Anatomy of a Crash Repository

Joshua Charles Campbell¹, Eddie Antonio Santos¹, and Abram Hindle¹

¹Department of Computing Science, University of Alberta, Edmonton, Canada

ABSTRACT

This work investigates the properties of crash reports collected from Ubuntu Linux users. Understanding crash reports is important to better store, categorize, prioritize, parse, triage, assign bugs to, and potentially synthesize them. Understanding what is in a crash report, and how the metadata and stack traces in crash reports vary will help solve, debug, and prevent the causes of crashes. 10 different aspects of 40,592 crash reports about 1,921 pieces of software submitted by users and developers to the Ubuntu project were analyzed, plotted, and statistical distributions were fitted to some of them. We investigated the structure and properties of crash reports. Crashes have many properties that seem to have distributions similar to standard statistical distributions, but with even longer tails than expected. These aspects of crash reports have not been analyzed statistically before. We found that many applications only had a single crash, while a few applications had a large number of crashes reported. Crash bucket size (clusters of similar crashes) also followed a Zipf-like distribution. The lifespan of buckets ranged from less than an hour to over four years. Some stack traces were short, and some were so long they were truncated by the tool that produced them. Many crash reports had no recursion, some contained recursion, and some displayed evidence of unbounded recursion. Linguistics literature hinted that sentence length follows a gamma distribution; this is not the case for function name length. Additionally, only two hardware architectures, and a few signals are reported for almost all of the crashes in the Ubuntu dataset. Many crashes were similar but there were also many unique crashes. This study of crashes from 1,921 projects will be valuable for anyone who wishes to: cluster or deduplicate crash reports, synthesize or simulate crash reports, store or triage crash reports, or data-mine crash reports.

Keywords: ...
have experienced one of these systems if you have ever seen a dialog box that
popped up asking if you wanted to send feedback when an application crashed
(Figure 2).

Most crash repositories rely on automated systems [37] that deduplicate and
cluster the crash reports, so that hopefully each crash bucket (cluster) has crashes
relating only to one fault in the program. Crashes must be deduplicated automati-
cally, as the quantity of crash reports being uploaded is vast. Campbell et al. [7]
indicated that Mozilla received 2,189,786 crash reports in a single week alone.
There is a substantial body of prior work [17, 11, 45, 36, 4, 27, 9, 41, 7] that has
tackled automated crash report deduplication and triage; however none of these
works focused on the properties of crashes and buckets or how those properties are
distributed. Unfortunately the true accuracy (measured in precision and recall)
of these automated systems is unknown, because they have not been validated
with a gold set of true duplicates.

Because of the big data nature of the problem, sources of manually dedupli-
cated bug reports are rare. For this reason, we analyzed the Ubuntu Launchpad
dataset, which contains manually deduplicated buckets of crashes—crashes that
a human determined to be caused by the same underlying software fault. The
dataset also contains crashes that may have been neglected, receiving little or no
attention. The Ubuntu dataset contains crashes from 1,921 different applications,
programs, and libraries made available on Ubuntu. This type of crash repository
data mining and fundamental descriptive statistics has not been performed in the
literature previously, to our knowledge.

Understanding crash repositories and characterizing the properties of crashes
that can be found within is important not only to deduplication and triage. For ex-
ample, large crash repositories could be used to provide feedback on how crashes
are collected, how software is designed, or what hardware is problematic. One
can leverage crash repositories by instrumenting crashes in a way to aide debug-
ing [32]. Another question that can be answered by analyzing crash repositories
is how often recursion was involved in a crash. We can also find out what actions
cause a large number of crashes. Intuitively, functions on the top of stack traces
are viewed as the most important features for deduplication, and we can use the
Ubuntu dataset to investigate those functions.

The Ubuntu dataset is a valuable resource because it contains thousands of
buckets of crash reports that were hand-marked as duplicates by Ubuntu develop-
ers and volunteers. The Ubuntu project has limited human resources to carry out
this task, however, leaving many crash reports neglected and not yet deduplicated.
Unfortunately it is difficult to discern crash reports which are truly unique from
ones that were merely never given enough attention to be assigned a duplicate.
Thus, the Ubuntu dataset does not contain a gold set of unique crashes.

The lack of larger datasets, and datasets with known good examples of unique
(not duplicate) crash reports motivates the need to create datasets where such
knowledge exists. A major reason to understand the distributions of crash report
data is for creating synthetic crash report datasets. For synthetic datasets to be
helpful, we require a deep understanding of what real crash repositories contain.
This must include understanding properties of real crashes and their statistical distributions within real crash repositories. Because of the scarcity of datasets that contain gold sets of human-deduplicated crashes that can be used to study crash report deduplication techniques, we feel it would be beneficial to produce large datasets of artificial automatically synthesized crash reports.

Fortunately, the Ubuntu dataset contains crashes from 1,921 different applications, programs, and libraries made available on Ubuntu. This includes a wide variety of software such as, GUI applications, command line tools, specialized libraries, standard libraries, compilers, server software, video games, scientific software, and more. While there are a large number of aspects of the dataset that could be studied we chose 10 aspects in the form of research questions (RQ) for this paper. RQ1 to RQ3 focus on how crashes are organized into groups in the dataset. RQ4 to RQ5 focus on categorical properties of crash reports. RQ6 to RQ8 focus on the size of stack traces, recursion and names. RQ9 and RQ10 focus on common functions and libraries that crash.

2 TERMINOLOGY

To understand the domain of crash report deduplication, we provide definitions of the terms used in this paper.

A crash is any unintended, abrupt termination of a program. These are often due to bugs, or erroneous programming caused by human errors. Human errors lead to faults in programs, and faults can lead to failures which can include crashes. We use bugs as an umbrella term to cover human error, faults, and failures, while crashes are the result of bugs. One common crashing bug occurs when a program tries to access memory that is not allocated to it. This is called a segmentation fault on Linux and Unix-based systems.

Crash reports (Figure 1) can be generated when a program crashes. Such reports typically contain information about what was going on in the program when it crashed such as the stack trace, file and line number of code that was executing when the program crashed, and what was in memory at the time of the crash. They also typically include information about the system the program was running on such as what type of computer it was running on, what other programs were running, and what version the program and any libraries it used.

A stack trace (Figure 1, middle) is a snapshot of the execution state of a program. A stack trace is a sequence of stack frames (Figure 1, bottom) which describe the current function, on the top of the stack, followed by the function which called that function—the second function on the stack—and so on. The entry point of the program is the bottom of the stack. Included in a stack frame is usually, the address of the function, the name of the function, and the source file that defined the function. Sometimes, a stack trace also includes the line number in the source file that defined the function, or the line number which was executing at that point in time. Additional information such as arguments and their values at the time each function is called may also be included.
Context data

ProblemType: Crash
DistroRelease: Ubuntu 11.04
Package: rhythmbox 0.13.3-0ubuntu6
ProcVersionSignature: Ubuntu 2.6.38-8.42-generic 2.6.38.2
Uname: Linux 2.6.38-8-generic x86_64
Architecture: amd64
Date: Wed Jul 6 06:50:47 2011
ExecutablePath: /usr/bin/rhythmbox
Signal: 11 (SIGV)
SourcePackage: rhythmbox

Stack Trace

#0 0x00007f08c4128740 in ?? ()
#1 0x00007f08dd22a56 in g_hash_table_lookup_node () at ghash.c:313
#2 g_hash_table_remove_internal () at ghash.c:1093
#3 0x00007f08e1ae059c in remove_child () at rhythmdb-tree.c:1514
#4 0x00007f08e1ae0631 in remove_entry_from_album () at rhythmdb-tree.c:1535
#5 0x00007f08e1ae4675 in rhythmdb_tree_entry_set () at rhythmdb-tree.c:1608
#6 0x00007f08e1ac8b6 in rhythmdb_entry_set_internal () at rhythmdb.c:3345
#7 0x00007f08b5f98fc1 in update_tracks () at rb-audiocd-source.c:1178
#8 0x00007f08b5f9919a in update_tracks_string () at rb-audiocd-source.c:1191
#9 0x00007f08b5f9924c in update_album_cb () at rb-audiocd-source.c:1212
#10 0x00007f08e0ca8578 in _gtk_marshal_BOOLEAN__BOXED () at gtkmarshalers.c:86

[37 stack frames omitted...]

