared based on the outcome of a statistical test of hypotheses,

$$\begin{cases} H_0 & : \quad P(cset = true|G_i) = P(cset = true|G_j) \\ H_1 & : \quad P(cset = true|G_i) \neq P(cset = true|G_j) \end{cases} \tag{2}$$

carried out for all couples of groups, G_i, G_j, with a user-defined false alarm level, α.

3.2 STUCCO

In STUCCO, contrast-set mining is cast as a tree search problem. The root node is an empty contrast-set. Then, for each step of the algorithm, existing nodes are specialized by appending new attribute-value pairs to existing ones. A canonical ordering of the attributes is used to avoid visiting the same node twice. With reference to our toy example, Figure 4 shows the search tree after two levels of specialization. Note that the nodes $g = N$ and $g = A$ have no children, as g comes after p in our ordered attribute list.

STUCCO performs a breadth-first level-wise search in the tree. We provide, here, a very high-level description of the algorithm, going into more details in the following subsection. For each node at a given level, the number of occurrences for each group in the dataset is counted. Based on such data, some heuristics are applied

Algorithm 1: STUCCO algorithm

```
Set of candidates C ← {};
Set of deviations D ← {};
Set of pruned candidates P ← {};
Let prune(c) return True if c should be pruned;
while C is not empty do
    scan data and count support ∀c ∈ C;
    foreach c ∈ C do
        if significant(c) ∧ large(c) then
            D ← D ∪ c
        end
        if prune(c) = True then
            P ← P ∪ c
        else
            C_new ← C_new ∪ GenChildren(c, P)
        end
    end
    C ← C_new
end
D_surprising ← FindSurprising(D)
```

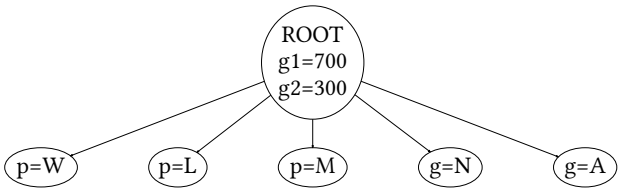
to decide on whether the node should be pruned, become a terminal node, or generate new children. The tree grows until no more child node can be generated, or a suitable stopping condition (applied to limit processing time) is met. After the whole tree is grown, each surviving node corresponds to a valid candidate contrast-set. Contrast-sets that are found to be both large and significant (deviations), and also surprising, are eventually kept, and submitted to the analyst as an ordered list, from largest to smallest. Algorithm 1 provides a pseudo-code description of the process. Figure 5, instead, shows the first few steps of the algorithm applied to our toy example. In particular:

(1) all possible attribute-value pairs ("candidates") are generated for each attribute in a crash report (figure 5a);
(2) the number of occurrences for each candidate in each group is counted (figure 5b);
(3) some nodes are pruned based on suitable heuristics (figure 5c);
(4) new candidates are generated by merging previous ones which survived pruning, for example $\{p = W\}$ and $\{g = N\}$ give rise to $\{p = W, g = N\}$ (figure 5d).

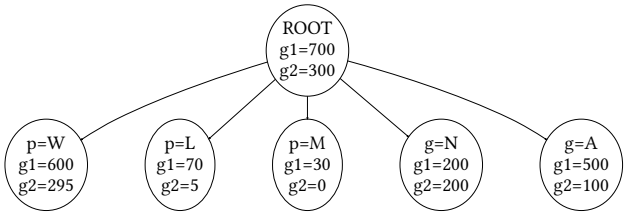
Steps 2-4 are repeated until there are no more candidates or a suitable stopping condition is met, for example, the maximum number of iterations. Eventually, all nodes/contrast-sets are tested, and only those that are large, significant, and surprising are submitted to the analyst.

The following subsections provide the necessary details for a full comprehension of the algorithm, describing the tests on largeness, significance, and surprise, as well as the heuristic rules for tree pruning.

