An Empirical Study of Runtime Files Attached to Crash Reports

Komal Panchal

A Thesis

in the Department of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Electrical and Computer Engineering)

at

Concordia University

Montreal, Quebec, Canada

January 2022

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Komal Panchal

Entitled: An Empirical Study of Runtime Files Attached to Crash Reports

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. D. Qiu________________________________ Chair

Dr. A. Youssef (CIISE)______________________ Examiner

Dr. D. Qiu________________________________ Examiner

Dr. Abdelwahab Hamou Lhadj_______________ Supervisor

Approved by: Dr. Jun Cai
Chair of Department or Graduate Program Director

____
Dr. Mourad Debbabi, Interim Dean,
Gina Cody School of Engineering & Computer Science

Abstract

An Empirical Study of Runtime Files Attached to Crash Reports

Komal Panchal

When a software system crashes, users report the crash using crash report tracking tools. A crash report (CR) is then routed to software developers for review to fix the problem. A CR contain a wealth of information that allow developers to diagnose the root causes of problems and provides fixes. This is particularly important at Ericsson, one of the world's largest Telecom company, in which this study was conducted. The handling of CRs at Ericsson goes through multiple lines of supports until a solution is provided. To make this possible, Ericsson software engineers and network operators rely on runtime data that is collected during the crash. This data is organized into files that are attached to the CRs. However, not all CRs contain this data in the first place. Software engineers and network operators often have to request additional files after the CR is created and sent to different Ericsson support lines, a problem that often delays the resolution process.

In this thesis, we conduct an empirical study of the runtime files attached to Ericsson CRs. We focus on answering four research questions that revolved around the proportion of runtime files in a selected set of CRs, the relationship between the severity of CRs and the type of files they contain, the impact of different file types on the time to fix the CR, and the possibility to predict whether a CR should have runtime data attached to it at the CR submission time. Our ultimate goal is to understand how runtime data is used during the CR handling process at Ericsson and what recommendations we can make to improve this process.

Acknowledgments

I would like to convey my heartfelt gratitude to Dr. Wahab Hamou-Lhadj who provided me with the incredible opportunity to conduct this research. I appreciate his faith in me, his constant support, keeping me motivated and the fact that he keeps his door open for constructive feedback and discussion.

I would like to express my gratitude to the entire team at Ericsson Global AI Accelerator (GAIA) in Montreal for their valuable input and contributions to this project. More particularly, I would like to acknowledge the great support I received from Alka Isac, Salman Memon, Zhongwen Zhu, and Pragash Krishnamoorth. I would also want to thank my Ericsson internship colleague Fatima Ait-Mahammed for her assistance with the implementation of my research work.

I would also want to express my gratitude to MITACS, Ericsson GAIA and Concordia University's Gina Cody School of Engineering and Computer Science for their financial support.

Many thanks to everyone in Professor Hamou-Lhadj's Lab for their friendship, inspiration, and thought-provoking conversations. I would also like to express my gratitude to Concordia University for their support and assistance, particularly the Concordia University Library, which provided a wealth of literature and guidance on report writing.

Words cannot express my gratitude to my parents and brother, who always supported my decisions from thousands of miles away and always encouraged me. A very special thanks to my friends for their loving support during my ups and downs of life, achievements, and setbacks. Without them, none of this would be possible.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. Problem and Motivation

Software systems are subject to crashes and failures during operation despite the effort spent on development and testing. Software companies use crash tracking systems to document system crashes from the time they occur to the time they are eventually solved. Examples of crash tracking systems include Bugzilla[1], Windows Error Reporting (WER)[2], and Apple Crash Reporter[3].

At Ericsson, the company in which this research study is conducted, the process of handling system crashes involves three lines of support:

- Level 1 – Network Operator Administrators: They are responsible for reconfiguring and restarting the system as well as collection detailed information that describes the crash.
- Level 2 – Network Operator Support Engineers: They are responsible of reviewing the crash reports, performing root cause analysis to understand the cause of the crash and providing a solution if the crash does not involve changes to the source code.

[1]https://www.bugzilla.org/
[2]https://docs.microsoft.com/en-us/windows/win32/wer/windows-error-reporting
[3]https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AnalyzingCrashReports/AnalyzingCrashReports.html

- Level 3 – Software Designers: They review the crashes, which may necessitate changes to the source code. They perform root cause analysis and provide a solution, usually in the form of system patches.

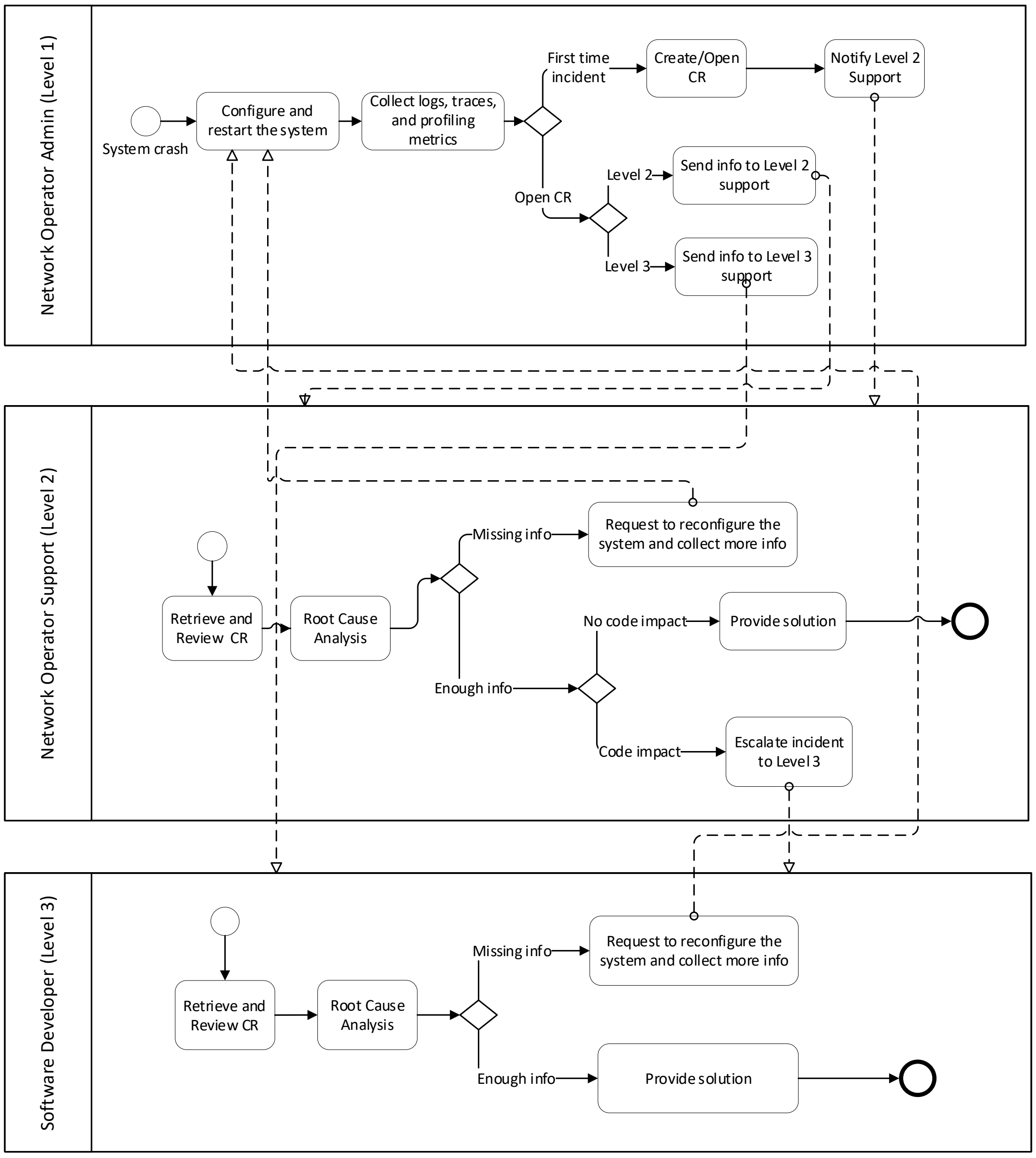
Network Operator Admin (Level 1)
System crash
Configure and restart the system
Collect logs, traces, and profiling metrics
First time incident
Create/Open CR
Notify Level 2 Support
Open CR
Level 2
Send info to Level 2 support
Level 3
Send info to Level 3 support
Network Operator Support (Level 2)
Retrieve and Review CR
Root Cause Analysis
Missing info
Request to reconfigure the system and collect more info
No code impact
Provide solution
Enough info
Code impact
Escalate incident to Level 3
Software Developer (Level 3)
Retrieve and Review CR
Root Cause Analysis
Missing info
Request to reconfigure the system and collect more info
Enough info
Provide solution