#48 0x00007f08dd349f2 in g_main_loop_run () at gmain.c:3299
#49 0x00007f08e0ca5af7 in IA__gtk_main () at gtkmain.c:1257
#50 0x000000000040f4f4c in main () at main.c:336

Stack Frame

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>0x00007f08dd22a56</td>
<td>g_hash_table_lookup_node ()</td>
</tr>
</tbody>
</table>

Filename  Line Number
at glib/ghash.c:313

Figure 1. An example crash report [40]: its contextual data, its stack trace, and a dissection of one of its stack frames. Note that some information was removed for space.
A crash stack trace is the stack trace captured at the time that a crash occurred. In this case, the top of the stack indicates the function that encountered the crash, but this function does not necessarily contain the bug that induced the crash. The contents and formatting of crash stack traces varies wildly depending on what language the software was written in, what platform it was running on, what tools were invoked to generate the crash stack trace, and what libraries the software was using.

Bug reports (sometimes called issue reports) are user- or developer-submitted reports, usually in a standard format, that detail a problem that the user or developer experiences with a piece of software.

Crash buckets are groups (or clusters) of crash reports, in our case, in the form of bug reports, that have been manually or automatically determined to represent crashes caused by the same underlying problem.

In addition to the stack traces, more information, which we refer to as contextual data may be included such as, what version the software and other pieces of software on the system are, the system architecture and OS version, settings the user may have modified, and information about hardware installed on the system such as sound and video cards.

Stack traces may contain recursion, which occurs when a function calls itself. Stack traces may also contain mutual recursion, when multiple functions call each other in a sequence with the last calling the first. In a stack trace with recursion, there are several instances of the same function.

Unbounded recursion is when a function calls itself endlessly without finding a terminating condition. In C and C++ programs, each function call allocates stack space. Thus, unbounded recursion will allocate stack space endlessly until the operating system refuses to allocate more pages to the program, crashing the process with a segmentation violation (SEGV signal as described in Section 5.4).

Applications, programs, and libraries are treated as packages, or pieces of software that can be installed onto Ubuntu systems, using the Ubuntu system’s package manager. These packages come from source packages which typically consist of a single piece of software from a single source repository, plus bug fixes and customizations made for Ubuntu.

3 PRIOR WORK

There have been a few prior works that have investigated the properties of crash report repositories. These works primarily seek to determine causes of crashes, in contrast to this this paper which seeks to describe and characterize the nature of the crashes from one particular crash repository. Much more prior work has created tools to deduplicate crash reports for the ultimate purpose of aiding developers in fixing common crashes in production. Our work is far more empirical, seeking to understand the properties and structure of crash reports, and is similar

---

1 A compiler may be able to apply the tail-call elimination optimization to recycle the same stack frame, however this is an optional feature provided by some compilers [2].
in intent to that of Herraiz et al. [22] who sought to characterize distributions of version control system repositories to study software growth.

3.1 Studying crash report repositories
Kechagia and Spinellis [2014] mined crashes from Android applications written in the Java programming language. The authors recorded the exceptions that crashed the software compared to the exceptions that were said to occur according to the Android API documentation and found that many of the crash causing exceptions were not mentioned at all in the documentation.

Ganapathi and Patterson [2005] collected crashes from local research machines running Windows XP SP1. Their analysis revealed that web browsers crashed most frequently of all types of applications, even more so than the more widely-used document preparation software. The software that did crash was due to users manual termination due to an application hang. The authors remarked that shared libraries (.dll files on Windows) invoked by multiple applications accounted for about 15% of all application crashes.

This was followed up by Ganapathi et al. [15], which analyzed Windows XP kernel crash data collected from machines on the Berkeley Open Infrastructure for Network Computing. They measured several aspects such as kernel crashes per user, and system uptime before a crash. They concluded that many kernel crashes are the result of unreliable device drivers; 10 device driver vendors were responsible for over 75% of kernel crashes out of the 110 vendors attested in the crash corpus.

Gómez et al. [21] created MoTiF to generate in vivo crash test suites that create the shortest sequence of actions to reproduce a crash experienced by a user. They used crowdsourced test suites to bucket crashes in order to give developers a sense of what needs to be fixed.

Like Campbell et al. [7], this paper focuses on one particular crash repository—the Ubuntu Launchpad dataset. Unlike the prior work, this paper places a greater emphasis on describing the statistical tendency of crashes. Our dataset is different from the ones studied in the prior work in a few important ways. We collected crashes from Linux machines, so some of the research questions in this work are Unix-specific, as opposed to Windows-specific as with Ganapathi’s two prior works [2005, 2006]. As well, the crashes came from lower level languages; despite the presence of exceptions in C++, ultimately what we captured were the Unix signals that crashed the applications.

3.2 Stack-trace crash report deduplication
In order to prevent as many crashes as possible, it is necessary to find out how many users are experiencing what kinds of crashes, categorize those crashes, prioritize them, and triage them. Deduplication is necessary to group and count similar crashes that are hopefully caused by the same fault in the software. Many papers use stack traces—either exclusively, or combined with contextual data—to deduplicate crash reports. Brodie et al. [6] introduced the concept of bug localization by comparing stack traces and finding the “best” sequence of consecutive
function names to create a signature for a crash. This was followed up by Brodie et al. [5], wherein the authors describe several heuristics for preprocessing stack trace data, including stop word removal and recursive call removal. Bartz et al. [4] created a call stack similarity classifier based on learned weighted edit distance. Modani et al. [36] proposed several stack trace similarity metrics including edit distance, longest common subsequence, and top-of-the-stack matching. Bartz et al. [11] described using average weighted Levenshtein distance between stack frames to measure the similarity between crashes. Kim et al. [27] deduplicated already bucketed crashes by constructing a directed graph using stack frames as nodes, and their relation to one-another as directed edges, and then applied a graph similarity metric. Seo and Kim [2012] studied crash stack traces in order to determine which previously bucketed crashes may reoccur, and are thus, worth the triaging and debugging effort to fix. Dang et al. [9] used agglomerative hierarchical clustering on crash stack traces for deduplication. Wang et al. [45] used crash stack traces to train a Bayesian belief network to find files that are likely the cause of a crash. Note that many deduplicators [5, 36, 9] who used stack trace edit-distance found it necessary to remove recursion in the preprocessing stages. Lerch and Mezini [2013] addressed deduplication by using tf–idf to determine the similarity between crash stack traces in order to deduplicate bug reports as they are being written. Campbell et al. [7] followed this up by directly addressing crash report deduplication by comparing various tokenization methods and the parameter of similarity threshold to bucket crashes.

