Analysing the Resolution of Security Bugs in Software Maintenance

Saad Bin Saleem
BS(Hons) and MS

Department of Computing and Communications
The Open University

A thesis submitted for the degree of
Doctor of Philosophy

June 2014

Abstract

Security bugs in software systems are often reported after incidents of malicious attacks. Developers often need to resolve these bugs quickly in order to maintain the security of such systems. Bug resolution includes two kinds of activities: *triaging* confirms that the bugs are indeed security problems, after which *fixing* involves making changes to the code.

It is reported in the literature that, statistically, security bugs are reopened more often compared to others, which poses two new research questions: (a) Are developers *"rushing"* to triage security bugs too soon under the pressure of deadlines? (b) Do developers need to spend more time fixing security bugs to avoid frequent reopening?

This thesis explores these questions in order to determine whether security bug fixing should take a higher priority than other bugs to avoid malicious attackers *exploiting* vulnerabilities before the problems are fixed, and whether security bug fixing should take a higher priority than other bugs.

In this thesis a quantitative approach has been adopted by conducting statistical empirical studies to observe the behaviour of software developers engaged in dealing with security bugs.

Firstly, the concept of "rush" has been borrowed from the time management literature to refer to the behaviour of people delivering work under the pressure of deadlines. By observing how developers deliver bug resolution before the deadline of releases,the degree of rush has been measured as the ratio between the actual time spent by developers during triaging and the theo-

retical time the developers have by delaying the fixes until the next regular *release*.

In this thesis, a suggest that delaying bug assignment helps find the right developer and gives the developer more time to prepare for the same workload with more relaxed planning constraints. Secondly, to analyse the complexity of security bug fixes, the fan-in complexity of functions relevant to security bugs has been measured, rather than simply measuring the time spent by the software developers on the fixing of such bugs.

The first null hypothesis is tested using a *Man-Whitney* method on five software case studies, *Samba*, *Mozilla Firefox*, *Red Hat*, *FreeBSD* and *Mozilla*. The second null hypothesis is tested by comparing the results of fixing security and non-security bugs from the *Samba* and *Mozilla Firefox* case studies. Statistically significant results suggest that security bugs are triaged in a rush compared to non-security bugs for *Red Hat*, *FreeBSD* and *Mozilla*. In terms of fan-in, the results of the *Samba* and *Mozilla Firefox* case studies suggest that security bugs are more complex to fix compared to non-security bugs.

Acknowledgements

I want to thank my supervisors Yijun Yu, Bashar Nuseibeh for their continuous support, encouragement, feedback and invaluable guidance in finishing this thesis.

I am especially grateful to my supervisor, Yijun Yu, for his support in making this work technically sound, and Bashar Nuseibeh for giving high level comments and support to complete the thesis. I also want to thank Charles Haley for his valuable reviews and comments at the early stage of my research and Thein Tun for his reviews and occasional discussions relevant to my research. To, Guenther Ruhe for discussions on the concept of planning releases. To Arosha Bandra and Marian Petre for their moral support and ideas to improve my work.

I would like to thank my examiners Professor Haris Mouratidis and Professor Andrea Zisman for their constructive feedback to improve my work.

Special thanks goes to my mother, Nusrat Saleem, and my brothers Kashif Saleem and Arslan Saleem for their continuous encouragement and support throughout this difficult time. I cannot forget to acknowledge my late father, Rana Muhammad Saleem, for encouraging and inspiring me to get a higher education.

I am grateful to my fellow postgraduate students at the Department of Computing and Communications and friends within and outside the Open University who were always available to help.

Finally, I am grateful to the Department of Computing and Communications

at The Open University. It would have been impossible to complete this thesis without their continuous help and support.

Publications

All the work in this thesis describes original contributions of the author.

- Svahnberg, Mikael; Gorschek, Tony; Feldt, Robert; Torkar, Richard; Saleem, Saad Bin; and Shafique, Muhammad Usman (2010). A systematic review on strategic release planning models. In: *The Journal of Information and Software Technology, vol. 52, no. 3, pp. 237-248*. A measurement metric for rush inspired by the release planning literature is detail in Chapter 3.
- Saleem, Saad Bin; Montrieux, Lionel; Yu, Yijun; Tun, Thein and Nuseibeh, Bashar (2013). Maintaining security requirements of software systems using evolving crosscutting dependencies. In: *Chitchyan, Ruzanna; Moreira, Ana; Araujo, Joao and Rashid, Awais eds. Aspect Oriented Requirements Engineering, pp. 167-181*, Springer. An improved version of the proposed measurement is detailed in Chapter 4.

List of Figures

List of Tables

Chapter 1

Introduction

In the year 2013, on average 13 vulnerabilities were reported every day out of the annual total of 4,794 records in the National Vulnerability Database (NVD) [1]. Statistics show that the number of reported vulnerabilities in year 2013 was higher than the last five years (Florian, 2014). Most of these reported vulnerabilities have been exploited by malicious attackers (Nagaraju et al., 2013), which caused the loss of millions of dollars and damaged the reputation of vendors. Not only is the exploitation of reported vulnerabilities growing, but recent trends show an increase in the exploitation of unknown or zero-day vulnerabilities as well (Donohue, 2014). The exploitation of such types of vulnerability is even worse for the reputation of the vendors because the zero-day vulnerability attacks not only cause security failure in the software system but also show the vendors' lack of awareness about the security flaws in their system.

For example, credit and debit cards worth $130 million dollars were compromised in a security breach of *Heartland Payment Systems Inc.* The company has spent a total of $139.4 million dollars dealing with this security breach issue (Vijayan, 2010). The cause of such a massive data breach was a bug in the code of a web form which allowed access to the company's corporate

[1] http://nvd.nist.gov/

network (Cheney, 2010).

In another security breach, a flaw was found in the release 1.7 of the Java Virtual Machine (JVM) software owned by the Oracle Inc. This security flaw started a debate in the news media about the security standards' of Oracle (Finkle, 2013). During the investigation of the attack, security experts from the US Department of Homeland Security explained that hackers had exploited a bug in a version of Java using Internet Explorer to install malicious software.

In general, a bug is defined as a problem in the software specification, design and code or unexpected behaviour of the software program which leads to the failure or improper functionality of the software system (Grubb and Takang, 2003). Specifically, a bug is referred to as *security related* when it creates vulnerability in the software, which the malicious attackers could exploit to attack the system (Viega and McGraw, 2011).

For example, the malicious attackers can exploit a buffer overflow bug in which a program tries to store more data in a buffer (temporary data storage area or random access memory) than its intended range (Margaret, 2007). As the result of buffer overflow, the extra data of a program corrupts or overwrite data into the adjacent buffers. This extra data may contain code designed to do specific actions or send instructions to attack the software system. Resulting in damage to data or disclosing the confidential information. The buffer overflow bug is one of the common causes of security vulnerabilities relevant to random access memory. Usually, such vulnerabilities lead to the data integrity breaches.

It was reported by the institute of Security Leadership Essentials for Managers (SANS) that in 2008 just two bugs caused half a million security breaches [2]. The growing exploitation of system vulnerabilities due to bugs indicates the failure of software system security at two levels. The first source of failure is overlooking at the security requirements of the system up-front,

[2] http://www.sans.org/

with no or bad security planning of the system by designers and the ignorance of programmers on security aspects of the system while writing the code. The second source of the problem is giving insufficient attention to maintaining the security of the system by resolving security bugs.
As a matter of fact all the security relevant bugs (those which can cause potential vulnerability in the system) are difficult to detect at the testing stage (Pfleeger and Pfleeger, 2006). Therefore, software developers not only need to hunt for the security bugs (Arce, 2002) but also need to ensure the proper resolution of such bugs to protect the system from malicious attackers. For this reason, the focus of this thesis is to study about the resolution of security bugs.
The gaps relevant to security bugs has been identified in Chapter 2 literature review. In the following subsections, the key contribution of this thesis, the research methodology and the structure of the thesis are discussed.

1.1 Contributions

This thesis provides a novel contribution concerning the resolution of security bugs. Before explaining each contribution of the thesis, below is the definition of terms used.
In this thesis, *triaging* refers to a process to priorities and assign bugs to developers for fixing during their resolution. Throughout this thesis, "rush" is refered to the time management behaviour of software developers on the triaging of bugs. The "fan-in" is an already established concept refers to the number of times a function is invoked by the other functions. The Common Vulnerability and Exposure (CVE) bugs are used as the confirmed security bugs in this thesis.
The following are the contributions of this thesis.

- A technique to measure the degree of "rush" (the time management behaviour of software developers) on the *triaging* of security and the

other types of bug.

According to the knowledge of the author of this thesis, this thesis is the first to introduce the notion of "rush" in the triaging of bugs. The concept of "rush" is borrowed from the time management literature for referring to the behaviour of software developers delivering work under the pressure of deadlines. Originally, "rush" is a concept in the time management literature refers to the behaviour of people delivering work under the pressure of deadlines. During the triaging of bugs, it is challenging for the developers to priorities the fixing of bugs given the resource constraints and release deadlines. Especially in the case of security bugs, it is very important to priorities the fixing of high priority bugs and still deliver the fix on schedule. The proposed technique to measure "rush" in the triaging of bugs is developed keeping this concern in mind. Therefore, the author of this thesis claims that the proposed technique to measure "rush" in the triaging of bugs is unique and new.

- A technique to measure the complexity of functions relevant to security and the other types of bug fixes using the "fan-in" metric.

The fan-in is an already established concept, however according to the knowledge of the author of this thesis. This thesis is the first to apply the fan-in concept for developing a technique to measure the complexity of functions relevant to bug fixes.

- An empirical study to measure "rush" (the time management behaviour of software developers) on the triaging of security bugs and to measure the complexity of functions relevant to bug fixes.

In this thesis, the demonstration of applying the techniques to measure rush and the complexity of functions relevant to bug fixes to five case studies (*Samba*, *Mozilla Firefox*,*Red Hat*, *FreeBSD* and *Mozilla*) is unique. Especially, this thesis is the first to use *Samba* case study for analysing the activities of bug resolution.

1.2 Research Methodology

In this thesis, the research is organised in the following stages as shown in Figure 1.1: (a) search bug fixing literature, (b) identify gaps in the literature studies, (c) classify the gaps into groups, (d) Formulate the research questions to fill the gaps, and (e) validate that the research questions has been addressed and (f) validate that the gaps has been filled by addressing the research questions.

1.2.1 Search security bug's resolution literature

In the literature search phase, the terms such as "security bug resolution", "resolving security bugs", "fixing security bugs" and "security bugs", etc. has been provided to the literature databases. Mainly, the "Google Scholar" and "Computer Science Bibliography (DBLP)" has been used to search for the relevant studies. To include/exclude a study in the list of studies relevant to the resolution of security bugs, the author of this thesis has manually scanned the "title", "abstract", "introduction" and "conclusion" of each study. Although, the author has also skimmed through the studies in some cases when it was not straight forward to decide whether the study is relevant to the resolution of security bugs.

1.2.2 Identify gaps relevant to the resolution of security bugs

To find gaps in the studies relevant to the resolution of security bugs, first each study has been read by the author of this thesis to identify the "problem" addressed in the study, the proposed "solution" to the problem and the method of "evaluation". In the next stage, the "problem", "solution" and the "evaluation" method of each study has been synthesised and scrutinised by the author of this thesis. In some cases, the study has been discussed with

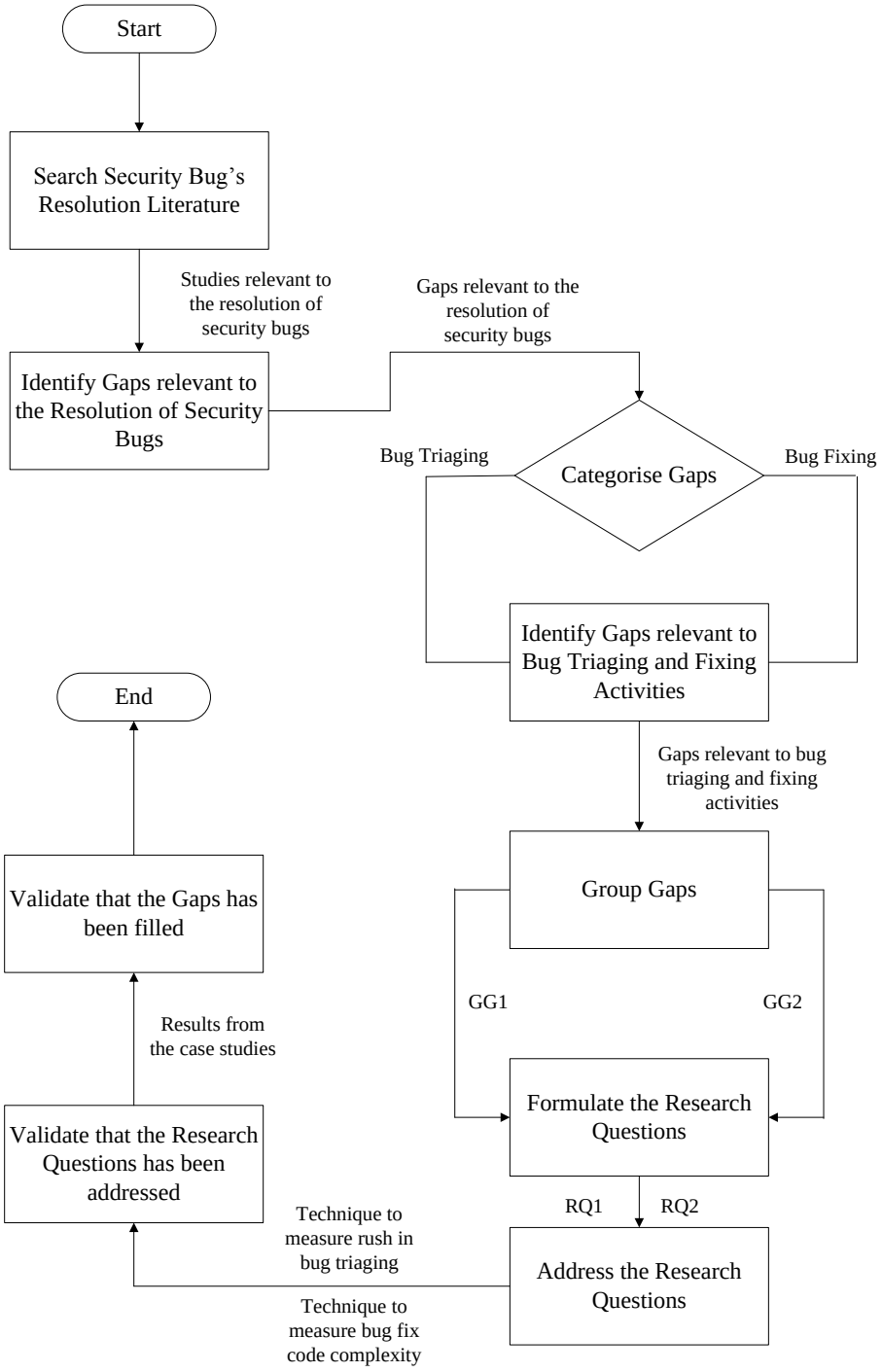

Figure 1.1: The graphical representation of research methodology with the stages and inputs/outputs.

the supervisors to take their opinion. Finally, a gap is determined in each study by the author of this thesis based on his synthesis and observations

about the study.

1.2.3 Categorise gaps

After finding gaps in the studies relevant to the resolution of security bugs, each gap has been categorised according to the triaging and fixing activities of bug resolution. The bug triaging involves reproducing and prioritising the bugs by the software developers before assigning it to fixing. The bug fixing involves making changes to the code. The outcome of bug triaging process is bug reports and the outcome of bug fixing process is code patches. Therefore, after categorisation of gaps relevant to the resolution of security bugs; further literature search has been conducted to find out studies relevant to triaging of security bugs by the software developers, fixing security bugs by making changes to the code and about the outcome of bug resolution as the bug reports and code patches.

1.2.4 Identify gaps relevant to triaging and fixing activities

The similar process to identify gaps relevant to the resolution of security bugs as discussed in Sub-section 1.2.2 has been adopted to identify gaps relevant to the bug triaging and fixing activities. In this way, the list of gaps has been prepared that contains gaps relevant to the triaging of security bugs by the software developers, the fixing of security bugs by making changes to the code and about the outcome of bug resolution as the bug reports and code patches.

1.2.5 Group gaps

As the gaps relevant to the resolution of security bugs are categorised into two main categorise (bug triaging and fixing). Therefore, each gap is related

to bug triaging and fixing from the list of gaps. The gaps relevant to the triaging of security bugs has been grouped together and named as the "first group of gaps" and the gaps relevant to the fixing of security bugs are named as the "second group of gaps".

1.2.6 Formulate the research questions

During the gaps analysis of security bug resolution, it is found in the literature that security bugs are triaged faster and reopened more frequently compared to the other bugs. Therefore, two research questions relevant to the triaging of bugs has been identified as listed in Section 2.6. Similarly, a gap relevant to the fixing of security bugs has been identified during the gaps analysis of security bug resolution. The bug fix complexity is relevant to the fixing activity of bug resolution; therefore the gaps relevant to the fixing of bugs are grouped into one category. The two research questions are formulated to know about the bug fix complexity of security bugs as reported in Section 2.6.

1.2.7 Address the research questions

To address the formulated first research question, a technique to measure rush in the triaging of bugs has been developed. Similarly, to address the second research question, a technique to measure bug fix code complexity has been developed.

1.2.8 Validate that the research questions has been addressed

For the validation that research question has been addressed, the following two options has been scrutinsed for conducting the case studies.

Deeply involved with development process:. The first option is to observe the security bug resolution process in a real industrial setting by

actually resolving the bugs or interviewing the software developers. For this purpose, An industrial case study needs to be searched that already has a bug resolution process in place and whose developers have some experience of resolving security bugs. However, the following challenges are identified in conducting the study in such a way.

1. As a researcher, getting access to an industrial security critical project is difficult if not impossible, because usually organisations cannot share their security breach data with outsiders;
2. Software developers are generally reluctant to share detailed inside knowledge about their personal process for resolving security bugs because of the fear to lose privacy;
3. Researchers may impose confirmation bias and subjectivity by getting deeply involved with the developers which may cause the change of the data;
4. All organisations have different processes in place for bug resolution, therefore repeating one study requires changing the settings to another case study which may lose the generality of the findings.

On the other hand, the strength of conducting this type of case study is that developers can immediately confirm and verify the findings.

Collecting and analysing historical data from publicly available projects:. Another option is to analyse how the security relevant bugs were resolved in the past using historical bug resolution data. Although the threat of not getting the documentation of security bugs from the organisations is still valid for such types of case studies, a key advantage is that, bug reports and code repositories of open source security critical projects is readily accessible from their repositories and could be verified by other developers and researchers.

The following enumerates the advantages of conducting a case study using historical data.

1. It is easier to repeat the study using the historical data of bugs resolved in the past rather than using qualitative data that are subject to varying interpretations;
2. The findings are easier to generalise as the other researchers can verify them by repeating the study using the same settings.
3. The results produced are more reliable compared to the results interpreted from the process of bug resolution or through the interviews of the software developers.
4. Getting access to the historical data is relatively easier than arranging interviews with the software developers.

One drawback of conducting a case study using historical data is that the results might be different if the organisation changes their process in the future.