Figure 1.1. A typical process for handling crash reports at Ericsson

Figure 1.1 shows a typical crash report process flow at Ericsson. The process is triggered when a system crash occurs, at which point a Level 1 support operator creates a CR that contains information about the incident such as a description, the impacted system component, the severity of the crash, and runtime data (logs and traces) that are generated during the crash. The CR is saved in a CR database.

The Level 1 support operator sends a notification to a Level 2 support engineer who retrieves and reviews the CR and performs root cause analysis to identify the reasons behind the incident and potentially provides a solution. At this point, the Level 2 support engineer may realize that the CR does not contain sufficient information and sends a request back to Level 1 support to generate the missing information. This cycle might be repeated several times until the support engineer is satisfied with the detailed description of the incident that can help him/her to find the root cause and solve the problem.

If the solution requires modifications to the source code, the CR is forwarded to a Level 3 support line to be reviewed by a software designer. Similar to the previous situation, the software designer may engage in a cycle of interactions with the network support teams to obtain more detailed information about the incident that would help him or her uncover the root cause and provide a fix.

These repeated cycles of interactions between the various support lines often delay the provision of fixes, which may result in inefficiencies and high costs. Our discussion with Ericsson domain experts revealed that the main information that developers and network operators request consists of files that can provide insights into the execution of the system. These files are enclosed with CRs and used by developers and network operators to conduct root cause analysis.

Runtime data take typically the form of logs, execution traces, and profiling metrics. Software developers use this data to support various software engineering tasks including debugging, performance analysis, program comprehension, and performance analysis [28-32]. Additionally, there exist studies that rely on runtime data to improve the CR handling process. For example, Sabor et al. [1] introduced an approach to predict the severity of CRs using a combination of execution traces and CR categorical features [1]. Similar studies include the work of Koopaei et al. [20] where the authors proposed an approach based on Hidden Markov Models and execution traces to predict duplicate crash reports. Bettenburg et al. [33] observed that developers consider execution traces to be one of the most useful information that developers request when fixing a bug. Schroter et al. [34] went one step further by examining the use of traces in fixing bugs and showed that crash reports with traces are fixed sooner than those without. They also suggested to add explicit support to traces in bug/crash tracking systems. More studies are presented in the next chapter.

1.2. Problem Statement and Thesis Contributions

Existing studies focus mainly on execution traces, paying less attention to the role of logs and profiling metrics in reducing the overhead of processing CRs. In addition, most studies are conducted using open source systems, which are inherently less complex than industrial systems, especially those used in large organizations such as Ericsson. In this thesis, we conduct the first empirical study to understand the types of runtime files attached to CRs and their impact on the CR fixing time at Ericsson. We achieve this by examining a large number of CRs and study the attached files with the objective to answer the following research questions:

- RQ1: What is the proportion of each type of runtime files (i.e., traces, logs, profiling metrics, etc.) in our dataset?
- RQ2: What is the relationship between the severity of CRs and the type of files they contain?
- RQ3: What is the impact of different file types on the CR fixing time?
- RQ4: Can we predict if a CR should have a file attached to it before the CR is relayed to other lines of support?

Answering RQ1, RQ2, and RQ3 will help us understand the types of files that are attached with Ericsson CRs. This will, in turn, allow Ericsson domain experts to prioritize the type of files they need to collect during a crash. The answer to RQ4 will determine if it is possible to develop a recommendation system that can suggest to the first support line the files they should include when submitting a CR. The ultimate objective is to reduce the interaction time between the different lines of support, which should yield a faster CR resolution time.

The answers to these questions will also help us understand better the role of runtime files in the resolution of crashes from an industrial perspective, contributing further to the advancement of the field of software maintenance and evolution.

1.3. Thesis Outline

The rest of the thesis is structured as follows:

Chapter 2 – Background and Related work

This chapter describes the background and related work relevant to the thesis. The crash tracking system of Ericsson is introduced in the first section. We go over the various components of this system. Then, we discuss the steps for reporting and resolving the crash. In the second section, we review recent studies on handling crash reports, followed by a discussion.

Chapter 3 – Empirical Study of CR Enclosing Files

In this chapter, we show our approach for answering RQ1, RQ2, and R3 through an empirical study. We start by discussing the type of files that are attached to CRs. We continue by reporting on our findings and provide insights into the usefulness of the CR enclosing files in terms of the impact they have on the CR fixing time. We also examine the relationship between the CR enclosing files and the CR severity.

Chapter 4 – Prediction of CR Enclosing Files

In this chapter, we focus on RQ4 where we investigate if we can predict the need to attach a file while the CR is being created. To this end, we experiment with three machine learning algorithms, namely Support Vector Machine (SVM), Random Forest (RF), and K-Nearest Neighbor (KNN).

Chapter 5 – Conclusion and Future Work

In this chapter, we revisit the main contributions of this thesis. We conclude with comments about this study and present opportunities for future research.

Chapter 2. Background and Related Work

2.1. Crash Reporting at Ericsson

A crash reporting system is used to keep track of maintenance tasks [7]. It acts as a central repository for tracking the status of CRs, managing the information flow between the parties involved in handling the CR, and discussing the fixes. The crash reporting system used at Ericsson is an integrated Web-based environment for maintenance and customer support. It has many components to allow design, maintenance, and support teams to share information that facilitate the CR handling and analysis process [8][10].

