ABSTRACT

SYED, RAZA ABBAS. Investigating Intermittent Software Failures. (Under the direction of Dr. Laurie Williams).

Intermittent software failures and nondeterministic behavior complicate and compromise the effectiveness of software testing processes. Software engineers spend a significant amount of time attempting to reproduce such intermittent and non-reproducible software failures, but often are unable to do so and instead close the failure as “Not Repeatable” or “Not Reproducible”. Whether such failures are detected during internal testing or by customers in the field, they represent a need for the development of versatile testing strategies that minimize non-deterministic behavior in software systems.

We explored the realm of intermittent software failures by conducting three case studies on a large industrial system (ABB 800xA), and an open source system (Mozilla Firefox). For our first study, we explored the observability of intermittent failures by altering processor speeds, memory capacities, and processor loads. In our second study, we analyzed failure reports describing intermittent software failures and developed a classification scheme based on common characteristics we identified from the failure reports. Additionally, we identified common attributes of intermittent behavior in software systems. For our third study, we extracted useful metrics from software failure repositories of two large software systems to establish the statistical significance of effort spent debugging intermittent failures. We also used the extracted metrics to train a prediction model for classifying a failure report as intermittent or reproducible at the time the failure is reported.
We found that decreasing processor speed and increasing processor load had a statistically significant relationship with frequency of occurrence of intermittent failures. Also, using metrics such as number of comments and failure resolution time, we found that software engineers spent significantly more time debugging intermittent failures than reproducible failures. Finally, we demonstrate that metrics extracted from failure reports and metrics based on natural language processing techniques can be used to predict that a failure may be intermittent at the time the failure is reported. Our prediction models were able to correctly predict more than 60% of failure reports as being intermittent.

Software engineers can supplement existing testing techniques with our findings to better deal with intermittent failures in their systems. By using our identified attributes and classification of intermittent failures, software engineers can improve the quality of their failure reports by incorporating this knowledge in their failure reports. Additionally, by using our failure prediction model, potential intermittent failures can be identified earlier in the failure lifecycle. Identifying intermittent failures early can lead to significant savings in terms of cost and time, and will improve the triaging process.
Investigating Intermittent Software Failures

by
Raza Abbas Syed

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Science

Raleigh, North Carolina
2011

APPROVED BY:

__________________________  __________________________
Dr. Laurie Williams        Dr. Tao Xie
Committee Chair

__________________________
Dr. Emerson Murphy-Hill
DEDICATION

To the *City of Knowledge*, and it’s *Gate*.

To *Papa, Baray Abbu, Bari Ammi, and Daadi Jaan* (my grandparents).

And especially to *Ammi* and *Abbu* (my parents).
BIOGRAPHY

Raza Abbas Syed was born in Karachi, Pakistan. He got his elementary and high school education at Army Public School and College, Malir Cantonment, Karachi. He went on to get his undergraduate degree in Computer and Information Systems Engineering from NED University in 2006. Following graduation, he worked as a Software Developer for two local companies for 16 months before coming to NC State for his graduate degree. Raza was part of the Realsearch software engineering research group at NC State. He will be graduating with a Master’s degree in May 2011.
ACKNOWLEDGMENTS

First and foremost, I would like to thank Laurie for being the best advisor ever; for putting up with me for two years; for her guidance, and showing me my path forward. I really couldn’t have asked for a nicer, more understanding advisor.

Thank you to Brian Robinson at ABB for being incredibly helpful on my research, and for being patient when things weren’t going so well. It was a pleasure working at ABB over the summer of 2009 and 2010. For that, I would like to thank the entire ISS team -- Andy, Pat, Will, Mithun, Vinay, and Xiao. Thanks for all your help guys.

Thank you to the entire Realsearch group – Andy, Ben, Yonghee, John, Pat, Sarah, Mei, Andrew, and Jerrod. It was a great experience working alongside such smart people. Our reading groups taught me the importance of peer review and getting feedback on one’s work, not just in academia, but on a more general level.

Thanks to Aapi (my sister) for those home-cooked meals. Thanks to Haider bhai (my brother-in-law) for helping me get settled in a new country and for countless other things over the last three years. You guys are incredibly generous, and I love you.

Thank you to Bhayya (my brother) and his family for always being available for help and advice, and for showing me around the beautiful Pacific Northwest.
Thank you to my fellow Pakistani friends at NC State – Sajeel, Ahmer, Maqbool, Asim, Fakeha, Qasim, Zohaib, Imran, Sadaf, Kashif, Aown, and Easa. Thanks for keeping me sane all this time at grad school.

This work was sponsored by ABB Corporate Research.

Some of the work in this thesis incorporates material from the following conference publication, which is copyright of IEEE © 2010:

# TABLE OF CONTENTS

LIST OF TABLES ......................................................................................................................... ix  
LIST OF FIGURES ....................................................................................................................... x 
1. INTRODUCTION ...................................................................................................................... 1 
2. BACKGROUND ......................................................................................................................... 5 
   2.1 Prevalence of Intermittent Behavior ............................................................................... 5 
      2.1.1 iPhone Alarm Faults ............................................................................................... 5 
      2.1.2 Therac-25 Accidents ............................................................................................... 6 
   2.2 Types of Intermittent Behavior ....................................................................................... 6 
      2.2.1 Configuration-dependent failures ........................................................................... 7 
      2.2.2 Unanticipated conditions ..................................................................................... 8 
   2.3 How do modern-day software organizations deal with intermittence? ....................... 9 
      2.3.1 Deterministic Replay Debugging ........................................................................... 9 
      2.3.2 Fuzz-testing .......................................................................................................... 9 
      2.3.3 High Volume Test Automation ............................................................................. 10 
      2.3.4 Load testing ......................................................................................................... 10 
      2.3.5 Disturbance Tests ................................................................................................. 11 
   2.4 Repeatability of Classification Scheme .......................................................................... 11 
3. RELATED WORK ................................................................................................................... 14 
4. EFFECTS OF HARDWARE CONFIGURATION ON OCCURRENCE OF INTERMITTENT FAILURES ......................................................................................... 20 
   4.1 Empirical Study .............................................................................................................. 20 
      4.1.1 Step One: Identify Intermittent Failures ................................................................. 21 
      4.1.2 Step Two: Determine Hardware Configurations .................................................. 22 
      4.1.3 Step Three: Hardware Configuration Testing ....................................................... 24 
      4.1.4 Step Four: Processor Load Testing ........................................................................ 25 
      4.1.5 Establishing a Baseline ......................................................................................... 26 
      4.1.6 Observability Data Collection and Tools .............................................................. 26 
      4.1.7 Threats to Validity ................................................................................................. 27 
   4.2 Selected Failures ............................................................................................................. 28 
   4.3 Results and Analysis ..................................................................................................... 32
4.3.1 Hardware Configuration Study ................................................................. 32
4.3.2 Processor Load Generation Study ............................................................. 38
4.4 Baseline Failure Analysis .............................................................................. 44
4.5 Conclusions ................................................................................................ 45
  4.5.1 Is the observability of a software fault impacted by processor speed? If so, how? 45
  4.5.2 Is the observability of a software fault impacted by the amount of memory? If so, how? .................................................................................................................. 46
  4.5.3 Is the observability of a software fault impacted by the size of the hard drive? If so, how? .................................................................................................................. 46
  4.5.4 Is the observability of a software fault impacted by processor load? If so, how? 46

5. CLASSIFICATION OF INTERMITTENT FAILURES .............................................. 47
  5.1 Study ........................................................................................................... 47
    5.1.1 Systems Under Test ................................................................................. 48
    5.1.2 Data Collection ....................................................................................... 49
    5.1.3 Classification Methodology ..................................................................... 50
  5.2 Classification of Intermittent Failures ........................................................... 52
    5.2.1 Industrial System: ABB ......................................................................... 57
    5.2.2 Open Source System: Mozilla Firefox .................................................. 58
    5.2.3 Repeatability of Classification Scheme .................................................. 59
  5.3 Attributes of Intermittent Failures ............................................................... 61
    5.3.1 Repeatability of Assigned Attributes ..................................................... 64
  5.4 Additional Observations .............................................................................. 67
  5.5 Limitations and Threats to Validity ............................................................. 68
  5.6 Conclusion ................................................................................................ 69

6. EFFORT SPENT DEBUGGING INTERMITTENT FAILURES ................................. 71
  6.1 Study ........................................................................................................... 71
    6.1.1 Data Collection ....................................................................................... 72
  6.2 Number of Comments .................................................................................. 73
  6.3 Failure Resolution Time ............................................................................... 75
  6.4 Length of Failure Reports ............................................................................ 77
  6.5 Failure Severity ............................................................................................ 78
6.6 Additional Observations ................................................................. 80
6.7 Summary ..................................................................................... 80
6.8 Limitations .................................................................................. 81
7. PREDICTION OF INTERMITTENT FAILURES ................................... 82
  7.1 Prediction Modeling ................................................................. 83
    7.1.1 Modeling Techniques ......................................................... 84
    7.1.2 Natural Language Processing Techniques .......................... 84
  7.2 Results .................................................................................... 85
    7.2.1 Industrial System: ABB ...................................................... 86
    7.2.2 Open Source System: Mozilla Firefox ............................... 89
  7.3 Limitations ............................................................................. 91
  7.4 Conclusions .......................................................................... 91
8. CONCLUSION AND RECOMMENDATIONS ..................................... 93
  8.1 Identifying Intermittent Failures ............................................... 93
  8.2 Increasing reproducibility of intermittent failures .................... 94
9. REFERENCES ............................................................................. 98
APPENDICES .................................................................................. 105
Appendix A. Common Words Identified by NLP Techniques ............. 106
  A.1 ABB ..................................................................................... 106
  A.2 Mozilla Firefox ....................................................................... 107
LIST OF TABLES

Table 1. Example Rater Classifications ................................................................. 12
Table 2. Levels of Independent Variables Used In Our Study .................................. 23
Table 3. Hardware Configurations Used In Our Study ............................................ 25
Table 4. Hardware Configurations Used In Processor Load Study ............................. 26
Table 5. ANOVA Results for Failure Observability Data .......................................... 33
Table 6. ANOVA Results for Failure Observability Data Using Load Generation ......... 39
Table 7. Rater classifications for intermittent failures of industrial and open source system 60
Table 8. Rater classifications for attributes of intermittent failures of industrial and open source system ................................................................. 66
Table 9. Results summarizing estimation of effort spent debugging failures in industrial system ............................................................................................... 75
Table 10. Results summarizing estimation of effort spent debugging failures in the open source system ................................................................. 76
Table 11. Performance of prediction model for industrial system ............................... 87
Table 12. Performance of prediction model for industrial system (with common word occurrences) ....................................................................................... 88
Table 13. Performance of prediction model for open source system ............................ 89
Table 14. Performance of prediction model for open source system (with common word occurrences) ....................................................................................... 90
LIST OF FIGURES

Figure 1. Observed frequency of failure # 124750 .......................................................... 33
Figure 2. Observed frequency of failure # 380417 .......................................................... 34
Figure 3. Observed frequency for failure # 396863 .......................................................... 35
Figure 4. Observed frequency for failure # 410075 .......................................................... 35
Figure 5. Observed frequency for failure # 494116 .......................................................... 36
Figure 6. Observed frequency for failure # 264562 .......................................................... 37
Figure 7. Observed frequency for failure # 332330 .......................................................... 37
Figure 8. Observed frequency of failure # 124750 under simulated processor load .......... 39
Figure 9. Observed frequency of failure # 264562 under simulated processor load .......... 40
Figure 10. Observed frequency of failure # 332330 under simulated processor load .......... 41
Figure 11. Observed frequency of failure # 380417 under simulated processor load .......... 42
Figure 12. Observed frequency of failure # 396863 under simulated processor load .......... 42
Figure 13. Observed frequency of failure # 410075 under simulated processor load .......... 43
Figure 14. Observed frequency of failure # 494116 under simulated processor load .......... 44
Figure 15. Distribution of failure statuses in the ABB system ......................................... 48
Figure 16. Distribution of classified intermittent failure reports for the ABB system ....... 57
Figure 17. Distribution of classified intermittent failure reports for Mozilla Firefox ........ 58
Figure 18. Distribution of attributes of intermittent failure reports for the ABB system ..... 64
Figure 19. Distribution of attributes of intermittent failure reports for Mozilla Firefox ..... 64
Figure 20. Distribution of number of comments for intermittent failure reports for ABB system ........................................................................................................................................ 73
Figure 21. Distribution of number of comments for previously intermittent failure reports for ABB system .................................................................................................................. 74
Figure 22. Distribution of percentages of severity for failure groups in ABB system ....... 79
Figure 23. Distribution of percentages of severity for failure groups in Mozilla Firefox ..... 79
1. INTRODUCTION

Software maintenance accounts for more than 65% of all costs incurred in the lifecycle of software development [1]. A major component of this cost is debugging and fixing faults\(^1\). Faults may escape to the field as a result of the difficulty in software systems adequately before release. A significant amount of software testing research has focused on improving software testing, leading to techniques and tools that aid software engineers in exposing faults. In recent years, advances have been made in test case generation [3], [4], model-based testing [5], regression test selection [6], and graphical user interface (GUI) testing [7]. However, intermittent failures and nondeterministic behavior continue to complicate and compromise the effectiveness of software testing despite advancements in these testing techniques.

According to a study by National Institute of Standards and Technology (NIST), software failures that arise as a result of inadequate testing infrastructure cost the US economy as much as $59.5 billion annually [8]. With software’s increasing complexity, debugging and fixing failures is an increasingly challenging task for developers and testers, especially due to tight deadlines in today’s competitive market. Among the most challenging failures for testers are those that are intermittent i.e., those that cannot be reliably reproduced.

Software testers and developers spend a significant amount of time debugging programs due to intermittent failures, and often make no quality-improving change to the code.

\(^1\) A fault is defined as “An incorrect step, process, or data definition in a computer program” [2]
Software testing and maintenance processes are adversely affected by failures that are hard to reproduce. Such intermittent failures, although investigated in great depth, are often closed as being “Not Repeatable” with no fault fixed. Whether such failures are detected during internal testing or by customers in the field, they represent a need for a versatile testing strategy that minimizes non-deterministic behavior in software that may lead to intermittent failures.

*The goal of this research is to help software engineers understand the characteristics of intermittent behavior in software systems, and to aid in developing efficient techniques for detecting and fixing such failures.*

The Toyota Prius “unintended acceleration” software fault\(^2\) is an example of the implications of potentially dangerous intermittent software failures. Toyota was forced to recall 133,000 vehicles in the US because one intermittent software fault escaped to the field, and was not detected until some customers complained of the car’s strange acceleration problem. This incident demonstrates a need for the development of adequate techniques for detecting intermittent failures, given how software has now permeated such safety-critical systems as cars and airplanes [9].

Another example is that of the Mars Pathfinder mission. In an article on LATimes.com\(^3\), David Cummings describes how a simple test on the Pathfinder software failed: the test case asserted the result of a computation, which was always expected to be even. The test case passed for thousands of test runs, except once. Further investigation revealed a fault in the


operating system's interrupt-handling routine that had triggered at just the right time (down to the exact microsecond) and had caused the operating system to “misremember” the carry on the previously-executed arithmetic operation. Such a subtle yet significant fault could have resulted in a catastrophic failure to the Pathfinder mission en route to Mars that was fortunately avoided in this case.