Taking into account of the advantages and disadvantages of the two options, the decision has been made to conduct the case studies using the historical data of open source projects (e.g., *Samba* [3]) to observe their security bug resolution process. The five case studies (*Samba*, *Mozilla Firefox*, *Mozilla*, *Red Hat*, *FreeBSD*) has been conducted to validate that first research question has been addressed. On the other hand, the technique to measure code complexity of bug fixes has been validated on the two above cases (*Samba*, *Mozilla Firefox*).

1.2.9 Validate that the gaps has been filled

At-last, the results of case studies has been mapped with the research question to conclude that the identified gaps in the literature have been addressed.

[3] https://www.samba.org/

1.3 Justifying the choice of empirical case study

During the design of research, the author of this thesis has thought about the reliability of results and what to consider as a valid answer from the outcome of the research. The purpose of this thesis is "to investigate the resolution process of security bugs", which require observations of bug resolution process. Therefore, the positivists philosophical stance has been adopted for accepting the evidence in response to the designed research questions. The positivists philosophical stance states that "all knowledge must be based on logical inference from a set of observable facts" (Easterbrook et al., 2008).
In the above subsection about "validate that the research questions has been addressed" 1.2.8 of research methodology section, the options to chose a research method for collecting data are discussed. The confirmatory case study method has been chosen as a method of investigation because the purpose of this study is to test the hypothesis based on existing literature (Yin, 2014). Another reason of choosing the empirical case study is the availability of data in the open source projects as discussed in above section 1.2.8. This study is designed for the multiple cases for the greater validity. As the nature of this study is confirmatory, therefore same results are expected from the outcome of different case studies (*literal replications*). A security critical project is a unit of analysis for each case in this thesis. The process of resolving bugs might be different for different organisations and projects. It means changing context can influence on the observations. Therefore, the case studies are preferred over the controlled experiments for this study.

1.4 Structure of the thesis

The thesis is organised into five chapters. In Chapter 2, gaps relevant to security bugs has been identified and organised into groups. After the grouping of gaps, two research questions has been formulated to be addressed in the technical chapters. The Chapter 3 is about measuring rush the time

management behaviour of software developers on the triaging of bugs. The technique is applied to the five case studies that includes *Samba*, *Mozilla Firefox*, and *Red Hat*, *FreeBSD* and *Mozilla*. The Chapter 4 is about measuring the fan-in complexity of functions relevant to bug fixes. The fan-in complexity technique is applied to *Samba* and *Mozilla Firefox* case studies. The both chapters include the details of algorithms used for the development of technique. The chapters also include explanation of tools developed for data collection and analysis to conduct the case studies. Finally, Chapter 5 provides the conclusion and discusses the future work and vision.

Chapter 2

Literature Review

Security bugs are one of the primary concerns for software developers during maintenance of the systems as motivated in the introduction. The focus of this chapter is on categorising existing studies relevant to security bugs to address the problem of maintaining security in a software system.

2.1 Security Bugs

According to Zaman et al. (2011) on average a security bug reopens 2.5 times more frequently than a performance bug and significantly more often than the other types of bug. One can conclude from this finding that the security bugs are more often prematurely resolved compared to the other bugs by the software developers. Following this finding, the study has also reported that on average a security bug is triaged faster compared to the other bugs during their resolution. The study has explained that it might be possible that developers strategically triage security bugs faster to hide information about the fixing of such bugs from the attackers to avoid further exploit of the system. However the faster triage and then frequent reopening also pose a question whether the software developers rush to resolve security bugs on the triaging. The follow up question is whether rush by the software developers is

the reason of security bugs premature assignment and frequent reopening. As the focus of this thesis is on the resolution of security bugs, which involves bug triaging. Therefore, investigating whether developers rush to triaging security bugs will help to improve the overall process of resolving security bugs.

In another study, Bhattacharya et al. (2013) have measured the median bug fixing time of security and the other types of bug for the *Mozilla* and *Google* code-based apps. The study has found that the security bugs were given high priority in the *Mozilla* case study compared to the *Google* code-based apps. The finding indicates that security bugs are treated differently in different systems. Therefore, it is very difficult to generalise that faster triage of security bugs is because of their higher priority compared to the other bugs. On the other hand, Mitropoulos et al. (2012) has found that the number of unresolved security bugs are increasing in the projects compared to the other bugs. Similarly, Li et al. (2006) have reported that the percentage of unresolved security bugs is increasing in the *Mozilla Firefox* and *Apache* open source systems.The both studies has not provided any scientific explanation of this phenomena. Although, Li et al. (2006) has explained that the proper resolution of security bugs is becoming important for all types of software systems. This finding leads to a question that why more security bugs are found unresolved in such systems. one can argue that complexity in fixing of such bugs might attribute to their status being unresolved for the longer period of time compared to the other bugs. In another study, Caglayan et al. (2012) have argued that a bug fix takes more time of developers in case its affected code have higher complexity. However, it is difficult to justify this argument higher code complexity contributes to the status of security bugs being unresolved without knowing whether such bugs are more complex to fix.

As the bug fix complexity is related to the fixing stage of bug resolution, therefore knowing about the complexity of security fixes will also help to

improve the resolution process of security bugs.
In this section, focus has been on the studies about and relevant to security bugs, and the following gaps labelled by SB, which stands for Security Bugs has been identified.

SB1: There is a need to investigate whether rush by the software developers on the triaging of security bugs is one of the reasons of their premature assignment and then frequent reopening (Zaman et al., 2011),(Bhattacharya et al., 2013).

SB2: There is a need to investigate whether security bugs are more complex to fix compared to the other bugs (Caglayan et al., 2012).

The gap relevant to rush in triaging and bug fix complexity relevant to fixing activities of security bugs are related to the resolution process, which will be detailed in the next section.

2.2 Bug Resolution Process

Triaging and fixing are the main activities involved in bug resolution (Xia et al., 2013). Bug triaging is concerned with the prioritisation and assignment of bugs to the right developers for fixing (Mani et al., 2013). After the assignment of a bug to the right developer, the bug fixing involves changing the code affected by the bug (Zeller, 2002). The following sections are discussing studies relevant to triaging and fixing activities.

2.2.1 Bug triaging

The developer who triage a bug needs to make sure that enough information is available to reproduce the bug (Anvik et al., 2006). According to Baysal et al. (2012), the fixing and reopening of a bug depends on how well the bug is triaged. In two other studies, Guo et al. (2010) and Guo et al. (2011) have argued that there is a higher chance of bug re-assignment if a developer assign bug for fixing without understanding its cause.

Similarly, Jeong et al. (2009) have argued that there is a less chance of re-assigning bugs to other developers in case the developer who triages the bug (a) carefully reproduce the bug by understanding the cause of bug, and (b) has assigned the bug to the right developer according to the problem description. All the above studies have emphasised that the developer should carefully triages the bug. However, none of these studies have quantitatively analysed bug triaging activity by the software developers.

As bug triaging activity is relevant to the resolution of security bugs, therefore without analysing bug triaging activity one can not know whether security bugs are triaged in rush. Therefore, quantitative analysis of bug triaigng activity is necessary to fill a gap relevant to the security bugs.

In another study, Francalanci and Merlo (2008) have found that the most of bugs are fixed closer to the release dates. They have explained that this phenomenon shows when the release date approaches developers hasten to fix bugs with the aim of including changes in the forthcoming release. Usually, the delivery of a bug fix is prioritised during triaging. Therefore in this hustle to fix bugs in the forth coming release, practically the developers can prolong the fixing of low priority bugs those can still be delivered in the next release in parallel to fixing the high priority bugs. In this way, the developers can deliver more high priority bugs within their resource limits. However, in their analysis to fix bugs near to the release deadlines. The study has not considered this practical fact that the developers can prolong the fixing of some bugs to fix more high priority bugs.

As the focus of this study is to measure rush during the triaging of bugs reflecting the real life software project. Therefore, it is necessary to consider the practical situation of bug fixing when the developers have to fix high priority bugs within the given release deadlines.

Similarly, Hooimeijer and Weimer (2007) have found that the daily workload of the developers can affect on their ability to triage bugs. This finding indicates that there is a possibility that the developer who triages the bug

with the bigger workload might assign it to the other developer without understanding the cause of the problem. Therefore, there is a possibility that the developers might rush during the triaging of bugs due to the workload and can assign the bug to the wrong developer leading to the re-assignment or re-opening of bugs. However, the study has not measure rush during the triaging of bugs by the software developers.

2.2.2 Bug fixing

During the fixing of bugs, the developers make changes to the affected source code as a solution to recover the system in its operational state (Thung et al., 2013). According to Zhang et al. (2013) have found that the quickness and slowness in the fixing of bugs by developers depends on the complexity of code relevant to bugs. Similarly, Murphy-Hill et al. (2013) have found that the developers fix the bugs with the higher code complexity nearer to the release deadlines compared to the bugs with the lower code complexity. These finding indicate that the bugs with the higher code complexity can be difficult to fix. However, neither study have measured the complexity of code relevant to bug fixes to argue that the bugs with the higher code complexity are indeed difficult to fix.

In their study, Zhang et al. (2012), have argued that the bugs with the higher code complexity are not as simple to fix. Similarly, Shihab et al. (2010) have argued that the bugs with the higher code complexity are complex to fix for the software developers. In other studies, Marks et al. (2011), Kim and Whitehead, Jr. (2006) and Weiss et al. (2007) are agreed with the findings of Shihab et al. (2010) that code complexity of bug fixes can increase their fixing time. But these studies have used the bug fixing duration and average bug fixing time as the measure of the complexity of code relevant to bugs. However, it is not necessary that the developers immediately start making changes to the code without spending some time in understanding the cause of the bug. Therefore, the claim that those bugs took more time to be fixed by

developers are complex without actually measuring their code complexity is not fully quantified. The above studies are relevant to this thesis as they are supporting the argument presented in this thesis to measure the complexity of bug fixes.

Other studies by Anbalagan and Vouk (2009) and Giger et al. (2010) have acknowledged that claiming the bugs those are more complex to fix take the more time of developers is very shallow without measuring the complexity of code relevant to bugs using the code complexity metric. However, these studies themselves have not measured the complexity of code relevant to bugs using any other technique than the fixing time.

2.2.3 Gaps in the study of bug resolution

In this section, the focus was on the studies about and relevant to the bug resolution process which include bug triaging and fixing. The following is a summary of gaps identified from the studies, the gaps are labelled as BR, which stands for Bug Resolution.

BR1: There is a need to quantitatively analyse how the developers triage bugs during their resolution (Guo et al., 2010),(Guo et al., 2011),(Jeong et al., 2009).

BR2: There is a need to measure rush during bug triaging considering the practical handling of bugs by the software developer near to the release deadlines (Francalanci and Merlo, 2008),(Hooimeijer and Weimer, 2007).

BR3: There is a need to measure the complexity of code relevant to bug fixes to argue that bugs with the higher code complexity are difficult to fix (Zhang et al., 2013),(Murphy-Hill et al., 2013).

BR4: There is a need to verify that the bugs with the higher code complexity are more difficult to fix using different complexity metrics from the fixing time (Zhang et al., 2012),(Shihab et al., 2010),(Kim and Whitehead, Jr., 2006),(Weiss et al., 2007),(Anbalagan and Vouk, 2009),(Giger

et al., 2010).

In previous two sections, the two key gaps has been identified: (a) there is a need to measure rush by the software developers in the triaging of bugs, and (b) the bugs with the higher code complexity are difficult to fix. Therefore, in the next two sections the focus of study is on the rush behaviour of software developers during triaging of bugs and on studies about the complexity of code relevant to bug fixes.

2.3 Software Developers

In software maintenance, the developers' personal traits, such as the behaviour towards time management, play an important role in their outputs (Wynekoop and Walz, 2000). Therefore, a detailed discussion is provided to describe the relationship between the time management behaviour of software developers and how such behaviour can be measured.

According to Ye and Kishida (2003), Ko et al. (2006) and Xiao and Afzal (2010) that not only the technical skills but the personality traits, such as the time management behaviour of the developers are equally important to successfully resolve the bugs. Usually, the developers are not working on a single project or doing only one task at a time. Therefore, it becomes extremely important how the software developers manage their time to complete the assigned tasks within the deadlines (Sillitti et al., 2003). Bad time management behaviour means that either the developer delays the completion of the task or quickly finishes the task in a rush without caring much about doing the work properly. Rush is a time management behaviour of software developers under the pressure of deadlines as suggested by the König and Kleinmann (2005).

Other studies by Rasch and Tosi (1992) and Höst et al. (2000) have discussed how time pressure can affect the rate of task completion by developers, because under time pressures either the developers try to finish the task in

a rush or slow down the work on some tasks. All these studies have indicated that time pressure affects developers' outputs. However, none of these studies has discussed how rush the time management behaviour of software developers, can be measured when they are under the pressure of time.

Peralta et al. (2010) have measured the daily and weekly utilisation of time by software developers to analyse how they managed their time to complete the assigned tasks. However, the study did not measure rush the time management behaviour of software developers on the triaging of bugs.

In other studies, Zhong et al. (2000), Prechelt and Unger (2001) and Yanyan and Renzuo (2008) are of the view that the time spent by software developers to complete a certain task is an indication of their time management behaviour. For example, Johnson et al. (2003) have measured the time spent daily by the software developers in completing the assigned tasks to evaluate their behaviour towards achieving the set deadlines. They have argued that such behavioural evaluation helps to determine the developers' personal process of completing the tasks. However, the study did not discuss whether rush the time management behaviour of software developers on the triaging of bugs, can be measured by calculating the time spent by them on their daily task of bug assignment.

In another study, Shihab et al. (2012) have argued that reopening of bugs depends on how fast or slow a bug is resolved by the software developers. Based on this finding, one can conclude that the bugs which are resolved quickly are more likely to reopen. Similarly, Jongyindee et al. (2012) have found that the bugs marked as NEW or CLOSED by expert software developers are less likely to re-assign or reopen compared to the bugs which are marked by the incautious developers. But neither study discusses whether rush the time management behaviour of software developers can be one of the reasons of their premature assignment and subsequently reopening.

In this section, the focus of study was on the studies about and relevant to the time management behaviour of the software developers. The following

gaps are identified from the studies. The gaps are represented by the symbol Software Developer (SD).

SD1: There is a need to measure rush the time management behaviour of software developers on the triaging of bugs (Rasch and Tosi, 1992) , (Höst et al., 2000), and (Peralta et al., 2010).

SD2: There is a need to verify whether rush the time management behaviour of software developers on the triaging of bugs can be measured by calculating the time spent by them on bug assignment (Prechelt and Unger, 2001) and (Johnson et al., 2003).

SD3: There is a need to verify whether rush the time management behaviour of software developers can be one of the reasons of premature bug assignment and subsequently their reopening (Jongyindee et al., 2012), (Shihab et al., 2012).

In brief, a discussion is provided to study the code complexity metrics to measure the complexity of code relevant to bug fixes. Techniques for such measurement will be compared in the next section.

2.4 Code Complexity

The amount of time spent by the developers during the fixing of bugs is directly related to the complexity of code (Banker et al., 1989). The code metric is a quantitative method to measure the complexity of code relevant to software systems (Henry and Selig, 1990). The source lines of code (SLOC), Cyclomatic Complexity (CYC), and fan-in are examples of source code metrics based on the call graph of the system (Abandah and Alsmadi, 2013). The call graph reflects the complexity of the code based on the relationship between the functions (Ryder, 1979). The source line of code (SLOC) is a code metric to count the size of a program or software system (Albrecht and Gaffney, 1983). The Cyclomatic Complexity (CYC) helps to measure the linearly independent paths in a program or software system (McCabe,

1976). However, neither technique helps to count the function calls. But the fan-in metric is useful to count the number of times a function has been called by the other functions (Marin et al., 2007).
In the following sub-sections, an explanation about the use of call graph based metrics to measure the complexity of code relevant to bug fixes.

2.4.1 Call graph based metrics

The complexity of code relevant to bugs could increase the fixing time significantly. As discussed above, there are different code metrics to measure the complexity of the code statically. This sub-section focuses on measuring the complexity of code relevant to bug fixes.
According to Zhang et al. (2007), the modules relevant to bugs with more lines of code are time consuming to fix. However, the study does not measure the complexity of code relevant to bug fixes. Rather, it uses the average module fixing time as the complexity metric. Multiple reasons, such as the functions of the module, are tightly coupled and changing one function requires changes in the other functions, which can cause a longer fixing time for bugs. But the study does not measure the complexity of functions relevant to bug fixes.
In another study, Nistor et al. (2013) has measured the complexity of bug fixes inside a patch using the lines of code metric (SLOC). Based on their results, they argued that the bugs with the larger sized patches are complex to fix. The larger size of the patch does not necessarily mean that the functions inside the patch are complex. It is possible that the logic used by the programmer to solve a problem needs more lines of code, which actually increases the size of the patch. Therefore, it is very difficult to generalise that all the bug fixes with the larger patch size are complex without actually measuring the complexity of functions relevant to bug fixes inside the patch.
Similarly, Kula et al. (2010) has measured the complexity of code relevant to bug fixes using the cyclomatic complexity (CYC). They have found that

the affected code of bug patches with a complex structure of control flow have higher fixing durations. It is possible that the developers have to search through the whole patch to make code changes because of the broken control flow, which causes the longer fixing duration. But this does not mean that the functions inside the patch are also complex to fix. Therefore, it is very difficult to generalise that the affected code relevant to bugs takes a longer fixing duration without measuring the complexity of functions relevant to the affected code of the bugs. However, the study does not measure the complexity of functions inside the patch.

Shin and Williams (2008) have even gone one step further for computing the code complexity of security and non-security functions by measuring their lines of code and control flow structure. But the complexity of functions relevant to bug fixes has not been computed. Secondly, in this study the metrics of lines of code and cyclomatic complexity are used for measuring the complexity of functions. In this way of measuring the complexity of functions, the many functions which are invoked by the other functions inside a patch have been ignored. When a function code is invoked by the other functions inside a patch then the developers need to make changes to the multiple functions relevant to bug fix, which increases the fixing duration. Therefore, there is a need to use different complexity metrics to measure the complexity of functions relevant to bug fixes.

2.4.2 Fan-in metric

Fan-in is one of the complexity metrics that is likely to tell the difference between security and functionalities. It computes how many times a function has been called by the other functions. In this section, the studies relevant to fan-in analysis are discussed.

Marin et al. (2004) have argued that code relevant to security implementation crosscuts many functions and have more fan-ins. However, the study has not measured the fan-ins of functions relevant to security bug fixes.

In an earlier study, using a concrete example program (Saleem et al., 2013), the author of this thesis has argued that all the crosscutting functions that implement a security requirement shall be updated when the change to the system is relevant to protection and asset (Haley et al., 2008). However, that study has not quantified the complexity of crosscutting security functions in terms of fan-ins.

In another earlier study, Yu et al. (2004) have discovered that crosscutting implementation of a system often coincides with the crosscutting refinement of quality requirements, which include security. The number of clones used in the case study of the work has not been re-factored into high fan-ins functions. Therefore, it was not possible to consider the fan-in impact of crosscutting functions on fixing security bugs.

According to Mubarak et al. (2010), a bug fix with high fan-in classes is more complex to fix but usually the developers have to change the affected functions of the class relevant to a bug and that changed function might be called by the other classes as well. Therefore, one cannot say whether a bug is simple or complex to fix without measuring the fan-in of changed functions relevant to a bug, rather than measuring the fan-in of classes relevant to a bug.