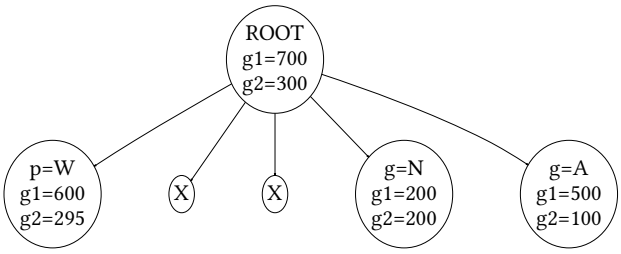
3.2.1 Selecting Large Contrast-Sets. This is a straightforward test: For a contrast-set to be large, its support must be larger than the threshold, δ, defined by the user.



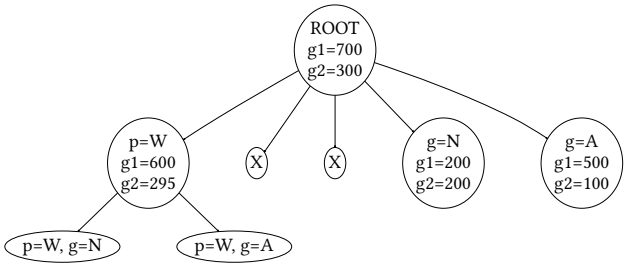
(a) Generation of all possible attribute-value pairs



(b) Count of the occurrences for all candidates



(c) Pruning of candidates using a set of heuristics



(d) Generation of a new level of candidates

Figure 5: Sample run of the algorithm in the context of crash reports

3.2.2 Selecting Significant Contrast-Sets. To evaluate whether a contrast-set is significant, we rely on the test of hypotheses of Eq.2. The null hypothesis is that the support of the contrast-set is equal across all groups or, differently said, it is independent of group membership. To this end, we build a contingency table like that shown in Table 3 reporting the occurrences of a contrast set across groups and the corresponding supports, our features of interest, that is, the frequencies of occurrence in the group.

Table 3: Example contingency table

	$p = W$	$p \neq W$	group size
Group 1	600 (85%)	100 (15%)	700
Group 2	295 (98%)	5 (2%)	300
Overall	895 (90%)	105 (10%)	1000

In our example we analyze $cset1$, namely, platform=Windows. If group and the platform were independent variables, the proportion of crash reports with the Windows platform should be about the same across all groups. This is not the case in our example. However, the supports may differ just because of random fluctuations, and the difference may not be statistically significant. Hence, we need to determine whether such differences are the effect of a true dependency between the variables or if it can be attributed to randomness, which is why we need a statistical test. The standard test for independence of variables in contingency tables is the chi-square test:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(o_{ij} - e_{ij})^2}{e_{ij}} \tag{3}$$

where o_{ij} is the observed frequency in cell ij and e_{ij} is the frequency expected under the hypothesis of independence between row and column variables. We then compare the resulting value against the χ^2 distribution under the null hypothesis, selecting level of significance α, which represents the probability of rejecting the null hypothesis when it holds (false alarm).

For a single test, a level $\alpha = 0.05$, implying a false alarm probability of 5%, could be considered acceptable for our application. However, since a large number of contrast sets are typically tested for significance, the overall number of false alarms may be disturbingly large. For example, if we ran 100 tests at $\alpha = 0.05$, and the null hypothesis were always true, we would detect on the average 5 significant differences that are not actually there. To keep the false alarm rate within acceptable limits, STUCCO reduces α according to the Bonferroni correction: given $H_1, H_2, \ldots, H_k$ hypotheses, and their corresponding p-values $p_1, p_2, \ldots, p_k$, the hypothesis H_i is rejected if $p_i < \alpha/k$. The Bonferroni correction controls the familywise error rate (FWER), which is the probability of incorrectly rejecting at least one true hypothesis H_i, at <= α.

$$\begin{aligned} \text{FWER} &= P\left\{ \bigcup_{i=1}^{k_{true}} \left(p_i \leq \frac{\alpha}{k} \right) \right\} \\ &\leq \sum_{i=1}^{k_{true}} \left\{ P\left(p_i \leq \frac{\alpha}{k} \right) \right\} \leq k_{true} \frac{\alpha}{k} \leq \alpha \end{aligned} \tag{4}$$

This holds no matter how many of the null hypotheses are true and even with dependent tests [26].