In software failure repositories, particularly the ones we studied, we observe that intermittent failure reports are often closed by testers due to lack of reproducibility of the failure. As a result, a significant investment of effort spent by testers goes to waste because often no quality-improving change is made to the code through fault fixing. Such wasted effort can compromise the effectiveness of the entire product testing effort, and needs to be minimized. Another important factor that affects this cost is whether such failures are discovered during internal testing or by customers in the field. Post-release failures cost more, since testers often have to travel to the customer site to investigate conditions that lead to the failure. With the increasing complexity of software, this trend in their maintenance costs is expected to continue to rise exponentially.

In this research, we present case studies conducted on intermittent failures reported for an industrial automation system developed by ABB Inc. and an open source system, Mozilla Firefox. For the ABB system, 10.2% of all reported failures were classified as “Not Repeatable”. In other industrial systems, this number is as high as 40% [10]. For Firefox, 11.5% of our sample of failure reports comprised of intermittent failures. Assuming these numbers to be a general representation of the industry to some extent, we can see that intermittent failures are highly prevalent.
To summarize, this research makes the following contributions:

- Shows the significance of hardware configuration factors (such as processor speed, processor load, and memory) in detecting intermittent failures
- Proposes a classification scheme for intermittent failures to aid in improvement of software processes in an organization
- Provides a list of common attributes of intermittent failures to aid in identification of common patterns, symptoms, and pre-conditions of intermittent failures
- Provides empirical evidence that developers spend more effort in debugging intermittent failures than reproducible failures; and finally,
- Presents a prediction model for intermittent failures based on metrics extracted from failure reports

The rest of this document is organized as follows. Section 2 presents a background of intermittent failures. Section 3 describes related work in the area. Section 4 describes our hardware configuration case study. Section 5 explains the classification scheme and identified attributes of intermittent failures. Section 6 elaborates on our analysis pertaining to estimating effort spent debugging intermittent failures. Section 7 describes our prediction model. Finally, Section 8 concludes and makes recommendations. The limitations and threats to validity are covered separately under each section, where relevant.
2. BACKGROUND

The term “intermittent” is defined as:

“occurring occasionally or at regular or irregular intervals”

In terms of software engineering, and particularly software testing, this term refers to failures that occur randomly and cannot be reliably reproduced. Often there are no obvious causes for intermittent behavior in software systems, and this behavior is generally marked as “non-reproducible” or “not repeatable” by software engineers. Practitioners have used the terms intermittent, non-reproducible, and not-repeatable to refer to failures that cannot be reliably reproduced. In this document, we use the term intermittent to refer to such behavior.

2.1 Prevalence of Intermittent Behavior

Intermittent behavior in software systems is highly prevalent [10]. The famous Therac-25 accidents are well-known⁵. In this section, we provide some more recent examples of intermittent failures disrupting everyday software systems.

2.1.1 iPhone Alarm Faults

In November 2010, users of Apple’s phone, iPhone 4, complained that their alarms went off an hour late immediately the day after Daylight Saving Time had come into effect. The fault was in the phone software’s alarm application and was triggered only when users set repeating alarms on their phones to go off on certain days of the week. More recently,

⁴ http://dictionary.reference.com/browse/intermittent
⁵ http://www.cc.gatech.edu/~spencer/courses/ethics/misc/therac.pdf
another fault in the phone’s software caused non-repeating alarms to not go off on New Year’s Day\textsuperscript{6}. Finally most recently, users of iPhone’s operating system iOS 4.1 (including the author) complained that their alarms went off an hour early the day after Daylight Saving Time had come into effect\textsuperscript{7} in Spring 2011.

2.1.2 Therac-25 Accidents

Therac-25 was a radiation therapy machine involved in at least six accidents from 1985 to 1987\textsuperscript{8}. Therac-25 inflicted patients with massive radiation overdose due to faulty software. Investigation of the accidents revealed bad coding practices, inadequate testing, lack of documentation, and no hardware locks allowed for race conditions in the machine’s software modules resulting in delivering 100 times the intended dose. The failure was dependent on timing of operator actions, and was never detected during testing since operators at that time were not very familiar with the machine.

2.2 Types of Intermittent Behavior

There are two major types of intermittent behavior we witness in software systems.

a) Failures that occur only on certain configurations (hardware or software)

b) Failures that occur in unanticipated conditions

\textsuperscript{6} http://www.nytimes.com/2011/01/03/technology/03iphone.html
\textsuperscript{7} http://www.slashgear.com/did-your-iphone-4-alarm-work-correctly-this-morning-14139740/
\textsuperscript{8} http://sunnyday.mit.edu/papers/therac.pdf
2.2.1 Configuration-dependent failures

Anomalous behavior in software systems is sometimes exposed on only specific hardware or software configurations that were not part of the testing phases of the software. Users of a software system may have hardware configurations ranging from very old systems to state of the art systems with multi-core processing units, large amounts of memory and hard drive. The total number of combinations for just these factors is huge. Moreover, the software system itself has an extremely large configuration space. For example, for Microsoft Internet Explorer\(^9\) Version 6 that shipped with Windows XP, the security tab under options alone had 31 different configurable options. Of these, 10 were binary, 19 had three levels, and two had four levels of settings. As a result, there were \(2^{10} \times 3^{19} \times 4^2 = 19,042,491,875,328\) total configurations for this tab alone. This demonstrates the vastness of the configuration space of a typical software system in use today.

Testing a software system across all possible configurations of hardware and software is obviously impossible. The term *configuration testing* refers to testing software system across different configurations, which usually includes the major operating systems in use along with both slow and advanced hardware. For example, for Firefox version 1.5.0.2 the Mozilla team\(^10\) designed configuration tests on Windows XP, Mac OS 10.3 and 10.4, and Linux Fedora Core 4.

To reduce configuration space, concepts such as *pairwise testing* can be adopted. Pairwise testing is a combinatorial testing technique that operates on the principle that most

---

\(^10\) [https://wiki.mozilla.org/Firefox:1.5.0.2:Test_Plan#Configuration_Testing](https://wiki.mozilla.org/Firefox:1.5.0.2:Test_Plan#Configuration_Testing)
faults are caused by interactions between two factors of a system [11]. Testing pairwise combinations of factors, therefore, generates a much smaller testable configuration space compared to an exhaustive technique that covers all possible combinations of factors. Testing for more than two-way combinations of factors becomes progressively expensive given the complex nature of software systems in use today [12].

2.2.2 Unanticipated conditions

Sometimes intermittent failures are observed in software systems when they run into unanticipated input or end up in an abnormal state resulting from an occurrence of an unlikely event or input. We, therefore, observe that software systems often malfunction when met with unforeseen and unanticipated conditions. For example, a real-time system designed to receive input A followed by input B after a certain time interval may malfunction when it receives input B twice or not at all in the specified time interval. There are instances where such faults may not be immediately observable or they manifest as different failures. Debugging such failures can be extremely challenging as testers often have to conduct root-cause analysis to find out where the fault originally occurred in the code. Additionally, there are instances where software simply does not scale well and thus causes intermittent failures.\(^{11}\)

\(^{11}\) http://kb.mozillazine.org/Firefox_hangs#Hyper-Threading
2.3 How do modern-day software organizations deal with intermittence?

A number of techniques are employed by software engineers nowadays to detect intermittent failures in their software systems.

2.3.1 Deterministic Replay Debugging

Deterministic replay debugging is the ability to record a program’s execution up to the minute it crashes, so that it can be deterministically replayed at a later point in time and a root cause of the crash can be identified. Tools such as the BugNet [13] help record every instruction executed by the program so that it can be exactly replayed up to the last second before a crash occurred. This solution has proven to be more useful than collecting and analyzing core dumps\textsuperscript{12,13}.

2.3.2 Fuzz-testing

Fuzz-testing is useful form of black-box testing technique for finding and fixing intermittent faults in software systems. Fuzz-testing or simply “fuzzing” involves providing a system with random, invalid, and unexpected data. This technique helps test whether a software system degrades gracefully or not. Tools such as jsfunfuzz\textsuperscript{14}, a JavaScript fuzzer for

\begin{itemize}
  \item \textsuperscript{13} http://support.mozilla.com/en-US/kb/Mozilla%20Crash%20Reporter
  \item \textsuperscript{14} http://blog.mozilla.com/security/2007/08/02/javascript-fuzzer-available/
\end{itemize}
Firefox, have helped uncover faults that could not uncovered using traditional testing techniques\textsuperscript{15}.

Fuzzing programs can either be mutation-based or generation-based. Mutation-based fuzzers mutate existing test input data to create new test data whereas generation-based fuzzers create new test input data based on models of the input [14]. Fuzz-testing is most useful in detecting failures related to data corruption, memory leaks, assertions, and crashes. Fuzzers must also record all data they create to produce test cases once faults are found in the code.

2.3.3 High Volume Test Automation

High Volume Test Automation (HVTA) is a broad class of testing techniques [15]. HVTA involves automated execution of a large number of test cases for finding intermittent and non-reproducible failures in software systems. HVTA techniques have shown to be particularly useful in finding failures such as buffer overruns, stack overflows, resource exhaustion, and timing-related errors. One example of HVTA tests is extended random regression tests (ERRT) [15]. ERRT involves running regression tests in random order until the software under test fails. These techniques augment conventional testing techniques by finding faults that are otherwise hard to find and reproduce.

2.3.4 Load testing

Load tests test a software system when it is operating at its defined limits. Load tests include volume and stress tests [16].

\textsuperscript{15} http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/
Volume tests test the ability of a program to operate normally when given large tasks to perform. For example, word processing software might be provided a constant stream of documents that are large in size.

Stress tests attempt to determine the maximum limit at which the program breaks. Ideally, this limit is well beyond the specified limit for the program under test. For example, a web server can be stress-tested by simulating a very high visitor load using bots and scripts.

2.3.5 Disturbance Tests

Some organizations employ disturbance tests to test the overall quality and fault-tolerance of a software system. Disturbance tests are defined as tests that disrupt normal operation of the application such as switching off network connection, unplugging the power cord, simulating some other physical failure etc. The anomalous behavior of the software seen as a result of some of the disturbance tests is generally very hard to reproduce, since it requires an intricate set of events to occur at specific time intervals (as discussed in Section 2.2.2).

2.4 Repeatability of Classification Scheme

Failure classification schemes are used extensively in software organizations, and serve as standards based on which product and process improvements take place. Well-known schemes include IBM’s Orthogonal Defect Classification scheme [17] and IEEE’s Classification of Software Anomalies [18]. We developed a classification scheme specifically targeted at intermittent failures, and also identified their common attributes
Such classifications need to be repeatable [19]. Several agreement coefficients have been proposed in literature for measuring inter-rater reliability such as Cohen’s kappa (κ) [20], Scott’s pi (π) [21], and intra-class coefficient [22]. Of these Cohen’s kappa is the most extensively used statistical measure [19]. Cohen’s kappa statistic measures agreement between two raters only. Several modifications such as weighted-kappa [23], kappa maximum [24], and Fleiss’s kappa (for more than two raters) [25] have been proposed, but Cohen’s kappa remains the most widespread in use with more than 11,400 citations to date\textsuperscript{16}.

Cohen’s kappa measures agreement between two raters who each classify N items into C mutually exclusive categories. Cohen’s kappa is defined as follows:

$$\kappa = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)},$$

where $\Pr(a)$ is the probability of agreement between raters, and $\Pr(e)$ is the probability of agreement among raters by chance. If there is complete agreement between raters, then $\kappa=1$, and if there’s complete disagreement between the raters, then $\kappa=0$.

As an example, consider two raters classified 50 emails as spam or not spam. Suppose the raters classified the emails as shown in Table 1 below. Rater A’s classifications are represented as rows, and rater B’s classifications are represented as columns.

<table>
<thead>
<tr>
<th>Table 1. Example Rater Classifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rater Classifications A/B</strong></td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>Spam</td>
</tr>
<tr>
<td>Not Spam</td>
</tr>
</tbody>
</table>

\textsuperscript{16} http://scholar.google.com/scholar?cites=1626806113593293536&as_sdt=5,34&sciodt=0,34&hl=en
For this example, probability of agreement, \( Pr(a) \), is \((20+15)/50 = 0.7\). To calculate probability of agreement by chance, \( Pr(e) \), consider that:

Rater A classified 25 emails as spam, and 25 emails as not spam. Therefore, probability of rater A classifying an email as spam or not spam is 0.5.

Rater B classified 30 emails as spam, and 20 emails as not spam. Therefore, the probability of rater B classifying an email as spam is \(30/50=0.6\), and the probability of rater B classifying an email as not spam is \(20/50=0.4\).

Therefore, the probability that both raters A and B classified an email as spam is \(0.5*0.6=0.3\). Similarly, the probability that both raters classified an email as not spam is \(0.5*0.4=0.2\). Thus, the overall probability of agreement by chance will be \(0.3+0.2=0.5\). If we apply our formula for Cohen’s kappa, we would get \( \kappa = (0.7-0.5) / (1-0.5) = 0.40 \).

Since kappa has been widely used in literature, guidelines for its interpretations have also been proposed [26]. Generally, kappa values less than 0.45 show inadequate agreement among raters. Kappa values above 0.62 demonstrate sufficiently good agreement, and kappa values above 0.78 indicate excellent agreement among raters.
3. RELATED WORK

Several studies have appeared in literature concerning nondeterminism [27], [28] as well as observation-based testing [29-31], but there are few existing studies that have established a correlation between the observability of faults in software and its base hardware. White et al. [32] presented one empirical study carried out on RealPlayer. Their work involved testing RealPlayer on hardware running on different processor speeds, memory, and operating systems. They observed that RealPlayer behaved differently on different configurations, particularly in the way that it failed. They also detected an increased trend of faults with decreasing processor speed and memory. Our study on hardware configurations (discussed in Section 4) builds on their work using a more rigorous experimental design. In addition, they had no access to the underlying code, which limited their ability to study and understand the intermittent nature of the failures.

Duarte et al. [33] present a framework, GridUnit, for testing distributed applications on a grid of multiple heterogeneous environments simultaneously. Their results show that carrying out such tests increases the proof of correctness of software under test by achieving greater environmental coverage for their test suites in a singular or small number of environments. While not completely similar to our research, their work on testing on different environment confirms an important assertion that testing in one environment is not sufficient for verifying that the software executes correctly.

Cohen et al. [6], [34] have discussed the use of combinatorial interaction testing techniques to test different software configurations, for normal as well as regression testing.
They have shown that large software configuration spaces need to be sampled using appropriate combinatorial testing techniques (such as n-way testing). Findings from their case study suggest that combinatorial testing techniques help find configuration-dependent faults, and also increase the overall fault-detection capability of individual test cases.

Combinatorial testing techniques have been extensively advocated in literature for increasing the efficiency of the testing phase by achieving high code coverage while utilizing significantly fewer test cases [35-38]. Kuhn et al. in [39], [40] found that combinatorial testing is effective for testing software such as Mozilla Firefox. In addition, they found that 75% of failures in Firefox were dependent on the interaction of two or more software configuration parameters. Our work uses combinatorial testing techniques to assess the impact of hardware configuration and environment factors on the observability of intermittent faults.