Zhang et al. (2008) have used the fan-in metric to identify the methods whose functionality is needed across the system. They argue that such methods generate high fan-in values. This finding indicates that the developers have to spend more time on changing the high fan-in functions. However, the study have not measured the fan-in of functions relevant to bug fixes.

2.4.3 Gaps in the study of code complexity

In this section,the focus has been on the studies about and relevant to code complexity metrics to measure the complexity of code relevant to bug fixes. The following gaps are identified from the studies. The gaps are represented by the label CC, which stands for Code Complexity.

CC1: There is a need to measure the complexity of functions relevant to a bug fix using different complexity metric from the SLOC and CYC (Zhang et al., 2007),(Nistor et al., 2013),(Kula et al., 2010),(Shin and Williams, 2008).

CC2: There is a need to verify whether functions relevant to bug fixes have high fan-in values (Mubarak et al., 2010),(Marin et al., 2004),(Zhang et al., 2008).

In the sections of studies relevant to software developers and code complexity, two key gaps has be found: (a) the need to measure rush the time management behaviour of software developers on bug triaging, by calculating the time spent by the software developers on bug assignment, and (b) the need to measure the fan-in complexity of functions relevant to bug fixes. The historical data of bug triaging is available in the bug reports, therefore to realise the first gap, the literature relevant to bug reports for measuring rush the time management behaviour of software developers is listed in the next section. Similarly, the fixed code relevant to bugs is available in the code patches, therefore the studies relevant to code patches to realise the second gap are also included in the next section.

2.5 Software Artefacts

The bug reports and code patches are related types of software artefacts produced during software maintenance. Usually, both artefacts are treated as rich historical data to analyse the bug attributes and the quality of the fixing code. In the following sub-sections, existing studies relevant to bug reports and code patches are discussed.

2.5.1 Bug reports

In software development, whenever a new bug is found, it is documented using a "bug tracking system" such as `Bugzilla` (do Rego et al., 2008). The

`Bugzilla` system requires from the reporter of the bug to fill relevant fields for the documentation of the bug (Serrano and Ciordia, 2005), so the developers can understand, reproduce and fix the problem. Filling information in these fields is necessary to have a good quality bug report (Bettenburg et al., 2008). The bug report not only helps to understand and reproduce the problem but is also useful to keep track of the current state of bug.

In the `Bugzilla` system at the time of documenting the bugs, usually the following fields are recorded (The Bugzilla Team, 2012). First, a bug is assigned a unique identifier and its status field is updated according to its stage in the life-cycle. A bug life-cycle starts from the time when the bug is reported and it ends when the bug is finally closed. The names of components and versions are recorded in which the bug is found. The importance of the bug based on its priority and severity are also logged in the bug reports. The name of the person to whom the bug is assigned for triaging or fixing, the bug ids on which the bug depends on for fixing and the bug ids which are dependent on fixing of this bug, the time when the bug is reported and the time when the bug was last modified are all recorded in the bug report.

In research, the bug reports are used as rich historical information for the analysis of past bugs. For example, Jain et al. (2012) have extracted the bug reports marked as FIXED and VERIFIED from the historical bug reports data. This past bug report data has been used for relatively quick assignment of the newly reported bugs to the right developers based on their past experience of fixing similar types of bug. The draw-back of assigning bugs to developers based on the past bug reports assignment of bugs to the developers can reduce the triaging time, but usually the quick assignment of bugs during triaging can lead to the re-assignment and reopening of bugs. However, the study has not extracted the bug reports data in a way to validate whether the developers quickly assigned the bugs during the triaging.

Similarly, Ahmed and Gokhale (2009) have extracted the time stamps when the bug status is reported and resolved from the bug modification histories

to measure the time spent on the resolution of bugs. In another study, Ahmed and Gokhale (2008) have used the states of bugs to measure the bug resolution duration according to their severity levels. Usually, the time spent on the resolution of bugs is measured by taking the difference of time when the bug is RESOLVED as fixed till the time period it was REPORTED. However, the data extraction fields used in these studies are not usable for the analysis of bug triaging.

Lamkanfi and Demeyer (2012) have extracted the time stamps when the bug status is changed to reported and resolved, but usually the overall duration spent on the resolution of bugs depends on the amount of time spent in the triaging and fixing of bugs. In particular, the triage duration depends on the availability of developers and their understanding of the problem. However, the study has not extracted the bug status time stamps when the bug is reported till it is assigned to the developers for measuring the time management behaviour of software developers, such as rush during triaging. Wu et al. (2011) have extracted the bug time stamps when the bug status is resolved without taking into account the time stamps when the bug status is reopened. Another study by Wang et al. (2012) has extracted the bug reports based on time stamps when the bug status is resolved or the bugs are marked as closed. In this way, both studies have extracted the bugs which are either marked as closed or their status is updated to resolved, but in some cases the bugs marked as resolved might reopen later. Therefore, ignoring the time stamps when the bug status is reopened while extracting the bug resolution time-stamps data might mislead on the duration of some bugs.

2.5.2 Code patches

The software system development is managed using the version control systems such as `git` (Loeliger and McCullough, 2012). Each version control system maintains a log entry in the "commit log" whenever a developer commits a patch in the source code (Chen et al., 2004). A patch reflects the

modifications made by the developers and the description of code changes. Usually, a patch has a section of log message and another section describing the code changes.

The log message contains a unique commit number to distinguish it from the other patches, author name and email, the date of commit and a list of names and emails of the other people to inform about the commit. The description of code changes part begins with a file name and the bunch of code lines started with the affected line numbers, old and new versions of the file followed by the code fragments.

Zhou et al. (2012) have extracted the patched code relevant to bug fixes from the code patches to analyse the size of patched code to fix the bugs. Such an analysis is useful for determining the characteristics of the whole code segment rather than the individual functions. However, the study did not extract the functions relevant to fixed bugs to measure their complexity.

In another study, Fischer et al. (2003) and Cubranic and Murphy (2003) have used the BUG and FIXED keywords to find the bugs in the patches relevant to bug fixes. Usually, the fix in the code is marked with the bug number at the time of commit by the developers. Therefore, one can parse the static code of patches to extract the fixed code relevant to bugs. However, it is not necessary that the developers always use the same keywords such as BUG and FIXED to commit the code relevant to the bugs at the time of patching the fix.

The changes to buggy code for fixing a bug may affect the multiple code fragments of a patch or of the multiple patches (Sliwerski et al., 2005). Therefore, such modified parts of the code can be extracted by taking the difference of two patches using the `git` $diff$ command (MacKenzie et al., 2003). However, the patch difference has not been used in these studies to extract the functions relevant to bug fixes.

Kim and Ernst (2007) and Tian et al. (2012) have extracted the changed sets from the patches relevant to bug fixes. Extracting the complete changed set

helps to know which lines or code fragments have been changed during the bug fix, but one cannot analyse those functions relevant to the bugs which have been changed during the fix without extracting the names of functions from the changed set.
On the other hand, Kim et al. (2006) have parsed the patches relevant to bug fixes to extract the components from the code hunks. Extracting the names of components relevant to the bug fixes helps to analyse those components which need more attention for the fixing of bugs. However, the developers will still face problems in the fixing of bugs because there are many functions in each component, so they have to go through all the functions relevant to the bugs for fixing them properly. On the other hand, in cases where the developers know which functions relevant to a bug fix have been called more often such knowledge can reduce their effort in searching through the whole system to fix the functions relevant to a bug. Therefore, measuring the code complexity of functions relevant to a bug fix rather than a component will help developers, but measuring the code complexity of functions relevant to a bug fix is not possible without extracting the changed functions from the code hunks of patches.

2.5.3 Gaps in the study of software artefacts

In this section, first the studies about and relevant to extracting the bug status time stamps to measure the "rush" time management behaviour of software developers has been discussed. Then, the studies relevant to and about extracting the changes functions from the changed set of the code patches are focused upon. The following gaps are identified from the studies. The gaps are labelled by SA, which stands for Software Artefacts.

SA1: There is a need to extract the bug status time stamps when the bug is reported till it is assigned to developers for measuring rush the time management behaviour of developers on bug triaging (Jain et al., 2012), (Ahmed and Gokhale, 2009),(Lamkanfi and Demeyer, 2012).

SA2: There is a need to extract the bug time stamps considering the reopening time of bugs (Wu et al., 2011),(Wang et al., 2012).

SA3: There is a need to extract the changed functions from the patches relevant to a bug fix to measure the complexity of functions relevant to bug fixes fix (Kim and Ernst, 2007),(Tian et al., 2012),(Kim et al., 2006).

2.6 Summary of Literature Gaps

The motivation of this study is to investigate the bug resolution as discussed in the Chapter1. Based on the motivation of the study, two key gaps has been identified involving triaging and fixing activities for the resolution of security bugs. Therefore, all the gaps identified from the literature survey are organised into two groups. The first group of gaps is related to the triaging activities of bug resolution. On the other hand, the second group of gaps is related to the fixing activities of bug resolution.

The following are gaps organised in the first group. The gaps related to the first group are labelled by GG1, which stands for the first Gap Group.

GG1.1: There is a need to investigate whether rush by the software developers on the triaging of security bugs is one of the reasons of their premature assignment and then frequent reopening [SB1, §2.1].

GG1.2: There is a need to quantitatively analyse how the developers triage bugs during their resolution [BR1, §2.2.3].

GG1.3: There is a need to measure rush during bug triaging considering the practical handling of bugs by the software developer near to the release deadlines [BR2, §2.2.3].

GG1.4: There is a need to measure rush the time management behaviour of software developers on the triaging of bugs [SD1, §2.3].

GG1.5: There is a need to verify whether rush the time management behaviour of software developers on the triaging of bugs can be measured

by calculating the time spent by them on bug assignment [SD2, §2.3].

GG1.6: There is a need to verify whether rush the time management behaviour of software developers can be one of the reasons of premature bug assignment and subsequently their reopening [SD3, §2.3].

GG1.7: There is a need to extract the bug status time stamps when the bug is reported till it is assigned to developers for measuring rush the time management behaviour of developers on bug triaging [SA1, §2.5.3].

GG1.8: There is a need to extract the bug time stamps considering the reopening time of bugs [SA2, §2.5.3].

The first group of gaps is organised based on the resolution of security bugs involving triaging activities. Therefore, the following research question about the triaging of security bugs is derived from the first group of gaps. In fact, this research question will fill the first gap relevant to the security bugs as listed in Section [SB1 2.1].

> `RQ1:` How can 'rush' of software developers be measured from their time management behaviour exhibit in the bug repository? If rush can be measured, is the hypothesis of 'security rush' be confirmed statistically? Is rush a reason for prematurely assigning security bugs to software developers and subsequently reopening the bugs more frequently?

The following are gaps organised into the second gap group, which is labelled by GG2:

GG2.1: There is a need to investigate whether security bugs are more complex to fix compared to the other bugs [SB2, §2.1].

GG2.2: There is a need to measure the complexity of code relevant to bug fixes to argue that bugs with the higher code complexity are difficult to fix [BR3, §2.2.3].

GG2.3: There is a need to verify that the bugs with the higher code complexity are more difficult to fix using different complexity metrics from the fixing time [BR4, §2.2.3].

GG2.4: There is a need to measure the complexity of functions relevant to a bug fix using different complexity metrics from the source lines of code (SLOC) and cyclomatic complexity (CYC) [CC1,§2.4.3].

GG2.5: There is a need to verify whether functions relevant to bug fixes have high fan-in values [CC2, §2.4.3].

GG2.6: There is a need to extract the changed functions from the patches relevant to a bug fix to measure the complexity of functions relevant to bug fixes [SA3, §2.5.3].

The second group of gaps is organised based on the resolution of security bugs involving fixing activities. Therefore, the following research question about the fixing activities of security bugs is derived from the second group of gaps. In fact, this research question will fill the second gap relevant to the security bugs as listed in Section [SB2 2.1].

> `RQ2:` How can the complexity of a bug fix be measured from the code repositories? Is the complexity of a security bug fix higher on average than the complexity of a non-security bug?

All the identified literature gaps are summarised in Figure 2.1.

The first research question `RQ1` will be addressed in Chapter 3 and the second research question `RQ2` will be addressed in Chapter 4.

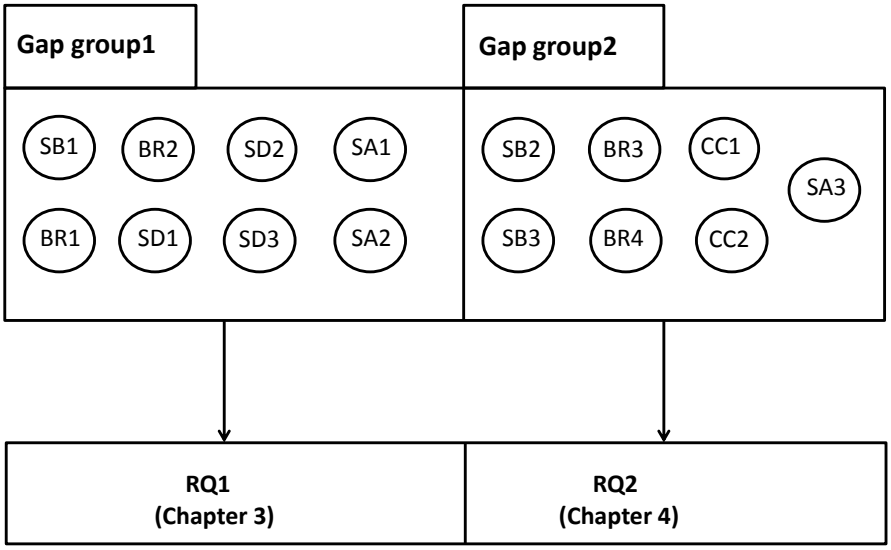

Figure 2.1: The summary of literature gaps grouping and the derived research question from each gap group.

Chapter 3

Measuring Rush

To maintain the security of software systems, often there is a perception that security bugs need to be resolved quickly in response to malicious attacks. Therefore, usually software engineers treat them as urgent problems. However, it is evident from a literature study by Zaman et al. (2011) reported in Section 2 of Chapter 2 that security bugs reopen more often than other bugs. The reopening of security bugs could be attributed to rush.

To know that the developers really triage security bugs in a rush, there is a need to understand the behaviour of developers when triaging a security bug, their psychology and their time management skills. However, it is not practical in a real life project to monitor the developers' behaviour while they are triaging the bugs, because the developers might not act normally when they are aware that somebody is assessing their rushing behaviour.

On the other hand, the artefacts produced by developers can give clues about the nature of task done by them in a specific time period. But one cannot say that the task done by the developers in a specific time reflects the fact that security bugs are triaged in a rush.

On the other hand, to address the "yes and no" part of `RQ1`, the following null hypothesis is derived.

`RQ1:` How can 'rush' of software developers be measured from their time management behaviour exhibit in the bug repository? If rush can be measured, is the hypothesis of 'security rush' be confirmed statistically? Is rush a reason for prematurely assigning security bugs to software developers and subsequently reopening the bugs more frequently?

In order to address the "how" part of `RQ1`, a technique to measure 'rush' is developed and explained in this chapter.

Hypothesis 1 *Median rush on the triaging of security bugs is not higher compared to the non-security bugs.*

In this thesis, the notion of 'rush' is introduced in the triaging (process to priorities and assign bugs to the developers for fixing) of bugs. The *rush* is defined as the ratio between the actual time spent by the developers during triaging and the theoretical time the developers have by delaying the fixes until the next regular release. In this way, the developers have more flexibility to relax the constraints for fixing high priority bugs earlier and still deliver the bug fixes on the release time within the resource constraints.

By intuition, rush is a relative measure that relates the time used for planning and the time used for the work. The metric of rush is based on the concept of planning towards the deadlines (Svahnberg et al., 2010) to complete the assigned tasks.

The theoretical measurement of rush considers the effect of delaying the work. Suppose the maintenance project originally plans to deliver a fix of a certain bug at the time of a certain release. If there is a rush, the bug may be delivered before that time. The delivery of fix with the smallest rush requires delaying the assignment of the bug till the moment when the work has to be started. In other words, the triaging duration can be prolonged. Therefore, the lower the ratio, the higher the rush in relation to the next release.

The following example is used to demonstrate the rush measurement in the

triaging of a bug. The bug 7494 of the *Samba* case study has been reported on 04/06/2010 at the time 17 : 55 by an internal developer "Jeremy Allison". The time-stamp of bug report has been logged in the *Samba* `Bugzilla` system. After the report of bug, the developer "Lars Müller" has reproduces the bug and prioritised and assigned it to the developer "Jeremy Allison". The time-stamp 04/06/2010 18 : 11 when the bug is assigned for fixing has also been logged in the *Samba* `Bugzilla` system. In this case, the bug is reported and assigned for fixing to the same developer. However, it is not necessary that a bug is assigned to the developer for fixing who has reported it. After the assignment of bug, the developer "Jeremy Allison" has fixed the bug and set the status of bug being resolved as fixed. The time-stamp 16/06/2010 06 : 21 when the bug status is changed from assigned to resolved has been logged in the *Samba* `Bugzilla` live system. All the time-stamps when the bug is reported, assigned and resolved are shown in the Figure 3.1.

Now in retrospective, let's say in the weekly release fix there are three releases schedule to be delivered on 07/06/2010, 14/06/2010 and 21/06/2010 respectively. In fact, the bug is already resolved after the delivery of first and second release as shown in the Figure 3.1. Therefore, it can only be included in the third release, which means in reality postponing the delivery of fix for this bug till the release deadline. Effectively, it is the same as postponing the assignment of this bug to fix till the maximum delay time. So, it can be included in the third release. In this way of using the maximum delay time for the assignment of bug, the more high priority bugs can be fixed within the same resources. Software developers are one of the examples of resources available for the fixing of bugs. The theoretical delay time is denoted by the symbol $t_{planned}$, which the developers have to postpone the assignment of bug for fixing till the next regular release.

Remember, the rush is defined as the ratio of actual and theoretical triage duration. Therefore, first we need to calculate the actual triage duration, maximum delay time for postponing the assignment of bugs and theoretical

triage duration to measure rush in the triaging of example bug 7494.
The actual triage duration is the difference of $t_{assigned}$ and $t_{reported}$, which is denoted by the symbol d_1 in the Figure 3.1. The actual fixing duration is the difference of $t_{resolved}$ and $t_{assigned}$, which is denoted by the symbol d_2. For the example bug 7494, the actual triage duration is $0.0115625 - days$ and actual fixing duration is $11.51 - days$. By taking the difference of release deadline which is 21/06/2010 in this case and actual fixing duration d_2 in this case $11.51 - days$. The result is the maximum delay time $6.73505787 - days$. It means the assignment of bug 7494 for fixing can be delayed till this time for delivering it in the third weekly release.
After the calculation of delay time, the theoretical triage duration for the bug 7494 is calculated by taking the difference of $t_{planned}$ and $t_{reported}$. The calculated theoretical triage duration is $6.72349537 - days$ denoted by the symbol d'_1. At-last the rush in the triaging of example bug 7494 is measured by taking the ratio of d_1 and d'_1, which is $.00171 - day$.
The rush has been computed by measuring the duration from the bug reports. Therefore, the set of logs used to recover the bug reports from the `Bugzilla` issue tracking system is defined in Definition 1.