There are two problems in the application of the Bonferroni correction in the context of STUCCO: first of all, if we report results in a level-wise fashion (shorter first, then longer), we cannot know how many tests we will perform in total, which makes it impossible to know the exact value of k. Moreover, as α gets smaller, the statistical power of the tests decreases, increasing the probability of producing false negatives. This cannot be avoided, since we want

to reduce the probability of false positives. However, we can use different values of α for tests concerning different levels of the tree, ensuring a high power for tests at higher levels (which are more general and easier to understand) and accepting a lower power for tests more down the tree. Since the Bonferroni method holds as long as $\sum_i \alpha_i \leq \alpha$, STUCCO adopts level-dependent values

$$\alpha_l = \min\left(\frac{\alpha}{2^l}/|C_l|, \alpha_{l-1}\right) \quad (5)$$

where α_l is the cutoff for level l, and $|C_l|$ is the number of candidates at level l. This way we assign $\frac{1}{2}$ of the total α risk to tests at level 1, $\frac{1}{4}$ to tests at level 2, etc. The min rule ensures that, as we move to deeper levels, the α cutoff can only decrease, making the tests more likely not to reject the null hypothesis.

3.2.3 Selecting Surprising Contrast-Sets. As already said, contrast-sets are shown in a level-wise fashion given higher priority to higher levels (e.g. level 1, with a single attribute-value pair) as they are easier to interpret. Further specializations are then included only if they are "surprising", namely, when the observed frequencies depart significantly from the expected frequencies. For example, if for all G_i's, $S(p = W, g = N|G_i) \simeq S(p = W|G_i) \times (g = N|G_i)$, that is the support of the specialization can be derived based on an independence conjecture, than the specialization itself does not add information (is not surprising) and thus can be discarded even when it is a deviation according to the definition.

3.2.4 Pruning the Search Space. When building the contrast-set tree, a number of heuristics can be applied to limit its size and hence reduce the computational burden.

Minimum deviation size. When a contrast-set has support less than δ for every node, it can be pruned. In fact, if the support is smaller than δ for any given group, the difference between any two supports cannot be larger than δ.

Expected cell frequencies. The validity of a test depends on the size of the available sample, becoming scarcely reliable when only a small number of items are available. A typical lower bound for the χ^2 test is 5 [11]. Therefore, when we reach a contrast-set with a number of occurrences smaller than 5, we can safely prune it, since any further specialization can only further reduce the number of occurrences.

χ^2 bounds. Bay and Pazzani showed that it is possible to define an upper bound on the χ^2 statistic. This can be used to prune nodes, when we know that the corresponding statistic will not exceed the α cutoff.

Identical support. Specializations with the same support as the parent might be not interesting and can be discarded. They target the same set of dataset entries as the parent and often represent findings that are common knowledge (e.g. the support of {platform_detail = Debian Wheezy} will obviously be the same as the support of {platform = Linux, platform_detail = Debian Wheezy}: the addition of {platform = Linux} provides no information).

Fixed relations. Often a group has larger support for a given contrast-set than any other group and specializing the contrast-set with additional attribute-value pairs does not change the situation. In those cases, the node can be pruned.