Porter et al. [41] present a framework for distributed testing of applications called Skoll. The Skoll system focuses on systems with large configuration spaces and distributes testing tasks to user communities worldwide in an effort to use greater computing resources and reduce testing time. They found that their distributed effort found faults sooner than traditional non-distributed techniques. Yilmaz et al. [42] found that using covering arrays to select a subset of configuration space for fault characterization allowed for much greater scalability of the Skoll framework.

Problems involving nondeterministic behavior in software have often appeared in literature in conjunction with concurrency problems and parallel execution of programs [43], [44]. Musuvathi et al. [44] at Microsoft Research have developed a tool called CHESS for
eliminating nondeterminism in concurrent programs. CHESS explores thread schedules of programs under test in a deterministic manner and uses model-checking techniques to expose any discrepancies in terms of interleaving of events or race conditions. While not directly related to our work, it represents one of the recent research studies undertaken in the area of concurrency testing and intermittent failures.

Artzi et al. [45] developed a tool called Recrash that stores copies of methods-under-execution in memory. When the target program crashes, Recrash generates unit test cases based on the method arguments stored in memory. Using a Java implementation of their approach, called RecrashJ, they have evaluated the tool with real crashes from applications such as Eclipse, SVNKit, among others. Recrash aims to reduce the amount of time spent by developers in reproducing failures, and according to the authors can also be adopted to work with non-crashing faults and errors.

Another tool, Krash, was developed by Perarnau et al. [46]. Krash supports the findings of our hardware configuration study (discussed in Section 4) that CPU load generation leads to higher occurrence of software failures in many cases. The tool itself can record and reproduce “dynamic load profiles” for multi-core machines. Their methodology cooperates with the system scheduler to ensure that a load profile is accurately replayed once it has been recorded. Krash is particularly helpful for parallel application developers who want to test for concurrency issues with their application.

Baars et al. [47] conducted a case study on hard-to-reproduce faults using their techniques based on evolutionary testing. They reported a successful application of their technique on software that suffered from memory and data corruption failures. These failures
were not reproduced after significant effort from the testers, but using evolutionary testing techniques enabled them to discover new test scenarios that helped reproduce those failures. The authors, however, admit that there are a number of hurdles and future work is needed before their approach can be fully adopted by the software industry.

Memory leaks are one of the sources of intermittent behavior in software systems. Bond et al. [48] describe their approach towards tolerating memory leaks in software systems. Their leak tolerance approach, called Melt, seeks to reduce performance degradations of the target application in the event of memory leak. Melt transfers portions of memory it deems as “likely leaked” to physical disk (analogous to operating system paging operation), thus delaying memory exhaustion, and in turn, a program crash. This approach is meant to give developers more time in identifying the memory leak and fixing it.

A problem that falls under the class of intermittent failures is software hangs i.e., when a software system goes unresponsive. Wang et al. [49] developed a tool, HangWiz, as part of their research into such software hangs. HangWiz requires hang patterns from runtime traces of the target application, which is then used to find places in the source that make blocking calls i.e., potential hang points. They evaluated their tool on several real-world large software systems, and found several “hang bugs” in those systems.

For estimating effort spent debugging intermittent failures, we used metrics such as number of comments and failure resolution time (discussed in Section 6). A study that employed the same metric for estimating effort (i.e., number of comments) was conducted by Ho et al. [50]. They analyzed the relationship between “Not a Problem” failure reports and performance requirements specified at the beginning of the development process. They found
that similar to intermittent failure reports, “Not a Problem” failure reports also cost the company significant resources, as each report had to be investigated thoroughly, especially if the report was filed by a customer.

An interesting study revolving around automated extraction of information from failure repositories was published by Aranda et al. [51]. The authors asserted that histories of even simple failures are strongly dependent on social, organizational, and technical knowledge that cannot be extracted via any automated process. Their study analyzed failure reports to identify coordination patterns among team members that were validated through a survey of software professionals.

Investigating intermittent field failures in software systems is heavily reliant on the amount of information available pertaining to the failure. Therefore, the important and quality of failure reports cannot be overemphasized. Zimmerman et al. [52] conducted a survey amongst users and developers of Apache, Mozilla, and Eclipse to gain a better understanding of what the developers look for in failure reports, and what the users can provide with ease. They present their findings as recommendations on how best to design failure tracking system so that they are easy to use and yet provide the crucial information required by developers and testers in debugging a failure.

Brown et al. [53] performed a study on the implications of “human” or “user” mistakes on software dependability. They show that user errors account for more than half of all failures, and need to be accounted for in system dependability designs. In our study, we found that there failure reports were filed as a result of such errors, and the system administrators have not anticipated dealing with such unlikely scenarios.
Nagappan et al. [54] analyzed five projects at Microsoft and evaluated several code complexity metrics as predictors of post-release failures. They performed cross-project analysis and found that there were no metrics that predicted failures universally across all projects. Therefore, they concluded that predictive factors are accurate when they are obtained and applied to the same project. Similarly, we extracted common word occurrences from intermittent failure reports and found that very few words overlapped across our two datasets. Therefore, our prediction models for both systems were significantly different.

Panjer [55] demonstrated that by mining change history of failures from a large open source software repository, one can predict failure lifetimes. The proposed models accurately predicted lifetimes for 34.9% failures. An extension of this study was performed on FreeBSD by Bougie et al. [56]. Their models were able to predict 19.49% of failure lifetimes accurately. Our prediction models based solely on failure report attributes had similar levels of accuracy, which we were later able to improve by incorporating common word occurrences in our models.

Many researchers have used software metrics for fault prediction [54], [13], [57-60]. In our work we utilize a similar methodology for predicting when a failure may exhibit intermittent behavior.
4. EFFECTS OF HARDWARE CONFIGURATION ON OCCURRENCE OF INTERMITTENT FAILURES

Our preliminary study on intermittent behavior in software systems explored the effect of hardware configurations on the *observability* of such failures. We define the term “observability” as the number of times the intermittent fault propagated to the fore, and was thus “observed” as an error or a failure by the user. To increase the observability of software faults, we explore the effect hardware configurations and processor load have on intermittent failures and the nondeterministic behavior of software systems. We conducted two case studies of a set of reported intermittent Mozilla Firefox failures. Ten times per failure, we replicated the conditions that caused the reported failures on nine hardware configurations running Windows XP Service Pack 3 and measured the frequency with which the failure was observed. We also replicated the failure conditions ten times with four different processor loads (0%, 25%, 50%, and 75%).

4.1 Empirical Study

The goal of this study was to *improve the observability of software faults by exploring the impact hardware configuration has on intermittent failures and the nondeterministic behavior of software systems*. Therefore, the research questions of interest to us are:

**RQ1:** Is the observability of a software fault impacted by processor speed? If so, how?

**RQ2:** Is the observability of a software fault impacted by the amount of memory? If so, how?
RQ3: Is the observability of a software fault impacted by the capacity of the hard drive? If so, how?

RQ4: Is the observability of a software fault impacted by processor load? If so, how?

We address these research questions through a study of selected set of failures from a large open-source application, Mozilla Firefox\textsuperscript{17}. Firefox was chosen because it is an active open-source project with a large repository of failures online, many of which have traceability to the code change that were made to fix the underlying faults. The rest of this section describes our procedures for selection of failures, study setup, observed measures, and threats to validity.

4.1.1 Step One: Identify Intermittent Failures

Our first step was to collect a set of reported failures with indications that these failures were intermittent and/or non-reproducible. We ran natural language queries on Mozilla's online repository of failures, Bugzilla\textsuperscript{18}. We did not restrict our search to any particular version of Firefox. Additionally, no failures related to plug-ins or add-ons were considered, as these represent another factor of configuration outside the scope of this study. The search was, therefore, restricted to Firefox's core codebase. We analyzed hundreds of failure reports returned from our searches, and found 75 failures that matched our criteria. The natural language queries used variations on the keywords such as "timing", "page file", "operating system", "system dependent", "race condition", "deadlock", "concurrency", "pentium", "older machines", and "slow computers".

\textsuperscript{17} http://www.mozilla.com/en-US/firefox/
\textsuperscript{18} http://bugzilla.mozilla.org
Once we felt we had gathered a substantial number of intermittent failures, we attempted to reproduce each failure ten times on a machine running Windows XP. The machine used for testing failures had the same specification as reported on Bugzilla. In addition to being able to reproduce a particular failure, we also considered its present status on Bugzilla, which told us whether the failure had been fixed or not. From our initial set of 75 failures, we selected those that (1) exhibited nondeterministic behavior on our test machine by being intermittently reproducible; and (2) had already been fixed by the Mozilla developers. Using failures that have been fixed assures us that there are identified code changes that can be studied to better understand why the failures were only observable on certain hardware configurations. Using these criteria we narrowed down our sample size to 11 failures. Each failure is described in detail in Section 4.2 along with an explanation of its code fix.

4.1.2 Step Two: Determine Hardware Configurations

We define a set of base hardware configurations on which to replicate our selected failures. These configurations vary in terms of their processor speed, memory, and hard drive capacity. Each of the three factors has multiple levels defined (see Table 2). The levels were chosen to best model the range of hardware configurations in use today. Together processor speed, memory, and hard drive capacity constitute the independent variables for our study.
Table 2. Levels of Independent Variables Used In Our Study

<table>
<thead>
<tr>
<th>Processor</th>
<th>667Mhz</th>
<th>1Ghz</th>
<th>2Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>128MB</td>
<td>256MB</td>
<td>1GB</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>2.5GB</td>
<td>10GB</td>
<td></td>
</tr>
</tbody>
</table>

The application under test was the open-source web browser Mozilla Firefox. The operating system on all configurations was chosen to be Microsoft Windows XP Professional Service Pack 3. While the operating system itself is a potential factor in observability, this study is focusing only on the effect the underlying hardware has on observability. According to Net Applications\(^{19}\), Windows XP is the most popular operating system in use today with an estimated market share of 71.8%, and the Windows family of operating systems together has 93% of market share\(^{20}\). This large market share was the primary motivation in choosing Windows XP for this study.

Hardware configurations were constructed as virtual machine images using the freely-available VMWare ESXi Server v4.0\(^{21}\) software. The three factors of our study were manipulated by using the VMWare vSphere v4.0\(^{22}\) client application. Although it can be debated how accurately a virtual machine models a real one, we believe such differences are not significant and outside the scope of this study. The ESXi server provides adequate controls for controlling processor speed and memory of a Windows system.

\(^{20}\) Both statistics are true for the time the study was conducted i.e., 2009
\(^{22}\) [http://www.vmware.com/download/vsphere/](http://www.vmware.com/download/vsphere/)
4.1.3 Step Three: Hardware Configuration Testing

A full-factorial design of the factors listed in Table 2 results in 18 total combinations. However, using only pairwise interactions [11], the total number of combinations was reduced by half (see Table 3). Pairwise testing enables us to reduce the actual number of test cases by testing only the two-way interaction of variables rather than exhaustively testing all of their possible combinations. Empirical results show that faults are rarely dependent on larger combinations of values, but are more dependent on pairwise interactions of different variables [11]. All of the variables are evenly distributed in the nine combinations shown in Table 3.

We tested each selected failure ten times on each hardware configuration by replicating the steps to reproduce provided in the report associated with the failure. Thus, the total number of tests conducted was 90, for the experiment in question. Tests were run ten times to capture the frequency a failure was observable on each configuration. The order of testing was randomized for each configuration to reduce bias.
Table 3. Hardware Configurations Used In Our Study

<table>
<thead>
<tr>
<th>No.</th>
<th>Processor Speed</th>
<th>Memory</th>
<th>Hard Drive Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>667Mhz</td>
<td>128MB</td>
<td>2.5GB</td>
</tr>
<tr>
<td>2</td>
<td>667Mhz</td>
<td>256MB</td>
<td>10GB</td>
</tr>
<tr>
<td>3</td>
<td>667Mhz</td>
<td>1GB</td>
<td>2.5GB</td>
</tr>
<tr>
<td>4</td>
<td>1Ghz</td>
<td>128MB</td>
<td>10GB</td>
</tr>
<tr>
<td>5</td>
<td>1Ghz</td>
<td>256MB</td>
<td>2.5GB</td>
</tr>
<tr>
<td>6</td>
<td>1Ghz</td>
<td>1GB</td>
<td>10GB</td>
</tr>
<tr>
<td>7</td>
<td>2Ghz</td>
<td>128MB</td>
<td>2.5GB</td>
</tr>
<tr>
<td>8</td>
<td>2Ghz</td>
<td>256MB</td>
<td>10GB</td>
</tr>
<tr>
<td>9</td>
<td>2Ghz</td>
<td>1GB</td>
<td>2.5GB/10GB</td>
</tr>
</tbody>
</table>

4.1.4 Step Four: Processor Load Testing

This study involved varying the processor load for all of the selected failures. In addition to the previous study that tested all failures on 0% processor load, we defined three levels of processor load to test - 25%, 50%, and 75%. The steps needed to invoke the failure were run ten times for each of the selected failures on each of three additional processor load conditions. Since the load generation study only involved manipulating processor load, the failures were tested across a smaller number of configurations that varied in processor speed (see Table 4). Including test runs conducted in the previous study, each of the selected
failures was run ten times on three configurations with four processor loads making the total number of test runs 120 for each failure.

Table 4. Hardware Configurations Used In Processor Load Study

<table>
<thead>
<tr>
<th>No.</th>
<th>Processor Speed</th>
<th>Memory</th>
<th>Hard Drive Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>667Mhz</td>
<td>128MB</td>
<td>2.5GB</td>
</tr>
<tr>
<td>2</td>
<td>1Ghz</td>
<td>128MB</td>
<td>10GB</td>
</tr>
<tr>
<td>3</td>
<td>2Ghz</td>
<td>128MB</td>
<td>2.5GB</td>
</tr>
</tbody>
</table>

4.1.5 Establishing a Baseline

To establish a base of measurement for the observability of failures in Firefox's codebase, a further 12 random failures were selected and tested on the slowest virtual machine configuration i.e., a configuration with processor speed of 667 MHz, 128 MB memory, and 2.5 GB hard drive capacity. The failures chosen for the baseline test all had failure descriptions that indicated the failures were not intermittent.

4.1.6 Observability Data Collection and Tools

The data of interest to us in this study included performance indicators of the system and the software under test (such as processor load, memory footprint of software under test, page file size etc.). Using different observability tools listed below, we were able to measure Firefox's memory consumption (working set, page faults per second, and page file size), processor usage, thread count, handle count, and priority base. For the system's performance measures, we measured total processor usage, average disk queue length, and memory pages.
per second. The sequence of events fired by the application under test was monitored using an event-capturing tool. Finally, we manipulated processor load using a free utility (CPU Grabber) that simulates load for Win9x/2000 systems. All the preceding performance measures constitute the dependent variables for our study.

The tools used for this purpose included:

- **Spy++**\(^23\) - An event-capturing utility that ships with Microsoft Visual Studio. Helpful in monitoring behind-the-scenes activity of a windows application.

- **VMMap**\(^24\) - Provides a map of virtual memory for analyzing an application’s memory usage.

- **Perfmon**\(^25\) - A windows system diagnostic utility used for monitoring total processor usage as well as process-specific measurements.

- **CPU Grabber**\(^26\) – A utility that lets us simulate processor load. Part of Microsoft DirectShow SDK Framework.