Definition 1 *The set of logs chosen in this study are:* $LOGS = \{log_B, log_A\}$. *The set of logs contains the bug entry log* log_B *and the bug activity log* log_A. *The set of bug entry logs* log_B *is defined as* $log_B = \{b_i \mid 1 \leq i \leq n \wedge n \in \mathcal{N}$, *where* $b_i \in B$ *and* B *is the set of bugs.*$\}$
The bug activity log log_A *is defined as* $log_A = \{(b_i, (l_i, t_i)) \mid 1 \leq i \leq n \wedge n \in \mathcal{N}$, *where* $b_i \in B$, l_i *is the set of bug life cycle labels* $l_i \subset L$, *and* t_i *is a time-stamp of* l_i, *and* $t_i \in T$. $\}$

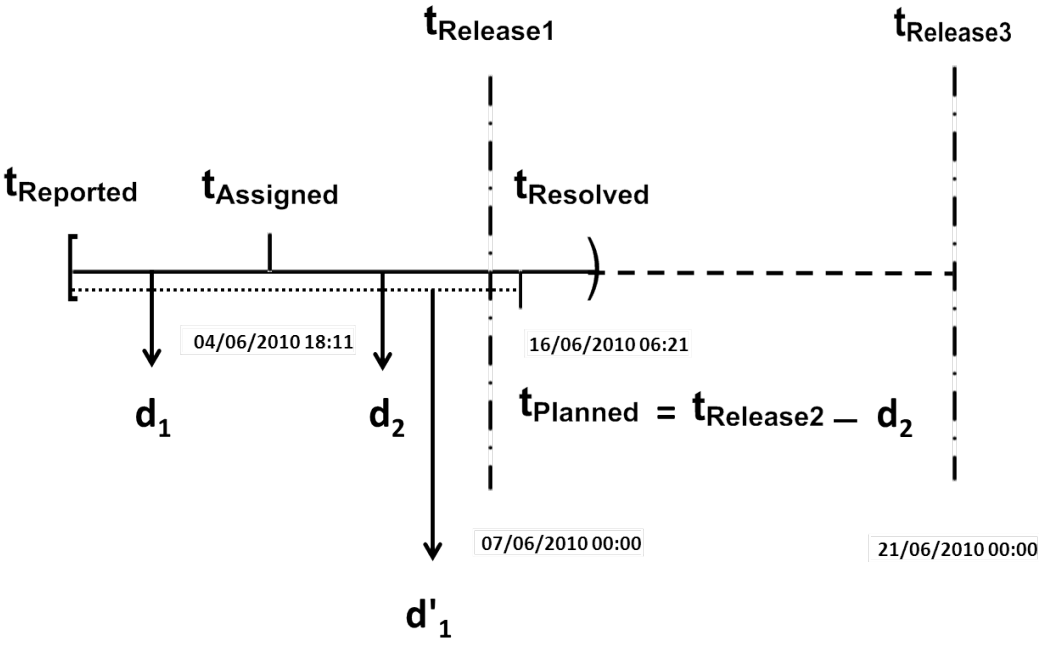

Figure 3.1: An example to explain the concept of rush using the time-stamps of the bug number 7494 of *Samba* case study.

3.1 Example Logs

The bug entry log log_B contains a pair of elements with each pair as a single bug report. In each pair, the element b represents the bug id.An example bug entry log's screen-shot is shown in Figure 3.2. In the figure, the retrieved bug id and bug report time stamp are highlighted with arrows. In this definition the set of security bugs B_s is the subset of bugs B.
The bug activity log log_A contains pairs of elements with each pair as a single bug status label and time-stamp. In each pair, b represents the bug id, C represents the set of bug life cycle labels, and t represents the bug status time. An example screen-shot of the bug activity log is shown in Figure 3.3. We have used the script to obtain the bug entry log log_B, and bug activity log log_A from `Bugzilla` bug tracking system. The scripts to obtain both logs log_B and log_A are available in Appendix A.1 and A.2 respectively.
These logs are used to recover the bug time stamps data. In the next section,an explanation has been provided about the choice of measuring rush from the bug reports.

3.2 Computing Rush from the Bug Reports

In this section, the actual bug triage and fixing durations are defined and the procedure to compute each duration is described using the algorithms. The triage and fixing durations are based on the status labels of bugs. A bug status label indicates the current stage of bug fixing. Zeller (2005) have described the `Bugzilla` bug life cycle based on the *Mozilla Firefox* case study. Zaman et al. (2011) have extended the `Bugzilla` bug life-cycle by adding a "reopened" arrow when the bug status is ASSIGNED as shown in the Figure 3.4.
In this study, the same `Bugzilla` bug life cycle as reported by Zaman et al. (2011) has been followed. However, it has been found that the figure is incomplete without two more kinds of arrows, which has been observed during

Figure 3.2: An example bug entry screen-shot.

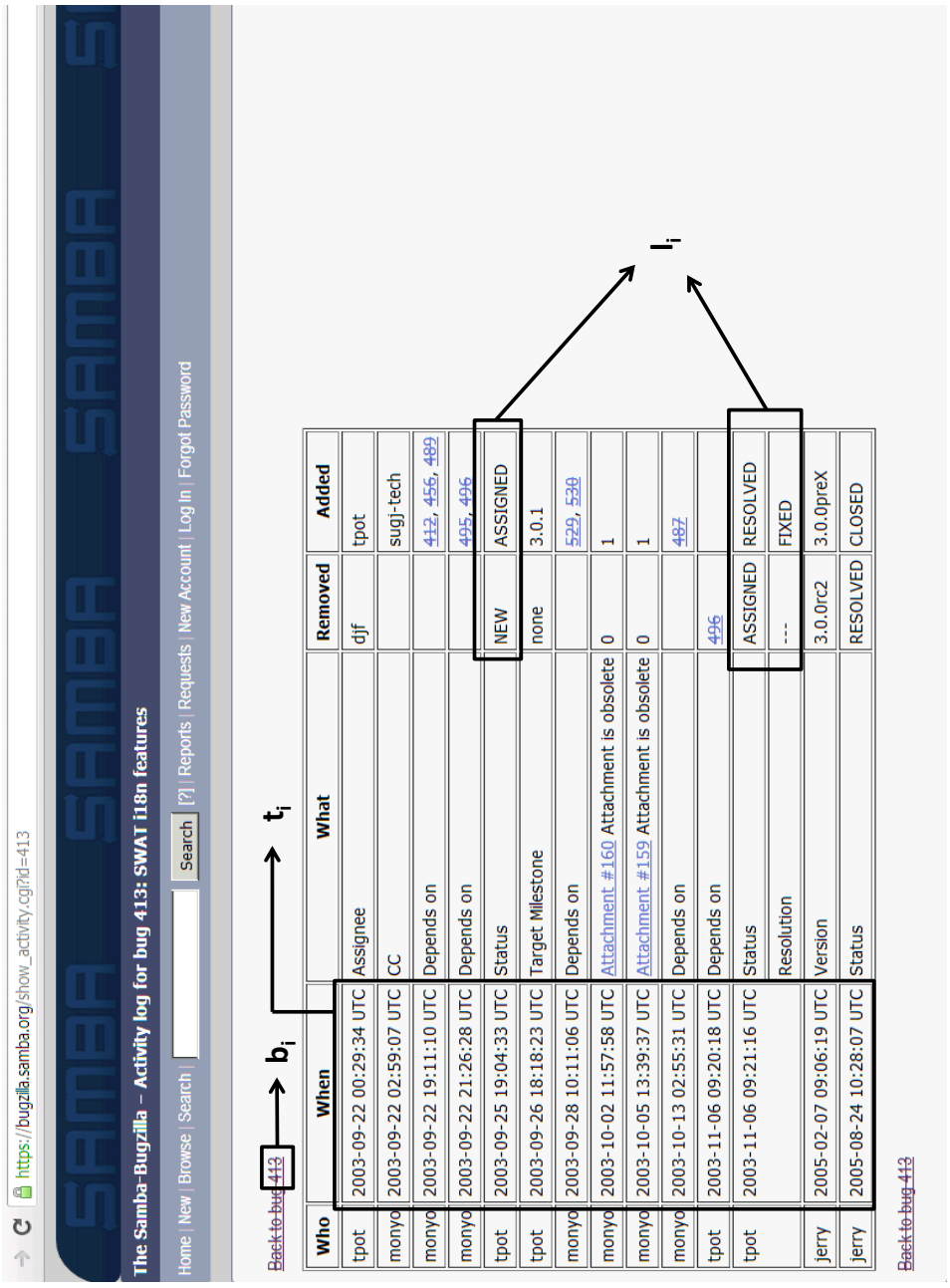

Figure 3.3: An example bug activity screen-shot.

the analysis of *Samba* case study, These are absent from the extended figure of Zaman et al. (2011). The first arrow is added to indicate that a bug can be tossed from the ASSIGNED state to the NEW state during fixing. The second arrow is added to indicate that a bug can reach the state of RESOLVED directly from the NEW state, skipping an intermediate ASSIGNED state. In this latter case, the bug does not need to pass through the ASSIGNED state for resolution. In the legend of the figure, the dotted line and circles are used to represent our added lines from the original figure of Zaman et al. (2011).
The bug status labels in the bug life cycle are used to identify the current health of the bug. From Figure 3.4, one can derive the set of Status labels L, which can include L={ REPORTED, UNCONFIRMED, NEW, ASSIGNED, RESOLVED, VERIFIED, CLOSED, REOPENED }.
A bug is assigned the status of REPORTED when an internal or external user of the system reports a bug at the start of the bug fixing process. The bug is assigned the status label of UNCONFIRMED to check that the reported bug is valid or invalid. If the developers agree that the reported bug is a valid bug then the status of NEW is assigned to the bug. Otherwise, the bug is given the status of RESOLVED to ensure that the bug has been validated by the testing team.
The status label of a bug is changed from NEW to ASSIGNED once the bug is assigned to some developers for fixing. From ASSIGNED status, if the bug is fixed then it is marked as RESOLVED for the testing team to verify the fix. Otherwise, the bug is given the status label of REOPENED either if there is some information missing or the developer does not have enough knowledge to fix it. Once the bug is given the status of RESOLVED, then it is the responsibility of the testing team to verify the fix and assign the status label of VERIFIED if the bug was properly fixed. Otherwise, if the testing team does not agree with the bug resolution, the bug is assigned the status label of REOPENED. All these status changes represents the life cycle of a bug in the `Bugzilla` system as shown in the Figure 3.4.

To retrieve the status labels relevant to triage and fixing durations, three states of a bug's life cycle are important, namely REPORTED, ASSIGNED, RESOLVED. Although obvious in Figure 3.4, these states are not always found in the bug database. For example, when RESOLVED is not found, the status of CLOSED as its placeholder has been used in this thesis. However, the bugs whose "ASSIGNED" state is absent due to one of RESOLVED state via the edges INVALID, DUPLICATE, WONTFIX, or WORKSFORME has not been considered in this thesis. In this thesis,the bugs those come from the transition "FIXED" edge are chosen for the analysis because only then is the fixing time not regarded as zero.

It has been observed during the study that there can be multiple states in a bug's life-cycle. For example, because of reopening, there can be more than one "RESOLVED" state on the original life-cycle diagram in the life-cycle of a bug. For the sake of simplicity, only the latest RESOLVED state to regard all the reopening activities as a somewhat prolonged fixing process has been considered in this thesis. There may be more than one ASSIGNED state due to the reopening as well. In these cases, the very last ASSIGNED state in the trace has been chosen for the bug analysis in this thesis. Formally, the set of bug life cycle status labels C used to retrieve the bug status labels is defined in Definition 2.

Definition 2 *The set of derived bug life cycle : C. The set of bug status is defined as $C = \{REPORTED, ASSIGNED, RESOLVED\}$.*

Algorithm 1 is used to get the derived bug life cycle. The set of bug life cycle labels L given in the bug activity log log_A is input. The set of derived bug life cycle C is output. The procedure starts by initialising the set of derived bug life cycle to the empty set. A multi branch switch statement is used to retrieve the different cases of the set of bug life cycle labels L. In case the element l of the set of bug life cycle labels L is UNCONFIRMED, an element c with the value REPORTED is added in the set of derived bug life cycle C. Similarly, a new element RESOLVED is added in the set of derived bug life

cycle C in case the element l of the set of bug life cycle labels L is equal to RESOLVED. In any other case, the bug is assigned the status label of NEW.

Data: The set of bug status labels $l \in L$

Result: $C = \{REPORTED, ASSIGNED, RESOLVED\}$

```
1  C = {};
2  switch (l ∈ L) do
3      case REPORTED
4          c = REPORTED;
5          break;
6      end
7      case ASSIGNED
8          c = ASSIGNED;
9          break;
10     end
11     case RESOLVED
12         c = RESOLVED;
13         break;
14     end
15 endsw
```

Algorithm 1: ObtainAssignedResolvedStatus : Obtain the set of bugs derived life cycle labels C.

After these pre-processing steps, now each bug is only associated with at most three states. The rather complex state transition diagram on the upper part of Figure 3.4 is now simplified to convert it into the much simplified state model on the lower part of the diagram. With these status labels, it is now sufficient to compute the values of bug event stamps.

In the `Bugzilla` issue tracking system, the time at each stage of bug status change is recorded in the bug activity log. Therefore, the set of derived status labels are used to derive the bug event stamps whenever the bug status is changed. The event stamps of each bug according to the current status of bug resolving keep track of the time whenever a bug status is changed from

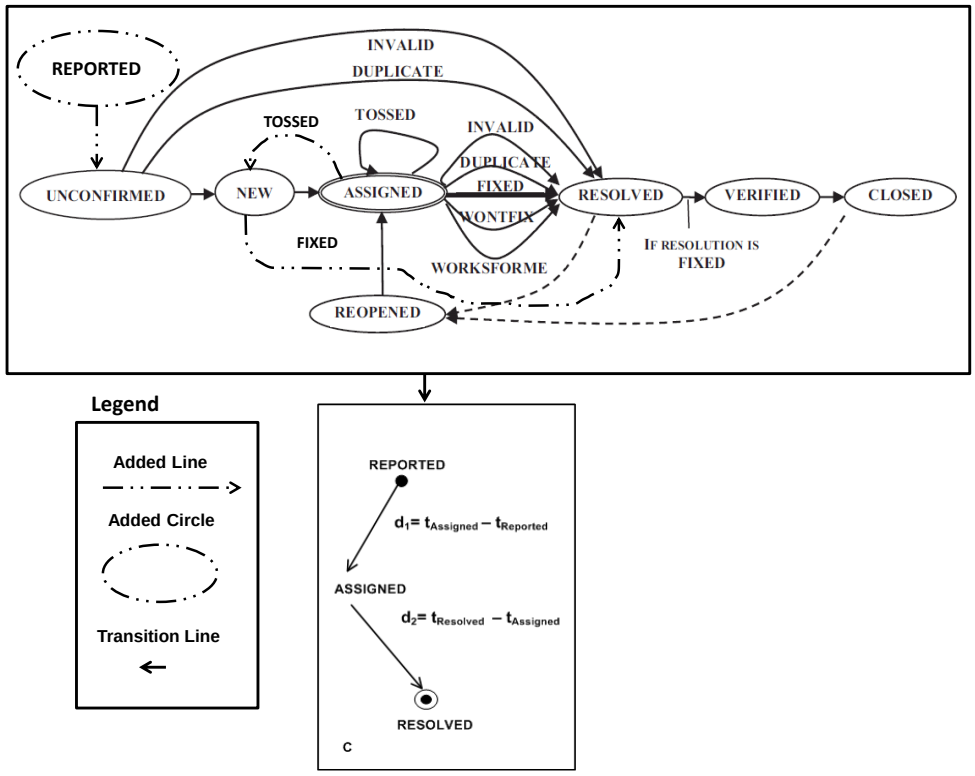

Figure 3.4: Relationship between bug life cycle labels L of Zaman et al. (2011) and the derived bug life cycle labels C.

one to another.

For example, the bug no. 7067 has been reported by the *Samba* developer "Jeremy Allison" at $2010-01-2618:48:57UTC$ in the *Samba* system. The bug has been reported in the product "Samba3.5". For reporting a bug, it is necessary that the bug reporter have a valid account at the `Bugzilla` live system for *Samba*. After login the system with a valid user account, the user must use the option "File a Bug" available at the home page of `Bugzilla` system [1]. The process of reporting a bug starts with selecting a product of the system in which the reporter found the bug. After selecting the right product to report the bug, the reporter needs to fill the bug report form by filling the fields that includes "Component", "Version", "Summary", "Severity", "Hardware", "OS" and "description". The `Bugzilla` live system logs the time of reporting a bug at the time of completing the bug report form by the developer, which is logged as the bug reporting time in the system. Once the bug is reported in the system, the *Samba* developer verifies the

[1] https://bugzilla.samba.org/

bug whether it is a valid by reproducing it based on the description given in the bug report. In the case of aforementioned bug 7067, the same person "Jeremy Allison" has reported and reproduced the bug. The bug reporting guidelines for the *Samba* system has been provided on the following link "https://bugzilla.samba.org/page.cgi?id=bug-writing.html".

By using the similar method of bug reporting the bug 7104 has been marked as reopen by the developer "Jeremy Allison". A bug is marked as reopen if the developer found that an already fixed bug is the source of problem in the program. The reopened bug is not marked as "REPORTED" and "NEW" because it is not a new problem. Therefore, it is directly assigned to a developer or to a group of developers for fixing. An arrow from the "RESOLVED" stage to "REOPENED" and from "REOPENED" stage to 'ASSIGNED" stage is indicating the phenomenon of bug reopening in the `Bugzilla` bug life cycle figure 3.4. It is observed that the example reopened bug 7104 has been directly assigned to developer "Jeremy Allison" for fixing rather than marking it as "REPORTED" and "NEW".

After recovering the bug resolution event stamps, it is possible to measure the actual triage durations. The set of bug event stamps is defined in Definition 3.

Definition 3 *The set of bug event stamps : E. The set of bug event stamps E is defined as:* $E = \{(b_i, (c_i, t_i)) \mid 1 \leq i \leq n \wedge n \in \mathcal{N}\}$.

Algorithm 2 is used to retrieve the bug event stamps. The set of bug activity log log_A, and the set of derived bug status is input. The set of bugs event stamps E according to the derived status is output. The algorithm starts by initialising the set of event stamps E to the empty set, then for each element of the set of bug event stamps E. The value of each bug status label l_i is obtained using the algorithm 1.The obtained value is assigned to the element c_i of the set of derived bug life cycle labels C. The value of bug id b_i, derived life cycle label c_i, and the time stamp for one particular event t_i is assigned to the set of event stamps.

Data: The set of bug activity log $log_A = \{(b_i, (l_i, t_i))\}$, and the set of derived bug life cycle C

Result: $E = \{(b_i, (c_i, t_i))\}$

```
E = {};
foreach (b_i, (l_i, t_i)) ∈ log_A do
    c_i = ObtainBugsDerievedStatus(l_i);
    E = E ∪ {(b_i, (c_i, t_i))}
end
```

Algorithm 2: ObtainBugEventStamps: Obtain the set of bug fixing event stamps E.