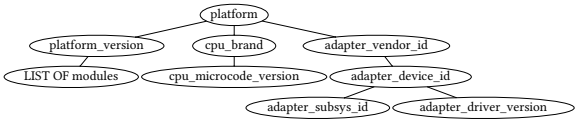


Figure 6: Detail of the dependency graph

3.3 Domain-Specific Variations

The implementation of the algorithm must take into account the large number of items to deal with in our real-world application. At the time of writing, around 500000 crash reports per week are generated for a single Firefox version[1]. Moreover, each report contains a large number of attributes (more than 200) spanning different possible values. This means that the number of possible candidates explodes very rapidly as soon as contrast-sets are specialized beyond level 1. Testing candidates for each couple of groups is clearly infeasible. Therefore, in our implementation, we test each group against the rest of the dataset, that is, we look for features that present anomalies w.r.t. the average behavior over the whole dataset. In addition, for performance reasons, we have implemented the tool using Apache Spark [28].

Another specific feature of our application is the existence of strong dependencies among groups of attributes. For example, the presence of a given DLL might be directly linked to a particular version of Windows; the CPU microcode version is directly linked to the CPU vendor; etc. We modified STUCCO to take into account such information by means of a graph of dependencies (see Figure 6 for a detail of the dependency graph). When a dependency is found, the percentage of occurrence is recalculated restricting the group to the reports where the dependency holds true. For example, in a group we studied, the module "bcryptPrimitives.dll" was present in 83.9% of crash reports vs. 33.91% overall, qualifying for a likely deviation. However, if we take into account the operating system (Windows 10), the percentages change to 100% vs 98.44%, and hence this rule could be ignored.

One of the fields of the crash reports is a small text area where the user who experiences the crash can write a short comment. Most users do not provide useful information but express only their frustration, which makes the comments field widely different from usual bug reports. Nonetheless, in our manual inspections, we have found comments to be sometimes useful, even if just as hints.

With the aim to extract some useful information from the comments field, we employed a well known information retrieval technique, term frequency and inverse document frequency (TF-IDF), which highlights the words most frequently used in the comments for a given crash group vs. other groups. This allows developers to quickly glance if there is something wrong with a particular setting. For example, in one particular instance, many users were mentioning "playing", and the crash turned out to be due to a resource exhaustion due to videogames running in the background.

[1] https://crash-stats.mozilla.com/search/?product=Firefox&version=51.0.1&date=>%3D2017-02-14&date=<2017-02-21#crash-reports

Table 4: Summary of the results of the validation.

Type	Number of bugs
Very useful – results that directly helped fixing the bug.	19
Compatible – results that were compatible with the resolution of the bug, but were not useful for fixing the bug.	19
Misleading – results not compatible with the resolution of the bug.	3

4 VALIDATION OF RESULTS

In order to validate the results, we have selected a set of bug reports where we knew developers used our tool and we have verified whether the tool

- helped in the resolution of the bug,
- gave compatible clues but did not help solving the bug,
- gave some misleading clues.

The tool has been integrated in Socorro, but we do not know when the developers use it for their investigations. Some developers, when using the tool, copied the results of the tool in the bug report they are working on. This allows us to select a set of real world cases that we can analyze, given that developers have fixed them already, so we can evaluate if the results of the tool have been useful for fixing the bug.

We considered about 800 crash bug reports (approximately 400 closed) generated from September 2016, when our tool has been put in production, to February 2017, mostly from Mozilla developers. For 90 of these reports (41 closed) we have definitive evidence that our tool was used. We have manually analyzed this set of bug reports and the code changes that are attached to them, finding 19 cases where the tool has been really useful; 19 cases where the tool generated compatible results, but did not help solving the bug; 3 cases where the tool has produced misleading results. These results are summarized in Table 4.

In some of the cases where the tool has been useful, we believe the bug would not have been solved if not with very large investigative effort. Out of the three cases where the tool has been misleading, we believe that, by improving the initial clustering algorithm, two misleading results would have been avoided. These are analyzed in more detail in section 4.1.4 and 4.1.5. As already said in the Introduction, the quality of clustering can strongly affect the results of the algorithm, polluting group statistics with unrelated reports, or generating groups too small to provide meaningful statistics at all.