A CR is described using mainly the following information:

- CR reference: This is a unique reference for the CR.
- CR severity: The severity of a CR, measured based on the impact of the fault on the system functionality. The severity can be "A", "B", or "C" with "A" referring to the most severe CRs. A CR of severity "A" is usually attributed to a crash that causes a complete failure of the system (e.g., stoppage of network traffic). The CR of severity "B" is assigned to CRs that cause important performance degradation of the system. CR of severity "C" is for minor faults and configuration with minimum impact on network traffic.

- CR Status: A CR goes through different stages from the time it is submitted to the time it is fixed. CR status includes “registered”, “assigned”, “cancelled”, “fixed”, etc.
- Product information: Several CR fields in Ericsson CR tracking system are used to describe the faulty product and version that is affected by the fault.
- History data: In this field, a track of the CR handling steps including the people who reviewed and processed the CR at various stage is kept for future reference.

In this study, we only consider CRs with the status "Finished". That is, we exclude ongoing CRs. It should also be mentioned that we exclude duplicate CRs from the dataset. Duplicate reports are marked as duplicate by developers.

2.2. Literature Review

Maiga et al. [10] conducted an empirical study using Ericsson data to better understand how internal and external CR are handled. Internal CRs refer to CR that are reported internally by testing teams, whereas external CRs refer to crashes that occur in operation. The study focused on the percentage of internal vs. external CRs, time taken to fix the CRs, and time to assign the CRs. To answer the research issues, they ran several statistical tests on the Ericsson CR dataset, which covered two years of maintenance activity. The authors showed that internal CRs account for 64% of the total reported CRs, while external CRs account for just 36%. The authors also showed that internal CRs take less time than external CRs using the Mann-Whitney test results. They attributed this to the fact that design teams have more understanding about the system than external CR reporters, and hence were able to provide more data that helped in solving the CRs. Finally, the authors showed that the severity of CRs did not affect CR fixing time or assignment time.

Zimmermann et al. [11] showed that the most prevalent issue that developers face when addressing CRs is incomplete information. By surveying developers and users, the authors synthesized the information that makes a good CR. According to their findings, the most useful information for developers to efficiently fix CRs is not something that is easy for CR submitters to provide. The authors went on creating the CUEZILLA recommendation tool to assess the quality of submitted CRs and make recommendations on how to enhance them. CUEZILLA automatically mines historical CRs related to bugs that are fixed successfully and use these as good examples for CR reporting.

Zhang et al. [9] assessed existing bug report analysis efforts in a survey study. Their main contribution is a classification of bug report analysis techniques that have been studies in recent years including optimization of bug reports, triaging of bug reports triage, bug-report misclassification reduction strategies, bug-report severity prediction techniques, bug localization, etc.

Bhattacharya et al. [12] conducted an empirical study to learn more about the bug fixing procedure in Android apps and platforms. Three primary research threads were investigated. First, the authors looked at the use of multiple metrics to analyze CRs and the bug fixing process. By calculating CR quality metrics, the authors discovered that most

Android app bug reports are of high quality, with lengthy textual descriptions, steps to reproduce errors, and explanations of expected output. A lengthy description in a CR usually indicates a high-quality bug report, which aids in the fast resolution of the bug. The authors also found that only 11% of the defects are fixed by the first assigned developers, and 75% of bugs take 2 to 13 developer tosses. In the second thread, the authors conducted a comparative study between Google Code and Bugzilla life cycles and argued that the introduction of Bugzilla's issue tracker and modification history into Google Code improved the bug fixing process. The last thread focused on a study of security flaws where the authors compared the quality of app vulnerabilities to the quality of non-security bugs. Also, the authors found that security bugs are resolved quicker than performance bugs.

An et al. [13] proposed an approach for mining information from CRs while highlighting the challenges of existing techniques. They looked at the problem of user identification of a CR. The authors also discussed the challenges of mapping crash reports to their corresponding bugs to understand the distribution of crash-related bugs in the user base. A similar study was conducted to understand the challenges of mapping CRs to bug fixes and to investigate whether developers efficiently addressed the reported field crashes by estimating the effort required to fix the crash.

Rastkar et al. [3] examined if current conversation-based automatic summarizers can be used to summarize CRs. They found that the quality of the generated CR summaries is comparable to summaries extracted from email threads and other conversations. The authors trained a summarizer on a CR corpus and showed that their model can be used to create summaries from CRs that can save time when analyzing CRs without sacrificing accuracy.

Karim [5] performed a research study to identify the important features of CRs that CR submitters often overlook. Then, he went on creating an automated key feature prediction model to recommend features to CR submitters based on historical reports. Using text classification techniques, he developed prediction models for each CR feature. The model performed well, achieving an F1-score of 70%.

Anbalagan et al. [14] conducted an empirical analysis of CRs from an open-source project to develop a model for predicting the project's corrective maintenance effort in terms of the time it takes to fix the bugs. This research was based on problem reports from nine Ubuntu versions. The author focused on user engagement in open-source project corrective maintenance and predicted the time it would take to fix bugs. The authors found that there a linear correlation between the number of users who report a defect and the median time it takes to fix it.

Chen et al. [15] proposed a method for automatically extracting bug entities and their relationships from CRs that combines Recurrent Neural Networks (RNN) with a dependency parser. This idea was shown to be effective in extracting bug entities and their relationships from issue reports in the studied bug repositories. The author first examined the categorization criteria for bug entities and their relationships before introducing the RNN models and dependency parser that were used to extract bug entities and interactions.

The CR description is preprocessed to extract words and then a baseline corpus is built to manually classify the types of bug entities and relations. The author used the Bi-LSTM-RNN model and dependency parser to predict the categories of entities and relations from CRs.

Sarkar et al. [16] developed an automated bug triaging tool tailored to the Ericsson bug tracking system. The author used a methodological valid time split evaluation where they sequentially train and test on large industrial data. Simple text and categorical features of CRs, combined with alarms and crash dumps are used for training a model.

Mahfoodh et al. [17] suggested a risk estimation method that can predict a quantifiable software risk value, allowing developers to improve software availability, security, and project management. This study used the number of bug-fixed samples for each priority category to determine the actual bug-fix time average, which was then combined with the last bug-fix time projected value to get the risk percentage of the bug-fix time prediction. This study used one crash report dataset to assess software components, yielding risk ratings ranging from 27.4 percent to 84 percent.

Xia et al. [18] looked at the impact of CR field reassignments on issue fixing experience. They contacted the developers of the subject software project to figure out the causes behind reassignment of CR fields. Three broad root causes were identified in the study: fresh CR correction, going through the procedure, and admin batch tasks. According to the researchers Around 80% of CRs have their field reassigned. Furthermore, the author concluded from the overall data that CRs with field reassignments took longer to fix than those without.