4 DATA

Figure 2. The dialog that appears when an Ubuntu application crashes. A user can opt-out of sending the crash report to Canonical.

This paper uses the Ubuntu Launchpad dataset which consists of 40,592 crash reports which are taken from bug reports for 1,921 packages. In Ubuntu Launchpad, crash reports are submitted as bug reports by a user, but with tooling that helps fill out the contextual data automatically. The tooling that collects the information from the crash report to the bug report also tags the bug report so that it is possible to separate out bug reports that were completely manually generated from those that were generated with the assistance of the crash reporting tool apport.
4.1 How do crashes make it into Launchpad?

On standard installs of Ubuntu Desktop and Ubuntu Server, the Apport system collects the stack trace and metadata, such as the CPU architecture and the signal that crashed the program. According to the Ubuntu Wiki: “If any process in the system dies due to a signal that is commonly referred to as a ‘crash’, […] the Apport back-end is automatically invoked” [43]. Apport does this by installing itself as the program that receives the core dump by setting 

```
/proc/sys/kernel/core_pattern
```

[1]. The Linux kernel invokes Apport with the core dump, which contains the full state of the program at the point of the crash. The core dump is loaded into gdb to extract a string representation of the crash stack trace (Figure 1, middle). Apport presents the user with a window indicating that their app has crashed (Figure 2). At this point, the user may choose to opt-out of sending the crash report. If the user does not opt-out they are prompted to log in to Ubuntu Launchpad, and add their comments to a form which is pre-filled with all of the data automatically collected by the Apport system.

4.2 Why Launchpad?

We chose the Launchpad dataset for several reasons: the dataset is freely available to researchers,² so our observations are reproducible. Similarly, most of the crashes occurred in open source software, allowing researchers to find the source code online and analyze any interesting faults (as we show in Section 5.7). Since the crashes come from a myriad of applications and packages on desktop machines, we can analyze crashes that were caused by shared libraries—in contrast with Mozilla’s crash database that only contains crashes from Firefox. Most remarkably, crashes that were marked as duplicates on Launchpad were labeled manually by volunteers and employees of Canonical. This creates somewhat of a gold set with which researchers can evaluate the precision and recall of their automated crash deduplication algorithms [7]. This set of human-curated crash buckets are invaluable for evaluating crash deduplication methods.

Since most of the software in Ubuntu is open source, most crashes can be debugged if they were not caused by system instability, cosmic rays [23], or other random phenomena. Even though some of the crashes are more than 6 years old, the version of the code that crashed is usually still available.

4.3 How were the crashes obtained?

The corpus was downloaded using a modified version of Bicho [39] which is available online.³ This process took a considerable amount of time (more than a month) due to the fact that the Ubuntu Launchpad API throttles requests. Additionally, much of the data required for the crash reports are uploaded as attachments which must be downloaded separately. These were downloaded using wget. 126,609 issues were downloaded, of which 44,464 contained stack traces from C or C++ programs. Of those, 3,872 were thrown out because they were unparsable, leaving

²https://pizza.cs.ualberta.ca/lp_big.json.xz
³https://github.com/orezpraw/Bicho
40,592 crash reports. These crash reports come exclusively from software that is compiled to a standard Linux binary and can be debugged with GDB, the GNU debugger. This limits the dataset to crashes from software written in C, C++ or other similarly compiled languages.

4.4 What do the crashes look like?

Each crash report consists of semi-structured text information. An example is shown in Figure 1. The text was parsed with a Python script written by the authors. The initial text is automatically generated by Apport, but the user is free to modify anything they want. This results in mostly machine-readable crashes. Unfortunately, some crashes are unparsable or contain unparsable metadata.

Additionally, because the dataset contains crashes from 2009 until 2015, the various software that produced the crash reports, the Launchpad system, and the standard format of reports changed over time. Some reports, when parsed contain the wrong information in the wrong place. For example, if there is a missing newline in the crash report data, information about which package a crash occurred in might mistakenly appear in the field meant to describe which signal crashed the application. Our parser cannot correct these types of problems in the data. However, this type of problem is rare, occurring in less than 1% of crashes.

Most crashes contain more than one stack trace. This is because there are stack traces of varying types available for a single crash. For example, for many crashes, both a normal stack trace is available along with stack traces for any other threads that might be running simultaneously in the application when it crashed. We only use the single stack trace (of the crashing thread) in our experiments.

Additionally, several versions of the same stack trace of the same crash are often available. This can be caused by information that was missing on the user’s computer being automatically filled in later by the Ubuntu Apport system. We attempt to use the stack trace with the additional information by default, if it is available.

5 RESEARCH QUESTIONS, METHODOLOGY AND RESULTS

We answer 10 research questions about the Launchpad crash repository. The first 3 answer how crashes are organized within the crash repository:

RQ1. How are crashes distributed among applications?
RQ2. How are crashes distributed among buckets?
RQ3. How long do crash buckets last?

The latter 7 research questions are focused on the properties of individual crashes and how they are distributed:

4https://github.com/naturalness/partycrasher/blob/master/partycrasher/launchpad_crash.py
Figure 3. Crashes per package. Both axes are logarithmic.

RQ4. What Unix signals are crashes associated with?

RQ5. What CPU architectures experienced the crashes?

RQ6. How long are crash stack traces?

RQ7. How are crashes associated with recursion?

RQ8. How long are function names in crashes?

RQ9. What are the most common functions in crashes?

RQ10. What are the most common crashing libraries?

5.1 RQ1. How are crashes distributed among applications?

In order to fix the causes of crashes, it is first necessary to find out what applications, programs, or libraries the crashes occur in. If a few pieces of software crash far more often, it may make sense to allocate developer time or other resources by
Frequency spectrum of crashes by package, indicating there are many packages with only one crash report, and a few packages with many crash reports. Both axes are logarithmic. The leftmost point on the plot indicates there are 736 packages with only a single crash.

In order to determine which applications crashed we looked at each crash report's `SourcePackage` metadata field, or if that was not available, the crash report's `Package` field. Ubuntu applications are often broken up into multiple packages, but they will usually all share a single `SourcePackage`. 44 crash reports were missing this information (or it was unparsable) so we looked at the remaining 40,548 crash reports.

The package with the most crashes was Nautilus with 1,895 crash reports. In total, 1,921 packages had crash reports. A log-log plot of the distribution of crash reports is shown in Figure 3. This plot shows the packages with the most crashes on the left and the fewest crashes on the right. Additionally, a log-log frequency spectrum plot of the distribution is shown in Figure 4. The frequency spectrum focusing on them. Or, if crashes are spread out among many pieces of software, it might indicate that widely-used libraries are a common cause.
Minimum  | Maximum | Median | Mean  | Std. Dev. | Skewness | Kurtosis
---|---|---|---|---|---|---
1 | 1895 | 2 | 21.1 | 94.8 | 10.8 | 150.4

**Table 1.** Descriptive statistics for crashes per package.

shows how many packages have 1 crash, how many packages have 2 crashes, and so on.