The event stamps of each bug according to the current status of bug resolution keep track of the time whenever a bug status is changed from one to another. Therefore, after recovering the bug resolution event stamps, it is possible to measure the actual triage and fixing durations. The set of bug triage and fixing durations τ_1 and τ_2 are defined in Definition 4

Definition 4 *The set of bugs' traige durations :* τ_1*, and the set of bugs' fixing durations:* τ_2*. The set of bugs' triage durations* $\tau_1 = \{(b_i, d_{1i}) \mid 1 \leq i \leq n \wedge n \in \mathcal{N}\}$*., where* $b_i \in B$ *and* $d_{1i} \in \tau_1$*. The set of bugs' fixing durations* $\tau_2 = \{(b_i, d_{2i}) \mid 1 \leq i \leq n \wedge n \in \mathcal{N}\}$*, where* $b_i \in B$ *and* $d_{2i} \in \tau_2$*.*

Algorithm 3 is used to get the set of bug triage τ_1 and fixing τ_2 durations. The input is the set of bug event stamps E. The output is the set of bug triage τ_1 and fixing τ_2 durations sets. The procedure starts by initialising the set of bug triage τ_1 and fixing τ_2 durations to the empty set, then for each bug b in the set of bug event stamps E. The value of derived bug life cycle label c is checked using switch statement. In case the value of variable c is REPORTED then the value of life cycle time t_i is assigned to the variable $t_{REPORTED}$. In case the value of variable c is ASSIGNED then the value of time t_i is assigned to the variable $t_{ASSIGNED}$. Similarly, in case the value of c is RESOLVED then the value of time t_i is assigned to the variable $t_{RESOLVED}$. Then the bug triage duration d_1 is measured by subtracting the time when the bug is assigned to developers for fixing $t_{ASSIGNED}$ to the time when the

bug is reported $t_{REPORTED}$. Likewise, the duration of bug fixing time d_2 is measured by subtracting the time when the bug is resolved $t_{RESOLVED}$ from the time when the bug is assigned to the developers for fixing $t_{Assigned}$. The computed bug triage d_1 and fixing d_2 durations are assigned to the bug triage and fixing durations sets τ_1 and τ_2 respectively.

Data: The set of bug event stamps according to the bug status labels E

Result: $\tau_1 = \{(b, d_1)\}$, $\tau_2 = \{(b, d_2)\}$

```
1  τ1 = {};
2  τ2 = {};
3  foreach (b ∈ log_B) do
4  |   foreach
   |   (c ∈ {REPORTED, ASSIGNED, RESOLVED}(b, (c, t)) ∈ E)
   |   do
5  |   |   if (b, (c, t)) ∈ E then
6  |   |   |   t_c = t;
7  |   |   end
8  |   end
9  |   if (t_ASSIGNED ∧ t_REPORTED) then
10 |   |   d1 = t_ASSIGNED − t_REPORTED;
11 |   |   τ1 = τ1 ∪ {(b, d1)};
12 |   end
13 |   if (t_RESOLVED ∧ t_ASSIGNED) then
14 |   |   d2 = t_RESOLVED − t_ASSIGNED;
15 |   |   τ2 = τ2 ∪ {(b, d2)};
16 |   end
17 end
```

Algorithm 3: ObtainBugDuration: Obtain the bug triage and fixing durations τ_1 and τ_2.

Through the above pre-processing steps, the values of bug triage d_1 and fixing d_2 durations has been calculated. Now, it is possible to measure the value of

planned rush in the triaging of bugs.

3.3 Model for Measuring the Weekly Release Fix Triage Durations

In this section, the model to measure the weekly release fix triage duration has been explained for considering the situation when the developers have time to plan the work. Figure 3.5 describes the model of measuring bug triage duration based on the weekly release fix. In this model, it is assumed that a bug can be planned in a release as long as it is reported before the delivery of a release. In this way, our model measures the time from when the bug is reported until it is released, which is the exact time spent before the release of a bug. It includes the bug fixing time and any waiting time before the release of the bug. For measuring the triage duration according to the weekly release fixes, the status labels C' has been used in the thesis, which is defined as the $C' = \{REPORTED, PLANNED, RESOLVED\}$. Figure 3.5 shows the relationship between the simplified life cycle of bugs to the planned bug status labels C'. The left hand part of Figure 3.5 is relevant to the set of derived bug life cycle status labels C. The right hand part of Figure 3.5 is relevant to the set of planned bug life cycle status labels C'. This relationship is used to compute the triage and fixing durations at the time of planning (represented by the d'_1 d'_2 respectively), although the planned triage duration d'_1 to compute the value of weekly release fix durations has been used. Before proceeding to compute the value of weekly release fix triage duration, it is required to have the knowledge about the release date of each bug. Formally, the set of release dates of bugs is defined in Definition 5.

Definition 5 *The set of weekly releases dates : R_τ. The set of weekly release dates R_τ is defined as : $R_\tau = \{(b_i, t_{release}) \mid 1 \leq i \leq n \wedge n \in \mathcal{N}\}$.*

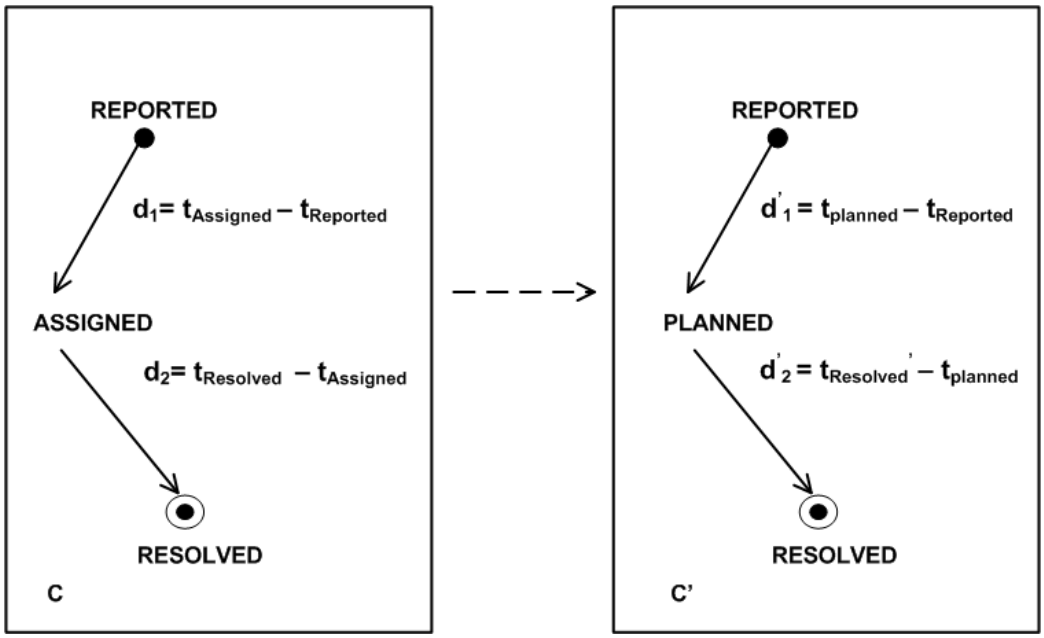

Figure 3.5: The relationship between bug derived life cycle C to the planned bug life-cycle C'.

https://bugzilla.samba.org/page.cgi?id=fields.html#see_also

The **status** and **resolution** fields define and track the life cycle of a bug.

STATUS	RESOLUTION
The **status** field indicates the general health of a bug. Only certain status transitions are allowed.	The **resolution** field indicates what happened to this bug.
UNCONFIRMED This bug has recently been added to the database. Nobody has validated that this bug is true. Users who have the "canconfirm" permission set may confirm this bug, changing its state to NEW. Or, it may be directly resolved and marked RESOLVED. **NEW** This bug has recently been added to the assignee's list of bugs and must be processed. Bugs in this state may be accepted, and become **ASSIGNED**, passed on to someone else, and remain **NEW**, or resolved and marked **RESOLVED**. **ASSIGNED** This bug is not yet resolved, but is assigned to the proper person. From here bugs can be given to another person and become **NEW**, or resolved and become **RESOLVED**. **REOPENED** This bug was once resolved, but the resolution was deemed incorrect. For example, a **WORKSFORME** bug is **REOPENED** when more information shows up and the bug is now reproducible. From here bugs are either marked **ASSIGNED** or **RESOLVED**.	No resolution yet. All bugs which are in one of these "open" states have the resolution set to blank. All other bugs will be marked with one of the following resolutions. **FIXED** A fix for this bug is checked into the tree and tested. **INVALID** The problem described is not a bug. **WONTFIX** The problem described is a bug which will never be fixed. **DUPLICATE** The problem is a duplicate of an existing bug. Marking a bug duplicate requires the bug# of the duplicating bug and will at least put that bug number in the description field. **WORKSFORME** All attempts at reproducing this bug were futile, and reading the code produces no clues as to why the described behavior would occur. If more information appears later, the bug can be reopened. **MOVED** The problem was specific to a related product whose bugs are tracked in another bug database. The bug has been moved to that database.
RESOLVED A resolution has been taken, and it is awaiting verification by QA. From here bugs are either re-opened and become **REOPENED**, are marked **VERIFIED**, or are closed for good and marked **CLOSED**. **VERIFIED** QA has looked at the bug and the resolution and agrees that the appropriate resolution has been taken. Bugs remain in this state until the product they were reported against actually ships, at which point they become **CLOSED**. **CLOSED** The bug is considered dead, the resolution is correct. Any zombie bugs who choose to walk the earth again must do so by becoming **REOPENED**.	

Figure 3.6: The bug status and resolution description.

Algorithm 4 is used to get the set of weekly release dates. The input is the set of bug event stamps E. The output is the set of weekly release dates R_τ. The procedure starts by initialising the set of bug event stamps E to the empty set. Then for each value of the set of bug event stamps E, the condition is checked that the resolved time of bug $t_{RESOLVED}$ should not be null to ensure that each bug has already been resolved. If the bug has already been resolved then the fix of bug is delayed to the latest release date rather than the same day by allowing the delay of three days. The delay of three days has been added because it is assumed that "Wednesday" as the release day of every week starting from "Monday". The retrieved release date of bug has been assigned to the release date variable $t_{release}$. Finally, the bug id b and the release date $t_{release}$ are added to the set of releases dates R_τ.

Data: The set of bug event stamps E

Result: $R_\tau = \{(b_i, t_{release}) \mid 1 \leq i \leq n \wedge n \in \mathcal{N}$

```
1 R_τ = {};
2 foreach (b, (RESOLVED, t_RESOLVED)) ∈ E do
3 |   if t_RESOLVED ≠< NULL > then
4 |   |   t_release = {t_resolved + ||7 − (t_resolved − 3), 7|, 7|};
5 |   |   R_τ = R_τ ∪ {(b, t_release)};
6 |   end
7 end
```

Algorithm 4: ObtainWeeklyReleaseDates: Obtain the set of weekly release dates R_τ.

After determining the release dates for each bug, now it is possible to measure the triage duration of a bug in case the developer have more time to triage the bug during the planning phase of the work (b, d'_1). The set of precise triage duration based on weekly release fix is defined in Definition 6.

Definition 6 *The set of precise triage duration based on the weekly release fix : τ'_1. The set of precise triage duration based on the weekly release fix is defined as: $\tau'_1 = \{(b_i, d_1{}'_i) \mid 1 \leq i \leq n \wedge n \in \mathcal{N}\}$.*

Algorithm 5 computes the bugs triage duration based on the bugs fixed in weekly releases. The input is the set of bugs and their weekly release fix dates R_τ, the set of bug event stamps E, and the set of bug fixing durations τ_2. The output is the set of weekly release fix triage durations ${\tau_1}'$. The procedure starts by initialising the set of weekly release fix triage durations ${\tau_1}'$ to the empty set. Then for each bug of $(b, t_{release})$ in the set of bugs fixed in the weekly releases R_τ, the $(b_i, (REPORTED, t_{reported}))$ in the set of E bug event stamps, and the (b, d_2) in the set of τ_2 bug fixing duration. The resolved date $t_{resolved}$ and the fixing duration d_2 of bug has been deducted from the release date $t_{release}$ to get the triage duration ${d_1}'$ spent by the developers when they have time to delay the triaging. The weekly release fix triage duration d'_1 is then added to the set of precise triage durations ${\tau_1}'$.

Data: The set of weekly released dates : R_τ, and the set of bug event stamps E

Result: ${\tau_1}' = \{(b_i, {d_1}')\}$

```
1 τ1' = {};
2 foreach (b, t_release) ∈ R_τ,(b_i, (REPORTED, t_reported)) ∈ E,(b, d_2) ∈ τ_2
  do
3 |   d1' = t_release − t_resolved − d_2;
4 |   τ1' = τ1' ∪ {(b_i, d1')};
5 end
```

Algorithm 5: ComputeWeeklyReleaseFixTriageDuration: Compute the set of weekly release fix triage durations ${\tau_1}'$.

After pre-processing of bugs fixed in the weekly releases, one can compute the rush in the triaging of bugs.

3.4 Measuring Rush in the Triaging of Bugs

In this section, the value of rush for the triaging of bugs has been computed. The planned rush in the triaging of bugs is a median ratio of the bug's actual triage duration d_1 and the delayed release triage durations ${d'}_1$. To verify

our hypothesis 1, the rush in triaging of all the bugs (resp. security and non-security) bugs as $B = B_s$ (resp. $B = B_{\bar{s}}$) has been measured, which is defined in Definition 7.

Definition 7 *The planned rush in the triaging of the bugs is :* $\lambda(B)$. *The rush in triaging of bugs is defined as:* $\lambda(B) = \tilde{M}\frac{(b,d_1)\in\tau_1}{(b,d'_1)\in\tau'_1}$

$$\lambda(B) = \tilde{M}\frac{d_1}{d'_1} \tag{3.1}$$

Equation 3.1 is used to get the rush in triaging of bugs $\lambda(B)$. The ratio is retrieved by dividing the median of actual triage duration $(b, d_1) \in \tau_1$ to the median of precise measured triage duration of the bugs $(b, d'_1) \in \tau'_1$.
The median duration is chosen for calculating the ratio because there can be outliers that distort the average of the data points.
Formally, the hypothesis 1 is restated as the set of security bugs B_s and non-security bugs $B_{\bar{s}}$, $\lambda(B_s) < \lambda(B_{\bar{s}})$.

3.5 Applying the rush metric to the case studies

In this section, the technique to measure rush is applied to case studies and values of rush are statistically tested.

3.5.1 Application of the technique

The median value of rush in the triaging of CVE relevant bugs for *Samba*, *Mozilla Firefox*, *Red Hat*, *FreeBSD* and *Mozilla* case studies is 0.002, 0.0042, 0.024, 0.002 and 0.0017 respectively. On the the other hand, the median value of rush in the triaging of non-CVE relevant bugs for the above case studies in the same order is 0.088, 0.0044, 0.091, 0.5023 and 0.0369 respectively.

Graphically, the values of rush are represented in Figure 3.7. In the figure 3.7, the y-axis represents the median value of rush in days and the name of case studies are shown at the x-axis. The bar charts with dark red color represents CVE relevant and the bar charts with blue color represents the Non-CVE relevant bugs in figure 3.7.
The results of all these case studies shows that there is a rush in the triaging of CVE relevant bugs compared to the non-CVE relevant bugs. Hence, one can draw conclusion that security bugs are rushed during triaging in the above case studies. The successful application of the technique to measure rush the time management behaviour of software developers is answering the first part (RQ1.1) of research question RQ1. Apart from the obvious difference between the rush values of CVE and Non-CVE bugs, one may also observe that *FreeBSD* has least median rush ratio for non-CVE bugs. One possible explanation is that the enterprise vendors behind *Samba*, *Mozilla Firefox*,*Mozilla* and *Red Hat* might give their developers more pressure in resolving bugs in general. Despite from this, it worth noting that *FreeBSD* has the tightest rush ratio for the CVE bugs. A possible reason is that *FreeBSD* products include primarily a dependable operating system that is more sensitive to security breaches. In contrast, *Red Hat* and *Mozilla Firefox* maintains many lines of products that may not be all as sensitive to security as its Enterprise Linux product line.

3.5.2 Statistical verification of the results

To statistically verify the rush in triaging of security bugs for all the case studies, the non-parametric *Mann Whitney U-test* has bee applied the CVE and non-CVE relevant bugs of all the case studies. The parametric and non-parametric are two types of statistical tests available to verify the samples from the population of data. Usually, the parametric tests are used for the data when the assumptions are made about the parameters of the population's data distribution. However, no assumptions are made in the

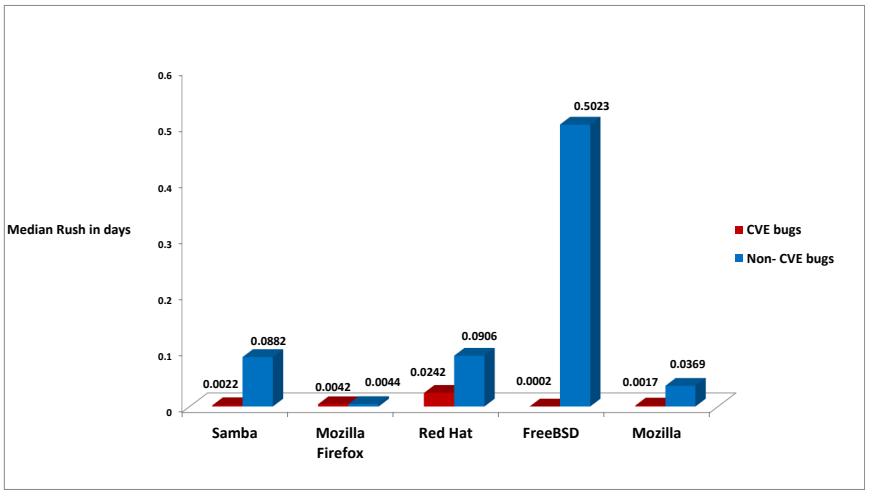

Figure 3.7: Rush in the triaging of CVE and Non-CVE bugs in the case studies for the *Samba*, *Mozilla Firefox*,*Red Hat*, *FreeBSD*, *Mozilla*.

non-parametric category of statistical tests about the distribution of data. In other words using the parametric test, difference between the populations

is determined based on the assumption that the data is normally distributed. But no assumptions are made on the population of data while using the non-parametric test. Usually, the standard Quintile Regression (QQ-plot) is used to determine the distribution of data. The rule is that in case the data is symmetric and plots on the straight line then data is normally distributed. The QQ-plot has been drawn for the security and non-security bugs of all the case studies. The results showed that the security and non-security bugs' data is not normally distributed. Therefore, the non-parametric category of statistical tests has been chosen to evaluate the difference between the samples of all the case studies.

In the non-parametric category of statistical tests, the *Mann Whitney U-test* is used to evaluate the difference between the population of two independent data sets based on their median values Kasuya (2001). In our case, the CVE relevant bugs are different from the non-CVE bugs, which means both data sets are different from each other. Therefore, the *Mann Whitney U-test* has been chosen to evaluate whether the values of rush for security bugs for all the case-studies are significantly different from the non-CVE relevant bugs for all the case studies.