Early detection of duplicate crash reports can reduce the amount of time required to process the CRs. CrashAutomata, a model proposed by Koopaei et al. [20], detects duplicate CRs using automata. The authors created a representative model using historical CRs and a combination of n-grams of varying lengths and automata to depict stack traces. CrashAutomata produces very good precision and recall results for Firefox CRs, higher than comparative tools. In a follow-up study, Koopaei et al. [21] improved CrashAutomat by introducing the use of Hidden-Markov Models (HMMs). The new approach resulted in an increase of the true positive rate by 10% over CrashAutomata.

From the same research lab as Koopaei et al., Sabor et al. [2] suggested a feature extraction strategy that reduces feature size while retaining the most important categorization information. They used this approach for the automatic prediction of duplicate reports. The author used a trace abstraction strategy in this investigation, replacing stack traces of function calls with traces of packages.

Dhaliwal et al. [23] conducted an empirical study on CRs gathered for Mozilla Firefox to determine the influence of CR grouping and the features of an efficient grouping. The author proposed a grouping technique to minimize bug resolving time, where each group comprises CRs generated by just one type of problem. Bug fixing time was decreased by 5.3% because of this effective grouping. A two-level method to CR categorization was presented. To construct crash types, the first level of grouping groups CRs based on the method signature of stack traces. A comprehensive comparison of stack traces is performed at the second level to establish a sub-group. The similarity of the two CRs was determined by comparing the 10 first frames of the CRs stack traces. When numerous bugs are collectively related to the same crash type, the author determined that it takes longer to fix the issue than when the bug is uniquely linked to one or more crash types.

2.3. Summary

The analysis of crash/bug reports has been an active research topic for the last decade. Techniques that can reduce or predict the time it takes to fix crashes are needed to reduce the maintenance overhead. Existing techniques focus on various aspects of CRs. This thesis complements these techniques by examining how runtime files including execution traces, logs, and profiling metrics, are used in an industrial setting to reduce the time and effort of solving crashes. To our knowledge, this is the first time that a study that focuses solely on the importance on various runtime files in the processing of CRs from an industrial perspective is conducted.

Chapter 3. Empirical Study of Crash Report Attached Files

3.1. Overview

Our main goal in this thesis is to gain a better understanding of the runtime files attached to the CRs and their impact on the CR resolution process. This is particularly important in a large company such as Ericsson where the various development teams may be logging different things (network devices, nodes, software applications, performance metrics, etc.). Adding to this, it is known that the practice of logging and tracing lacks guidelines and common practices in software engineering [24].

With the help of domain experts working on a large product at Ericsson, we conducted an empirical study of runtime files attached to many CRs of an active Ericsson project with the objective to answer the following four research questions:

- RQ1: What is the proportion of each type of runtime files (i.e., traces, logs, profiling metrics, etc.) in our dataset?
- RQ2: What is the relationship between the severity of CRs and the type of files they contain?
- RQ3: What is the impact of different file types on the CR fixing time?

- RQ4: Can we predict if a CR should have a file attached to it before the CR is relayed to other lines of support?

3.2. Study Setup and Results

Figure 3.1 shows the process of collecting the data and addressing the research questions RQ1, RQ2, and RQ3. RQ4 is deferred to the next chapter. First, we extract data from the Ericsson CR database. We then mine the reports to extract the files enclosed with the CRs. The input of domain experts is needed throughout the process to help understand the data and interpret the results.

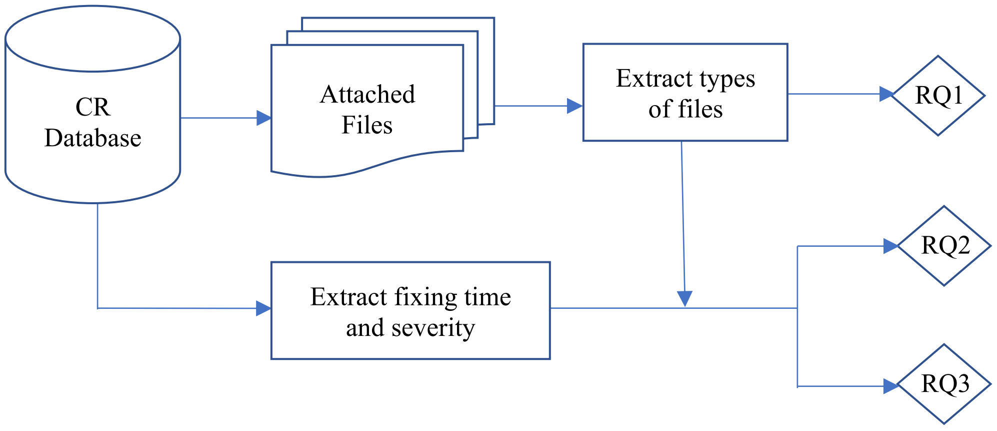

Figure 3.1. Data collection and analysis process

3.2.1. Dataset

We collected CRs of an active software product at Ericsson. The CRs cover almost two years of development. These CRs went through different lines of support until they were fixed. With the help of Ericsson experts, we wrote queries to extract CR information such as the CR heading, description, severity, the submitter, as well as the attached files. For each file, we retrieved the file name, the date at which the file was attached to the CR, the user who submitted the file, and the file size. Note that one CR may contain more than one file attached to it. In our dataset, the total number of files is five times larger than the number of CRs.

Ericsson developers use different types of files. For example, a file may contain performance indicators, which are profiling metrics collected at the time of the crash. Another file may contain execution information of a given process. Unfortunately, there is no mechanism that distinguishes between the different types of enclosed files in the CRs.

To address this, with the help of Ericsson experts, we manually analyzed the file attached to the CRs in our dataset. For each file, we reviewed the associated description. For some files, we had to open them and read their content. We also used the file name to infer information that may help us identify the type of runtime data it represents. Based on this, we developed a set of rules that can be automatically used to identify the type of files in other CRs or CRs of other systems. These rules use the file name, the CR description provided by the crash reporter in the CR, and e-mail threads (records of discussions that took place between the various lines of support). Some of these rules combine keywords extracted from all these sources. We find cases where file names alone are used as the main source for identifying the file type, which suggests that software developers at Ericsson use some sort of naming convention to label files. However, these conventions are not documented or enforced in the CR raising process, justifying the need for the rules that we have developed in the context of this project. These rules are implemented as regular expressions after validating them with Ericsson experts. We wrote a script that implements these regular expressions. The script takes the file description, file name, and email threads as input and return the file type as output. In the future, we intend to apply advanced data mining techniques to create rules that can generalize to CRs of other Ericsson systems. The rules are not included in this thesis for confidentiality reasons.

We identified five types of files, which we present here and discuss in more details in what follows:

- Node Dumps (NDs)
- Key Performance Indicators (KPIs)
- Execution Traces
- User-defined Logs
- Post-mortem Dumps (PMDs)

Type 1 – Node Dumps (NDs): At Ericsson, a ND file is required whenever a system crash is reported in a CR. A ND file contains the output of a set of commands that are executed on the node, providing us with a snapshot of the state of the node. This file is important for understanding what went wrong. Sometimes, the problem is due to a configuration issue. The content of a ND may help spot such issues by first lines of support, i.e., without having to relay to other lines of support.