### 4.1.7 Threats to Validity

Internal validity involves determining effect of other factors that can influence study results without the researcher’s knowledge. As this study specifically focuses on nondeterministic behavior of software, any factors that affect the software under test are a risk. This risk was mitigated by using a clean virtual image with only the operating system and Firefox installed. Since testing Firefox involves sending messages on the network,

---


network traffic and other noncontrollable factors could make up some of the unexplained variance in the ANOVA models. Also, the tests were run manually. For a few of the failures, this manual timing lead to some unexplained variance in the ANOVA models.

Threats to external validity are conditions that limit the generalization of the results. The primary threats in this study are the use of a single subject program and the use of a small set of failures. Future studies are needed to better understand how prevalent these issues are in Firefox and other types of software.

Threats to construct validity arise when measurement instruments do not properly capture what they are intended to capture. In this study, people were used to physically observe the failure, either through directly viewing the deviant behavior or indirectly observing the failure through the data collection tools. Subtle changes in the failure could be missed by the observers.

### 4.2 Selected Failures

Based on the criterion mentioned in the preceding section, we arrived at our sample size of 11 failures for in-depth study. This section describes each of those failures in greater detail along with their code fixes.

**Failure # 124750:** On Firefox version 0.8, and on computers with low processor speeds, this failure resulted in a stealing of focus from the active tab by another tab loading in the background. This failure was timing-related and was evident on web pages that shifted focus to an input field upon loading. For example, loading google.com (which shifts focus to its search box as soon as it loads) and quickly switching to another tab (say yahoo.com) to
type a query results in the typed text appearing on the search box in the google tab. This behavior occurred since there was only one focus controller object defined in source code for the entire window. This was part of Firefox’s underlying Gecko engine and needed significant modifications. Instead a workaround was put in place to avoid this behavior that simply blocked all focus-stealing calls by checking if the tab requesting focus was also the active tab at that time.

**Failure # 200119:** This failure occurred when a Certificate Revoke List (CRL) is configured to be automatically updated in Firefox version 1.0. This configuration lead to random crashes of the application when a user attempted to exit. Although this failure was intermittent in our tests initially, it was later found to be always observable upon visiting a particular website. The fix for this failure involved refactoring the code and removing some redundant method calls.

**Failure # 264562:** This failure involved the "Find As You Type"\(^{27}\) or incremental find functionality in Firefox. Upon pressing Ctrl-T to open a new tab and pressing some keystrokes in quick succession causes the find bar to appear at the bottom of the window, even though there is a blank page loaded in the browser. This failure was only reported on machines that had low processor speeds. The fix for this failure was to simply block all user input when no web page had been loaded in the browser window.

**Failure # 314369:** This failure has to do with the difference in minimum marquee speeds between Microsoft Internet Explorer and Firefox. In Internet Explorer, the minimum speed for a marquee is 60ms while in Firefox it is 40ms. This failure could be regarded as a

\(^{27}\) [http://www.mozilla.org/access/type-ahead/](http://www.mozilla.org/access/type-ahead/)
case of always failing, but failing differently (since marquee speeds were different each time the test was run). Some marquees in test cases exhibited inconsistent speeds and their behavior had to be mended in the source code fix to conform to standards similar to that of Internet Explorer.

**Failure #332330:** This failure occurred when Ctrl-N and Alt-D were pressed in quick succession of each other. Normally this action would result in shifting of focus to the Firefox address bar but, instead, it triggered the Open Location dialog box. Similar to the failure #124750, this failure was also observed on systems with low processor speeds. Furthermore, this failure occurred only when a user's homepage was set to about:blank. Analyzing the underlying source code, we found that a variable $gURLBar$ (representing the address bar) was uninitialized for cases when homepage was set to about:blank to improve application performance during startup. The fix was to assign the $gURLBar$ its usual value regardless of a user’s homepage setting.

**Failure #363258:** This failure was related to the inaccurate millisecond resolution used in earlier versions of Firefox on Windows XP. The millisecond resolution by default was 15 or 16ms which made profiling applications and add-ons for Firefox inaccurate. The fix involved doing a new implementation of the timer with a resolution of 1ms. This was another failure that was deemed deterministic since each test run produced different results i.e., the failure was always observed, just differently each time.

**Failure #380417:** This failure was related to the Standard Vector Graphics (SVG) functionality in Firefox. The failure was specifically dependent on timing execution of instructions. The test case for observing this failure had some JavaScript alerts that
corresponded to the execution of instructions. The test case had three alerts in all, and one of them corresponded to SVG load event, the appearance of which indicated a normal flow of execution (and hence, the non-observavility of the fault). The fix was to use an event dispatcher to make sure the SVG load event was not ignored and was always captured by a listener.

**Failure # 396863:** This failure resulted in two sub-menus of the same menu-item being open at the same time. Normally, when focus moves away from one sub-menu, it closes after a set time or earlier if another sub-menu is opened. On a particular build of Firefox containing this failure, the mouse has to be moved quickly from an item in the first sub-menu to the second sub-menu so that both are open at the same time. The fix corrected the behavior of the timer responsible for opening and closing sub-menus.

**Failure # 410075:** This failure is timing-related and is observed when Ctrl-T and Ctrl-K in quick succession on Firefox's main window. By default, Ctrl-T opens a new tab and Ctrl-K moves focus to the search box at the top-right of the window. When pressed in quick succession, these sequence of keystrokes result in focus moving back to the address bar instead of staying in the search box. The fix to this was to remove a timeout in legacy code that was causing the focus to shift back to the address bar after a specified time interval, and hence, was overriding Ctrl-K.

**Failure # 463635:** This failure was related with the All-Tabs functionality that was introduced in a prototype of an upcoming version of Firefox. The All-Tabs functionality is a new way of switching between tabs inside Firefox, and is modeled after how windows are switched in Windows Vista. In addition, the All-Tabs functionality also includes an
incremental search feature. This failure was observed when multiple tabs were open in Firefox and a tab search was run form the All-Tabs window quickly followed by pressing the Enter key. These actions result in focus switching to the presently selected tab in the All-Tabs window rather than the first tab returned in the search results. The fix to this failure involved waiting for the search to be over before shifting focus to the appropriate tab.

   **Failure # 494116:** This failure resulted in audio and video on a system running Microsoft Windows to go out of synchronization. The failure report described the failure as intermittent and that it was observed on normal as well as high processor load. The fix changed the method a Windows system’s audio timers were requested to ensure that audio and video were always in synchronization.

**4.3 Results and Analysis**

This section presents the analysis of study observations carried out on different hardware configurations and processor loads.

**4.3.1 Hardware Configuration Study**

The hardware configuration study involved replicating the conditions for each failure ten times on each configuration. In summary, of the 11 failures that were under test, the observability of five failures was related to the hardware configuration, and the other six had no such relationship. The ANOVA results for all observed failures are summarized in Table 5.
Figure 1 displays the results for failure # 124750. The figure shows a downward trend in the number of occurrences of this failure as configurations increase in terms of processor speed, memory, and hard drive capacity. The ANOVA results in Table 5 suggest a strong dependence on processor speed across all configurations. More than 70% of the observed variation in the data was due to processor speed alone.
Figure 2 displays the results for failure # 380417. The ANOVA results for this failure suggest a dependence on memory more than processor speed. The shape of the observed behavior suggests a decreased rate of occurrence with increasing memory and processor speed. The rate of occurrence is highest for cases with 128MB memory setting, hence its low p-value in the ANOVA results. Processor speed, being less significant, has a p-value of 0.235 and lastly, hard drive is statistically insignificant with a p-value of 0.727.

![Figure 2. Observed frequency of failure # 380417](image)

Figure 3 displays the observed behavior of failure # 396863. This failure was most susceptible to manual testing since it involved manually moving the mouse to expose the failure (i.e., opening of two sub-menus at the same time). The figure suggests a regular pattern of occurrence for this bug across all configurations. The ANOVA results show a dependence on all three factors with the most significant being memory with a p-value of 0.004. Processor speed and hard drive were also significant with p-values of 0.029 and 0.052 respectively.
Figure 3. Observed frequency for failure # 396863

Figure 4 shows the observed behavior for failure # 410075. The ANOVA results suggest a dependence on both processor speed and hard drive factors. The failure description (see Section 4.2) implies it being dependent only on processor speed, but the ANOVA results show that both hard drive and processor speed have low statistical significance p-values of 0.150 each. Memory is also found to be statistically insignificant with a p-value of 0.453.

Figure 4. Observed frequency for failure # 410075
Figure 5 shows the observed behavior for failure # 494116. There is a strong dependence of this failure on processor speed (with a p-value of 0.014) and it was only observable on the first three configurations i.e., configurations that had a processor speed of 667MHz. These slower configurations were the only ones that had difficulty in playing audio and video smoothly together for the duration of the test. Memory and hard drive were not significant for this failure, with p-values of 0.614 and 0.466 respectively.

![Diagram showing failure frequency across configurations](image)

**Figure 5. Observed frequency for failure # 494116**

There were six failures whose observability was not related to hardware at all. Failures 200119, 314369, and 363258 were always observable whereas failures 264562, 332330, and 463635 were never observed. Additionally, there were two failures (failures 264562 and 332330) that were not observable in this study but later became observable under high processor load. Their complete load test results are described in the processor load study results. Figures 6 and 7 show the occurrence rates for these failures respectively.
The ANOVA results for both failures 264562 and 332330 show a strong dependence on processor speed, which accounts for most variation in the data followed by memory. Although these failures are included in the normal load study results, they were never reproducible under normal (or 0%) processor load. For failure # 264562, we simulated processor load by opening 10 different tabs at once (through a drop-down Live Bookmark in Firefox) before running the test ten times. For failure # 332330, we simulated the load using
CPU Grabber at 25%. Under higher load, both of the failures showed signs of nondeterminism, which prompted us to explore this aspect of the failures in greater detail.

In summary, we found that five of the eleven failures selected displayed nondeterministic behavior and were observed intermittently on our test configurations. Our ANOVA results suggest that processor speed and memory capacity were the major contributing factors to such behavior. Except for failure # 410075, all p-values for our intermittent failures were significant (i.e., they were close to or less than 0.05). Additionally, there were several failures that were observed more frequently on the slowest configuration as compared to other configurations with more processor speed and memory. In terms of our research questions, we found that observability of software faults is impacted significantly by both processor speed and memory. We did not find evidence of hard drive capacity impacting the observability of any failure amongst our selected set of failures.

**4.3.2 Processor Load Generation Study**

This section describes the results for the processor load study conducted on our selected failures.

Figure 8 shows the results for failure # 124750 when tested under different processor loads. The frequency of the failure increases with increasing load. The ANOVA results in Table 6 show that more than 80% variance in the data is explained by processor speed and load. Both are statistically significant with p-values of 0.024 and 0.007 respectively.
Figure 8. Observed frequency of failure # 124750 under simulated processor load

<table>
<thead>
<tr>
<th>Failure ID</th>
<th>Significant Factor</th>
<th>P-value</th>
<th>Std. Dev.</th>
<th>R-Sq. (%)</th>
<th>R-Sq adj. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>124750</td>
<td>Processor Load</td>
<td>0.007</td>
<td>1.29</td>
<td>89.00</td>
<td>79.84</td>
</tr>
<tr>
<td>264562</td>
<td>Processor Speed</td>
<td>0.053</td>
<td>0.64</td>
<td>77.10</td>
<td>58.02</td>
</tr>
<tr>
<td>332330</td>
<td>Processor Speed</td>
<td>0.044</td>
<td>2.95</td>
<td>72.79</td>
<td>50.11</td>
</tr>
<tr>
<td>380417</td>
<td>Processor Load</td>
<td>0.001</td>
<td>1.11</td>
<td>92.57</td>
<td>86.39</td>
</tr>
<tr>
<td>396863</td>
<td>Processor Speed</td>
<td>0.097</td>
<td>1.25</td>
<td>58.55</td>
<td>24.00</td>
</tr>
<tr>
<td>410075</td>
<td>Processor Load</td>
<td>0.001</td>
<td>0.86</td>
<td>92.74</td>
<td>86.69</td>
</tr>
<tr>
<td>494116</td>
<td>Processor Speed</td>
<td>0.016</td>
<td>0.44</td>
<td>85.86</td>
<td>74.07</td>
</tr>
</tbody>
</table>

Figure 9 shows the observed behavior for failure # 264562 under high processor load. This failure was initially not observable when tested with no processor load. Furthermore, the failure was observed at 25% processor load and higher for the slowest configuration and at
75% processor loads for the other two configurations. Processor load proved to be more effective on the slowest configuration, which is reflected in the ANOVA results. The most significant factor for this failure is processor speed (with a p-value of 0.053) followed by processor load (with a p-value of 0.094).

![Graph](image)

**Figure 9. Observed frequency of failure # 264562 under simulated processor load**

Figure 10 shows the observed behavior for failure # 332330 under simulated load. Similar to the previous failure, the failure was observed most on the slowest configuration compared to normal load conditions when it could not be observed at all. This shows a dependence on slower computers and processor load for this failure. The ANOVA results for this bug show a dependence on processor speed more than processor load with p-values of 0.044 and 0.268 respectively.
Figure 10. Observed frequency of failure # 332330 under simulated processor load

Figure 11 shows observed behavior for failure # 380417 under high processor load. We observe this behavior because the increased load resulted in the application behaving normally, and thus, we observed the failure lesser number of times. The ANOVA results in Table 6 confirm a strong dependence on processor load. The slowest configuration stays relatively consistent across all load tests, but the other two configurations witness a sharp decline in the rate of occurrence. Of the two factors, processor load is statistically significant with a p-value of 0.001 whereas processor speed is relatively insignificant with a p-value of 0.154.
Figure 11. Observed frequency of failure # 380417 under simulated processor load

Figure 12 shows observed behavior for failure # 396863 under simulated processor load. As mentioned in Section 4, this failure was most prone to the effects of manual testing. The results are similar to the preceding study with the most significant factor being processor speed. The failure occurred due to a malfunctioning timer call (see Section 4.2 for details), and increased loads did not have an effect on the observability of this failure, expectedly.

Figure 12. Observed frequency of failure # 396863 under simulated processor load
Figure 13 shows observed behavior for failure # 410075 under simulated high processor load. For all configurations, the observability of the failure rises sharply to the point that it becomes always observable for 50% load (in some cases) and at 75% load (for all cases). The ANOVA results also show a strong dependence on processor load for this failure (with a p-value of 0.001) even though processor speed was still statistically significant (with a p-value of 0.178).

![Figure 13. Observed frequency of failure # 410075 under simulated processor load](image)

Figure 14 shows the observed behavior for failure # 494116 under simulated processor load. This failure was only reproducible on the slowest configurations under normal load (machines with 667MHz clock speeds), and a similar behavior was witnessed here. We, however, observe a downward trend of occurrence with increasing processor load. The ANOVA results show that both processor speed and load were statistically significant, with p-values of 0.016 and 0.029 respectively.
In summary, our results indicate that processor load has a significant impact on the observability of software faults. Our ANOVA results suggest both processor speed and load are responsible for exposing nondeterministic behavior of software. Except for failure #396863, all p-values obtained in our ANOVA results were statistically significant (i.e., they were close to 0.05 level). There were two failures that were not observed on any configuration, but later became observable under higher processor load. Software engineers can, therefore, use our approach to better observe intermittent failures and nondeterministic behavior of software systems.