The results of statistical test showed that the security bugs are rushed in the *Red Hat*, *FreeBSD* and *Mozilla* case studies with the statistical significance. However, the *Samba* and *Mozilla Firefox* case studies are inclusive. Our results are based on the probability significance $p - value$ obtained using the *Mann Whitney U-test* as described below. The $p - values$ of *Red Hat*, *FreeBSD*, *Mozilla*, *Samba* and *Mozilla Firefox* are 0.00001, .000, and .001, 0.008291, 0.692843 respectively. The results of statistical test proves that statistically there is a rush in the triaging of CVE bugs compared to the Non-CVE bugs. The purpose of bug triaging is to prioritise and assign bugs to the developers for fixing. Therefore, the statistical results verify that security bugs are rushed to triage, which is one of the reasons of their premature assignment and subsequently reopening. The calculation of rush on

the triaging of bugs considers the first reporting time-stamp and last assignment time-stamps. In this way a bug which reopens and reassigns is also considered in the calculation of rush. Therefore, the conclusion about the reopening of bugs is based on the values of rush. The statistical results of rush answers the second part (RQ1.2) of the first research question RQ1.
The statistical significance results for all the cases studies (*Samba*, *Mozilla Firefox*, *Red Hat*, *FreeBSD* and *Mozilla*) are shown in Figure 3.8, 3.9, 3.10, 3.11 and 3.12 respectively. The each above figure is an output for the statistical test *Mann Whitney U-test* showing two tables. The first table explains the number of bugs belonging to CVE and Non-CVE groups and their values of mean rank and sum of ranks. In the group column of first table, the first row 1.0 represents the CVE bugs and the second number 2.0 represents the Non-CVE group of bugs. The second column is representing the number of bugs in each group (CVE and Non-CVE) respectively. For example, the first row of second column is representing that there are 9 CVE bugs in the *Samba* case study. On the other hand, the second row is representing that there are 1702 Non-CVE bugs in the *Samba* case study as shown in Figure 3.8. The third column of the first table is representing the mean rank for the CVE and Non-CVE bugs. For example in the case of *Samba* case study, the mean rank of CVE bugs is 422.33 and Non-CVE 858.29 respectively as shown in Figure 3.8. The forth column of the first table is representing the sum of ranks for CVE and Non-CVE bugs. For example, the sum of ranks for the CVE bugs is 3801.00 and for the Non-CVE bugs is 1460815.00 as shown in the first and second row of table in Figure 3.8.
The second table shows the derived statistics of λ values. In the table, the row with the "*Mann-Whitney U*" represents the difference between the totals of CVE and non-CVE relevant bugs. The row "*Wilcoxon W*" represents the ranked sum of the smaller sample size of the data, which is security bugs for our case studies. The row "*Z*" represents the *z-values* for each case study. Usually, the "*z-value*" can be used to reject the null hypothesis instead of the

p-value. However, the *p-value* has been used in the analysis which represents by *Asymp. Sig. (2-tailed)* row. The definition of *p-value* is given in Appendix 5.2

Samba Ranks

	Group	N	Mean Rank	Sum of Ranks
Bugs	1.0	9	422.33	3801.00
	2.0	1702	858.29	1460815.00
	Total	1711		

Samba Test Statistics[a]

	Bugs
Mann-Whitney U	3756.000
Wilcoxon W	3801.000
Z	-2.640
Asymp. Sig. (2-tailed)	.008

a. Grouping Variable: Group

Figure 3.8: The results of *Mann Whitney U-test* for *Samba* case study from SPSS package.

Mozilla Firefox Ranks

	Group	N	Mean Rank	Sum of Ranks
Bugs	1.0	14	14770.36	206785.00
	2.0	31444	15729.93	494611826.0
	Total	31458		

Mozilla Firefox Test Statistics[a]

	Bugs
Mann-Whitney U	206680.000
Wilcoxon W	206785.000
Z	-.395
Asymp. Sig. (2-tailed)	.693

a. Grouping Variable: Group

Figure 3.9: The results of *Mann Whitney U-test* for *Mozilla Firefox* case study from SPSS package.

Red Hat Ranks

	Group	N	Mean Rank	Sum of Ranks
Bugs	1.0	1368	2832.20	3874455.00
	2.0	5508	3589.08	19768671.00
	Total	6876		

Red Hat Test Statistics[a]

	Bugs
Mann-Whitney U	2938059.000
Wilcoxon W	3874455.000
Z	-12.622
Asymp. Sig. (2-tailed)	.000

a. Grouping Variable: Group

Figure 3.10: The results of *Mann Whitney U-test* for *Red Hat* case study from SPSS package.

FreeBSD Ranks

	Group	N	Mean Rank	Sum of Ranks
Bugs	1	519	8096.87	4202276.00
	2	64171	32541.62	2088228119
	Total	64690		

Test Statistics[a]

	Bugs
Mann-Whitney U	4067336.000
Wilcoxon W	4202276.000
Z	-29.701
Asymp. Sig. (2-tailed)	.000

a. Grouping Variable: Group

Figure 3.11: The results of *Mann Whitney U-test* for *FreeBSD* case study from SPSS package.

Mozilla Ranks

	Group	N	Mean Rank	Sum of Ranks
Bugs	1	22	74.59	1641.00
	2	219	125.66	27520.00
	Total	241		

Test Statistics[a]

	Bugs
Mann-Whitney U	1388.000
Wilcoxon W	1641.000
Z	-3.275
Asymp. Sig. (2-tailed)	.001

a. Grouping Variable: Group

Figure 3.12: The results of *Mann Whitney U-test* for *Mozilla* case study from `SPSS` package.

3.6 Tool Support

To answer the first research question (RQ1) for measuring rush on the triaging of bugs, an automated tool has been developed. The tool has two components "Data Extraction" and "Data Analysis".

Data Extraction: In the data extraction, the first phase is to extract the bug ids from the `Bugzilla` live system by conducting a query based advance search. The query based search has been conducted to find-out the CVE and non-CVE relevant bug ids for extracting the bug reports later in the second

phase of data extraction. The CVE relevant bug ids has been retrieved by using the keyword "CVE-". On the other hand, no keyword has been used to retrieve the list of all the bugs. The searched bugs has been downloaded in the comma-separated values (CSV) format. There is a possibility that some CVE relevant bugs might exists in the list of all the bugs. Therefore, the difference of the list of CVE and all the bugs has been taken to filter out the CVE bugs from the list of all the bugs. After the filtration of CVE bugs from the list of all the bugs, the list of all the bugs has been considered as non-CVE relevant bugs. In the second phase of data extraction, the bug reports and their associated bug activity reports relevant to CVE and non-CVE bug ids has been retrieved. For retrieving the bug reports, the automated script A.1 has been used. On the other hand, for retrieving the associated activity reports with the bug reports A.2 has been used. The bug reports have been processed to get the reported, assigned and resolved time-stamps of bugs. The scripts A.3 and A.4 has been used for retrieving the bug time stamps data. Each script has been explained in detail in the appendix A.

Data Analysis: The date and time of reported, assigned and resolved time-stamps of bugs has been collected separately during the data extraction. Therefore, first the date and time of bug reported, assigned and resolved time-stamps combined together before further pre-processing. After that all the time-stamps has been stored in the set of bug event-stamps using the Algorithm 2. The set of bug event-stamps contains the time-stamps when each bug is reported, assigned and resolved. After getting the bug event-stamps, the Algorithm 3 has been used to get the actual triaging and fixing durations of bugs. As rush is the ratio of actual triage duration and the theoretical triage duration (time developers have to delay the fixes until the next regular release). Therefore, to calculate the theoretical triage duration, the weekly release dates of each bug has been calculated using the Algorithm 4. Then, the theoretical triage duration has been calculated using the Algorithm 5. After these pre-processing steps to calculate the actual and theoretical triage

durations, the rush has been calculated by taking the ratio of actual and theoretical triage durations for each bug. Finally, to compare the rush on the triaging of CVE and non-CVE bugs for each case study, the median rush has been taken using the Equation 3.1.

Advantages and the limitations of the tool:

The following are advantages of the tool to measure 'rush' in the triaging of bugs.

- The algorithms for data extraction and analysis phase has been provided. Therefore, it is easier to apply the technique to measure rush to the more case studies. It also makes easier to repeat the case studies conducted in this thesis to confirm the results.
- The data extraction phase of the tool is fully automated and the `AWK` scripts to extract the bug reports has been provided in Appendix A.

The following are limitations of the tool to measure 'rush' in the triaging of bugs.

- The human involvement is necessary for searching the bug ids from `Bugzilla` live systems during the data extraction phase.
- For some of the case studies, e.g. *Mozilla*, the `Bugzilla` live systems has limitation to show the number of search records. Therefore, there is a possibility of missing some bug reports during the data extraction phase.
- For some of the case studies, e.g *Mozilla Firefox*, the `Bugzilla` live systems has limitation to show the number of search records on the screen, therefore a manual yearly search is needed to find the yearly records of data and then such yearly records are integrated.

3.7 Threats to Validity

The following validity threats to this empirical case study has been discussed.

Internal Validity: The collected data is based on what an external developer could gather. The decision to label security bugs depends on the external security experts who maintain the CVE database for the public, while the developers inside each team must decide for themselves whether they have fixed these high-profile security problems. The number of CVE bugs (9, 14, 22) for the *Samba*, *Mozilla Firefox* and *Mozilla* case studies are comparatively smaller than the Non-CVE bugs (1702, 31444, 219) respectively. One can argue that the CVE and Non-CVE bugs are not comparable in terms of the number of bugs. Therefore, the median value of rush is taken to counter this threat to validity.

Construct Validity: The rush measurement approach is repeatable for the other open-source projects as long as they use `Bugzilla` to report bugs. However, the algorithms for the measurement of rush are applicable to the bug duration data set of any issue tracking system. All the five vendors in the study have security sensitive products, their `Bugzilla` data sets have different number of CVE-related bugs. Also due to the difference in documenting the status and time-stamps of the bugs, not all of their bugs are sampled to compute the rush ratio. However, it is sufficient to draw meaningful conclusions for checking the hypothesis.

In this study, rush the time management behaviour of software developers is measured by calculating the time spent by them to assign the bugs. To measure rush, the past data of bug assignment by the software developer is used. Therefore, one can argue about the validity of results due to the lack of the direct involvement of software developers in the study. Usually, there is is no control on the developers in the open source projects because they work on voluntary basis. Therefore, confirming the results of study by involving software developers is the out of scope of this study. But the technique to measure rush is well explained in this thesis. Therefore, the other researchers can easily repeat this study to verify the results. Taking the opinion of software developers about the results of this study is one of

the future works.

External Validity: This study intends to measure rush the time management behaviour of software developer for the security and non-security bugs. However, different bug reporting systems have different mechanism of tagging the bugs as security and non-security relevant. There was a risk that wrong tagging of bugs as either security and non-security related can pollute the results. Therefore, we have used the CVE and Non-CVE confirmed security and non-security bugs to measure rush.

In this thesis, five different security critical case studies has been conducted to measure rush the time management behaviour of software developers. Therefore, there is sufficient evidence about generalising the conclusion that statistically rush is one of the reasons of security bugs premature assignment and subsequently reopening.

3.8 Summary

In this Chapter, one of the contributions of this thesis "a technique to measure rush the time management behaviour of software developers" has been introduced. The technique to measure rush is based on the concept of delivering bug fixes in the weekly releases. The technique is applied to the five case studies *Samba*,*Mozilla Firefox*, *Red Hat*, *FreeBSD* and *Mozilla*. The median value of rush on the triaging of CVE and Non-CVE bugs has been calculated. It is found that the median 'rush' on the triaging of CVE bugs is higher for all the case studies compared to the Non-CVE bugs. To further statistically verify the null hypothesis 1, the *Mann Whitney U-test* has been applied on the median values of rush for all the case studies. The results of test shows that statistically security bugs are triaged in a rush in *Red Hat*, *FreeBSD* and *Mozilla* case studies. However, there is no statistical significant difference between the rush values of CVE and Non-CVE bugs for the *Samba* and *Mozilla Firefox* case studies.

As in this thesis the CVE bugs are used as the representative of security bugs and the Non-CVE bugs as the representative of Non-security bugs. Therefore, based on the results of statistical test, the null hypothesis1 has been rejected for the three case studies *Red Hat*, *FreeBSD* and *Mozilla*.

Table 3.1: Gaps filled by answering the first research question (RQ1).

Gap reference	**Gap**	**How the gap is filled**

$SB1$	There is a need to investigate whether rush by the software developers on the triaging of security bugs is one of the reasons of their premature assignment and then frequent reopening.	The purpose of bug triaging is to assign bugs to the other developers for fixing. Therefore, if a bug reopens it means the bug also need to reassign for fixing. In fact, the reassignment of a bug leads to the longer triage duration. In this study, rush is defined as the ratio of actual and theoretical triage durations. The shorter triage duration means less rush and on the other hand longer triage duration means more rush. This study has found that the statistical value of rush of security bugs for the *Red Hat*, *FreeBSD* and *Mozilla* case studies is higher than the non-security bugs. Therefore, in this study the conclusion is drawn from the finding that security bugs reopens more often because of rush by the software developers on the triaging in *Red HatFreeBSD* and *Mozilla* case studies. Therefore, this gap has been filled by the study.

$BR1$	There is a need to quantitatively analyse how the developers triage bugs during their resolution.	This study has measured rush by calculating the duration spent by developers on triaging. However, quantitative analysis can be done on the different aspects of bug triaging. Therefore, this gap has partially filled by this study.
$BR2$	There is a need to measure rush during bug triaging considering the practical handling of bugs by the software developers near to the release deadlines.	In this study, the rush is defined as a ratio of actual and theoretical triage durations. The theoretical triage duration consider the time developers have to delay the assignment of a bug fix in the weekly release. Therefore, this study has filled this gap.
$SD1$	There is a need to measure rush the time management behaviour of software developers on the triaging of bugs	This study has measured rush the time management behaviour of software developers on the triaging of bugs. Therefore, this gap has been filled by this study.
$SD2$	There is a need to verify whether rush the time management behaviour of software developers on the triaging of bugs can be measured by calculating the time spent by them on bug assignment.	This study has measured rush by the software developers by calculating the time spent by them on the assignment of bugs. Therefore, this study has filled this.

$SD3$	There is a need to verify whether rush the time management behaviour of software developers can be one of the reasons of premature bug assignment and subsequently their reopening.	This gap is related to the gap SB1.
$SA1$	There is a need to extract the bug status time stamps when the bug is reported till it is assigned to developers for measuring rush the time management behaviour of developers on bug triaging.	In this study, the bug status time-stamps has been extracted from the time when the bug is reported till it is assigned to developers for measuring rush the time management behaviour of software developers. Therefore, this study has filled this gap.
$SA2$	There is a need to extract the bug time stamps considering the reopening time of bugs.	In this study, the status time-stamp of a bug is extracted when the bug is first reported and last assigned, which include the reopen time stamp as well. Therefore, this study has filled this gap.

Chapter 4

Measuring Code Complexity

The previous chapter reports that on average a security bug is triaged in a rush compared to an average bug. However, the rush in the triaging of security bugs does not mean that such types of bug are simpler or easier to fix. The real time developers spend on such types of bug may vary and is very difficult to know and measure. Another way to know that such types of bug are easier or complex to fix is to measure their code complexity.
There are many ways to measure the complexity of code, ranging from simple SLOC to McCabe. Concerning changes, however, it is not clear whether touching one line of code is more complex than touching another, and whether the change to the control flow is indeed more complex than a change that does not introduce new control flows. Estimating the best complexity metric is beyond the scope of this thesis. Our aim is to obtain measurements that are suitable for verifying the hypothesis. Since the focus of this thesis is on security bugs, a metric based on the evidence that security aspects tend to crosscut many functions, and therefore have more fan-ins (Marin et al., 2004) is chosen for measuring complexity of security bugs in this thesis. Based on the literature evidence, the hypothesis is that security bug fixes also may touch more functions and have larger fan-ins.

RQ2: How can the complexity of a bug fix be measured from the code repositories? Is the complexity of a security bug fix higher on average than the complexity of a non-security bug?

In order to address the "how" part of RQ2, a technique to measure fan-ins of functions relevant to bug fixes is developed and explained in this Chapter. On the other hand, to address the "yes and no" part of the RQ2, the following null hypothesis is derived from the RQ2.

Hypothesis 2 *On average the fan-in of changed functions in the patches of security bugs is not higher than the fan-in of changed functions in the patches of Non-security bugs.*

4.1 Extracting Call Graph of the System

In order to check the above hypothesis, there is a need to compute the fan-in of all the functions, and then the fan-in of changed functions by obtaining first a call graph. Formally, a call graph as a directed graph is represented by $G =< V, E >$, where V is a set of functions, and E is the call relationship between a calling function to a "callee" function as shown in Definition 8.

Definition 8 *Directed graph $G =< V, E >$: A digraph is a type of graph to represent the objects in a specific order, where objects are referred to as nodes and their ordered pairs of nodes are referred to as edges. Usually, it is represented as $G =< V, E >$, where $V = \{v_i \mid 1 \leq i \leq n \wedge n \in \mathcal{N}\}$ is a set of vertices and $E = \{(v_i, v_j) \mid 1 \leq i, j \leq |V| \wedge i, j \in \mathcal{N}\}$ is a set of edges. Here $\mathcal{N}$ is the set of natural numbers.*

The *Samba* case study is implemented in C/C++ language. To obtain the call graph for such a system, A Google search has been conducted to find static source code analysis tools for C-program. For example, keywords such as "C fan-in analysis tools", "call graph tools" and "C static source code analysis tools" are used. The keyword "call graph tools" returned link to

the stack overflow forum (In Orbit, 2011) for obtaining the call graphs of C-programs. In this link, a list of tools to obtain the call graph is provided that includes `Egypt` as well. Our purpose is to conduct the source code analysis of security bugs. Therefore, it is found that `Egypt` is useful in comparison to the others tools because it is a *Perl* script to integrate the *GCC* compiler for source code analysis and *graphviz* to generate the call graph. However, it only used to study the source code analysis instead of generating the graph. The further introduction of `Egypt` and details about its functions are available on its home page[1]. Finally, `Egypt` has been selected to fit our purpose.

4.2 Computing Fan-in of Functions

The purpose of obtaining a call graph is to compute the fan-in of functions. Formally, fan-in is defined in Definition 9.

Definition 9 *Fan in, F_v The number of times a function is invoked by the other functions is called its* fan-in. *On the call graph, fan-ins are the in-degree of nodes, denoted by the following mapping set: $F_v = \{(v, |C_v|) \mid v \in V \wedge C = \{v' \mid (v', v) \in E\}\}$.*

To automatically compute the fan-in of all the functions, Algorithm 6 is used. The call graph $G < V, E >$ is the input, where V is the set of functions representing each node of the call graph, and E is the set of edges representing the ordered pairs of calling and callee functions. The edge set $\{(i, j)\}$ is used to represent the index of functions, where i represents the calling function (predecessor) and j represents the callee function (successor). The output is the set of changed functions F_v, which is the Cartesian product of the set of functions V and their fan-in is represented by $|C_v|$. C_v is a set of callers of the functions v. The procedure starts by initialising the output changed function set F_v to the empty set. Then it computes all the callers for each function v and assigns to the set of the caller functions C_v. In Line 4 of the

[1]http://www.gson.org/egypt/egypt.html

procedure, the callee function v and the cardinality value of its caller set C_v are a pair assigned to the output changed function set F_v.

Data: A digraph $G =< V, E >$, where the set of vertices $V = \{v_i \mid 1 \leq i \leq n \wedge n \in \mathcal{N}\}$ and the set of edges $E = \{(v_i, v_j) \mid 1 \leq i, j \leq |V| \wedge i, j \in \mathcal{N}\}$. Here $\mathcal{N}$ is the set of natural numbers.

Result: $F_v = \{(v, |C_v|) \mid v \in V \wedge C_v = \{v' \mid (v', v) \in E\}\}$

```
1 F_v = {};
2 foreach v ∈ V do
3 |   C_v = {v' | (v', v) ∈ E};
4 |   F_v = F ∪ {(v, |C_v|)};
5 end
```

Algorithm 6: ComputeFan-In: Compute the fan-in of nodes functions from the call graph.