Type 2 - Key Performance Indicators (KPIs): A KPI file provides information on the performance of the system through performance counters. For example, a KPI file may show the degradation of the network, which can help network operators identify the cause of the problem.

Type 3 - Execution traces: Execution traces are used to find a causal relationship between the system artifacts. Examples includes traces of function calls, inter-process communication traces, distributed traces, etc. Tracing requires instrumentation of the system, which consists of insertion of probes in places of interest. Using a trace, an analyst can replay the execution of the system to understand and diagnose the problem.

Type 4 - User-defined Logs: Complex telecom networking systems such as the ones developed at Ericsson are composed of many hardware and software platforms. It is common for developers to insert logging statements that would help them later to diagnose challenging problems. A logging statement typically contains a timestamp, a process id, a verbosity level, a logging function, a log message, and variables. Developers go through the log messages to debug the system and perform root cause analysis.

Type 5 - Post-mortem Dumps (PMDs): They contain whatever data is available in memory when a system crashes, including the processes that were executing at the moment of incident, the data exchanged between processes, etc. PMDs are rarely structured and may contain data related to multiple processes or even components.

3.2.2. RQ1: What is the proportion of each type of runtime files (i.e., traces, logs, profiling metrics, etc.) in our dataset?

Objectives: The answer to this question can help Ericsson's development teams understand the type of files that are used the most and that should be given priority. This is because generating these files infers an overhead and cost. Collecting all attached file types for each incident may turn out to be unproductive. This is particularly important in the context of runtime data analysis where one needs to develop parsers and analysis tools that are tailored towards a particular type of file.

Variables: For each file type $T_i \in \{ND, KPI, Log, Trace, PMD\}$, we use as variables the number of files of type T_i, the number of CRs that contain files of type T_i. We also need the total number of files and CRs to compute the ratios.

Method: To answer this research question, we use descriptive statistics. More precisely, we measure the ratio of the number of files of type T_i to the total number of files and the ratio of the number of CRs that contain logs of file type T_i to the total number of CRs.

Results:

Figure 3.2 shows the percentage of each file type in our dataset. As we can see, NDs are the most files that are collected. This is expected since NDs contain information about nodes after a crash occurs. It is also common to have multiple ND files for one crash report.

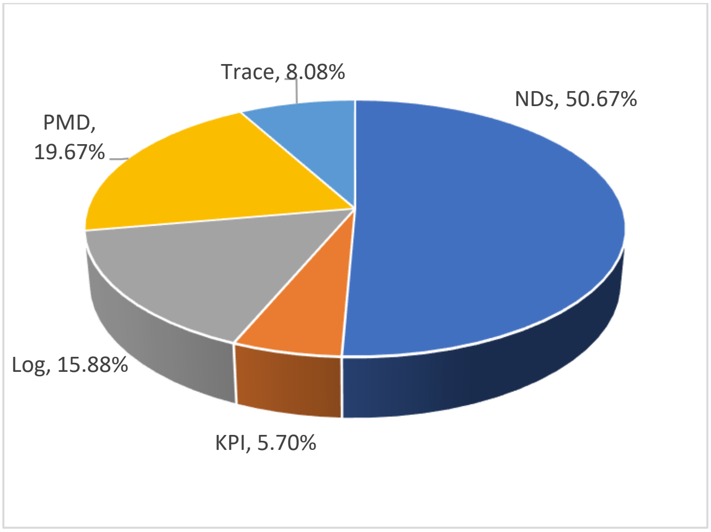

Figure 3.2. Percentage of file types in our dataset

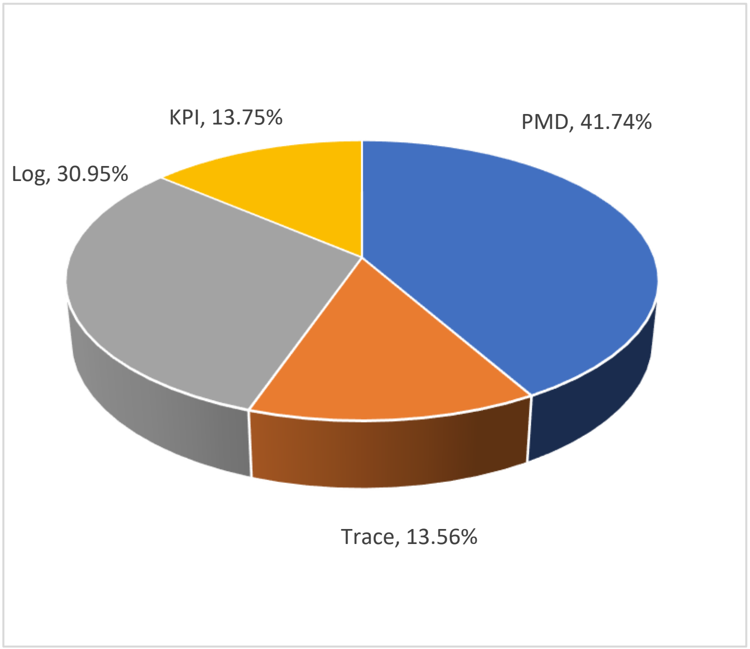

Figure 3.3. Percentage of CRs with respect to the enclosed file type

PMDs and Logs occupy the second and third position with 19.67% and 15.88% of all the file types. Traces and KPI files are used the least. This could be due to the fact that these two types of files incur an additional overhead when generating. For example, tracing requires an instrumentation tool and settings, which may deter users from generating this type of file. The same applies to KPI files that contain profiling metrics.

Figure 3.3 shows the percentage of CRs with respect to the type of files they contain. Note that we excluded Node Dumps from this analysis since Node Dumps are collected for every crash. The figure shows that PMDs and Logs are the most used files, which clearly supports the idea that traces and KPIs, which reveal a different perspective than PMDs and Logs, receive less attention in problem diagnosis.

3.2.3. RQ2. What is the relationship between the severity of CRs and the file types they contain?

In this research question, we examine if there is a relationship between the severity of the CRs and the type of files they contain. As mentioned in Chapter 2, the severity of a CR reflects the impact of the fault on the system functionality. The severity can be “A”, “B”, or “C” with “A” being the most severe CRs.

We want to establish whether the severity of a CR impacts the type of files that are attached to this CR. For this part, we state the following null hypothesis:

- H_{01}: The type of files that is attached to a CR is not dependant on the CR severity?

Variables: We use as variables the number of CRs with file type T_i and the severity level of the CRs (A, B, C). Note that we exclude node dumps from this analysis as well.

Method: To answer the research question, we build the contingency table with two qualitative variables, the CR severity (A, B, C) and the CR file type. The contingency table template is shown in Table 3.1. For example, in cell (A, KPI), we include the number of CRs that contain only files of type KPI (or KPI and node dumps since we are ignoring node dumps in this study). The last column shows the number of CRs that contain more than one of the file types.