When the clustering algorithm fails by generating groups that are too large (clustering together crashes that have no relation with each other), it is harder for the correlation tool to find interesting properties. Indeed, as many crashes which are actually really different from each other get clustered together, it gets more difficult to analyze them (both manually and automatically).

When the clustering algorithm fails by generating groups that are too small (allocating reports for the same crash to different groups), the correlation tool, and manual analysis, is more prone to find spurious correlations.

The clusters' dimensions can vary wildly between thousands of reports (the most crowded cluster contains around 20000 crashes) and a very small number of reports (even a single one). We only apply the tool to the largest 200 clusters, as they are the most important ones (after the 200th cluster, we only have clusters with less than 100 reports). These top clusters account for around 55% of all reports, but there is a very long tail of clusters with very few reports.

4.1 Deployment on Socorro

We tested the tool on crash groups which we already analyzed in the past, to assess its validity, and we put it in production for new crash groups. In this section, we summarize a few interesting results that we obtained during our analysis.

4.1.1 AMD CPU Bug. A group of crashes was found to be correlated with a particular family of AMD CPUs. We later found that the particular family of AMD CPUs that was involved in the crash group was affected by a hardware bug, and developers were able to find a workaround for it.

4.1.2 Antivirus-Related Crash. A group of crashes was found to be correlated with a version of an addon of an antivirus suite. In cases like this, the tool allows us to act quickly and simply block the addons (or modules) that cause problems, while we talk with the vendors to solve the problem in the long term.

4.1.3 Crash Without AdBlock. Interestingly, the tool also generates results that are quite open to intepretation. For example, there was a crash group that was more common to users without ad-blocking addons. It was a crash happening often with a very famous Flash game. We believe the crash was caused by some ad network serving particular advertisement that would cause the browser to crash. The crash disappeared quickly on its own, which supports that hypothesis.

4.1.4 Misleading Result Caused by Clustering Failure (Too Few Clusters). Crashes related to the JIT compiler for JavaScript are a clear example of how crash clustering can affect the results of the tool. The clustering algorithm employed by Socorro does not work well for those kind of crashes, often lumping unrelated crashes together. The correlation tool is only able to tell that the group of crashes is related to the JIT, but cannot say much more.

4.1.5 Misleading Result Caused by Clustering Failure (Too Many Clusters). There was a crash, which was later diagnosed to be due to concurrency issues, which was happening in different functions according to CPU brand or graphics card. This caused the clustering algorithm used by Socorro to generate a new cluster for each CPU brand / graphics card, making each cluster obviously correlated to those. Clearly, the correlations were spurious.

4.1.6 Analyzing Crash Reports Before/After a Change. The algorithm is really useful when analyzing a crash group generated by Socorro, but can be used for generic groups as well. For example, to analyze the differences in the properties of crash reports

before/after a change, e.g. to assess the effectiveness of the change and as another means to ensure that it did not cause regressions.

We employed the tool to analyze the differences between the crashes before/after a change that relaxed the blocklist for graphics acceleration on NVIDIA graphics cards. We found that the change improved the stability with a particular version of the NVIDIA drivers (one where hardware acceleration was previously blocked and unblocked by the change), probably because hardware acceleration is a more common and thoroughly tested code path.

4.2 Feedback from Developers

Developers and people triaging crash bugs generally expressed favourable opinions about the tool. We collected suggestions from them since the deployment to Socorro. Most of the suggestions were requests of addition of new possible fields to the analysis (sometimes meta-information dynamically generated from already existing fields). Some of the suggestions were instead related to the way results are shown, which is actually a pretty important aspect. Indeed, we empiricaly noticed that, if the information presented to the user is too crowded (e.g. too many useless attributes, too much information), the user is more likely to complain or overlook something. In the remainder of this section, we present some of the more specific suggestions that we received from developers.

4.2.1 Employing the Correlation Results Themselves to Improve Clustering. The correlation analysis itself might be useful to improve the clustering algorithm. For example, two groups which present similar correlations might be clustered together. Groups which do not have any interesting correlation, might be candidates to be split.