Figure 3 indicates that the data might follow a broken power-law (Zipf’s law) distribution because it appears to have several linear appearing decreasing segments when plotted with both axes being logarithmic. The plot is very similar in shape to plots of word frequency in English-language texts, such as the plot of the word frequency in English Wikipedia text by Grishchenko [2006]. This indicates that the distribution of crashes per application has a piecewise power law distribution.

We fit a finite Zipf-Mandelbrot model to the frequency spectrum using the `zipfR` R package [13]. In Figure 4, the red line indicates the model distribution and the black line indicates the empirical data. We used the $\chi^2$ goodness-of-fit test to determine how well the model fit the data. The test determined a $p$-value of 0.17. However this $p$-value is not small enough to reject the hypothesis that the data came from the model distribution.

We can conclude that the distribution of crashes among packages follows a power-law distribution. Therefore, a few applications will cause a large number of crashes, and there will also be a large number of applications with only a few crashes. Crash deduplication systems must be able to handle deduplicating both crashes from software that crashes rarely and software that crashes frequently.

5.2 RQ2. How are crashes distributed among buckets?

Minimum  | Maximum | Median | Mean  | Std. Dev. | Skewness | Kurtosis
---|---|---|---|---|---|---
1 | 148 | 1 | 1.4 | 2.5 | 24.0 | 989.4

**Table 2.** Descriptive statistics for crashes per bucket.

Understanding the distribution of crashes in buckets is critical to effective crash report deduplication. It might also be useful for predicting how many major buckets (groups of crashes which are caused by a single underlying bug) will appear based on how many new crashes appeared. Thus it may be useful for predicting how many new crash-causing faults have appeared in the software once the number of new crashes is known. It is also essential if crash report repositories are to be synthesized, in order determine how many synthetic crashes a synthetic bug should produce.

We downloaded all of the crash reports available at the end of 2015. Our modified version of Bicho also downloaded information about which crash each crash was marked as a duplicate of, if any. This information is available from the Launchpad API [28] along with the crashes. It is provided by the API as a field.
Figure 5. Crashes per bucket. Both axes are logarithmic.
Figure 6. Frequency spectrum of crashes by bucket, indicating there are many buckets with only one crash report, and a few buckets with many crash reports. Both axes are logarithmic. The leftmost point indicates there are 25,594 buckets with only a single crash.
indicating which bug report each bug report is a duplicate of (crashes are stored as bug reports in the same system). We used this information to group crashes into buckets. Then we sorted the buckets by size and plotted the result.

Figure 5 shows the distribution of crashes among buckets. The largest bucket had 148 crashes. The crashes concern a library package called gtk-sharp which is used as a GUI toolkit for many applications in Ubuntu. These packages include CD burning applications, photo management applications, and others, all of which crashed due to this library bug.

In addition, there are 1,853 buckets with only 2 crashes in them, and there are 25,594 buckets with only a single crash. Buckets with only a single crash were not used in Campbell et al. [7] because it is not known if they are truly unique or simply have not been evaluated by Ubuntu developers and volunteers enough to be assigned to a larger bucket. In this paper, we consider crashes with no known duplicates as being in a bucket by themselves.

Buckets with only a single crash in them account for more than half of the crashes. There are only 14,998 crashes in the dataset which are in buckets with other crashes.

We fit a finite Zipf-Mandelbrot model to the frequency spectrum using the zipfR R package [13]. In Figure 6 the red line indicates the model and the black line indicates the empirical data. We used the $\chi^2$ goodness-of-fit test to determine how well the model distribution shown fit the data. The test produced a $p$-value less than $10^{-13}$, which indicates the data did not come from the fit distribution. However, Figure 6 clearly shows that the model distribution is able to capture the general shape of most of the empirical distribution.

Much like RQ1, we can conclude that a many crashes will be lonely, that is, being the only crash in a bucket, with no duplicates, while some buckets will be significantly larger, containing a large number of crashes. Crash deduplication systems must be able to correctly place crashes into both small buckets and very large buckets. This means that any deduplication technique that tends toward some average size of bucket will perform poorly. Most importantly, this dataset indicates developers can address large amounts of the crashes that users report by focusing on only a few crash buckets.

5.3 RQ3. How long do crash buckets last?

<table>
<thead>
<tr>
<th>Minimum</th>
<th>Maximum</th>
<th>Median</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$1.28 \times 10^8$</td>
<td>0</td>
<td>$6.3 \times 10^6$</td>
<td>2.5</td>
<td>9.8</td>
<td>124.0</td>
</tr>
</tbody>
</table>

Table 3. Descriptive statistics for lifetime in seconds.

A bucket is first created when a user experiences a crash and reports it. After that, other users might experience the same crash, and those crashes will be added to the bucket. Eventually, the code causing the crashes could be fixed, replaced, or become irrelevant. However, even if it is fixed, it is still possible that later
Figure 7. Empirical and model cumulative distribution functions of bucket lifespan. Only buckets with at least 2 crashes are shown.
changes undo the fix and more crashes are reported. Ideally, all of these crashes are in the same bucket.

It is important to understand how long crash buckets last in order to deduplicate them, simulate their creation over time, and manage crash repositories. For example when deciding on data retention policies, if we found that buckets have a very limited lifespan it would indicate that crashes could be removed after some time, limiting the amount of work that processes working with crashes, such as databases, would need to do.

We analyzed the lifespan of buckets in order to determine whether there are any patterns in bucket lifespan. The lifespan is the time between when the first and last crashes were posted to Launchpad, for each bucket. Of course, there could have been crashes added to each bucket after the data was collected (and could be in the future) so it is impossible to know definitively how long each bucket lasts. We used a Cullen and Frey graph produced by the `fitdistrplus` R package [10] to determine what distribution to attempt to fit to the data.

Figure 7 shows the lifespan of each bucket. The longest-lived bucket lasted for over 4 years, while there are several buckets containing multiple crashes that lasted less than an hour.

The Cullen and Frey graph indicated that the skewness and kurtosis of the data matched that of a beta distribution. An example beta distribution is shown in Figure 7. The distribution was fit using the method of moments. When fitting a distribution and plotting, only buckets with two or more crashes were used. A method of moments fit matches the mean and the variance of theoretical distribution to the empirical distribution. The fit beta distribution shown in Figure 7 has parameters $a = 0.30$ and $b = 2.88$.

We used the Kolmogorov-Smirnov, Cramer-von Mises, and Anderson-Darling goodness-of-fit tests to determine how well the model fit the data. The tests all produced $p$-values less than $10^{-6}$, which indicates that the data did not come from a pure beta distribution of the estimated parameters. However, Figure 7 clearly shows that the beta distribution is able to capture the general shape of the empirical distribution. It is possible that the empirical distribution is a sum of two distributions or the result of multiple processes that a single beta distribution is not able to capture completely.

The beta distribution predicts that more buckets will have lifetimes less than a month than are actually observed. This could be the result of only considering buckets with two or more crashes in them. We feel that the beta distribution is a good starting point for sampling the lifespans of crash buckets when creating synthetic crash repositories.