Table 3.1. Template contingency table

Severity	**KPI**	**Log**	**Trace**	**PMD**	**More than one of these types**
A					
B					
C					

We use the Pearson's chi-squared independence test to accept or reject the null hypothesis H_{01}. The test is commonly used to examine the relationship between two independent qualitative variables, which are in our study, the CR severity and the CR attached file type. The result is considered statistically significant at alpha = 0.05. If p-value < 0.05 then we reject the null hypothesis H_{01} and conclude that the severity of the CRs and the types of files attached to the CRs are dependant.

Results:

Table 3.2. Contingency Table for RQ2

Severity	KPI	Log	Trace	PMD	More than one of these types
A	3.57%	37.50%	7.14%	32.14%	19.64%
B	7.78%	14.39%	7.34%	41.56%	28.93%
C	19.64%	21.43%	5.36%	19.64%	33.93%

Table 3.2 shows the percentage of each file type with respect to severity. Note that we show the percentages instead of the real values for confidentiality reasons. To compute the Chi-square test, we used the real value and not the percentages. We found that the p-value is < 0.00001. The result is significant at p-value < 0.05. Therefore, we reject the null hypothesis H_{01} and conclude that there is a relationship between the CR severity and the file types.

Figure 3.3 shows the distribution of the file types based on the CR severity. The data varies depending on the severity. The figure shows that the most severe CRs (CR with severity "A") have logs as the most important files. This may be explained by the fact that logs contain information introduced by developers and therefore can be very useful in debugging these systems. It should be noted that CRs with severity "A" are almost always sent to developers to provide fixes. CRs of severity "B" rely on PMDs and more than one type to help diagnose the problems. The least severe CRs contain a combination of all file types. We found that in all CRs, traces seem to receive the least attention.

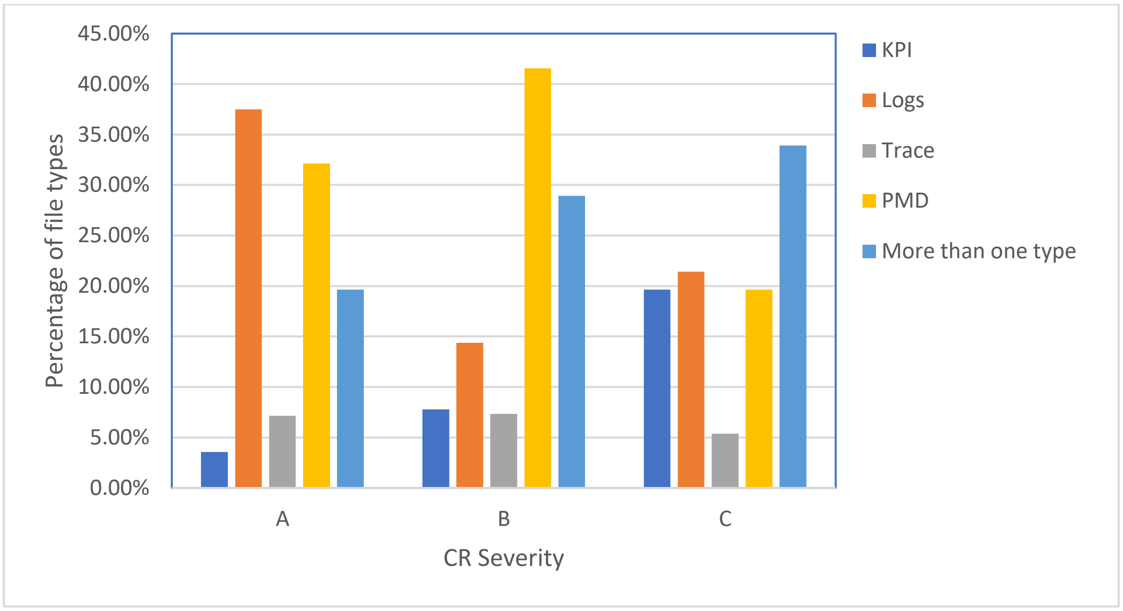

Figure 3.4. Percentage of file types attached to CRs with respect to CR severity

3.2.4. RQ3: What is the impact of file types on the time to fix the CRs?

Objective: For this question, we analyze the impact of file types on the time it takes to solve the CR. The fixing time of a CR is measured in days. The answer to this question is useful to understand whether including a certain file type would improve the resolution process of CRs. For example, knowing that a CR that contains a PMD takes less time than a CR that does not have a PMD would encourage Ericsson designers to collect PMDs if they are available when a crash occurs.

To answer this question, we run a statistical test for each file type $T_i \in$ {KPI, Log, Trace, PMD}. We state the following null hypothesis:

H_{02}: There is no statistically significant difference between the fixing time of CRs with file type T_i and that of CRs that do not have files of type T_i.

Variables: We use as independent variables the fixing time of a CR, which is in days and the file type $T_i \in \{ND, KPI, Log, Trace, PMD\}$. Unlike the other two questions, we do include node dumps in this question. In other words, we want also to know if the presence of a node dump file impacts the fixing time.

Method: We compute the non-parametric Mann-Whitney test to compare the CR fixing time with respect to the file type T_i attached to the CR and analyze. We use the Mann-Whitney test because we cannot assume that the data follows a normal distribution. The result of the test is considered as statistically significant at alpha = 0.05. Therefore, if p-value < 0.05, we reject the null hypothesis H_{02} and conclude that the fixing time of CRs with file type T_i is significantly different from the fixing time of CRs without file type T_i.

Results:

Table 3.3 shows the results of the statistical tests. As we can see from the table, there is a statistically significant difference between CRs that contain Node Dumps and those that do not in terms of the fixing time. The same applies to KPI, Log, and Trace file types. The corresponding p-values are less than 0.05. However, we did not find a statistical difference between CRs with PMDs and those that do not.

Table 3.3 Results of statistical analysis on the impact of file types on CR fixing time

Log Type	**p-value**	**H_0**
Node Dumps	$0.006273 < 0.05$	Reject
KPI	$0.000293 < 0.05$	Reject
Log	$8.761203e-05 < 0. 05$	Reject
Trace	$0.0102392 < 0.05$	Reject
PMD	$0.6895507 > 0.05$	Do not reject

In other words, the fact that a CR contains a PMD does not necessarily result in a faster resolution. This may be due to the fact that PMDs are memory dumps and may contain a large amount of unstructured data related to different parts of the system. Interpreting this data is usually a challenging task. Ironically, in RQ1, we found that PMDs are the most collected runtime data. Based on this finding, we suggest to review the relevance of PMDs for fixing CRs for improved efficiency and cost saving. The current practice of collecting PMDs whenever a crash occurs may not be productive.

Chapter 4. Prediction of CR files

In this chapter, we answer the fourth research question that was introduced in the previous chapter, which consists of:

- RQ4. Can we predict if a CR should have a file attached to it before the CR is relayed to other lines of support?

Predicting whether additional files need to be attached during the creation of a CR is an important step towards reducing the time and effort it takes to solve the CR. We can implement a recommendation system that recommends to the CR reporter whether a file should be attached or not. This will reduce the number of interactions between the various lines of supports.