We observed that this operation was done manually by developers in the results validation. Concerning two bugs where the correlations were very similar, the developers noticed that the two groups were actually a single one (and closed a bug as a duplicate of the other).

4.2.2 Extract Information from Unstructured Crash Report Fields. The algorithm we presented only works with discrete fields, but crash reports often contain unstructured information too. The user comment is a clear example. The TF-IDF solution works for simple cases and it could be greatly improved. For example, if several users mention the same thing in different ways, TF-IDF will not notice it. Using a more powerful text mining algorithm might improve the results, although it is still not clear to us how much information is actually contained in the users' comments. We noticed some cases where it turned out to be useful, but devolving time and resources for this might not be too valuable.

4.2.3 Driving Automated Tests Configuration. At Mozilla, we developed a tool which automatically tries to reproduce crashes with different settings and under different configurations, called BugHunter [20]. The correlation results could help in driving the tool to directly test under a configuration that is more likely to reproduce the crash, both saving running time (e.g. if a crash is only happening with a specific graphics card vendor and a specific driver, there is no point in trying to reproduce it with a graphics card from a different vendor) and making reproducibility easier.

4.2.4 Predicting Volume of a Crash in a Release Channel from Pre-release Channels. By linking the data generated by the correlation tool with data about the user population distribution, we can estimate how a crash that is affecting a pre-release version will affect the release version. The reader can refer to the work by Khomh et al. [14] for an explanation of the Firefox pipelined release model. This has been attempted in the past using machine learning techniques: in Kim et al. [10] it was used to predict which crash stack is more probable to become a "top crash" and should be fixed first. For example, Firefox Beta users are predominantly from the United States. The percentage of those users is fairly lower in Firefox Release. This means that crashes that are easily reproducible on a website that is not in the English language, are very likely to go unnoticed during the Beta cycle and explode when Firefox is released. If we had a way to re-rank the crashes considering the attributes to whom they are correlated and the incidence of those attributes in different channels, then those crashes would less likely go unnoticed.

5 THREATS TO VALIDITY

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. We evaluated our tool on 41 closed bugs, which might not be a representative dataset. We have chosen to evaluate the results on the fixed bugs as we needed to check if the fix was compatible with the findings of the tool. *External validity threats* are concerned with the generalizability of our results. In this paper, we only evaluated the results of the tool applied to Mozilla Firefox crashes, because its crash data, bug reports and source code are publicly available.

6 RELATED WORK

Bird et al. [5] studied the effect of extrinsic factors on software reliability. In our experience we found evidence that corroborates their findings: there are several crashes that are due to external software badly interacting with Firefox. In our case though we often noticed security applications being the root cause of the crashes.

6.1 Automatic Crash Reporting Systems

Several past studies have shown how a crash reporting system, such as Socorro, can be very valuable for discovering and fixing crashes. For example, Glerum et al. [12] presented their experience with WER (Windows Error Reporting). Ahmed et al. [1] studied the Mozilla crash reporting system. One of the problems presented in [1] is the overwhelming amount of data that is made available through a crash reporting system. Our work tries to solve this problem by using data mining techniques to handle the complexity of the data and provide a way to automatically understand it.

6.2 Crash Clustering

The crash clustering problem has been studied extensively in the literature and is closely related to the technique presented in our paper. Indeed, a good clustering technique is needed in order to avoid false positives or false negatives. Lohman et al. [18] and Modani et al. [19] adapted stop-word removal to call stacks, removing recursive calls, and using similarity measures like edit distance, longest common subsequence, and prefix matching. Bartz et al. [2] used

edit distance, proposing seven types of edits assigned with different weights. Dhaliwal et al. [9] proposed a two-level grouping of crash reports, using Levenshtein distance [25] to evaluate the similarity between stack traces. Dang et al. [8] presented *ReBucket*, an algorithm for clustering crashes based on a custom method (called PDM, Position Dependent Model) that uses the position of a function in the stack trace and the offset between matched functions for calculating the similarity between stack traces. Lerch et al. [17] proposed using a well known information retrieval technique, term frequency and inverse document frequency, to rate stack traces. Campbell et al. [6] presented an overview of several clustering algorithms, including the one presented by Lerch et al., evaluating their results in the same setting (Ubuntu Apport crashes). They found traditional information retrieval techniques to outperform techniques specifically designed for crash clustering. The proposed algorithm is strongly related to crash clustering, as it operates on clusters of crashes. Thus, its performance is directly affected by the quality of the clustering algorithm employed.