The distribution of fan-ins of all the functions in the *Samba* case study are shown in Figure 4.1. A total of 2553 functions' fan-in is obtained from the release 3.6.12. Most functions have fan-in in the range of 1 to 2, and there is no function with zero fan-in because all such functions from the output of `Egypt` have been filtered out. In the figure 4.1, the y-axis represents the frequency of fan-ins of functions in the *Samba* case study. The a-axis represents the ranges of fa-in. The range of fan-in starts from (1 to 10). One can observe from the figure that the most functions' fan-in lies in the range of (1 to 10).

4.3 Extracting Changed Functions Relevant to Bugs

When a bug is fixed, the change happens to the code that directly reflects in the functions of the system. However, computing the fan-ins of all the functions does not tell us anything about the changes in functions in response

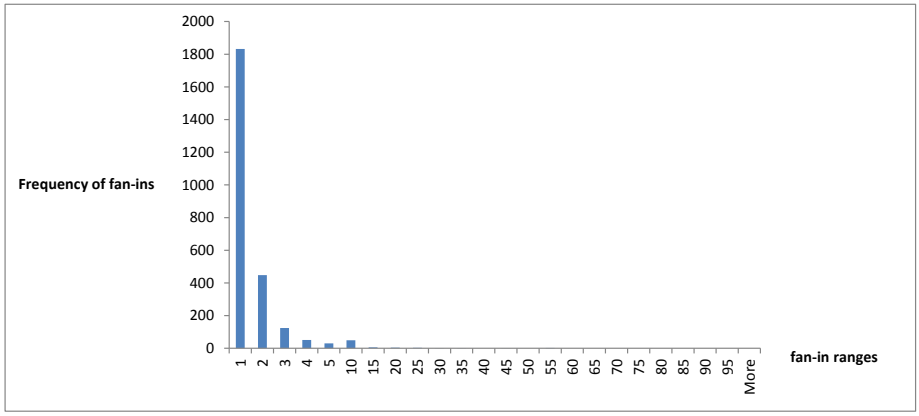

Figure 4.1: The distributions of the fan-ins among functions.

to the bug fix. To verify our hypothesis for the changed functions, the change functions need to be obtained first by taking into account the difference of the two patches. Hence, the set of all the *Samba* patches is required to get the changed functions. The following procedure describes how the set of patches in the *Samba* case study has been obtained. Another case study may involve different change management systems, but the data extract steps should be similar. The `git` version control system is used by the *Samba* development team to keep track of changes made to the code, which maintains a commit log whenever a fix to buggy code is patched. One can obtain commit logs from `git` repository using the `git` command as discussed below and the set of patches can be obtained from the commit logs. In general, the patches are defined in Definition 10.

Definition 10 *Patches : P and Commit logs : L. A set of patches P is defined as:* $P = \{p \mid contains(L, \text{“}commit\text{”} + p.id)\}$ *where L is the set of logs of a* `git` *repository.*

The set of patches is obtained from Algorithm 7. The log output from the `git` repository is the input, and a set of patches is its output. The procedure starts by initialising the set p to the empty set. The condition that a line starts with the “commit” and has a commit id is checked using regular expressions. When the condition is true, the whole line splits by the string “commit” and returns the first element as the patch id into the variable p. In the last step at Line 5, the commit id is stored as the set of patches by the end of the procedure.

Data: L: the log output from a `git` repository obtained by the command `git log`

Result: $P = \{p \mid contains(L, \text{“}commit'' + p.id)\}$

```
1 P = {} ;
2 foreach line ∈ L do
3 |   if contains(line , "^commit) then
4 |   |   p = split(line , “commit'')[1];
5 |   |   P = P ∪ {p};
6 |   end
7 end
```

Algorithm 7: ObtainPatches: Obtaining the list of patches.

Now, getting the changed functions it is possible that one can retrieve the set of patches. To get changed functions, the `git diff` command is used to get the change set Δ by taking the difference of two patches. From each change set, the list of changed functions can be retrieved. The patch p and changed functions V_p are defined in Definition 11.

Definition 11 *Patch : p , and Changed functions : V_p. A patch is defined as a set of changes to the program: $p = \{\Delta\}$ where Δ is a change. Every patch p can be identified by a hexadecimal number* `id`*, denoted by p.*`id`*. Usually a patch happens as the result of a bug fix. The set of functions changed by a patch p are denoted as $V_p = \{v \mid v \in V \wedge contains(\Delta, v) \wedge \Delta \in p\}$. Here V is the set of functions already defined in Definition 8.*

In Algorithm 8, the list of changed functions is obtained. The patch p is the input and the set of changed functions V_p is the output. The patch p is a set of changes Δ obtained by taking the difference of two patches p_i and p_j. Each element of the changed functions set V_p belongs to Δ and the set of functions V. The patch p is collected using the `git diff id..id^` command, where the id of each commit is provided to `git` to know the difference between two patches. The procedure starts by initialising the set of changed functions V_p to the empty set. Then for each change Δ belonging to p and each function v belonging to functions set V, the condition if Δ contains v is checked. When

the condition is true, the function v is added to the set of changed functions V_p.

Data: $p = \{\Delta\}$: a patch is a set of code changes `id` obtained using the command `git diff id..id^` , V: the set of functions

Result: $V_p = \{v \mid v \in V \wedge contains(\Delta, v) \wedge \Delta \in p\}$

```
1 V_p = {};
2 foreach Δ ∈ p , v ∈ V do
3 |   if contains(Δ , v) then
4 |   |   V_p = V_p ∪ {v};
5 |   end
6 end
```

Algorithm 8: ObtainChangedFunctions:Obtaining the list of changed functions from a patch.

4.4 Computing Fan-in of Changed Functions Relevant to Bugs

Until now for the *Samba* case study the set of patches P and changed functions V_p has been obtained. In order to know the patches which have high fan-in changed functions relevant to a bug, the relationship between a patch and the changed functions has to be analysed. Therefore, getting the accumulative fan-in of all the changed functions inside all the patches will help to know which patches have high fan-in changed functions. The indication of high fan-in changed functions inside a patch tells us that a patch code is complex.

Formally, the accumulative fan-in of all the changed functions in a patch is given in Definition 12.

Definition 12 *The accumulative fan-ins of all the changed functions inside all the patches: F_P , and the accumulative fan-in of all the changed functions inside a patch : f_p. The set of the accumulative fan-ins of all the changed*

functions inside all the patches F_P is defined as: $F_P = \{(p, \sum_{v \in V_p} F_v) \mid p \in P\}$, *where V_P is the set of changed functions, and F_v is a Cartesian product of the set of functions and their fan-ins. The accumulative fan-in of all the changed functions inside a patch f_p is defined as:* $f_p = \sum_{v \in V_p} F_v(v)$.

The set of accumulative fan-ins of all the changed functions inside all the patches is obtained using Algorithm 9. The set of changed functions fan-ins inside the patches F_P contains the pair of each patch and its fan-in. In Algorithm 9, the set of functions' fan-ins F_v and the set of patches P are the input parameters. The set of functions' fan-ins F_v is the result of applying Algorithm 6 and the set of patches P is the results of applying Algorithm 7. The procedure starts by initialising the set of the changed functions' fan-ins inside the patches F_P to the empty set. This set is the Cartesian product of patches and their fan-ins. Then for each patch p belonging to the set of patches P, the value of the set of changed functions V_p is obtained using Algorithm 8. Line 6 is aggregating the fan-ins of all the changed functions inside each patch for the given function v. Here, the function v belongs to the set of changed functions V_p and is assigned to the accumulative fan-in of changed functions inside a patch f_p.

Line 7 inside the loop is used to store the patch p and the sum of the fan-ins of changed functions f_p into the Cartesian product of patches and their changed functions' fan-ins F_P.

Data: F_v, P

Result: $F_P = \{(p, \sum_{v \in V_p} F_v) \mid p \in P\}$

1 $F_v = ComputeFan - In(G)$;

2 $P = ObtainPatches(L)$;

3 $F_P = \{\}$;

4 **foreach** $p \in P$ **do**

5 $\quad V_p = ObtainChangedFunctions(p)$;

6 $\quad f_p = \sum_{v \in V_p} F_v(v)$;

7 $\quad F_P = F_P \cup \{(p, f_p)\}$;

8 **end**

Algorithm 9: ComputeOverallFan-in: Compute the accumulative fan-in of all the functions inside a patch.

The accumulative fan-ins of all the changed functions inside all the patches F_P has been calculated. Now one can find the bugs relevant to the set of patches to know whether the changed functions of patches relevant to bugs have higher fan-in. Therefore, the Cartesian product of the set of bugs and patches is required, which is formally defined in Definition 13.

Definition 13 *The set of bugs : B, and The set of bugs and their corresponding patches : B_P. The set of bugs B is defined as $B = \{b \mid b = matcher(L(p), regex)\}$.*

In this set, the regex is $.*[\hat{}\mathtt{a}-\mathtt{z}](\mathtt{bug}|\mathtt{fix}|\mathtt{correct})\backslash\backslash\mathtt{D}*(\backslash\backslash\mathtt{d}+).*$ *with b being $\backslash\backslash(d+)$ group1. The matcher is a function to match the regular expression with the given pattern. The Cartesian product of B and patches P is defined as $B_P = \{(b, p) \mid b \in B \wedge p \in P\}$.*

The set of bugs and their corresponding patches are obtained using Algorithm 10. Here, the set of logs output L from all the branches in `git` repository is the input parameter. The output is B_P. The procedure starts by initialising the value of B_P to the empty set, then for each line of code belonging to the corresponding patch $L(p)$, and for each patch p belonging to the set of patches P. The regular expression in the line *line* is matched with

the bug id b. If the regular expression matches the value of bug b and patch p are added as an element to the set of bugs and their corresponding patches B_P.

Data: L: the log output from a `git` repository obtained by the command `git log`

Result: $B_P = \{(b, p) \mid b \in B \wedge p \in P\}$, where
$B = \{b \mid b = matcher(L(p), regex)\}$ in which
$\texttt{regex} = .*[\hat{}\texttt{a}-\texttt{z}](\texttt{bug}|\texttt{fix}|\texttt{correct})$
$\backslash\backslash\texttt{D}*(\backslash\backslash\texttt{d}+).*\texttt{withbbeing}\backslash\backslash(\texttt{d}+)\texttt{group1}$

```
1 B_P = {};
2 foreach p ∈ P do
3 |   foreach line ∈ L(p) do
4 |   |   if containPatterns(line, b, regex) then
5 |   |   |   B_P = B_P ∪ {(b, p)};
6 |   |   end
7 |   end
8 end
```

Algorithm 10: ObtainBugs'Patches: Obtain the list of bugs and their corresponding patches in which they are fixed.

After applying Algorithm 10, the set of bugs and their corresponding patches B_P and the set of accumulative fan-in of changed functions inside the patches F_P has been collected. In *Samba* it is observed that the fix to each bug can be patched in one or multiple patches. Therefore, to know how many bugs have high fan-in changed functions it is required to obtain the accumulative fan-ins of all the changed functions inside all the patches relevant to the bug fix. Formally, the accumulative fan-ins of changed functions inside all the patches relevant to a bug fix is given in Definition 14.

Definition 14 *The set of fan-ins of all the changed functions inside all the patches relevant to the bugs :F_B, and the accumulative fan-in of all the changed functions inside all the patches relevant to a bug : f_b. The Cartesian product of bugs and their relevant patches' changed functions fan-ins is de-*

fined as $F_B = \{(b, f_b) \mid b \in B_p \wedge f_b = f_b + F_p.get(p)\}$*, where* f_b *is the fan-in of changed functions inside all the patches relevant to a bug. The* f_b *is obtained by summing the changed functions fan-ins of all the patches corresponding to a bug.*

Algorithm 11 is used to obtain the changed functions fan-ins of all the patches relevant to the bugs. The Cartesian product of patches and their changed functions fan-in F_p and the Cartesian product of bugs and their corresponding patches B_P are input. The output is the fan-in of changed functions inside the patches relevant to the bug fixes F_B as defined in Definition 14. The procedure starts by obtaining the values of F_p and B_P from Algorithms 9 and 10 respectively. In Line 3, the value of F_B is initialised to the empty set. Then for each value of the bug b belonging to the B_P, the fan-ins of changed functions inside the patches relevant to a bug fix f_b are initialised to zero. Since each bug might have multiple patches, a "for" loop is used to iterate all the patches P in the F_P relevant to the bug b. At each iteration the changed functions' fan-in of each patch is added to f_b to get the accumulative changed functions' fan-in of all the patches relevant to the bug b. Finally, the pair of bug b and its fan-in f_b are added to the F_B.

Data: F_p, B_P
Result: $F_B = \{(b, f_b) \mid b \in B_P \wedge f_b = f_b + F_p.get(p)\}$

```
1  F_p = ComputeAccumulativeFanin(F_v, P);
2  B_P = ObtainBugs'Pastches(L);
3  F_B = {};
4  foreach b ∈ B_P do
5  |   f_b = 0;
6  |   foreach p ∈ F_p do
7  |   |   f_b = f_b + F_p.get(p);
8  |   end
9  |   F_B = F_B ∪ {(b, f_b)};
10 end
```

Algorithm 11: ObtainBugsFanin: Obtain the fan-in of all the bugs.

After obtaining the set of changed functions' fan-ins inside all the patches relevant to the bugs, one can get the average fan-in of changed functions inside all the patches relevant to security and all the bugs. The average changed functions fan-in inside all the patches relevant to all the bugs is given in Definition 15.

$F_B = ObtainBugsFanin(F_P, B_P)$ // write about this equation

Definition 15 *Given the set of bugs B, and their average fan-in $\bar{f_B}$, the average fan-in is defined as. The average fan-in of all the bugs $\bar{f_B}$ is defined as $\bar{f_B} = \sum_{b \in F_B} f_b / |F_B|$.*

$$\bar{f_B} = \sum_{b \in F_B} f_b / |B| \tag{4.1}$$

It is not possible to verify our hypothesis for security and non-security bugs by only calculating the average fan-ins of non-security bugs. Therefore, the security bugs are defined in Definition 16. The security bugs are retrieved from the *Samba* security releases web-page *http* : *//www.samba.org/samba/history/security.html*. At the time of study, only 14 of the retrieved CVE relevant bugs have `Bugzilla` entries. In these 14-bugs, the patches of 9 bugs are available in the *Samba* `git` repository.

Definition 16 *The set of given security bugs B_S, The fan-in of a security bug f_{b_s}, and The average fan-in of security bugs $\bar{f}_{B_S}$. For this algorithm, the security bugs are already given. The average fan-in of security bugs $\bar{f}_{B_S}$ is defined as $\bar{f}_{B_S} = \sum_{b_s \in B_S} f_{b_s} / |B_S|$.*

$$\bar{f_{B_S}} = \sum_{b_s \in B_S} f_{b_s} / |B_S| \tag{4.2}$$

Formally, the null hypothesis 2 is restated as $\bar{f}_B > \bar{f_{B_S}}$.

4.5 Applying the fan-in metric to the case studies

The technique to measure fan-in complexity of functions relevant to CVE and Non-CVE bugs has been applied to two case studies *Samba* and *Mozilla Firefox*. In the *Samba* case study, the average fan-ins of CVE bugs is $\bar{f_{B_S}} = 26.42$ and average fan-ins of Non-CVE bugs is $\bar{f}_{B_{\bar{S}}} = 8.39$. In the *Mozilla Firefox* case study, the average fan-ins of CVE bugs is $\bar{f_{B_S}} = 4.5$ and the average fan-ins of Non-CVE bugs is $\bar{f}_{B_{\bar{S}}} = 0.44478723$. Graphically, the average fan-in values of CVE and Non-CVE bugs are presented in Figure 4.2. In the figure, the y-axis represents the average fan-in values and the name of case studies are presented on the x-axis. The bar chart with red color represents the CVE relevant bugs and the bar chart with blue color represents the Non-CVE relevant bugs. By applying the technique to measure fan-in complexity of functions relevant to bug fixes. The first part (RQ2.1) of the second research question (RQ2) has been answered.

The results of case studies shows that on average a CVE bugs has higher fan-in compared to the Non-CVE bugs in the *Samba* and *Mozilla Firefox* case studies. Based on the fan-ins comparisons of CVE and Non-CVE relevant bugs. It is concluded that the null hypothesis 2 is rejected that on average the fan-in of security bugs is lower than the non-security bugs. One can conclude from these results that on average security bugs are more complex to fix than the other bugs in *Samba* and *Mozilla Firefox* case studies. By comparing the fan-ins complexity results of CVE and Non-CVE bug fixes, the second part (RQ2.2) of the second research question RQ2 has been answered.

There is a difference in the fan-in results of *Samba* and *Mozilla Firefox* case studies. A possible explanation of the lower fan-in of *Mozilla Firefox* case study is that there a lot of patches delivered for the purpose of regression testing. Therefore, in fact there are no function changes in such patches. Another possible reason is that there is a less trace-ability between the bug

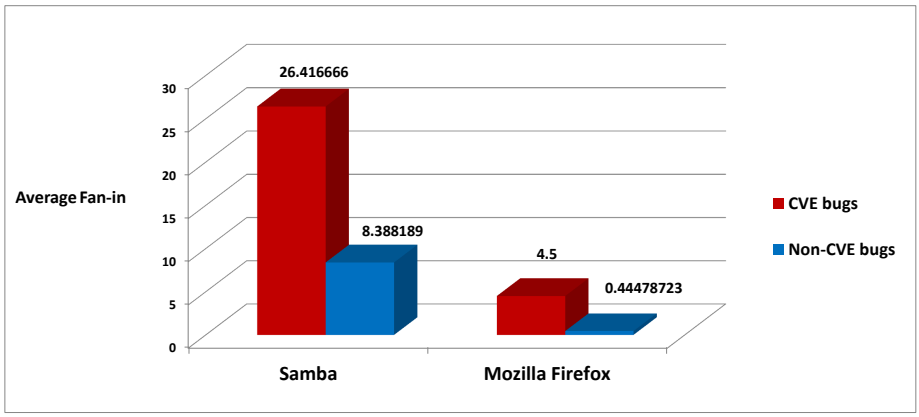

Figure 4.2: The average fan-in of *Samba* and *Mozilla Firefox* case studies.

ids and code patches in the case of *Mozilla Firefox* case study.

4.6 Tool Support

To answer the second research question (RQ2) for measuring the complexity of functions relevant to bug fixes an automated tool has been developed. The tool has two components "Data Extraction" and "Data Analysis".

Data Extraction: In the data extraction, the first phase is to download and build the code of the system. To download and build the code of the system, the instructions given for the development of open source system has been used. For example, the instructions for the *Samba*'s developers on their wiki

to "Build *Samba*" [2] and "Contribute to *Samba*" [3] has been used to download and build the code of the *Samba* system. The *Samba* system has been build by creating the clone of its code base in the `git` version control system. In the second phase of data extraction, the call graph of the *Samba* system has been collected from the branches of `git` version control system. The steps to extract the call graph has been described in Appendix A.5. The call graph of *Samba* contains information about which function is calling which other function in the system.