There is little evidence to support a strong correlation between bucket size and bucket lifetime. The Pearson’s (linear) correlation is 0.17, which is a weak, correlation, but it is statistically significant, with a $p$-value of $2 \cdot 10^{-12}$. Similarly, the Spearman’s (rank) correlation is 0.23, also statistically significant with a $p$-value of $2 \cdot 10^{-16}$. Despite the weakness of the correlation, it is present. Our initial hypothesis was that larger buckets would have a longer lifespan, because
Figure 8. Lifespan of each bucket. Only buckets with at least two crashes are shown.
with more crashes in a bucket, the bucket has more opportunities to have crashes which are spaced further apart in time. This relationship is shown in Figure 8.

However, we also expect that there is a competing effect: the more crashes a single bug causes, the more likely it is to be fixed by drawing the attention of Ubuntu developers or volunteers. The bucket with the largest lifespan, lasting from 2007-08-13 until 2011-09-12 (\(\sim\) 4 years, 30 days), had 39 crashes. However, the bucket with the second largest lifespan lasted from 2009-10-03 until 2013-10-16 (\(\sim\) 4 years, 12 days) and only contained two crashes. This is possibly due to a bug which was present in the code for a long time. In fact, the bug report for this crash is still open at the time of writing (2016-11-07).

The age of a crash may be artificially shortened due to the Launchpad maintainers deeming a crash report to be “unreproducible.” Crashes (which are treated as a type of bug in the Launchpad system) are marked as “unreproducible” if the report does not contain enough information to replicate the conditions under which the crash occurred. A crash may be considered unreproducible even if it contains a stack trace, context about the machine, and a comment from the user who experienced the crash. The issue will be closed with a status of “Invalid” [46].

The empirical data shows that the lifespan of buckets is highly variable, and that it would be incorrect to assume that it is limited. If the dataset contained 60 years of data rather than 6, crashes with lifetimes even longer than 4 years would probably be observed. Thus, crash repositories should keep data for as long as possible, because some old crashes will be experienced again. In addition, crash deduplication systems must be able to find duplicates of any age.

Duplicate crash reports can also arrive in fairly rapid succession. In this dataset there are buckets with more than one crash, which last less than an hour. This indicates that crash deduplication systems must be able to form buckets quickly. For example, if the deduplication system only looked for similar crashes in an archive, and the archive was only updated once an hour, it might miss duplicate crashes.

5.4 RQ4. What Unix signals are crashes associated with?

Signals are sent to processes running on Linux when they crash. They can indicate what kind of crash the process experienced. Sometimes they are sent by the Linux kernel when it detects a crash, and sometimes they are sent by the program itself or by other programs on the system when they detect some fault condition in order to crash the process. These are similar to exceptions in managed languages like Java, where there has been prior work [24] to understand which exceptions crash programs.

Because the signal indicates what general type of crash occurred, 2 crashes with different signals are unlikely to be caused by the same underlying bug. Therefore the signal is very useful for crash deduplication. Furthermore, it would be desirable for synthetic crash reports to contain similar metadata that can be used to distinguish large categories of crashes.
Figure 9. Signals causing crashes.
In order to investigate the ways in which applications crash, we used the crash metadata which usually contains a field indicating what signal a program crashed with. An example can be seen in Figure 1.

Figure 9 shows the amount of crashes associated with each signal in the dataset. Due to the semi-structured text nature of the data, 101 crashes either did not have signals or the signals were not parsable.

The most common signal, **SEGV** (a **segmentation violation**), indicates that memory has been accessed in an illegal way. This can happen when a program tries to access memory that does not belong to it, has yet to be allocated, or has been deallocated. It can also happen if the program tries to write to read-only memory. **SEGV** can also occur when a program uses up all of the available stack space due to unbounded recursion (Section 5.7). It is sent to the program by the Linux kernel.

**ABRT** was the second most common signal and is typically caused by the program sending **ABRT** to itself. This can happen, for example, when a program attempts to deallocate memory which has already been deallocated. In this case, the fault is not detected by the OS kernel, but rather, by a library in use by the program.

The third most common signal causing crashes was **TRAP**. This signal is typically used to invoke a C debugger such as **gdb** [12]. However, when a program receives **TRAP** and is not being debugged, the default action is to terminate the program. Thus it is used to either exit the program or pause the debugger when some fault condition is detected [44].

**BUS** was the fourth most common signal. Originally it was intended to indicate that there was something wrong with the hardware memory bus, but this usage is not common anymore. This is because serious hardware faults are now handled by the kernel and typically crash the entire system. **BUS** signals do still occur in other situations, such as when a program tries to read data past the end of a file [33].

The fifth most common signal, **FPE** indicates a floating-point math error, such as division by zero. It was followed by **XCPU** and **XFSZ** which indicate that a program has used more than its allowed CPU time and that a program has made a file which is larger than the maximum allowed size, respectively. The least commonly seen signal was **SYS** which is described as unused.

From this information we can conclude that C and C++ software running on Ubuntu mostly crashes due to illegal memory access. This may indicate that there could be fewer crashes if memory-safe languages were used for these applications instead.

**5.5 RQ5. What CPU architectures experienced the crashes?**

When using a crash repository to evaluate deduplication methods, it is important to understand how the computer architecture will affect those findings. For example, Campbell *et al.* [7] found that the addresses of functions was a possible method of deduplicating crashes using an earlier version of the Launchpad dataset used in this analysis. However, we can see that the dataset contains two types of addresses: 32-bit and 64-bit. A 32-bit address will never match a 64-bit address.
Figure 10. Crashes by architecture.
We used the “Architecture” metadata field provided in most Ubuntu crash reports to examine how often crashes were reported on different CPU architectures. An example can be seen in Figure 1. Due to the semi-structured text nature of the data, 55 crashes either did not have architecture information or the architecture information was not parsable.

Figure 10 shows the amount of crashes occurring in each CPU architecture. The crashes in the dataset consist primarily of the two PC architectures, i386 and amd64. The most popular is the 32-bit i386 architecture, even though it is becoming less common. The second most popular is the amd64 architecture, which is the 64-bit version of the i386 architecture. amd64 is also commonly referred to as x86_64 in other places, and does not necessarily indicate that the CPU in use is an AMD.

Less popular architectures are armhf and the older armel. Both armhf and armel are ARM architectures, which differ on how they support floating-point math. The least popular architecture is ppc, indicating PowerPC CPUs. There is one crash labeled i686, but this is likely due to the person who submitted the crash mistakenly editing i386 before they submitted it to Launchpad.

Because crash repositories can contain crashes from several hardware architectures, any system that works with crash reports—whether for storage, data mining, or deduplication—needs to function regardless of changes in metadata caused by differences between architectures. These differences include handling both 32-bit and 64-bit addresses.

### 5.6 RQ6. How long are crash stack traces?

<table>
<thead>
<tr>
<th>Minimum</th>
<th>Maximum</th>
<th>Median</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2654</td>
<td>16</td>
<td>42.7</td>
<td>201.8</td>
<td>9.5</td>
<td>91.6</td>
</tr>
</tbody>
</table>

Table 4. Descriptive statistics for stack length.

Because stack traces are a major component of a crash report, the longer a stack trace is, the more data a crash report contains. That data may then be used as a source of features for a crash report deduplicator. For example, if all crashes only had one function in its stack trace, it would be impossible to distinguish crashes from each other based on the stack trace alone. A factor that may affect the length of stack traces is recursion (Section 5.7).

Using our crash report extraction scripts (described in Section 4), we were able to count the amount of stack frames to determine the stack trace length for each crash. Then we fit a geometric distribution to the empirical data.