4.4 Baseline Failure Analysis

Ten of the twelve baseline failures showed no nondeterministic behavior and were always observable. There were, however, two failures that had differences in their observability - 329892 and 332493. Both of these failures were related to the then newly-
introduced *Places* toolbar functionality in Firefox and occurred in builds released close to each other. The nondeterministic behavior of these failures arose due to inconsistencies in implementation of data structures underlying Places toolbar (specifically, the functionality governing folders and separators). The failures and their fixes were not related to processor speed, memory, or processor load.

### 4.5 Conclusions

We summarize the findings for our hardware configuration study in terms of the research question we stated earlier.

#### 4.5.1 Is the observability of a software fault impacted by processor speed? If so, how?

Our observations indicate that processor speed is a major contributing factor to nondeterministic behavior in software systems, and can be manipulated to increase the observability of software faults. In our hardware configuration study, five of the seven failures that showed nondeterministic behavior depended on the variation in processor speed. Except for one failure, all results were statistically significant with low p-values for processor speed. In addition, our results show that hardware configurations that had less processor speed observed failures more frequently than other configurations with higher processor speeds.
4.5.2 Is the observability of a software fault impacted by the amount of memory? If so, how?

Our results show that memory capacity also influences the observability of software faults, similar to processor speed. Two of the seven failures that displayed nondeterministic behavior in our hardware configuration study were strongly dependent on memory. In both cases, our ANOVA results were statistically significant.

4.5.3 Is the observability of a software fault impacted by the size of the hard drive? If so, how?

Our observations did not indicate that observability of software faults was impacted by hard drive capacity. There was one failure (failure # 410075) where hard drive capacity had the highest p-value of the three factors (0.150), which is not statistically significant.

4.5.4 Is the observability of a software fault impacted by processor load? If so, how?

We conducted a separate case study for observing the effects of processor load on observability of software faults, and found that manipulating processor load increased the observability of all failures under test. In terms of ANOVA results, processor load was statistically significant for four of the seven defects i.e., it had a p-value of less than 0.05 (including cases where processor speed had a lower p-value than load).
5. CLASSIFICATION OF INTERMITTENT FAILURES

This study was primarily aimed at broadening the knowledge about intermittent failures that we encounter in present-day software systems. There are a number of classification schemes that exist in literature, most notably the IEEE Classification of Software Anomalies [18], the Orthogonal Defect Classification scheme [17] proposed by IBM, and a comprehensive taxonomy of failures by Beizer [61]. However, the classification scheme that we propose in this study is different from the preceding schemes or any other such schemes, since our scheme specifically seeks to classify intermittent software failures. To our knowledge, no such classification scheme exists in the literature. Classification schemes have been known to help improve software processes in an organization, which also serves as the prime motivation for the development of our scheme i.e., improvement of software process with respect to intermittent failures.

5.1 Study

The goal of this research was to reduce the effort spent on debugging intermittent failures by increasing the knowledge about them, and by providing actionable insight into their causes and underlying faults. Thus, we were interested in the following research questions:

**RQ5:** What are the different causes of intermittence in software failures?

**RQ6:** What are the common attributes of intermittent behavior in software?
We address these research questions by carrying out a detailed analysis of failure reports from an industrial software system developed by ABB Inc. and an open source system, Mozilla Firefox.

5.1.1 Systems Under Test

We used an industrial and an open source system for this study. The industrial system under test for our study was a large real-time desktop utility produced by ABB. The failure tracking database for this industrial system referred to intermittent failures as “Not Repeatable”. The proportional distribution of final failure status in the failure tracking database is shown in Figure 15. A total of 10.2% of all failures in the said database were closed as being “Not Repeatable”. The failure reports for such failures contained important clues about the nature of the failures, and the amount of effort spent investigating the cause of the failure by developers and testers.

![Figure 15. Distribution of failure statuses in the ABB system](image)

---

28 Intentionally not identified to protect ABB proprietary information
In this study, we use the term intermittent to refer to “Not Repeatable” failures in the industrial system. We found a number of failure reports that had initially been marked as intermittent by the investigating team, but after further investigation (either immediately or after a future occurrence of the failure) were changed to some other status. Other statuses that the reports were later changed to included “Not a Problem”, which represented cases when the investigating team eventually decided that the failure report did not actually constitute a problem in the system, and that the customer had used a system in an incorrect way.

For the open source system, we chose Mozilla Firefox for our study. We had gathered a total of 82 intermittent failure reports for Mozilla Firefox. We used intermittent failure reports from both the industrial system and Mozilla Firefox as basis of our classification scheme as well as for identification of common preconditions that expose intermittent failures. We present more details about these processes in the following sections.

5.1.2 Data Collection

Prior to data collection, we divided the failure reports into three groups: intermittent, previously intermittent, and reproducible reports. Intermittent reports described failures that were found to be intermittent by the investigation team and could not be reliably reproduced. Such intermittent failures were, therefore, closed by the investigating teams as unresolved due to lack of reproducibility. Intermittent failure reports are represented by “Not Repeatable” failures in Figure 15. Previously intermittent refers to failure reports that had initially been classified as intermittent by their reporters or the investigation team, but were later changed to another status following further investigation. Finally, reproducible failure
reports refer to all failure reports that described regular reproducible failures. In other words, this group consisted of reports that could not be ascribed to the preceding two groups. The failure reports in intermittent and previously intermittent groups were of primary interest to us for this study.

For Mozilla Firefox, we had gathered 82 intermittent failure reports from Bugzilla as part of our hardware configuration study (Section 4) [62]. Bugzilla does not have a separate status for intermittent failures. We, therefore, gathered failure reports by running natural language queries on Bugzilla based on variations of different keywords such as "timing", "page file", "operating system", "system dependent", "race condition", "deadlock", "concurrency", "pentium", "older machines", and "slow computers". We observed that the incidence rate of intermittent failures was similar to the industrial system i.e., 11.5%. We read a total of 710 reports that were returned from our searches, and manually identified 82 of them as being intermittent.

5.1.3 Classification Methodology

Development of the classification scheme was essentially a grounded theory exercise [63], where we read a randomly selected sample of intermittent failure reports from both systems under test, and designed classes around the different types of intermittent failures we read. Each failure report had a description field, which was populated by the person who reported the bug, and described the nature of the failure. For the ABB system, a field Repeatable that had the values Yes or No specified whether the reporter was able to reproduce the failure on their machine consistently or not. For Firefox, the Reproducible field
specified whether the failure was *Always* reproducible or *Sometimes* reproducible. Additionally, the comments posted by other developers and testers on the failure report helped identify the exact nature of the failure, its symptoms, causes / pre-conditions, and proposed fixes / workarounds. All this preceding information helped identify a *classification* of the failure and an *attribute* of the failure.

We chose to develop both a classification scheme and a list of common attributes, because we felt neither fully encompassed the breadth of information available in the failure reports, and that attributes of failure supplement their respective classes. For example, a problem with a third-party library being used in the industrial system was not exposed until a customer installed an update supplied by ABB. Such information is useful, and can be used to improve testing processes before rolling out service packs and updates to a software product.

We also read a significant number of reports where it was not possible to identify the class and/or the cause of the failure due to insufficient information available. In our sample of intermittent failure reports, 54% reports were unclassified due to lack of information. Such unclassified reports described one-time occurrences of failures, and were not deemed serious enough to warrant further investigation by the team. In contrast, 10% of intermittent failures in the open source system were classified as one-time occurrences. We also carried out the same classification exercise for previously intermittent failure reports in both systems. We omit actual failure counts for the industrial system on purpose, since ABB does not wish to share those details.
The initial categories and attributes were designed by the author as an output of the grounded theory exercise. To remove bias and improve repeatability [19] of the scheme, two researchers (the author and an undergraduate student) read a selected sample of intermittent failure reports separately, and assigned appropriate classes and attributes to them. We demonstrate the repeatability of our classification scheme by listing kappa values and agreement coefficients for both classification and attributes in the following sections.

5.2 Classification of Intermittent Failures

This section answers RQ5, which we stated earlier as:

\textit{RQ5: What are the different causes of intermittence in software failures?}

We performed the classification by reading intermittent failure reports from an industrial and an open source system. Sections 5.2.1 and 5.2.2 discuss classification with respect to the industrial system and open source system respectively. In this section, we explain the 11 different classes and provide examples from our systems under test.

\textbf{Timing problems} – These failures depend purely on the timing of execution of actions. These actions may be manual or automated. For example, in our hardware configuration study (Section 4) [62], we were able to observe that Mozilla Firefox was susceptible to some unanticipated behavior on startup when given a certain input quickly. For our system under test, a failure report described running a test where the system communicated some information back and forth (for verification) with another instance of itself. The said system was operating under strict timing restrictions, and was unable to recover the entire communicated information due to low memory and processor speed.
**Race conditions** – Race conditions (or deadlocks) can be intermittent in nature and can have a number of underlying causes themselves including (but not limited to): operator speed (timing), third-party libraries (interoperability), badly written multi-threaded applications etc. For example, a failure in the database described a deadlock of the system under test that occurred only once when the system was shutting down. Investigation revealed that two threads were locked and waiting on each other after interacting with a third-party Microsoft component.

**Network problems** – Sometimes the communication medium is responsible for abnormal behavior of software that can be intermittent in nature. For example, the system recorded an event when a message sent from one system node to another was received twice, even though it was sent once. Although not serious in this instance, such failures can have unintended effects that may be damaging.

**Third-party libraries/drivers** – Third party components being used in software can also lead to intermittent failures. Our system under test used third-party libraries from several vendors in its different modules. For example, a failure report described an instance where the system on one deployment node kept crashing for no apparent reason. Investigation into the matter revealed that the graphics driver by nVidia was interacting abnormally with some graphics’ components of the system resulting in a crash. The investigation team requested new drivers from nVidia, and the failure was no longer seen.

**Memory Leak** – If memory is not properly allocated/de-allocated, the footprint of a program can increase (especially with prolonged usage) resulting in intermittent freezes and

---

29 [http://www.nvidia.com](http://www.nvidia.com)
crashes. For example, a failure report described an instance where memory of one megabyte was getting leaked when user opened a certain module. Further investigation revealed that this leakage occurred only when user carried out a certain series of actions (doing some maintenance action, and opening the module through a certain menu).

**Unhandled exceptions** – These failures usually represent extreme or corner cases in a program that are executed intermittently in unanticipated circumstances. Even software that is well-written may not be tested to the fullest extent under all possible input combinations. There can be underlying causes to unhandled exceptions that can be investigated further, but we use this category as a blanket for all failures that are caught by exceptions that were previously not detected.

**Disk errors** – This class of failures represents anomalous behavior of software resulting from errors in physical disk drives. For example, a failure report in our database described an instance where an important file belonging to a service was lost in the process of being written to disk. There was a disk write error, and all the data in the file was lost. On restarting the system, the service could not start since the file could not be recovered. This instance was not repeated again, so the failure was closed as being intermittent.

**Concurrency-related** – Concurrency-related intermittent failures have been the subject of extensive research in both academia and industry [46], [48], [64-66]. In our system under test, concurrency-related intermittent failures were relatively few. One such failure report described the name-client server cache being inconsistent following heavy usage of the system i.e., when concurrent transactions were being posted. This failure did not occur under normal operating conditions.
**User privileges and configuration settings** – Some failures arise due to improper assignment of user privilege and configuration information. The system under test in this study requires user permissions and other configuration settings on the deployment machine to be set up precisely. If these settings are not set up exactly as specified, some intermittent behavior may be encountered. For example, a report in the failure tracking database reported an occurrence of anomalous behavior due to some configuration settings. On certain occasions, when a new user logged on to the system, the new user could not carry out certain operations. When the user logged out and logged in again, the system behaved expectedly. This behavior occurred only for users that logged in the system for the first time. This category is represented as “User Settings” in Figure 16 for brevity.

**Miscellaneous or Software** – We created this blanket category for all other types of intermittent failures that lie undetected in the code, but do not necessarily fall under any of the preceding categories. There is usually no precedent for such failures, whether they lie inside a library or some third-party component interacting with the system. For example, a report described the intermittent occurrence of faulty tooltip messages. Switching back and forth between two windows inside the industrial system caused different tooltip messages to appear. This behavior was completely random, and only appeared sometimes. No cause for this behavior was found by the investigation team. Another interesting example is of a report which was concerned with setting background color of a Windows form to transparent after an action. This expected behavior of the form was only seen on some of the machines running our system under test, and not all. Investigation into the matter revealed inappropriate use of ActiveX calls in a .NET managed environment. These two examples
represent cases which may not necessarily fall under any of the other groups we define, but are still intermittent.

**One-time Occurrences** or **Unclassified** – These failures represent the most typical intermittent (or “truly intermittent”) failures. Such failures occur rarely and are hard to reproduce. In most occurrences in the industrial system, such failures occurred on specific builds of the software, but could not be repeated on the next incremental build. Hence, such failures were never looked at again unless they occurred again at some point in the future. For example, one failure report described an access violation exception in a dynamic link library (DLL) file being used by a component of the system. The crash logs led the team to the DLL file in question, but they were never able to repeat the said exception even after conducting several tests simulating a myriad of conditions using both old and new version of the component, or by recreating usage patterns that lead to the failure. The investigating team put such failures “under observation”, but typically very few of these failures occurred again. In other words, we can say that failure reports that were unclassified due to lack of information were assigned to this class.

The classification categories proposed in this section serve as the answer to our RQ5. These categories, however, are not mutually exclusive. There may be instances of failures that belong to more than one category. For example, a race condition could occur due to some underlying timing problem resulting from a network communications failure. We assigned each failure to its primary classification category. Figures 16 and 17 show a proportional distribution of classification of intermittent failures in both systems under test.
Since we assigned to each failure its primary classification, no failure was double-counted for the categories shown in Figures 16 and 17.

5.2.1 Industrial System: ABB

We read a randomly selected sample of intermittent and previously intermittent failure reports in the industrial system’s failure tracking database. In our sample, we were able to classify only 48% of all failure reports. The other 52% reports were unclassified due to lack of information present in the reports (see Figure 16). These unclassified reports described one-time occurrences of failures. Additionally, these reports had little to no information available that could have been used to classify the report, or even identify the preconditions that lead to the failures. We will look at preconditions in more detail in the next section. Of the entire sample, a total of 12% intermittent failures were fixed by the developers following some investigation.

![Figure 16. Distribution of classified intermittent failure reports for the ABB system](image-url)
5.2.2 Open Source System: Mozilla Firefox

As stated in Section 5.1.2, we also read 82 intermittent failure reports for Mozilla Firefox as part of our grounded theory exercise. The distribution of classified Mozilla Firefox failures is shown in Figure 3. Of the 82 intermittent failure reports at our disposal, we were able to classify 74 of them, and the rest were unclassified i.e., classified as one-time occurrences. Our previous study describes these failures in greater detail [62]. In our sample of intermittent failures for Mozilla Firefox, 58% of failures were fixed by the developers and testers following some investigation. The difference in the percentages of intermittent failures fixed across both systems was not further studied, and is left as future work.