4.1. Approach

To answer this question, we use machine learning techniques. Our approach uses a two-phase process: Training and validating the models (shown in Figure 4.1). We build a training model that learns from past CRs that can later be used in the second phase to predict the inclusion of attached files (the inference phase). These two phases are discussed in more details in what follows.

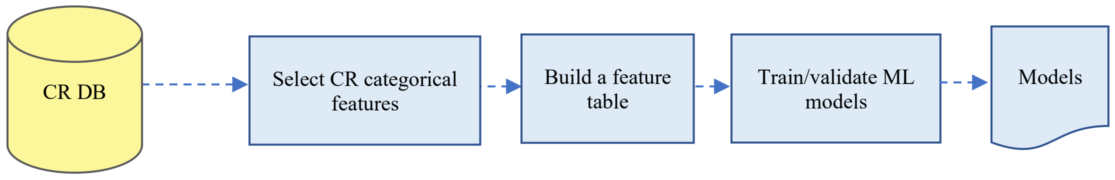

Figure 4.1. Approach used to answer RQ4

4.2. Feature Extraction

With the help of Ericsson domain experts, we selected the following features of crash reports to use for classification. Some features contain categorical values. We used one-hot encoder [38] to convert these features to numerical values.

- Submitter Priority: It is the severity of the defect and the CR's priority (i.e., A, B, or C).
- Priority Rate: This is an internal priority assigned to the CR.
- Observation Fault: It represents the fault type.
- Faulty Product Design Responsible Office: This field represents the team within Ericsson that is responsible for the defective product. This field changes as the CR is reassigned over the course of its life.
- Node Level Name: The node level product name.
- Node Level Product Number: The product number at the node level that is affected by the failure.
- Faulty product: This fields refers to the faulty product.
- Product affected system area: This field represents the functionality that is affected by the failure.

4.3. Training and testing

We use three machine learning algorithms to predict whether a CR should have a file attached to it. These are Support Vector Machine (SVM) [35], Random Forest (RF) [37], and K-Nearest Neighbor (KNN) [36].

SVM is a classification algorithm that uses a hyperplane to differentiate different classes of examples in high-dimension space. KNN is a lazy learning algorithm based on instances. KNN returns the K most similar instances to a feature vector when given one. As a result, the method provides the K closest relevant instances based on the value of (K), which is a fixed variable that defines the number of returned neighbours. RF is a machine learning technique based on the decision tree algorithm. The model was created using a logic-based approach.

We use ten-fold cross-validation to test the model. This method randomly shuffles the dataset and divides it into 10 folds. The training is done using 90% of the data, which results in a model that is later tested with the remaining 10%. The approach is repeated 10 times by choosing different folds each time. The accuracy of the classification algorithm is the average accuracy obtained using the 10-fold cross-validation.

4.4. Evaluation Metrics

We used precision, recall, and F1-score to assess the performance of the algorithms. These metrics are used extensively in the literature, which are calculated as follows:

- Precision = TP/(TP + FP).
- Recall = TP/(TP + FN).
- F1_score = 2*Precision*Recall / (Precision + Recall)
- Accuracy = (TP+TN)/(TP+FP+FN+TN)

Where TP (True Positive) is the number of CRs that are classified properly. FP (False Positive) represents the number of CR for which the algorithm wrongly predicted that an attachment is needed. FN (False Negative) represents the number of CRs that have file attached files, but the algorithm missed. TN (True Negative) represents the number of CRs that do not have files attached to them and that the algorithm predicted properly.

Table 4.1. Prediction of CR runtime file attachments

	SVM	RF	KNN
Precision:	75%	81%	76.8%
Recall:	88.5%	87.5%	88%
F1_Score:	81%	84%	82%
Accuracy:	70.3%	76%	72%

4.5. Results

Table 4.1 shows the results of applying SVM, RF, and KNN. All algorithms perform relatively well. However, Random Forest yields best results with an F1-score of 84%. The precision and recall are 81% and 87.5%, respectively. Our comparative study of the different classifications techniques reveals that RF outperforms the other techniques when predicting the presence of files in CRs. These results are very promising and suggest that we can develop a recommendation system that predict if a file should be attached to a CR, which may reduce the number of interactions between the various lines of support.

Chapter 5. Conclusion and Future Work

5.1. Research Contributions

In this thesis, we conducted an empirical study on the runtime files attached to Ericsson CRs. We used a dataset of CRs covering two years of development. We show that there are five type of files that are used to diagnose the problem that led to system failures. These are: Node Dumps, Logs, KPIs, traces, and PMDs. In addition to Node Dumps, we found that logs and PMD files are used the most. We also found that all file types have an impact on the CR fixing time except PMDs.

Additionally, we found that despite their usefulness, traces and KPIs are used the least. We conjectured that this could be due to the fact that these types of runtime data require an overhead when generating them.

Further, we investigated whether we can predict if a CR should contain a file during the creation of the CR. This is important for reducing the time it takes to fix the CR. We used three machine learning algorithms – SVM, KNN and RF – and found that RF performs the best with an F1-score of 84%.

5.2. Opportunities for Further Research

Future work should focus on three main directions:

We should investigate whether we can predict the type of file that should be included with the CR. By doing so, we can build a comprehensive tool that not only predicts the presence or absence of a file but also the type needed. We can use other features of the CRs such the CR description and headings, combined with multi-class classification algorithms for this purpose.

Another future direction is to apply this research to predict other important features of a CR such as the steps to reproduce the CR. We found that many CRs that we analyzed did not have these steps despite the fact many research studies showed that steps to reproduce are the most needed by developers

Furthermore, we need to conduct a user study with Ericsson domain experts to understand the impact of this research. More particularly, we are interested in a qualitative analysis that can reveal more on the way Ericsson uses traces, KPIs, logs and PMDs. We also would like to investigate if one type of files may be suitable for a certain category of crashes more than another one.

Finally, the application of more advanced machine learning methods is another prospective direction for advancement. For our research, we used supervised machine learning algorithms. More advanced machine learning methods, such as deep learning methods, could be used in the future to increase the prediction accuracy of the current methodology. We can also calculate how much time can be saved by using our method, and use this as one of our criteria for assessing the effectiveness of a recommendation system for file prediction. This will make the crash report more thorough, and the developer will request less information throughout the CR handling process, resulting in a more effective and efficient CR handling process.

Bibliography

[1]. K. K. Sabor, M. Hamdaqa, and A. Hamou-Lhadj, “Automatic prediction of the severity of bugs using stack traces and categorical features,” Information and Software Technology, vol. 123, 2020.

[2]. K. K. Sabor, A. Hamou-Lhadj, and A. Larsson, “DURFEX: A feature extraction technique for efficient detection of duplicate bug reports,” in Proc. of the 2017 IEEE International Conference on Software Quality and Reliability (QRS'17), 2017 pp. 240–250.

[3]. S. Rastkar, G. C. Murphy, and G. Murray, “Automatic summarization of bug reports,” IEEE Transactions on Software Engineering, vol. 40, no. 4, pp. 366–380, 2014.