Figure 11 shows a log-log plot of the lengths of stacks found in crashes. The peak at stack length 2000 is caused by a hard limit coded into gdb which prevents it from generating traces with more than 2000 frames. There were 413 such stack traces truncated at 2000 frames. The code that performs this truncation is defined in tracefile.c in the gdb source with #define MAX_TRACE_UPLOAD 2000 [38]. The truncated stack traces always include the top of the stack (the
Figure 11. Stack trace lengths.
point of the crash), but discard the bottom of the stack (the entry point of the program). There was only one crash which exceeded the truncation limit with 2654 stack frames which was an internal error from the Mono C# runtime [14].

In Figure 11, the red line indicates the model and the black bars indicate the empirical data. The model distribution is geometric distributed with parameter $p = 0.060$. We used the $\chi^2$ goodness-of-fit test to determine how well the model fit the data. The test determined a $p$-value of less than $10^{-13}$, which indicates that the data did not come from a geometric distribution with the parameters listed. However, it appears to capture the shape of the data with stack lengths between 10 and 50 quite well.

The fact that some of the data seems to follow a distribution which appears similar in shape to a geometric distribution, indicates that there may be some fixed chance of any function calling (or not calling) another function, and making the stack one frame longer. However, the small $p$-value, and the tail that extends past the fit geometric distribution indicates that this is only a part of the process that determines stack length.

The truncation behaviour in gdb is problematic for a number of reasons. First, this discarded information may be crucial for debugging, because the point that triggered the crash may be missing—even though it is essential for debugging the root cause of the crash. Second, it is incomplete information that may produce incorrect results in a naïve tool, relying on the fact that the bottom of the stack may contain the entry point of the program.

Future crash reporting systems should not truncate the stack if possible, to avoid losing information about what the software was doing before it began to crash. In addition, crash deduplication systems need to be able to determine if stacks are similar regardless of stack length.

5.7 RQ7. How are crashes associated with recursion?

When synthesizing crash reports, it is important to know whether to model recursion, both bounded and unbounded; when creating crash report deduplication algorithms, it is useful to know whether unbounded recursion can be used as a feature to cluster crashes together.

We found all cases of single-function recursion or trivial recursion. To find cases of trivial recursion, we considered all stack traces wherein at least 2 consecutive stack frames contained the same function name. Then, we counted the number of consecutive stack frames with the same function to find the recursion depth. Note that each stack trace with recursion may contain more than one instance of recursion; thus, we counted the instances of recursion contained within one stack trace. We considered any trivial recursion as being unbounded if the recursive function was at the top-of-stack and the stack length exceeded gdb’s hard-coded limit, described in Section 5.6.

Summary descriptive statistics for recursion depths are provided in Table 6. We plotted the depth of all instances of trivial recursion in Figure 12, bottom. There appears to be at least two different distributions at play: one wherein there is shallow recursion, never exceeding 25 stack frames deep; and one wherein there
Figure 12. Recursion depth. The top plot shows lengths 2 to 2000 and bottom right plot shows lengths 2 to 30 of the same data. A function is considered recursive if there are at least two consecutive stack frames with the same function name, hence the minimum depth of recursion is 2.
Table 5. Descriptive statistics for recursion depths.

<table>
<thead>
<tr>
<th>Minimum</th>
<th>Maximum</th>
<th>Median</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2000</td>
<td>2.0</td>
<td>4.8</td>
<td>72.3</td>
<td>26.9</td>
<td>732.1</td>
</tr>
</tbody>
</table>

is deep, perhaps unbounded recursion. There is an additional clump of recursion depths around 1000 stack frames deep.

5,947 crash reports had trivial recursion associated. In those reports, 28,172 instances of recursion for 2 or more frames were identified. Thus, many crash reports had multiple instances of recursion within their stack traces. Of the 413 truncated stack traces (Section 5.6), 123 of these stack traces exhibited unbounded trivial recursion, leaving 290 truncated stack traces unaccounted for.

We fit a negative binomial distribution to recursion depths less than 20 stack frames. The fit distribution is shown in red in Figure 12. An offset of 2 was applied to the distribution, because recursion must have a depth of at least 2. The $\mu$ (mean) parameter was 0.086 and the dispersion parameter (size) was 0.047. The fit distribution had a $p$-value of 0.0014 according to the $\chi^2$ test, indicating that the data likely did not come from the fit distribution. However, based on Figure 12, the fit distribution captures the general shape of the empirical data for recursion depths less than 8.

A crash in Rhythmbox exhibited unbounded trivial recursion [46]. All 2000 reported stack frames represented the same function: `rb_removable_media_source_should_paste_no_duplicate();` thus, the entry point to this C program, `main();`, and all ancestors to the function call that caused the unbounded recursion were discarded from the stack trace that was ultimately uploaded to Launchpad. This function was defined within a Rhythmbox plugin, Rhythmbox-Spotify-Plugin [35]. The bug is interesting, because there is no explicit call to the function; rather, a function pointer is assigned to a member of a `struct` [34], and a pointer to that `struct` is eventually passed to the crashing function itself. This function innocently calls the function pointer, implicitly calling itself [35].

For the 15 percent of crash reports that had trivial recursion, we can not necessarily say whether or not the recursion caused the crash, except in the case of the 123 stack traces with trivial unbounded recursion. However, we can conclude that recursion occurred in a crash in a significant fraction of crash reports, and that it is capable of causing a crash. Synthetic crash report data must contain examples of both bounded recursion and unbounded recursion. Crash deduplication systems should be able to use information from both recursive functions and non-recursive functions.

5.8 RQ8. How long are function names in crashes?

Function names are an important source of information when debugging, categorizing, or deduplicating crash reports because they indicate what the software was doing when it crashed. The longer (and hopefully more descriptive) a function name is, the more information it contains. This is supported by prior work on the
Figure 13. Function name lengths in tokens as produced by the CamelCase tokenizer.
quality and efficacy of identifier names [29, 30]. However, in this study a single “function name” may include multiple identifiers, such as its declaring class, and any C++ template parameters. In order to extract that information, function names must be tokenized.

In order to evaluate function name lengths in words, we used the CamelCase tokenizer used in Campbell et al. [7] for crash report deduplication. We re-implemented the tokenizer in Python (originally, it was implemented in Java) and then applied it to every function name in the dataset. The CamelCase tokenizer splits words into tokens on symbols and at lowercase-to-uppercase transitions. It is a regular expression suggested by the ElasticSearch documentation [19].

Sigurd et al. [42] fit gamma distributions to word lengths in syllables and sentence lengths in words in various natural languages. Because words in a function name serve a similar purpose to words in a sentence, and often contain natural language vocabulary words [29, 30], we attempted to fit a gamma distribution to the tokenized function name lengths as well.

However, as shown in Figure 13, the Gamma distribution does not appear to be a particularly good fit for function name lengths in tokens. When only unique tokens are considered, the tail of the distribution extends past where it is predicted by the gamma distribution. The gamma distribution fit for every function name seen in every frame had parameters $\alpha = 3.6$ and $\beta = 0.74$. The gamma distribution fit for unique function names had parameters $\alpha = 6.3$ and $\beta = 1.3$. $p$-values were computed using the Kolmogorov-Smirnov, Cramer-von Mises, and Anderson Darling tests for both distributions, but they were all less than $10^{-16}$, indicating the data did not come from the fit distributions. The fit distributions only appear to capture the shape of the data for function names less than around 10 tokens.