Figure 17. Distribution of classified intermittent failure reports for Mozilla Firefox
5.2.3 Repeatability of Classification Scheme

To demonstrate the repeatability of our classification scheme, we calculated Cohen’s kappa (κ) statistic using free utility called ComKappa\textsuperscript{30}. The utility took as input a table similar to Table 7 showing the differences in classification amongst the two raters. Classifications done by rater A (the author) are represented as rows, and classifications done by the undergraduate student are represented as columns. For example, row 1 and column 1 represents the number of failures classified as Timing problems by both raters A and B. Similarly, row 2 column 2 represents failures classified as Race conditions by both raters A and B. All diagonals, therefore, represent failures that were classified the same by both raters. The rest of cells in Table 7 represent disagreements with their row representing rater A’s classification and column representing rater B’s classification. For example, row 4 column 1 cell value represents the number of failures that were classified as Third-party by rater A, and Timing by rater B. The kappa (κ) value for Table 7 is 0.65, which indicates sufficiently good agreement (see Section 2.4).

\textsuperscript{30} http://www2.gsu.edu/~psyrab/ComKappa2.zip
Table 7. Rater classifications for intermittent failures of industrial and open source system

<table>
<thead>
<tr>
<th>Rater Classifications A/B</th>
<th>Timing</th>
<th>Race Condition</th>
<th>Network</th>
<th>Third-party</th>
<th>Memory Leak</th>
<th>Unhandled Ex.</th>
<th>Disk Error</th>
<th>Concurrency</th>
<th>User Settings</th>
<th>Software</th>
<th>Hardware</th>
<th>Unclassified</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing</td>
<td>36</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>9</td>
<td>54</td>
</tr>
<tr>
<td>Race condition</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Network</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>Third-party</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>Memory Leak</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>29</td>
</tr>
<tr>
<td>Unhandled Exception</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>0</td>
<td>26</td>
<td>58</td>
</tr>
<tr>
<td>Disk error</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>Concurrency</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>User Settings</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>Software</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>96</td>
<td>0</td>
<td>11</td>
<td>114</td>
</tr>
<tr>
<td>Hardware</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>Unclassified</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>19</td>
<td>1</td>
<td>209</td>
<td>244</td>
</tr>
<tr>
<td>Total</td>
<td>42</td>
<td>9</td>
<td>26</td>
<td>27</td>
<td>25</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td>16</td>
<td>152</td>
<td>15</td>
<td>291</td>
<td>645</td>
</tr>
</tbody>
</table>
5.3 Attributes of Intermittent Failures

This section explains our RQ6, which we stated earlier to be:

**RQ6:** What are the common attributes of intermittent behavior in software?

We identified common attributes of intermittent failures by reading intermittent failure reports from an industrial system (ABB) and an open source system (Mozilla Firefox). These attributes represent preconditions and causes of intermittent behavior in our systems. In this section, we explain each attribute in detail, and provide a representative example.

**High Processor Load** – As demonstrated in our earlier study [62], processor load can significantly impact the occurrence of a failure. Processor load, in such instances, need not be as a result of a specific process, but refers to a general working load of the system under test. For example, one failure report described an instance where one of the services used by ABB system kept restarting every two minutes due to high processor load. The service would crash and then restart. This failure occurred only on some networks and was intermittent.

**Hardware (Processor Speed and Memory)** – Similar to the previous attribute, this attribute also alludes to hardware configuration study (Section 4) [62]. We found several failures in both our systems under test that occurred on deployment machines running on low hardware specifications, specifically low processor speed and memory. Resource-intensive operations conducted on the ABB system caused the machine to become unresponsive. The system would sometimes recover from this state once it had completed the operation, and at other times it would stay unresponsive.
**Continuous Operation** – There are failures that only tend to occur in systems that have been running continuously for a long period of time. Such failures are not necessarily dependent on memory leaks or excess resource consumption. For example, one ABB failure report indicated that the system had been running continuously for two days, and its main module stopped responding completely. The system had to be restarted so that it could return to normal operation.

**Synchronization** – Synchronization failures generally occur as a result of two process calls failing to acknowledge each other (or missing each other’s calls completely). These failures may be timing related, network-related, or dependent on processor load or memory. The ABB system could be set up in an environment where it communicates with other instances of itself and regularly sends information back and forth. Therefore, the system needed to be in complete synch with other nodes on a network.

**Operating system** – Sometimes, the underlying cause of intermittence of the system is the operating system. In our systems, some failures were only observed on a certain operating systems. For example, one ABB failure report described an occurrence of a Microsoft Visual C++ runtime library error that was only exposed on machines running Windows 2008 Server. The fact that a failure is only exposed on a certain operating system makes operating system a variable of interest when dealing with intermittent failures.

**Failures related to restarts** – A significant number of failures in the ABB system disappeared after a restart of the machine. For example, there was a failure that caused one of the major modules of the system to malfunction after an update was applied. The occurrence
of this failure was intermittent, but it was common enough for the development team to make a restart of the system necessary after application of the said update.

**Failures resulting from updates** – Similar to the previous attribute, this attribute deals with failures that occur after an internal software or third-party update is applied. For example, there was a report in our database that described a failure that occurred on machines that had upgraded Internet Explorer from Version 7 to 8. This update caused unanticipated consequences at some customer sites, and the update had to be immediately uninstalled before a workaround was identified by the investigation team.

**Endless Loops** – There were a number of failures in our database where the system sometimes ended up in an infinite loop, and used up extensive memory and processor resources. Such failures are different than race conditions or deadlocks. Endless loops lead to the system becoming unresponsive, and the process had to be terminated manually, sometimes resulting in data loss.

**Failures resulting from disturbance tests** – Disturbance tests are defined as tests that disrupt normal operation of the application, such as switching off network connection, unplugging the power cord, or simulating some other physical failure [67]. The anomalous behavior of the software seen as a result of some of the disturbance tests is very hard to repeat generally. ABB conducts disturbance tests during its comprehensive testing phase.

Figures 18 and 19 show the distribution of the identified attributes for ABB and Firefox respectively. Developers and testers can make use of this information when dealing with
intermittent failures in their systems. The identified attributes show that testing in unanticipated conditions can often lead to intermittent failures in systems.

Figure 18. Distribution of attributes of intermittent failure reports for the ABB system

Figure 19. Distribution of attributes of intermittent failure reports for Mozilla Firefox

5.3.1 Repeatability of Assigned Attributes

Similar to the repeatability of the classification scheme (Section 5.2.3); we demonstrate the repeatability of the attributes of intermittent behavior in Table 8. The Cohen’s kappa (κ)
statistic for the attributes of intermittent failures came out to be 0.61, which is termed as good agreement (see Section 2.4). This value was also calculated using the free utility ComKappa. Classifications done by rater A (the author) are represented as rows, and classifications done by the undergraduate student are represented as columns.
### Table 8. Rater classifications for attributes of intermittent failures of industrial and open source system

<table>
<thead>
<tr>
<th>Rater Classifications</th>
<th>Hardware</th>
<th>High Processor Load</th>
<th>Continuous Operation</th>
<th>Endless Loops</th>
<th>Operating System</th>
<th>Synchronization</th>
<th>Failures from Updates</th>
<th>Failures from Restarts</th>
<th>Failures from Dist. Tests</th>
<th>Unclassified</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>21</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>9</td>
<td>42</td>
</tr>
<tr>
<td>High Processor Load</td>
<td>4</td>
<td>18</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>38</td>
</tr>
<tr>
<td>Continuous Operation</td>
<td>5</td>
<td>6</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>32</td>
</tr>
<tr>
<td>Endless Loops</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>37</td>
</tr>
<tr>
<td>Operating System</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>36</td>
</tr>
<tr>
<td>Synchronization</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>Failures from Updates</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>3</td>
<td>1</td>
<td>16</td>
<td>44</td>
</tr>
<tr>
<td>Failures from Restarts</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>41</td>
<td>5</td>
<td>19</td>
<td>71</td>
</tr>
<tr>
<td>Failures from Dist. Tests</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>19</td>
<td>6</td>
<td>34</td>
</tr>
<tr>
<td>Unclassified</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>280</td>
<td>287</td>
</tr>
<tr>
<td>Total</td>
<td>34</td>
<td>34</td>
<td>27</td>
<td>28</td>
<td>18</td>
<td>22</td>
<td>31</td>
<td>54</td>
<td>25</td>
<td>372</td>
<td>645</td>
</tr>
</tbody>
</table>
5.4 Additional Observations

A number of other things stood out while gathering data for intermittent failure reports that deserve a mention. We enumerate them below.

**Spelling mistakes** – The majority of developers and testers of the industrial system were non-native English speakers. Therefore, spelling mistakes were fairly common across all failure reports. Mistakes such as “reproducible” instead of “reproducible” were common to the extent that “reproducible” was identified as an important keyword in our natural language processing results (Section 7).

**Communication** – As mentioned in the previous bullet, most of the contributors to the failure reports were non-native speakers of English, so sometimes there were communication issues among teams that contributed to the delay in getting to the root cause of an intermittent problem. Another aspect of communication is the blame game that tends to take place amongst teams in an organization to determine who should take ownership for fixing a failure. Such scenarios are more common in Firefox, but we also identified such cases in the industrial failure tracking system.

**Traps/hooks** – A practice that was common during investigation of intermittent failures in both Firefox and the ABB system was that of placing “traps” or “hooks” in the code to determine the failing component.

**Logging** – We observed that logging and tracing was often used in conjunction with traps/hooks to determine the root cause of an error. Developers of the systems under test
would put additional logging calls around the piece of code they suspect was the source of the intermittent fault.

**Experienced testers** - Often when dealing with intermittent failures, the investigators would defer to the opinion of a selected group of experienced testers who presumably had a much better understanding of the code, as well as holding a position of authority that enabled them to dictate the direction of the investigation. In addition, contributions by experienced testers generally offered more insight about the problem owing to their greater knowledge about the system. Typically, experienced testers (on both Firefox and ABB system) would suggest exactly where to look for the fault in the source code.

**Automated testing** - We observed some instances where once a repeatable pattern to an intermittent problem was identified, testers built automated test cases to test out different scenarios, and were able to successfully reproduce the behavior more frequently than before.

### 5.5 Limitations and Threats to Validity

In this study, we propose a classification of intermittent failures based on our case study of an industrial and an open source system. Classification schemes, however, run the risk of being subjective, and that they need to be repeatable [19]. For external validity, our findings may not be entirely representative of all software systems in use today. Therefore, further work is needed to show that our classification scheme generalizes and is not merely opinion-based.
For internal validity, we use intermittent failure reports as our basis of analysis so that we may be able to explain their behavior. We minimized selection bias in by randomizing our choice of intermittent failure reports to analyze. However, some level of selection bias may still persist.

For construct validity, we manually classified intermittent failure reports of an industrial and an open source system. As is the case with all classification schemes, our proposed scheme also runs the risk of being subjective and opinion-based. Therefore, further studies are and refinements may be needed to show that the scheme is orthogonal.

### 5.6 Conclusion

As stated in related work, a number of classification schemes have been proposed in the literature [18], [17], [61]. These classification schemes serve as standards that aid software developers and testers in analyzing software processes in their organization. However, these classification schemes are detailed, as well as being general in nature. Sometimes classification schemes tailored to specific areas need to be developed that can help software practitioners better understand specific problems. The motivation behind our classification scheme was similar i.e., to create a classification scheme targeted specifically at intermittent failures.

Given the pervasiveness of intermittent failures in industry and open source software, developers and testers can use our classification to better identify process problems in their organization that lead to intermittent failures. For example, using our proposed classification
scheme and identified preconditions, developers and testers can run tests on their software to look for similar patterns in behavior that can be further investigated using causal analysis. Detection and prevention of potentially intermittent failures early in the process can lead to significant reduction in cost and development time.

Additionally, developers and testers can incorporate the information presented in this paper in their testing strategies. For example, they can prioritize test cases based on patterns of behavior observed in their system. Organizations can also use this information to analyze different aspects of their software development methodologies, and make improvements as needed. For example, the identified classes and preconditions presented in this paper can be added to failure tracking systems to add relevant detail to the failure report, and in turn, also improve the quality of the report.
6. **EFFORT SPENT DEBUGGING INTERMITTENT FAILURES**

In previous chapters, we performed a hardware configuration study followed by identification of common causes and attributes of intermittent failures. In this chapter, we discuss the significance of extra effort that is spent investigating intermittent failures compared to reproducible failures. A significant challenge to effective and efficient software testing is exposing faults such that they can be observed as failures. Over the history of one large product at ABB, 10.2% of all reported software failures were determined to be non-reproducible by the development and test teams during the product development cycle. For failures detected in test, it is difficult to determine the underlying fault when reproducing the failure is not possible. When the underlying fault cannot be determined, the failure report is often closed and no additional effort is spent on it until additional occurrences are detected. For faults detected in the field, developers spent considerable effort in either remotely debugging the fault or traveling to the customer site itself to determine the fault and fix it.

6.1 **Study**

The goal of this research was to reduce the effort spent on debugging intermittent failures by increasing the knowledge about them, and by providing actionable insight into their causes and underlying faults. Thus, we were interested in the following research question:

*RQ7: What is the difference in effort spent debugging intermittent failures as opposed to reproducible failures?*
We describe data collection methodology and answer this research question in the subsequent sections, using different metrics we used.

### 6.1.1 Data Collection

We conducted the failure report analysis on the ABB system and Mozilla Firefox. Since neither of the repositories contained information that shows the actual effort spent by software engineers in debugging intermittent failures, we used surrogate measures to calculate the relative effort spent. We used two metrics for this purpose: number of comments and failure resolution time. We wrote an application in Visual C#.NET that parsed all failure reports we had collected for each system, extracted metrics and other information from the reports and saved these to a MySQL database. We used these metrics to measure statistically significant differences among intermittent, previously intermittent, and reproducible failures (for the ABB system) and between intermittent and reproducible failures (for Firefox).

From the failure reports, our parser application calculated metrics such as Number of comments, Failure resolution time, Length of the report (in words), Number of change sets, and Number of attachments. We were also able to determine whether reports had stack traces or steps to reproduce in their description.

The choice of using number of comments on each failure report as a means of calculating effort has its pros and cons. Other modes of communication not privy to us, such as one-on-one conversations, email, and instant messaging would have been used by teams to
communicate with each other. This aspect alone warrants another study, based on suggestions proposed in [51].

6.2 Number of Comments

For intermittent failures in the industrial system, we found that the average number of comments were 2.8 per failure report. However, we can dissect this number and analyze this statistic in greater detail by looking at sub-classes of intermittent failures. Figure 20 shows the distribution of intermittent failure reports in terms of number of comments. As we stated earlier in Section 5.2.1, around 54% of our sample were one-time occurrences of failures. Among failure reports describing such one-time occurrences of failures, 67% of reports had less than three comments on them, thus bringing the total average down. For the other 33% of reports, the average number of comments was higher (3.6).

Figure 20. Distribution of number of comments for intermittent failure reports for ABB system
For previously intermittent failures, the average number of comments were much higher i.e., 6.1 comments per failure report. Of these, the failures that were later changed to Change Applied (meaning they were fixed), the average was even higher i.e., 7.6 comments per failure report. Figure 5 shows the distribution of number of comments in previously intermittent failure reports.