6.3 Visualization of Crash Reports

Another related area of research is the visualization of crash reports to aid in the understanding by developers. For example, Kim et al. [15] proposed an approach based on an aggregated graph view of multiple crashes. They also presented a way to use the crash graphs for clustering. Chan et al. [7] presented three types of graphs to analyze field testing results under three different perspectives. The above approaches could be combined with our proposed approach to improve understanding of group of crash reports.

6.4 Triaging of Crash Reports

Kim et al. [10] presented a machine learning technique to predict which crash stacks are more probable to become "top crashers" and should be fixed first. Khomh et al. [13] proposed an entropy evaluation approach, taking into account volume of crash groups and distribution among users, to rank the crash clusters by importance. The above approaches focused on prioritizing the groups of crash reports for bug fixing. Our approach instead identifies generic properties of the groups, which can be later used by developers and managers, not only for prioritization, but also to directly understand possible causes.

7 CONCLUSION

Crashes are one of the main pain points for users of a software. Fixing them promptly can improve the users' perception of the quality of a software. We found that analyzing crash reports in an automated manner can help developers in fixing crashes, by removing manual analysis burden from developers, or by finding properties that would have been really difficult to find with manual analysis, or can give clues in the characterization of crashes. Software organizations can use these data mining techniques to speed up and simplify the resolution of crashes and to reduce the amount of manual tedious work for developers.

7.1 Future Work

We identified two interesting directions for future work. First, as discussed in the Validation section (section 4), with examples in section 4.1.4 and 4.1.5, the results of the crash clustering can greatly affect the results of our tool. Thus, improvements to the clustering algorithm used by Socorro, other than being useful by themselves, would benefit our results as well. Second, it could be useful to have a dashboard to simplify finding reproducible crashes. At Mozilla, we are often helped by volunteers in reproducing crashes that are specific to some configuration that we do not have readily available. The correlation results might be useful to create a way for volunteers to automatically find the crashes that they might be able to reproduce, by showing them the crash groups that are related to their hardware or software (e.g. installed addons, antivirus, etc.) configuration.

ACKNOWLEDGMENTS

The authors would like to thank the Socorro developers for their help in the deployment of the tool in the Socorro crash reporting system. The authors would also like to thank the Mozilla developers that have used the tool and gave us important feedback and comments.

This research was carried out within the Fault-Injection-Driven Approach project in the frame of Programme STAR, financially supported by University Federico II of Naples and Compagnia di San Paolo foundation.