Occasionally, function names can become very long when using C++ templates as a metaprogramming device. One example contains function names with over 300 tokens in over 2000 characters using the CamelCase tokenizer [25].

From this data we can conclude that function names can range from very short, single-token names to very long, multiple identifier names involving C++ templates. Thus, crash repositories and crash deduplication systems must be able to handle function names which range from short to very long. Synthetic crash repositories should contain function names of a wide variety of lengths.

5.9 RQ9. What are the most common crashing functions?

An important piece of information about the cause of a crash is what the program is doing at the time of the crash. The functions present in the stack at the time of the crash indicate what the software was doing when it crashed. This is usually

<table>
<thead>
<tr>
<th>Minimum</th>
<th>Maximum</th>
<th>Median</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>331</td>
<td>3</td>
<td>2.9</td>
<td>5.7</td>
<td>38.2</td>
<td>1987.6</td>
</tr>
</tbody>
</table>

Table 6. Descriptive statistics for function name lengths.
<table>
<thead>
<tr>
<th>#</th>
<th>Count</th>
<th>Function name</th>
<th>Most common signal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Name</td>
</tr>
<tr>
<td>1</td>
<td>3390</td>
<td>_kernel_vsyscall</td>
<td>ABRT</td>
</tr>
<tr>
<td>2</td>
<td>1442</td>
<td>_GI_raise</td>
<td>ABRT</td>
</tr>
<tr>
<td>3</td>
<td>1440</td>
<td>g_logv</td>
<td>TRAP</td>
</tr>
<tr>
<td>4</td>
<td>1237</td>
<td>raise</td>
<td>ABRT</td>
</tr>
<tr>
<td>5</td>
<td>544</td>
<td>g_type_check_instance_cast</td>
<td>SEGV</td>
</tr>
<tr>
<td>6</td>
<td>299</td>
<td>strlen</td>
<td>SEGV</td>
</tr>
<tr>
<td>7</td>
<td>245</td>
<td>free</td>
<td>SEGV</td>
</tr>
<tr>
<td>8</td>
<td>230</td>
<td>pthread_mutex_lock</td>
<td>SEGV</td>
</tr>
<tr>
<td>9</td>
<td>220</td>
<td>malloc_consolidate</td>
<td>SEGV</td>
</tr>
<tr>
<td>10</td>
<td>205</td>
<td>g_slice_alloc</td>
<td>SEGV</td>
</tr>
</tbody>
</table>

Table 7. The top 10 most common function names in crash stack traces. Each function name is counted at most once per crash.

Table 8. The top 10 most common function names at the top of crash stack traces. The most common signal associated with the function is given. An explanation of what each signal means is provided in Section 5.4.
Figure 14. Frequency spectrum of crashes by the function on top of the stack trace, indicating there are many functions that only appear on top of the stack trace once, and a few functions that appear on top of the stack trace in many crash reports. Both axes are logarithmic.
indicated by the function on the top of the stack at the time of a crash. It is also considered to be the most important part of the stack in many papers on crash deduplication [45, 17, 36]. Thus, by examining the functions on top of the stack, we can determine how reliable the top of the stack is for determining the cause of a crash.

To collect function names, we iterated through every crash in the dataset, and collected every unique function name present in the crash stack trace. Not every stack frame had a function name. Function names are often missing due to missing debugging symbols which gdb relies on to produce this information. Once collected, we counted the number of crashes each function name appeared in. Only 36,511 out of the 40,592 crashes (89.9%) had any parsable function names at all. Of these, 27,406 crashes had parsable function names on the top of the stack (67.5% of all crashes). Thus, the results presented here are a subset of the total corpus. The results of the most common unique function names within a stack trace are presented in Table 7, and the most common functions at the top of the stack, along with their crashing signal, are presented in Table 8.

Figure 14 shows the frequency spectrum of the top functions. We fit a finite Zipf-Mandelbrot model to the frequency spectrum using the zipfR R package [13]. In Figure 14, the red line indicates the model distribution and the black line indicates the empirical data. We used the $\chi^2$ goodness-of-fit test to determine how well the model fit the data. The test determined a $p$-value of less than $10^{-75}$. However, the shape of the model distribution seems to follow the empirical distribution quite closely, with the exception of functions that only appear in a single crash (the leftmost points plotted in Figure 14). This could indicate some other process is affecting the number of functions which are only seen once.

Despite being the entry point for most C and C++ programs (thus, most likely to be the top function present in stack traces), main() is present in only 15,477 crashes (42.4% of crashes with function names). Part of this may be explained by the stack truncation reported in Section 5.7.

Because the most common function on the top of the stack merely indicates that some signal was raised, but not which, it alone is often a poor indicator of what the software is truly doing at the time of the crash. This is why we included its most common crashing signal. In particular, the function __kernel_vsyscall is a generic way to invoke any system call in Linux, and as such, its most common signal, ABRT, does not account for 22.2% of all its crashes. The second, third, and fourth most common functions on top of the stack (__GI_raise, g_logv, and raise, respectively) merely indicate that the program is crashing itself intentionally after detecting some fault condition. This is supported by ABRT (intentionally aborts the program) and TRAP (causes the debugger to break here, or crash if not attached to a debugger) being the most common signals for these crashes. The rest of the top ten are caused by the top signal overall SEGV, or segmentation violations (Section 9). These are often caused due to passing an invalid pointer or a NULL pointer to some function argument, causing the function at the top of the stack to access memory in some invalid manner. For example, the fifth most common crasher (g_type_check_instance_cast) indicates that an
<table>
<thead>
<tr>
<th>#</th>
<th>Count</th>
<th>Library name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2950</td>
<td>libc</td>
</tr>
<tr>
<td>2</td>
<td>1854</td>
<td>libglib-2</td>
</tr>
<tr>
<td>3</td>
<td>469</td>
<td>libpthread</td>
</tr>
<tr>
<td>4</td>
<td>376</td>
<td>libgobject-2</td>
</tr>
<tr>
<td>5</td>
<td>341</td>
<td>libgtk-x11-2</td>
</tr>
<tr>
<td>6</td>
<td>309</td>
<td>libGL</td>
</tr>
<tr>
<td>7</td>
<td>268</td>
<td>ld-linux</td>
</tr>
<tr>
<td>8</td>
<td>152</td>
<td>libQtCore</td>
</tr>
<tr>
<td>9</td>
<td>143</td>
<td>libdbus-1</td>
</tr>
<tr>
<td>10</td>
<td>141</td>
<td>libQtGui</td>
</tr>
</tbody>
</table>

Table 9. The top 10 most common library names at the top of crash stack traces.

An invalid pointer was passed into a glib function. The sixth most common function on top of the stack does not give a clear indication of the bug: either a bad pointer was passed to `strlen`—a C function that computes the length of a string—or the string was not null-terminated, perhaps due to being invalid or corrupted data.

The top of a stack trace is often not a good indication of what exactly the fault is. Thus, crash deduplicators must not rely on the top of the stack trace. Instead, they should examine the entire stack for clues, and contextual data like the crashing signal for clues. Additionally, synthetic crash repositories should contain crash reports with functions specific to the crash on top of the stack, and general purpose functions on top of the stack.