We used Mann-Wilcoxon-Whitney test to determine the significance of the difference between average number of comments among the three failure groups (intermittent, previously intermittent, and reproducible). The difference in average number of comments on intermittent reports was statistically significant when compared to reproducible reports (P < 0.01 at α=0.05 level). Similarly, the difference in average number of comments on previously intermittent reports and reproducible reports was also statistically significant (P < 0.01 at α=0.05 level). We also compared intermittent and previously intermittent failure reports, and found their difference to be statistically significant as well (P < 0.01 at α=0.05 level).

![Figure 21. Distribution of number of comments for previously intermittent failure reports for ABB system](image)

74
For Firefox, the average number of comments on reproducible and intermittent failure reports was 17.27 and 34.33 respectively. This difference was statistically significant (P < 0.01 at α=0.05 level). Tables 9 and 10 summarize the effort metrics for ABB and Firefox systems respectively.

Table 9. Results summarizing estimation of effort spent debugging failures in industrial system

<table>
<thead>
<tr>
<th>Failure Report Metrics / Failure Report Groups</th>
<th>Intermittent</th>
<th>Previously Intermittent</th>
<th>Reproducible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Number of Comments</td>
<td>2.8</td>
<td>6.1</td>
<td>3.7</td>
</tr>
<tr>
<td>- With Status <em>Change Applied</em></td>
<td>6.6</td>
<td>7.6</td>
<td>5.0</td>
</tr>
<tr>
<td>- Submitted by Customer</td>
<td>3.9</td>
<td>9.5</td>
<td>4.9</td>
</tr>
<tr>
<td>Average Resolution Time (days)</td>
<td>81</td>
<td>161</td>
<td>105</td>
</tr>
<tr>
<td>- Submitted by Customer</td>
<td>69</td>
<td>586</td>
<td>136</td>
</tr>
<tr>
<td>Average Length (number of words)</td>
<td>59.64</td>
<td>58.75</td>
<td>59.98</td>
</tr>
<tr>
<td>Average Severity (1 to 5)</td>
<td>2.74</td>
<td>2.72</td>
<td>2.49</td>
</tr>
</tbody>
</table>

6.3 Failure Resolution Time

We also looked at the average resolution time for failures of different groups as a further means of estimating effort spent. We define resolution time as the number of days the failure report was open i.e., the difference between the report close date and submit date. As can be seen in Table 9, the resolution time for intermittent reports in the industrial system is lower (81 days) than other two groups. The resolution time for previously intermittent reports
was much higher (161 days) indicating that some prior investigation had been carried out by
the developers and testers as a result of which, the failure was either reproduced and fixed or
changed to some status (such as “Not a Problem”). Similarly, for comparison, we also list the
average resolution time for a reproducible failure report i.e., 105 days.

We compared the difference in failure resolution among the three failure groups
mentioned previously. Similar to number of comments, the difference in failure resolution
times for intermittent and reproducible failures was statistically significant (P < 0.01 at
α=0.05 level). The difference in failure resolution times of previously intermittent and
reproducible failures was also statistically significant (P < 0.01 at α=0.05 level). Finally, the
difference between resolution times of intermittent and previously intermittent failures was
statistically significant as well (P < 0.01 at α=0.05 level).

For Firefox, the average resolution time for reproducible and intermittent failures was
362.82 and 394.25 days respectively. This difference was statistically significant (P < 0.01 at
α=0.05 level).

Table 10. Results summarizing estimation of effort spent debugging failures in the open
source system

<table>
<thead>
<tr>
<th>Failure Report Metrics</th>
<th>Intermittent</th>
<th>Reproducible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of comments</td>
<td>34.33</td>
<td>17.27</td>
</tr>
<tr>
<td>Resolution time (days)</td>
<td>394.25</td>
<td>362.82</td>
</tr>
<tr>
<td>Length of report (words)</td>
<td>96.23</td>
<td>120</td>
</tr>
<tr>
<td>Severity</td>
<td>2.6</td>
<td>1.98</td>
</tr>
</tbody>
</table>
These results support our earlier finding that developers and testers spent more effort debugging intermittent failures than reproducible failures, and that effort spent was directly proportional to the likelihood of intermittent failure being reproduced.

Customers are the ultimate judge of quality of any software system [68]. Another subset of failure reports that we analyzed were reports submitted by customers in the industrial system. Calculating the average number of comments metric for each failure group with respect to *Submitter Class* (Customer/ Dev/ Test); we found that customer-submitted reports had the highest number of comments for each of the failure groups. For intermittent reports, average number of comments was 3.9; for previously intermittent reports, the average was 9.5; and for reproducible reports, the average was 4.9. This confirms the belief that failure reports submitted by customers are given the highest priority by testing teams.

We list average resolution times in Table 9 for all failure reports submitted by customers. The average resolution time for intermittent, previously intermittent, and reproducible reports was 69, 586, and 136 days respectively. The resolution time for previously intermittent failure reports was high due to relatively lesser number of such reports compared to other failure groups. However, these differences were still statistically significant (P < 0.01 at α=0.05 level).

### 6.4 Length of Failure Reports

We hypothesized that failure reports that were longer in length would have higher chances of being reproduced (and subsequently fixed) as they were more likely to
meaningful information for developers and testers, such as stack traces and execution logs.
For the industrial system under test, the average lengths of failure reports were similar among all three groups of failure reports (see Table 9). The difference between average length of previously intermittent failure reports when compared to intermittent reports, and reproducible reports was not found to be statistically significant, with p-values of 0.48 and 0.44 respectively.

For Firefox (see Table 10), we found that the average length of the report varied across the two groups. For reproducible failures, failure reports had 120 words on average, and for intermittent failures, this number was much less 96.23. The difference was expectedly found to be statistically significant (Mann-Wilcoxon-Whitney test, P < 0.01 at α=0.05 level).

6.5 Failure Severity

The failure tracking database for the industrial system under test assigned five values as indicative of severity of a failure report. We converted these values to a numeric scale: one to five, with five being the highest severity. The text-to-numeric mappings were: 1-Low, 2-Medium, 3-High, 4-Critical, 5-Project Stopper. Table 9 shows the average severity for failure reports for all three groups in the ABB system. The difference between severities of reported intermittent and reproducible failures was statistically significant (Mann-Wilcoxon-Whitney test, P < 0.01 at α=0.05 level). Similarly, the difference between previously intermittent and reproducible failures was also statistically significant (Mann-Wilcoxon-Whitney test, P < 0.01 at α=0.05 level). The difference between intermittent and previously intermittent
failures, however, was not statistically significant (Mann-Wilcoxon-Whitney test, P=0.83 at α=0.05 level).

In Bugzilla the severity scale is numeric, and runs from zero to six. The mappings are: 0-Trivial, 1-Minor, 2-Normal, 3-Major, 4-Critical, 5-Blocker. The distribution of severity for intermittent and reproducible failures in Firefox is shown in Figure 23. In terms of average, we found that the average severity for intermittent failures was 2.6 compared to reproducible failures, which was 1.98. The difference was statistically significant (Mann-Wilcoxon-Whitney test, P < 0.01 at α=0.05 level).
6.6 Additional Observations

We tested several other hypotheses regarding the occurrence of intermittent failures in the ABB system. They included testing for the significance of *Submitter Location* (due to distributed nature of developers); *Submitter* (whether certain people report more intermittent failures); and *System Components*. We did not have access to the complexity or size of different system components; therefore, no conclusion about their significance could be reached. For Submitter location and Submitter, the percentage of locations and submitters respectively were similar across all three groups of failure reports and were, therefore, were not statistically significant. For Firefox in particular, we found that on average reports describing reproducible failures had higher number of attachments than intermittent failure reports. This difference in number of attachments was statistically significant (Mann-Wilcoxon-Whitney test, $P < 0.01$ at $\alpha=0.05$ level).

6.7 Summary

In this section, we presented our answer to RQ7. We used two metrics – number of comments and failure resolution time – to indicate that developers and testers spent more effort debugging intermittent failures compared to reproducible failures. We also showed that reports that were submitted by customers, on average, had more comments than reports submitted by developers and testers. Finally, we found that intermittent failures were generally more severe than reproducible failures. However, due to a large proportion of our
intermittent failure sample being one-time occurrences, these failures may not have been fixed despite their higher severity.

6.8 Limitations

As mentioned earlier, number of comments and failure resolution time metrics may not be representative of total effort spent investigating intermittent failures. We cannot preclude the possibility that other modes of communication such as one-on-one meetings, email, or instant messaging, were also used in conjunction with comments. We also used the metric failure resolution time as another measure of effort spent debugging intermittent failures. We, however, considered only a certain window of time i.e., the elapsed days between failure report submit date and close date. Factors outside this time window that may have affected the failure in some way were, thus, not considered in our study. Also, failure resolution time in some cases may have been higher due to included vacation time, or it may have been higher as resources may have been diverted towards fixing more severe failures. To summarize, we believe our findings are indicative of relative effort spent despite these limitations.
7. PREDICTION OF INTERMITTENT FAILURES

We mined metrics from failure repositories of a large industrial software system developed by ABB Inc. and a large open source system, Mozilla Firefox, for the purpose of predicting intermittent failures. We have shown in Section 4 that load and configuration testing techniques such as altering processor speed, memory, and processor load can help reproduce intermittent failures. Therefore, we propose an “early warning” system that can flag potential intermittent failures at the time the failure is reported. Successful identification of the intermittent nature of a software failure at the time the failure report is filed can be critical, as developers and testers can immediately direct their efforts towards reproducing the failure based on their past experiences. Additionally, this information can be incorporated in a triage tool so that the failure can be assigned to the right people, thus saving considerable effort [69].

The goal of this research is to help software engineers understand the characteristics of intermittent behavior in software systems, and to aid in developing efficient techniques for detecting and fixing such failures. Thus, we are interested in the following research questions:

**RQ8:** Can metrics collected from failure repositories be used to develop a prediction model for intermittent failures?

**RQ9:** Can natural language processing techniques aid us in predicting intermittent failure reports?

We answer these research questions in the sections that follow.
7.1 Prediction Modeling

We used the extracted metrics to train a model for predicting whether a failure report is Intermittent or Reproducible given the information available when the failure report is entered in the system. We had access to the entire ABB failure tracking database. Using the “Not Repeatable” status to distinguish intermittent and reproducible failure reports, we were able to use all reports in the database for training and testing our prediction model. For the Firefox dataset, we had a limited number of intermittent failure reports manually gathered, and downloaded a further 780 reproducible failure reports that were randomly selected. Our application ensured that no failure report was downloaded more than once, and that there were no overlapping failure reports between intermittent and reproducible groups of reports.

Failure reports in our systems under test contained significantly different information. Reports from the industrial system had a lot of product-specific information as opposed to Bugzilla reports. Therefore, we created separate prediction models for each system. Furthermore, we used the information gain attribute selection technique in Weka\(^\text{31}\) to identify the strongest predictors of the class attribute. For both systems under test, we used two-thirds of the datasets for training, and the rest for testing the prediction model. These predictor metrics represented the independent variables in our model, and the binary prediction class of \textit{Intermittent}/\textit{Reproducible} was the dependent variable.

\(^{31}\text{http://www.cs.waikato.ac.nz/ml/weka/index.html}\)
7.1.1 Modeling Techniques

We used three classification techniques to predict intermittent failures: Naïve Bayes, Bayesian network, and J48 Decision Trees. These techniques have been extensively employed in the literature for predicting faults [70-72].

Naïve Bayes is based on the Bayesian theorem of conditional probability, and assumes that all attributes of a model are independent of each other. This assumption seems oversimplified, but Naïve Bayes has shown good prediction performance [71].

Bayesian network classifier represents a set of random variables along with their conditional probabilities via a directed acyclic graph. The graph represents causal relationships between nodes. Each node has a conditional probability for each value provided its parent nodes evaluate to true.

J48 Decision Trees are an extension of the C4.5 algorithm. Decision trees have been known to perform well in machine learning tasks that involve structured data and numerical attributes. Each node in the tree represents a predictor variable that divides the tree based on a range of values. The algorithm selects nodes at each stage based on the criteria that the traversal path from root node to the leaf node is minimal.

7.1.2 Natural Language Processing Techniques

To answer RQ9, we identified the top 50 most commonly used words in intermittent failure reports in both systems using natural language processing techniques. We used the R32 statistical computing and graphics tool to construct a term-document matrix, and used term

32 http://www.r-project.org
frequency – inverse document frequency weights to identify the 50 most commonly used words in intermittent failure reports (see Appendix A). We added columns for each of these words in our database tables, and used our parser application to populate these fields with occurrences of these common words for all reports in the dataset. The occurrences were represented as 0 or 1 for each report in the dataset. We then built another prediction model that incorporated the common word occurrences as predictor metrics of intermittent failures. We describe the performance implications of doing so in Section 7.2.

**Term Frequency - Inverse Document Frequency** is a statistical measure used to evaluate the importance of terms in a document corpus. The weights increase with frequent of occurrences of terms in a document, but is offset by the frequency of the word in the entire corpus. This algorithm finds many applications in search engines ranking schemes.

### 7.2 Results

Our predictive goal was to identify from the full set of failure reports those that were likely to exhibit behavior such that techniques for improving the chances of reproducing the failures would be executed. The training set for the prediction model constituted of metrics mined from failure reports that had been marked as being *Intermittent* or *Reproducible*. Only metrics that would be available when the report is first submitted were considered since we wanted our prediction model to work at the onset of processing the failure. For example, attributes such as *Assigned To* and *Number of Change sets* were not part of the model.
Our dataset for the ABB system contained 53 metrics that we were able to reduce to ten based on information gain attribute selection done through Weka. The ten predictor metrics for ABB system were:

- Component, Submitter, Submitter Class, Severity, Repeatable, System Function, Error Exists in Previous Version, Estimated Time to Change, Risk, and Test Case Reference

Similarly, failure reports in Bugzilla for Firefox initially contained 23 metrics that could be used for prediction. Using attribute selection techniques from Weka, we refined the number of predictor metrics to 12. They were:

- Reporter, Product, Component, Version, Platform, Priority, Severity, QA Contact, Number of Attachments, Length of Report, Stack Trace, and Steps to Reproduce

The Repeatable metric in the ABB prediction model should not be confused with the “Not Repeatable” status of failure reports in the ABB system. The Repeatable metric was set at the time the failure was reported, and indicated whether the reporter was able to successfully reproduce the failure or not.

For failure report prediction, we used three classification techniques: Naïve Bayes, Bayesian Nets, and Decision Trees. We next describe the most significant predictive factors for our models for both systems.

### 7.2.1 Industrial System: ABB

The performance of the different classification algorithms in our prediction model for the industrial system is shown in Table 11. Given the large number of reproducible failures in
our dataset (89.8%), a simple classifier that predicted all reports to be reproducible could achieve an 89.8% precision and a 100% recall. Therefore, the prediction model was designed so as to maximize the information gain, and appropriate attribute selection was carried out, as explained in Section 7.1.