[4]. C. Weiss, R. Premraj, T. Zimmermann and A. Zeller, "How Long Will It Take to Fix This Bug?," in Proc. of the 4th International Workshop on Mining Software Repositories (MSR'07), 2007.

[5]. M. R. Karim, "Key Features Recommendation to Improve Bug Reporting," in Proc. of the International Conference on Software and System Processes (ICSSP), 2019, pp. 1-4.

[6]. O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in Proc. of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17), 2017, pp. 396-407.

[7]. T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, "Improving bug tracking systems," in Proc. of the 31st International Conference on Software Engineering - Companion Volume, 2009, pp. 247-250.

[8]. L. Hribar and D. Duka, "Reporting and removing faults in telecommunication software," in Proc. of the 34th International Convention MIPRO, 2011, pp. 593-599.

[9]. J. Zhang, X. Y. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, “A survey on bug-report analysis,” Science China Information Sciences, vol. 58, no. 2, pp. 1–24, 2015.

[10]. A. Maiga, A. Hamou-Lhadj, M. Nayrolles, K. K. Sabor and A. Larsson, "An empirical study on the handling of crash reports in a large software company: An experience report," in Proc. of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 342-351.

[11]. T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter and C. Weiss, "What Makes a Good Bug Report?," Transactions on Software Engineering, vol. 36, no. 5, pp. 618-643, 2010.

[12]. P. Bhattacharya, L. Ulanova, I. Neamtiu and S. C. Koduru, "An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps," in Proc. of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13), 2013, pp. 133-143.

[13]. L. An and F. Khomh, "Challenges and Issues of Mining Crash Reports," in Proc. of the IEEE 1st International Workshop on Software Analytics (SWAN), 2015, pp. 5-8.

[14]. P. Anbalagan and M. Vouk, “On predicting the time taken to correct bug reports in open source projects,” in Proc. of the IEEE 25th International Conference on Software Maintenance (ICSM'09), 2009, pp. 523–526.

[15]. D. Chen, B. Li, C. Zhou and X. Zhu, "Automatically Identifying Bug Entities and Relations for Bug Analysis," 2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF), 2019, pp. 39-43.

[16]. A. Sarkar, P. C. Rigby and B. Bartalos, "Improving Bug Triaging with High Confidence Predictions at Ericsson," 2019 IEEE 25th International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 81-91.

[17]. H. Mahfoodh and Q. Obediat, "Software Risk Estimation Through Bug Reports Analysis and Bug-fix Time Predictions," in Proc. of the International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT), 2020, pp. 1-6.

[18]. X. Xia, D. Lo, M. Wen, E. Shihab and B. Zhou, "An empirical study of bug report field reassignment," in Proc, of the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 174-183.

[19]. P. Ramarao, K. Muthukumaran, S. Dash and N. L. Bhanu Murthy, "Impact of Bug Reporter's Reputation on Bug-Fix Times," in Proc. of the International Conference on Information Systems Engineering (ICISE), 2016, pp. 57-61.

[20]. N. E. Koopaei, A. Hamou-Lhadj, “CrashAutomata: An Approach for the Detection of

Duplicate Crash Reports Based on Generalizable Automata," in Proc. of the 25th Annual International Conference on Computer Science and Software Engineering (CASCON'15), 2015, pp. 201–210.

[21]. N. E. Koopaei, S. Islam, A. Hamou-lhadj, and M. Hamdaqua, "An Effective Method for Detecting Duplicate Crash Reports Using Crash Traces and Hidden Markov Models," in Proc. of the 26th nnual International Conference on Computer Science and Software Engineering (CASCON'16), pp. 75–84, 2016.

[22]. Y. Zhai, W. Song, X. Liu, L. Liu and X. Zhao, "A Chi-Square Statistics Based Feature Selection Method in Text Classification," in Proc. of the 9th International Conference on Software Engineering and Service Science (ICSESS), 2018, pp. 160-163.

[23]. T. Dhaliwal, F. Khomh and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox," in Proc. of the 27th IEEE International Conference on Software Maintenance (ICSM'11), 2011, pp. 333-342.

[24]. K. H. Patel, "The Sense of Logging in the Linux Kernel," Master's thesis, Concordia University, 2020.

[25]. K. K. Sabor, A. Hamou-Lhadj, A. Trabelsi, J. Hassine, "Predicting bug report fields using stack traces and categorical attributes," in Proc. of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON'19), 2019, pp. 224-233.

[26]. D. El-Masri, F. Petrillo, Y-G. Guéhéneuc, A. Hamou-Lhadj, A. Bouziane, "A Systematic

Literature Review on Automated Log Abstraction Techniques," Information and Software Technology (IST), vol. 122, 2020.

[27]. A. V. Miranskyy, A. Hamou-Lhadj, E. Cialini, A. Larsson, "Operational-Log Analysis for Big Data Systems: Challenges and Solutions," IEEE Software, vol. 33, no. 2, pp. 52-59, 2016.

[28]. S. He, Q. Lin, J-G. Lou, H. Zhang, M. R. Lyu, D. Zhang, "Identifying impactful service system problems via log analysis," in Proc. of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18), 2018, pp. 60–70.

[29]. A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, S. Bhattacharya, "Anomaly Detection Using Program Control Flow Graph Mining from Execution Logs," in Proc. of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16), 2016, pp. 215–224.

[30]. Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J-G. Lou, C. Li, Y. Wu, R. Yao, M. Chintalapati, D. Zhang, "Predicting Node failure in cloud service systems," in Proc. of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18), 2018, pp. 480–490.

[31]. A. Hamou-Lhadj, T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System," in Proc. of the International

Conference on Program Comprehension (ICPC'06), 2006, pp. 181-190.

[32]. N. Ezzati-Jivan, Q. Fournier, M. R. Dagenais and A. Hamou-Lhadj, "DepGraph: Localizing Performance Bottlenecks in Multi-Core Applications Using Waiting Dependency Graphs and Software Tracing," in Proc. of the IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 149-159.

[33]. N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16), 2008, pp. 308-318.

[34]. A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in Proc. of 7th IEEE Working Conference on the Mining Software Repositories (MSR), 2010, pp. 118-121.

[35]. I. Steinwart, A. and Christmann (2008). Support Vector Machines. 1st edition, New York: Springer (Information science and statistics). doi: 10.1007/978-0-387-77242-4.

[36]. P. Cunningham, and S. J. Delany, S. J. (2021) "K-Nearest Neighbour Classifiers - a Tutorial," ACM Computing Surveys (CSUR), 54(6), 2021, pp. 1–25.

[37]. R. Genuer, and J.-M. Poggi. Random Forests with R. Cham: Springer Use R! Book Series, 2020.

[38]. F. Pedregosa et al., "Scikit-Learn: Machine Learning in Python," Journal of Machine

Learning Research, Volume 12, 2011, pp. 2825–2830.

[39]. K. Jolly. Machine learning with scikit-learn quick start guide: classification, regression. and clustering techniques in python. Packt Publishing, Birmingham, UK, 2018.