The top of the stack is populated with intentional crashes, aborts, and syscalls, as well as calls to the standard library with inappropriate data. Again we see that improper memory management affects many crashes rather than resource depletion.

5.10 RQ10. What are the most common crashing libraries?

While crashes are often specific to a particular product, similar faults or misuses of a library can occur across the clients of a library. Even worse, if a library has a fault it could induce crashes across numerous client products. Thus we ask the question what are the most common crashing libraries. Are they just misuse of libc or is GUI client code causing instability? Are device drivers at play or concurrency? In this section we look for evidence of libraries whose use can cause or correlate with instability across many products.

In order to determine the libraries which crashed the most often, we examined the top of the stack trace. For some crash reports, frames have the library file listed in the crash report. For others, the source file is listed, or there is no information about the origin of the function. We only considered stack traces that had the library file specified, because it could be parsed the most reliably.
11,569 out of 40,592 crashes had a library file specified for the function at the top of the stack.

The 10 most common crashing libraries are listed in Table 9. The most common library in which crashes occurred was libc, the C standard library. Yet libraries about concurrency, libpthread, GUI libraries, libgtk-x11-2, libQtGui, libQtCore, and even 3D graphics, libGL, are ranked top crashers.

Using our methodology we could only determine what library functions software crashed inside. However, a library function may crash because it was supplied invalid arguments by another part of the software. Therefore, we cannot necessarily conclude that the libraries which crash the most often are the most buggy. For example, the library which crashed the most often, libc is probably the most well-tested C library because it is used by every C program on Linux. Thus, it is not likely that the crashes in libc are caused by faults in libc, but rather by software using libc. libGL also signals that graphical device drivers could also be at fault for instability.

The libraries and frameworks that tend to induce crashes are both popular and infrastructural. Many crashes seem to be the result of passing invalid parameters (invalid pointers) to library and framework code. Ubuntu’s UI are dominated by GTK and QT based programs so it is not surprising to see these functions appear in stack traces. We are left wondering if managed code environments such as C# or Java exhibit the same properties.

6 THREATS TO VALIDITY

6.1 Threats to construct validity

Construct validity is threatened by the fact that parsing the semi-structured textual crash reports is not completely reliable. As stated, 3,872 reports (9 percent) of the collected reports were thrown out because they could not be parsed. Additionally, the crashes that could be parsed were not always perfectly reliable. For example, a missing newline character can cause data from one field to appear in another field. However, we believe these issues to be very rare, affecting only a handful of crashes.

6.2 Threats to internal validity

Internal validity is threatened by the validity of the Ubuntu Launchpad dataset itself. We trusted that the manually tagged duplicates were true duplicates and that all true duplicates were found and clustered appropriately. This affects our conclusions about the distribution of crashes among buckets (Section 5.2) and the lifespan of crash buckets (Section 5.3).

Even though we only studied crashes for which we could parse stack traces, not all stack frames contained crucial information such as function names, line numbers, and filenames. As such, this fact may affect our conclusions depending on function names (Section 5.7, Section 5.9, and Section 5.8).

A threat to internal validity is alluded to in Figure 2 is that end-users have the ability to opt-out of submitting a crash report. As such, this biases the data that
is ultimately available for analysis—both in this paper, and for other purposes, such as crash report deduplication. In addition, the crashes that do appear in Launchpad require that the users have a Launchpad account, as the bug reports are intended to be uploaded by more technically-minded users. This further biases the data we have available for study. This affects our conclusions, as we cannot reason about crash reports that were never uploaded.

6.3 Threats to external validity

The crashes from the Launchpad dataset came exclusively from Ubuntu-based distributions of Linux, focusing on personal computing and server applications (such as Rhythmbox, a music player, and Nautilus, a graphical file manager). As such, our conclusions may not be applicable to other platforms and operating systems such as macOS and Windows, different crash report repositories, or even different distributions of Linux.

Since the stack traces we collected exclusively come from software debuggable by *gdb*, most of the crashes come from C and C++ software. The results here may not generalize to software written in other programming languages such as Java or Python.

Because the Launchpad dataset consists of crashes from many different applications, the results in this paper may not generalize to crash repositories that contain only crashes from a single application such as Mozilla's [2012].

7 CONCLUSIONS

We analyzed 40,592 crash reports for 1,921 packages from the Ubuntu Launchpad crash repository. From this dataset, we found that various properties of crash reports tend to have empirical distributions which, at least partially, resemble standard statistical distributions. However, crash reports are complicated, and no single standard distribution fit the entire range of crash reports we observed.

When grouped by package or by bucket, crashes exhibited power-law distributions. There are a few groups with many crashes, and many groups with few crashes. This fact should be taken into account when deciding which crashes are similar, and when determining which crashes should receive developer attention. Crash deduplication systems must be able to handle this wide variety, and synthetic crash repositories should generate crashes in groups that follow similarly extreme distributions. The dataset also shows that it is not possible to rely on buckets having a limited life span. Thus, crash repositories and deduplication techniques must allow crashes to be added to buckets regardless of their age.

Crashes in this dataset, which consists of programs written in natively compiled languages, such as C and C++, are heavily impacted by the hardware architecture. Crashes from two different architectures may be difficult to compare despite being caused by the same underlying software fault. In addition, the most common fault type for the crashes in the dataset was illegal memory access (*SEGV*). This indicates that preventing illegal memory access may be a valuable approach to preventing crashes.
Crash stack traces and function names can range from short to very long. Crash repositories and crash reporting tools should be able to handle any length of stack trace. Crash deduplication techniques must be able to handle any length of stack trace and a wide range of lengths of function names, despite the fact that long stack traces and long function names provide much more information that deduplication techniques can use than short function names and short stack traces. Additionally, deduplication techniques should handle crashes with no recursion, bounded recursion, and unbounded recursion.

Popular infrastructural, GUI, and standard libraries and functions appeared in stack traces. Graphical libraries such as libGL and concurrency libraries such as libpthread occurred as well. Essentially complicated and popular infrastructure libraries and functions appeared frequently in crashes. While these could be common errors that clients are susceptible to, it might be worthwhile for library maintainers to understand how their code becomes involved in these crashes. Perhaps library maintainers could provide checks for the more common crashes?

Overall, this data indicates that there are many crashes that have similar characteristics, and many crashes that have unique and exceptional characteristics. Crash repositories, dedupicators, or other systems cannot rely on crashes following common patterns or expressing limited ranges, because every property has significant outliers. For example, crash repositories cannot rely on buckets having a consistent size, applications causing similar numbers of crashes, or buckets having limited lifespans. Crash dedupicators cannot rely on crashes having functions with addresses or names of similar length, or stack traces of similar lengths, and they should examine the entire stack, not just the first function. Recursion can cause problems by making the stack too long or by hiding the bottom of the stack if it gets truncated. In order to test crash dedupicators with synthetic crash reports, synthetic crash reports must be generated using processes which result in long-tailed distributions.

Future work includes analyzing other crash repositories to determine if crashes from those repositories follows similar distributions to the crashes in the Ubuntu Launchpad dataset. In addition, studying crashes in languages such as Python, which crash due to exceptions rather than signals, and are unlikely to experience a SEGV signal, might give better insight into why software crashes.

REFERENCES


SysML.


[40] Schaaf, M. (2012). rhythmbox crashed with SIGSEGV.