We first describe our prediction model based solely on attributes extracted from the failure report, and no common word occurrences. As seen in Table 11, Naïve Bayes and Bayesian Nets demonstrated similar levels of performance based on our predictive metrics whereas Decision trees classified all reports as Reproducible. Naïve Bayes and Bayesian Net both had false positive rates (FPR) of less than 0.05, due to a large number of reproducible failure reports being correctly classified. For both Naïve Bayes and Bayesian Nets, the most significant predictors were Submitter, Component, Submitter Class, and System Function. Overall, models based solely on attributes extracted from failure reports did not yield high prediction accuracy.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Precision</th>
<th>Recall</th>
<th>ROC Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>0.35</td>
<td>0.21</td>
<td>0.76</td>
</tr>
<tr>
<td>Bayesian Net</td>
<td>0.33</td>
<td>0.24</td>
<td>0.76</td>
</tr>
<tr>
<td>J48 Decision Trees</td>
<td>0</td>
<td>0</td>
<td>0.50</td>
</tr>
</tbody>
</table>

The second prediction model for the industrial system included columns specifying occurrences of common words in the failure reports. The performance of the three classifiers on this extended model is shown in Table 12. The precision of all classifiers in this instance
improved, but recall improved only slightly. For Naïve Bayes and Bayesian Nets, the most significant predictive metrics were *Submitter Class, Risk*, followed by commonly used words such as *timer, repeatable, under observation, unable to repeat, reproducible, cannot repeat,* and *bring out developer.* The word *reproducible* is common misspelling, and was used so often that it became a significant predictor of intermittent failure reports.

Decision trees had the highest precision of all the classifiers in the second model for ABB. At the root of the tree was the word *repeatable*, followed by *not reproducible*, the metric *Risk*, and further by words *cost, solution, validation,* and *timer.* At the base of the tree, failure reports that did not have the words *cost* and *solution* were most often predicted as *Intermittent.* This had been our observation as well, since investigators of the failure would use those words when a potential fix for the failure had been identified.

**Table 12. Performance of prediction model for industrial system (with common word occurrences)**

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Precision</th>
<th>Recall</th>
<th>ROC Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>0.51</td>
<td>0.28</td>
<td>0.82</td>
</tr>
<tr>
<td>Bayesian Net</td>
<td>0.51</td>
<td>0.28</td>
<td>0.82</td>
</tr>
<tr>
<td>J48 Decision Trees</td>
<td>0.60</td>
<td>0.26</td>
<td>0.63</td>
</tr>
</tbody>
</table>

In terms of common attributes of intermittent failures, we found that the category of *Failures related to restarts* and *Failures resulting from updates* had the highest number of failures correctly predicted as intermittent.
7.2.2 Open Source System: Mozilla Firefox

The performance of our prediction model for Mozilla Firefox is shown in Table 13. Similar to the model for the industrial system, the Naïve Bayes and Bayesian Net classifiers had the same performance level. Decision trees, again, classified all failure reports as Reproducible. Both Naïve Bayes and Bayesian Net models relied the most on predictive factors of Reporter, Product, Severity, and Stack trace for their predictions. However, similar to the industrial system model, the performance of both classifiers for this model was not satisfactory.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Precision</th>
<th>Recall</th>
<th>ROC Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>0.30</td>
<td>0.34</td>
<td>0.78</td>
</tr>
<tr>
<td>Bayesian Net</td>
<td>0.33</td>
<td>0.31</td>
<td>0.78</td>
</tr>
<tr>
<td>J48 Decision Trees</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The second prediction model based on common word occurrences as well as metrics from the failure reports performed better than the first model for Firefox (see Table 14). Both Naïve Bayes and Bayesian Nets had reasonable precision and recalls, as well as Area under ROC Curve values. For both Naïve Bayes and Bayesian Nets, the most significant predictive factors were Reporter, Stack trace, Steps to reproduce, Length of the report and words CPU, timer, and memory.
Decision tree had the highest precision of all classifiers with 0.63, but had very small recall. At the top of the tree was the node for the word *CPU*, followed by *memory* and metric Length of the report. Additionally, at the base of the tree, Steps to reproduce and Stack trace metrics helped to distinguish between reproducible and intermittent failure reports. Failure reports that contain stack traces and steps to reproduce tend to be repeatable, and get fixed sooner [52].

**Table 14. Performance of prediction model for open source system (with common word occurrences)**

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Precision</th>
<th>Recall</th>
<th>ROC Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>0.50</td>
<td>0.50</td>
<td>0.89</td>
</tr>
<tr>
<td>Bayesian Net</td>
<td>0.52</td>
<td>0.57</td>
<td>0.89</td>
</tr>
<tr>
<td>J48 Decision Trees</td>
<td>0.63</td>
<td>0.17</td>
<td>0.52</td>
</tr>
</tbody>
</table>

In terms of common attributes of intermittent failures, we found that failures belonging to *Hardware* and *High Processor Load* were most correctly predicted to be intermittent. Additionally, for both systems under test, we observed that the inclusion of common word occurrences in the prediction model helped increase its accuracy. This finding answers our RQ9 that natural language processing techniques can be useful in identifying potentially intermittent failure reports at the time the failures is reported.
7.3 Limitations

For internal validity, we use intermittent failure reports as our basis of analysis so that we may be able to explain their behavior. We minimized selection bias in by randomizing our choice of intermittent failure reports to analyze. However, some level of selection bias may still persist.

For external validity, we conducted our case studies on an industrial and an open source system. Our findings, especially regarding prediction, may not generalize to all software systems. Further studies are required to determine the degree to which our findings generalize.

7.4 Conclusions

In this study, we explored the feasibility of extracting metrics from software failure repositories for predicting intermittent failure reports at the time the failure is reported. We performed a case study on an industrial system developed by ABB and an open source system, Mozilla Firefox. We examined failure reports from both systems and found that:

- Software engineers spent more effort investigating intermittent failures than reproducible failures
- Intermittent failures often result when systems encounter unanticipated conditions, such as excess processor load or application of a faulty third-party update
- Intermittent failures often come to the fore when the system ends up in an abnormal state, and thus can be avoided by a simple restart of the system
- Incorporating information such as stack traces and steps to reproduce can be useful in reproducing intermittent failures
- Detailed failure reports are much more likely to be fixed sooner when compared to shorter failure reports
- The vocabulary used to describe problems in failure reports varies from one system to another, and indicates differences in company culture, background of the reporter etc.
- Natural language processing techniques can be used to extract metrics and terms from failure reports for training a model that can predict whether a failure report is intermittent at the time the report is filed.
8. CONCLUSION AND RECOMMENDATIONS

This research was aimed at broadening the knowledge about intermittent failures that plague software systems. We established the prevalence of intermittent behavior in today’s software systems, and conducted case studies to establish the significance of effort that is wasted debugging and fixing intermittent failures. Additionally, we proposed a classification scheme and identified common attributes to help software engineers identify common characteristics and patterns intermittent failures in their systems. Finally, we demonstrated that a prediction model based on natural language processing techniques can be used to identify intermittent failures at the time the failure is reported. We summarize our findings in this section, and offer general advice we gathered based on our experience of investigating intermittent problems.

8.1 Identifying Intermittent Failures

Fixing faults requires finding them.

A major component of software testing and maintenance costs involves finding and fixing faults in code. Finding faults in today’s software systems can be challenging given their complex nature and provision of third-party plug-ins and extensions. In Section 2.3, we list several techniques in use today for finding intermittent faults in code. In this research, we have proposed a classification scheme and identified common attributes of intermittent failures that can be used by software engineers to identify such failures in their system based
on common patterns and characteristics. Additionally, we demonstrate that a prediction model based on metrics extracted from failure reports can be used with reasonable success in finding intermittent failures. Such models can be tailored to each system since metrics and common word occurrences differ for repositories of each software system.

We also show that conducting hardware configuration tests can be useful in finding intermittent failures. Section 4 explains an empirical study we conducted on intermittent Firefox failures, and found that altering processor speed, memory, and processor load can impact the frequency of occurrence or “observability” of intermittent failures in software.

8.2 Increasing reproducibility of intermittent failures

Developers cannot fix faults they cannot reproduce.

The first step towards improving the reproducibility of a software failure is to enable the production of high quality failure reports [52]. Helpful information such as the identified attributes of intermittent behavior in this research can be added to failure tracking systems. Additionally, information such as stack traces and steps to reproduce must be made part of the failure reports via automated crash reporting tools such as Windows Error Reporting and Breakpad\(^3\).

Another aspect that can help in reproducing software failures is communication among failure reporters and the investigating team. In our systems, we observed that some failure reports lacked any useful information and the investigation team could not begin identifying

\(^3\) http://code.google.com/p/google-breakpad/
the root cause of the failure until after having several back and forth communications with 
the reporter. The time difference between the locations of the reporter and the investigating 
team resulted in a prolonged delay in debugging the failure.

**Intermittent failures are normal software failures, and will repeat each time the**

**exact conditions occur.**

Getting to the bottom of an intermittent failure takes time and effort, but once the exact 
conditions are known, intermittent failures become normal reproducible failures. As we 
demonstrate in our hardware configuration study, many intermittent failures are timing-
related, and can be reproduced by altering configuration factors such as processor speed, 
memory capacity, and processor load. Common intermittent failures such as synchronization 
issues, race conditions, and other timing-dependent faults must be tested on slower systems 
and under increased load.

**Many minds make bugs shallow.**

What may be common knowledge to one person may not be common knowledge to 
another. Therefore, increased developer and tester collaboration is required when dealing 
with intermittent failures. Diversity of experiences of team members can help in investigating 
intermittent failures. During our grounded theory exercise, we observed that experienced 
testers offered greater insight into intermittent failures compared to other relatively
inexperienced testers on the team. A thorough knowledge of the inner workings of the system is, therefore, crucial when finding subtle intermittent faults in the system.

**Journey is its own reward.**

We showed in Section 6 that software engineers spend significantly more time debugging intermittent failures than reproducible failures. Such investigations, although costly (in terms of time), can lead to overall improvement of software quality if appropriate lessons are learned by software engineers once a fix is identified, and common mistakes that lead to such errors are avoided in the future. Also, the thorough debugging effort leads to an overall improved understanding of the intricate workings of the system that can prove to be useful in reproducing intermittent failures in the future.

**Order of testing may also be important.**

One failure in the industrial system occurred only when a certain order of installations was followed. For example, a user action was carried out on a specific version of the system that had a service pack installed as well as one major update to the service pack. The failure only occurred sometimes on systems where the user action was done before the installation of the major update to the service pack.

**Intermittent failures may point to a different problem in the system.**

A failure in the industrial system was reported on the latest build, but was reproduced on a much earlier build that led investigators to the root-cause of the problem, which existed
in the latest build as well. The fault, in essence, was lying dormant in the future builds of the system and would have occurred had the exact conditions for the fault been met.

Some more observations worth mentioning based on our experience of working with intermittent failures:

- Often, infrequently-used components of the system are not as thoroughly tested as others. Using operational profiling [73], such components can be identified and tested for the presence of intermittent failures.

- A good logging mechanism can prove to be useful when debugging intermittent failures. Identification of patterns in logs and traces can pinpoint the source of the intermittent fault.

- A common technique used by testers when finding intermittent failures is to add “hooks” and “traps” in the code. Such techniques, in addition to logs, can help get to the source of the intermittent fault.

- Using automated testing tools can help cover more testing scenarios, and also give testers better control over the manner of execution of test cases. For example, they may be able to control the timing of certain actions being performed in test cases via automated testing tools.
9. REFERENCES


disagreement or partial credit,” *Psychological bulletin*, vol. 70, no. 4, pp. 213–220, 1968.


testing and debugging, pp. 61-68, 2006.


APPENDICES
### Appendix A. Common Words Identified by NLP Techniques

#### A.1 ABB

<table>
<thead>
<tr>
<th>#</th>
<th>Keyword</th>
<th>#</th>
<th>Keyword</th>
<th>#</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sometimes</td>
<td>23</td>
<td>sporadic</td>
<td>45</td>
<td>server</td>
</tr>
<tr>
<td>2</td>
<td>timer</td>
<td>24</td>
<td>sporadically</td>
<td>46</td>
<td>validation</td>
</tr>
<tr>
<td>3</td>
<td>timing related</td>
<td>25</td>
<td>synchronization</td>
<td>47</td>
<td>alarm</td>
</tr>
<tr>
<td>4</td>
<td>race condition</td>
<td>26</td>
<td>repetition</td>
<td>48</td>
<td>system</td>
</tr>
<tr>
<td>5</td>
<td>deadlock</td>
<td>27</td>
<td>under observation</td>
<td>49</td>
<td>aspect</td>
</tr>
<tr>
<td>6</td>
<td>repeatable</td>
<td>28</td>
<td>multi core</td>
<td>50</td>
<td>plant explorer</td>
</tr>
<tr>
<td>7</td>
<td>reproducible</td>
<td>29</td>
<td>single core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>reproducible</td>
<td>30</td>
<td>dual core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>reproduce</td>
<td>31</td>
<td>bring out developer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>hard to reproduce</td>
<td>32</td>
<td>new tracker should be written</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>rare</td>
<td>33</td>
<td>if occurs again</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>intermittent</td>
<td>34</td>
<td>no one can repeat this problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>memory leakage</td>
<td>35</td>
<td>cannot repeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>high load</td>
<td>36</td>
<td>handle leak</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>time out</td>
<td>37</td>
<td>unable to repeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>memory</td>
<td>38</td>
<td>not able to reproduce</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>restart</td>
<td>39</td>
<td>not been able to reproduce</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>reboot</td>
<td>40</td>
<td>long time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>access violation</td>
<td>41</td>
<td>exceptional circumstance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>permission</td>
<td>42</td>
<td>trace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>unhandled exception</td>
<td>43</td>
<td>step</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>cannot reproduce</td>
<td>44</td>
<td>problem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## A.2 Mozilla Firefox

<table>
<thead>
<tr>
<th>#</th>
<th>Keyword</th>
<th>#</th>
<th>Keyword</th>
<th>#</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>background</td>
<td>23</td>
<td>dom</td>
<td>45</td>
<td>tabs</td>
</tr>
<tr>
<td>2</td>
<td>cpu</td>
<td>24</td>
<td>solution</td>
<td>46</td>
<td>windows</td>
</tr>
<tr>
<td>3</td>
<td>page</td>
<td>25</td>
<td>path</td>
<td>47</td>
<td>message</td>
</tr>
<tr>
<td>4</td>
<td>scrolling</td>
<td>26</td>
<td>ram</td>
<td>48</td>
<td>messages</td>
</tr>
<tr>
<td>5</td>
<td>tab</td>
<td>27</td>
<td>exe</td>
<td>49</td>
<td>bookmarks</td>
</tr>
<tr>
<td>6</td>
<td>window</td>
<td>28</td>
<td>dll</td>
<td>50</td>
<td>gecko</td>
</tr>
<tr>
<td>7</td>
<td>focus</td>
<td>29</td>
<td>cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>marked</td>
<td>30</td>
<td>crash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>clipboard</td>
<td>31</td>
<td>data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>malware</td>
<td>32</td>
<td>dialog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>branch</td>
<td>33</td>
<td>cpp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>script</td>
<td>34</td>
<td>trunk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>stack</td>
<td>35</td>
<td>registry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>extensions</td>
<td>36</td>
<td>temporary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>memory</td>
<td>37</td>
<td>code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>process</td>
<td>38</td>
<td>frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>profile</td>
<td>39</td>
<td>iframe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>running</td>
<td>40</td>
<td>patch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>task</td>
<td>41</td>
<td>system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>time</td>
<td>42</td>
<td>target</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>timer</td>
<td>43</td>
<td>file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>timers</td>
<td>44</td>
<td>build</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>