REFERENCES

[1] I. Ahmed, N. Mohan, and C. Jensen. 2014. The impact of automatic crash reports on bug triaging and development in Mozilla. In *Proc. of the International Symposium on Open Collaboration*. 1:1–1:8.
[2] K. Bartz, J.W. Stokes, J.C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. 2008. Finding similar failures using callstack similarity. In *Proc. of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques*. 1–1.
[3] S.D. Bay and M.J. Pazzani. 1999. Detecting change in categorical data: mining contrast sets. In *Proc. of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 302–306.
[4] S.D. Bay and M.J. Pazzani. 2001. Detecting group differences: mining contrast sets. *Data Mining and Knowledge Discovery* 5, 3 (July 2001), 213–246.
[5] C. Bird, V.P. Ranganath, T. Zimmermann, N. Nagappan, and A. Zeller. 2014. Extrinsic Influence Factors in Software Reliability: A Study of 200,000 Windows Machines. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 205–214.
[6] J.C. Campbell, E.A. Santos, and A. Hindle. 2016. The unreasonable effectiveness of traditional information retrieval in crash report deduplication. In *Proc. of the 13th International Conference on Mining Software Repositories*. 269–280.
[7] B. Chan, Y. Zou, A.E. Hassan, and A. Sinha. 2010. Visualizing the Results of Field Testing. In *IEEE International Conference on Program Comprehension*. 114–123.
[8] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. 2012. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proc. of the 34th International Conference on Software Engineering*. 1084–1093.
[9] T. Dhaliwal, F. Khomh, and Y. Zou. 2011. Classifying field crash reports for fixing bugs: a case study of Mozilla Firefox. In *IEEE International Conference on Software Maintenance (ICSM)*. 333–342.
[10] K. Dongsun, W. Xinming, K. Sunghun, A. Zeller, S.C. Cheung, and S. Park. 2011. Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering* 37, 3 (May 2011), 430–447.
[11] B.S. Everitt. 1992. *The Analysis of Contingency Tables*. Chapman & Hall/CRC.
[12] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *Proc. of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. 103–116.
[13] F. Khomh, B. Chan, Y. Zou, and A.E. Hassan. An entropy evaluation approach for triaging field crashes: a case study of Mozilla Firefox. In *18th Working Conference on Reverse Engineering*. 261–270.
[14] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. 2012. Do faster releases improve software quality? An empirical case study of Mozilla Firefox. In *IEEE Working Conference on Mining Software Repositories (MSR)*. 179–188.
[15] S. Kim, T. Zimmermann, and N. Nagappan. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. IEEE Compute Society, 486–493.

[16] P. Kralj, N. Lavrač, D. Gamberger, and A. Krstačič. 2007. Contrast Set Mining for Distinguishing between Similar Diseases. In *Proc. of the 11th conference on Artificial Intelligence in Medicine*. 109–118.

[17] J. Lerch and M. Mezini. 2013. Finding duplicates of your yet unwritten bug report. In *Proc. of European Conference on Software Maintenance and Reengineering*. 69–78.

[18] G. Lohman, J. Champlin, and P. Sohn. 2005. Quickly Finding Known Software Problems via Automated Symptom Matching. In *Proc. of the Second International Conference on Automatic Computing*. 101–110.

[19] N. Modani, R. Gupta, G.M. Lohman, T. Syeda-Mahmood, and L. Mignet. 2007. Automatically identifying known software problems. In *Proc. of the 23rd International Conference on Data Engineering Workshops*. 433–441.

[20] Mozilla. 2017. BugHunter. https://wiki.mozilla.org/Auto-tools/Projects/BugHunter. (2017). Online; Accessed February 21st, 2017.

[21] Mozilla. 2017. Socorro Crash Report Schema. https://github.com/mozilla/socorro/blob/master/socorro/schemas/crash_report.json. (2017). Online; Accessed February 21st, 2017.

[22] Mozilla. 2017. Socorro Crash Report Schema. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM. (2017). Online; Accessed February 21st, 2017.

[23] Mozilla. 2017. Socorro server. https://crash-stats.mozilla.com/. (2017). Online; Accessed February 21st, 2017.

[24] P.K. Novak, N. Lavrač, and G.I. Webb. 2009. Supervised descriptive rule discovery: a unifying survey of contrast set, emerging pattern and subgroup mining. *Journal of Machine Learning Research* 10 (2009), 377–403.

[25] D. Sankoff and J.B. Kruskal. 1983. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. 1–44 pages.

[26] J.P. Shaffre. 1995. Multiple Hypothesis Testing. *Annual Review of Psychology* 46, 1 (1995), 561–584.

[27] X. Yu, S. Han, D. Zhang, and T. Xie. 2014. Comprehending performance from real-world execution traces: a device-driver case. In *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 193–206.

[28] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: cluster computing with working sets. In *Proc. of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. 10–10.