Data Analysis: In the data analysis, the first step is to measure the fan-in of all the functions in the collected call graph. The fan-in of functions from the call graph has been measured using automated program in Appendix A.5. The call graph of the system is input of the program and the set of fan-ins of functions F_v is its output. The next step is to collect patches delivered for the fixing of bugs using the program in Appendix A.8. The patch commit log is input of the program and the set of patches P is its output. After collecting the set of patches P, the changed functions V_p has been collected by taking the difference of patches. The program to obtain the set of changed functions is provided in Appendix A.7. The set of patches P is input of the program and the set of changed functions V_p is its output. In the next step, the fan-ins of changed functions for each patch has been measured by mapping the functions retrieved from the call graph F_v with the changed functions V_p. The program to obtain the fan-ins of changed functions for each patch is provided in Appendix A.9. The set of patches P and the set of changed functions V_p are input of the program and a Cartesian product (p, f_p) is output of the program. In this Cartesian product, the f_p represents the fan-in value of changed functions corresponding to the patch p.

After getting the fan-in value of changed functions for each patch, the next step is to look for the bugs those fix are delivered in the patches. In this

[2] https://wiki.samba.org/index.php/Build_Samba

[3] https://wiki.samba.org/index.php/Contribute

way, the bugs and their corresponding patch are collected by parsing the commit logs. The program to obtain the set of bugs and their corresponding patches B_P is provided in Appendix A.11. The patch commit log is input of the program and a Cartesian product $(bugID, P)$ is output of the program. In this Cartesian product, the $bugID$ represents the identifier for each bug corresponding to the patch in which its fix is delivered. In the next step, the fan-in value of each bug is obtained using the program in Appendix A.12. The set of bugs and their corresponding patches B_P and the Cartesian product of the fan-in value of changed functions for each patch F_p are input of the program. The Cartesian product of bugs and their fan-in value F_B is output of the program.

At-last, the fan-in of bugs relevant to CVE and Non-CVE is obtained using the program in Appendix A.14. The set of bugs and their corresponding patches F_B and a file containing the list of bugs for either CVE and Non-CVE bugs are input of the program. The output of this program is the average fan-in value of either CVE and Non-CVE relevant bugs. As this program is used to get the fan-in of both CVE and Non-CVE relevant bugs. Therefore, the output of this program depends on the list of bugs in the input file. One of the prerequisite of getting the average fan-in value for CVE and Non-CVE bugs is to collect the CVE and Non-CVE bugs. In the earlier chapter, it is reported that CVE and Non-CVE bugs are collected from the `Bugzilla` system. Therefore, the CVE and Non-CVE bugs used for measuring rush for the *Samba* and *Mozilla Firefox* case studies are also used to compute the average fan-in.

Advantages and the limitation of the tool:

The following are advantages of the tool to measure 'fan-in' complexity of functions relevant to bug fixes.

- The algorithms for data extraction and analysis phase has been provided. Therefore, it is easier to apply the technique to measure fan-in of functions relevant to bug fixes for the more case studies. It also makes

easier to repeat the case studies conducted in this thesis to confirm the results.

- The data extraction and analysis phases of the tool are fully automated and the data analysis phase has been programmed using *Java*. All the *Java* programs are provided in Appendix A.

The following are limitations of the tool to measure 'fan-in' complexity of functions relevant to bug fixes.

- The tool is taking the difference of patches to extract the changed functions from the change set. Therefore, the calculation depends on the availability of the patches in the commit logs.
- The tool takes into account the patches delivered for the purpose of regression testing which might affect on the outcome of the results.
- The "program to get the list of functions from the change set" and the program to "obtain the list of patches" might take more time for the case studies in which the number of patches delivered are higher.

4.7 Threats to Validity

Briefly, the following threats to validity to this empirical case study has been discussed.

Internal Validity: For the *Samba* case study, only 14 CVE relevant bug reports has been found in the `Bugzilla` database for the *Samba* system at *samba.bugzilla.org*. In these 14 CVE entries, only 12 bugs have corresponding patches available in the `git` repository. The results of CVE and non-CVE bugs are based on 12 and 1945 bugs respectively. Therefore, smaller CVE bugs data set for comparing the fan-in values of *Samba* case study is one of the validity threats. To counter this threat to validity, the technique to measure fan-in complexity of functions relevant to bug fixes has been applied to another case study *Mozilla Firefox*. To measure the complexity of functions relevant to bug fixes for *Mozilla Firefox* case study, the 14 CVE and

31444 Non-CVE bugs are used. However, it is found that *Mozilla Firefox* does not disclose all of CVE relevant bugs in the public domain. Therefore, it is possible that the sample size to compare the fan-in values for the both case studies is not reflecting all the bugs. But it is sufficient enough to verify the hypothesis.

In this study, the average fan-in complexity of functions relevant to bug fixes has been compared. One can argue about the choice of comparing average, well the total fan-ins by a code patch follows a normal distribution. That means there are few outliers in the fan-in values because the complexity of functions relevant to CVE and Non-CVE bugs is computed using the call graph of the same system.

Construct Validity: Our complexity measurement on change impact complexity using call graphs also provides a chance for objectively checking the effort required for fixing the problems. One threat of validity is that not all code patches are related to the bug numbers and not all fixed bug numbers have comments in the code to indicate where their patches are. For example in the *Samba* case study only 15% of code patches have their corresponding bug id. The percentage of bug fixes with the lack of trace-ability between the code patches for *Mozilla Firefox* is 40%. Although in another case study, Yu et al. (2008) have extracted the trace-ability links between the source code and its security design in *UML* sequence diagram. However, such links cannot be used for the analysis of security bugs without an explicit *UML* design model or 100% accurate bug-code trace-ability. In their book Huang and Zisman (2012) has mentioned that managing trace-ability links between different artefacts of the system is difficult. However, the lack of bug-code trace-ability is still another area to work on for the research community. Therefore, it is outside the scope of this thesis to address the risk that fan-in computation only covers partial bugs.

External Validity: The `git` version control system and the open source project for the validation of our approach has been used. Therefore, our

approach is applicable to any open source project using `git`, and any other version of control system, if they can be converted to `git`. There are tools available to convert the `SVN` and `CVS` repositories into `git` version control system.
In this thesis, two different security critical case studies have been conducted to measure the fan-in complexity of functions relevant to bug fixes. Therefore, there is sufficient evidence about generalising the conclusion that on average a CVE bug has higher fan-in complexity compared to a Non-CVE bug.

4.8 Summary

In this Chapter, the contributions of this thesis "a technique to measure the fan-in complexity of functions relevant to bug fixes" has been introduced. This thesis is the first to apply the concept of fan-in to measure the bug fix complexity. The fan-in complexity technique is applied to the two case studies *Samba* and *Mozilla Firefox*. The average fan-in of changed functions relevant to the bug fixes relevant to CVE and Non-CVE bugs has been computed. It is found that on average the functions relevant to CVE bug fixes has higher complexity compared to the functions relevant to the Non-CVE bug fixes for the both case studies.
As in this thesis the CVE bugs are used as the representative of security bugs and the Non-CVE bugs as the representative of Non-security bugs. Therefore, by comparing the fan-in values of CVE and Non-CVE bugs, the null hypothesis2 has been rejected for the both *Samba* and *Mozilla Firefox* case studies.

Table 4.1: Gaps filled by answering the second research question (RQ2).

Gap reference	**Gap**	**How the gap is filled**

$SB2$	There is a need to investigate whether security bugs are more complex to fix compared to the other bugs.	This study has developed a technique to measure the complexity of functions relevant to bug fixes. The technique has been applied to two case studies *Samba* and *Mozilla Firefox*. The comparison of both case studies showed that security bugs have higher complexity compared to the non-security bugs. There are various metrics to measure the complexity of bugs as explained in the literature review section of this thesis. However, this study is the first to use the fan-in complexity metric to measure the complexity of bug fixes. Therefore, this study has partially filled this gap.
$BR3$	There is a need to measure the complexity of code relevant to bug fixes to argue that bugs with the higher code complexity are difficult to fix.	The results of fan-in complexity shows that security bugs have higher complexity. Existing studies in the literature have reported that bugs with the higher code complexity are difficult to fix. This study has provided an evidence to support this argument. Therefore, this study has filled this gap.

$BR4$	There is a need to verify that the bugs with the higher code complexity are more difficult to fix using different complexity metric from the fixing time.	This study has used the fan-in complexity metric to measure the complexity of bug fixes rather than the fixing time. Therefore, this study has filled this gap.
$CC1$	There is a need to measure the complexity of functions relevant to a bug fix using different complexity metrics from the source lines of code (SLOC) and cyclomatic complexity (CYC).	This gap is related to the BR4.
$CC2$	There is a need to verify whether functions relevant to bug fixes have high fan-in values.	It is found in this study that the functions relevant to security bug fixes have higher fan-in values. Therefore, this gap has been filled by this study.
$SA3$	There is a need to extract the changed functions from the patches relevant to a bug fix to measure the complexity of functions.	In this study, the fan-in complexity of functions relevant to bug fixes by extracting the call graph of the system. Therefore, this study has filled this gap.

Chapter 5

Conclusions and Future Work

The contributions of this thesis are two folds: (i) a technique to measure rush the time management behaviour of software developers on bug triaging has been developed. The technique has been applied to five security critical case studies namely *Samba*, *Mozilla Firefox*, *Red Hat*, *FreeBSD* and *Mozilla*. The values of rush are tested using statistical testing to compare CVE and Non-CVE relevant bugs. (ii) Another technique to measure the complexity of functions relevant to bug fixes has been developed. The technique has been applied to the *Samba* and *Mozilla Firefox* case studies to compare the fan-in complexity of functions relevant to CVE and Non-CVE bug fixes.
The following subsections discuss the conclusions drawn from the findings and their implications if any, the limitations of our study, and finally future work and future vision.

5.1 Conclusions

In this study, two research questions (RQ1) and (RQ2) has been answered. Both research questions have two parts, the first part of each research question is answering "how" to develop the technique and the second part is answering the "so what" part about the application of technique. The fol-

lowing is a summary of the work.

First, a technique to measure "rush", the time management behaviour of software developers in the triaging of bugs has been developed. The algorithms and formulas to measure "rush" in the triaging of bugs have been provided. The technique has been applied to five case studies *Samba*, *Mozilla Firefox*, *Red Hat*, *FreeBSD* and *Mozilla*. The findings of the study showed that a CVE bug has been triaged in a "rush" by the software developers in the above case studies compared to an average Non-CVE bug. Further, the values of "rush" are statistically verified using the *Mann-whitney U test*. The results of the test showed that the values of "rush" for CVE and Non-CVE bugs for *Red Hat*, *FreeBSD* and *Mozilla* case studies are significantly different. However, the value of "rush" for CVE and Non-CVE bugs for the *Samba* and *Mozilla Firefox* case studies are not significantly different. Based on these findings, it is concluded that CVE relevant bugs are triaged in a rush in *Red Hat*, *FreeBSD* and *Mozilla* case studies. The 'rush' measurement technique is useful in agile software development to determine which bugs has been triaged in a rush.

Second, a technique to measure the complexity of functions relevant to bug fixes using the call graph based fan-in metric has been developed. The algorithms and formulas to compute the complexity of functions relevant to bug fixes have been provided. The technique has been applied to the *Samba* and *Mozilla Firefox* case studies. The findings show that the functions relevant to a CVE bug have higher fan-in complexity compared to the functions relevant to an average Non-CVE bug. The 'fan-in' complexity technique is useful to measure the complexity of functions relevant to bug fixes.

5.2 The correctness and usefulness of findings

In this section, the correctness and usefulness of findings are discussed.

5.2.1 The Correctness of findings

In this study, two key research questions has been answered. The first research question is "whether rush the time management behaviour of software developers is one of the reasons of the premature assignment and subsequently reopening of security bugs". The second research question is "whether security security bugs are complex to fix". The two null hypothesis have been developed in addition to techniques to answer the research questions.

To verify the null hypothesis relevant to the first research question, the technique to measure rush is applied to more than one case studies. Additionally, the null hypothesis relevant to rush (the first research question) is tested using *Mann Whitney U-test*. However, the outcome of empirical research using statistical hypothesis testing can not be regarded as a proof. But based on the probabilities, statistical testing help to support or reject a hypothesis.

On the other hand, to verify the null hypothesis relevant to the second research question, the technique to measure fan-in complexity is applied to more than one case studies (*Samba* and *Mozilla Firefox*).

During the design of study, the same results are expected from the different cases to reject the null hypothesis. The median value of rush for security bugs is higher in the three case studies *Red Hat*, *FreeBSD* and *Mozilla*. Similarly, the fan-in complexity of functions relevant to security fixes are higher for the both *Samba* and *Mozilla Firefox* case studies. It mean the null hypothesis designed in this study for measuring rush and fan-in complexity has been rejected in more than one case studies. Therefore, the results produced by this study are reliable and valid.

One can argue that the author of this thesis can bias the data collection and analysis steps. Therefore, the tool kits for measuring rush and fan-in complexity are explained, so the other researchers can repeat the study to find the similar results.

5.2.2 The usefulness of findings

On the basis of the findings that CVE-relevant security bugs are triaged in a rush compared to the Non-CVE bugs, and CVE-relevant security bugs have higher fan-in complexity compared to the Non-CVE bugs. The following are recommendations for developers and vendors involved in the maintenance of software systems.

Delaying the bug assignment if there is time for the next regular release: According to the release planning model of rush ratio definition, the observation confirms that developers rush to deliver bugs before the deadline of releases. It suggests that delaying the assignment of some CVE bugs when the release planning constraints allow could help find the right developer. In practice, however, many vendor provide quick patches through *security releases* immediately after a CVE bug is 'fixed'. It would give malicious attackers two advantages: (a) the bug patch may not be thorough and complete, yet presenting an illusion of security; (b) the very announcement of bug fix earlier than a normal weekly release may lead to extra updates that are be mended on the regular basis unless all users accept the risk of immediate security updates. Of course, it is not to say that vendors should give up security releases as a best-effort workaround under difficult situations to save their reputation. The statistical findings merely suggest that more often than ever vendors security releases could be aligned better with the regular releases.

Incorporating rush and fan-in complexity techniques with the development workflow: Since it is not difficult to compute rush ratio according to the regular release dates and the bug reports, vendors could benefit from incorporating the metric within their development process to help assess how likely a CVE bug is handled in a rush. In this way, developers can not only estimate the time required in advance but also compare each other to find who can benefit more from a better time management. Such an awareness could also help the vendor in understating the reason of rushing

and which features of the bug reports tend to cause rushing in the past. On the other hand, incorporating fan-in technique in the development process can help vendors in better estimation of the fixing effort. So that they can allocate a single or team of developers on the fixing task accordingly.

Measuring the complexity of past security bug fixes to know the complexity of newly reported similar type of bug fixes: The technique to measure fan-in complexity of functions relevant to bug fixes is useful to measure the complexity of past fixes. In this way, the software developers can determine the complexity of similar type of newly reported bug fixes. Such complexity measure can help them for better estimation about the fixing time. The fan-in complexity will also help software developers at the triaging stage to chose a developer based on his experience of fixing complex bugs.

5.3 Future Work Directions

The research work presented in this thesis is extendable in the following directions.

5.3.1 Using natural language classification technique to retrieve security bugs

In this thesis, the CVE relevant bugs are used as the confirmed security bugs to measure the rush and code complexity of functions relevant to bug fixes. The CVE relevant bugs are already confirmed as the security bugs by the security experts and people working in the security domain have a consensus that these bugs are indeed security related. Therefore,the CVE relevant security bugs from the *Samba*, *Mozilla Firefox*, *Red Hat FreeBSD* and *Mozilla* security release *wikies* and `Bugzilla` bug reports has been retrieved.

However, the CVE bugs are not the only security related bugs that exist in

the system, because in the bug tracking systems the bugs are reported by the end-users, development and testing teams etc., and usually the bugs are labelled as security related at the time of reporting. For example, in the *Mozilla Firefox* system the bug reports labelled as security are treated with high priority Mozilla (2014). Therefore, the bug report helps to determine whether a bug is security related based on the label assigned to the bug report or through looking at its description. Gegick et al. (2010) have used the bug report labels and descriptions to classify them as CVE or non-CVE related. In this thesis that approach could have been used to retrieve the security bugs but one of the prerequisites of applying the approach was that the bugs should be labelled as security related. However, in the *Samba* system the security bugs are not labelled, and CVE relevant labelling is the most reliable source. Another challenge was that the natural language processing approach is not automated; therefore applying the approach to retrieve security bugs was a challenge. Hence, the future plan is to use the natural language bug classification approach to retrieve security bugs for future studies. Similarly, there is an automated tool support available for the classification of bug reports using the natural language technique Podgurski et al. (2003). However, such tool support needs to be extended for classifying security related bugs. In future, the plan is to perform disambiguation for the natural language description of security bugs (Hui Yang et al., 2011).

5.3.2 Applying rush and code complexity measurement approach to the other case studies

The technique to measure rush have been applied to five case studies *Samba*, *Mozilla Firefox*, *Red Hat FreeBSD* and *Mozilla* to know whether the security bugs are triaged in a rush. To the best of our knowledge, this study is the first to choose the *Samba* case study for bug analysis, and *Mozilla Firefox*, *Red Hat*, *FreeBSD* and *Mozilla* case studies to measure the rush. However, a lot of studies on bug analysis have been conducted on the *Apache* open source

systems (Mockus et al., 2002). The *Apache* Tomcat has been widely used as a web server to run web services for *Java* relevant platforms. Security is one of the primary concerns for such a system as well. Therefore, the plan is to apply the rush measurement technique for the other potential case studies in the future.

The technique to measure code complexity of functions relevant to CVE bugs have been applied to the *Samba* and *Mozilla Firefox* case studies to know whether the functions relevant to CVE bugs have higher code complexity compared to the other types of bug. In the future, the plan is to apply fan-in complexity technique to the more case studies as well.

5.3.3 Confirming the rush phenomenon in the triaging of security bugs in industrial settings

In this thesis, the rush in the triaging of CVE bugs has been measured through mining the data from the bug repositories of *Samba*, *Mozilla Firefox*,*Red Hat*, *FreeBSD* and *Mozilla* open source projects. However, bug triaging is a very human-oriented process in which a developer makes decision about the assignment of a bug to the other developer for fixing and whether there is enough information to reproduce the bug. In cases where the developer is convinced that the reported bug is valid and cannot find the duplicate bug then they assign it to another developer for fixing, but the existing evidence in the literature shows that the number of reported bugs everyday over-loads the developers for triaging in open source projects (Anvik et al., 2005). Therefore, taking the developers' view on the rush in the triaging of bugs by conducting a survey is also in our list of future works.

In open source projects, software developers usually work in the distributed teams on a voluntary basis compared to the closed source projects (Crowston and Scozzi, 2008). Therefore, the teams working in closed source projects face different challenges during the triaging of bugs. According to Kuan (2004), there are different practices in place for the resolution of bugs for the

closed source projects compared to the open source. Therefore, there is a plan to apply the rush measurement technique to the closed source projects.

5.3.4 Measuring complexity of functions for the closed source systems

In Chapter 4 of this thesis the complexity of functions relevant to CVE bugs for the *Samba* and *Mozilla Firefox* open source systems has been measured. Usually, the open source systems are considered more extensible and less coupled due to the limited control on the quality of the development process (Stamelos et al., 2002). Therefore, it is considered that the functions added and modified at the time of maintenance release in a closed source project are closely related compared to the functions added and modified in the open source (Paulson et al., 2004). However, it is an open question whether the coupling of systems impacts on the fan-in value of functions relevant to security bugs. Hence, one of the future tasks is to apply our code complexity approach to measuring the fan-in of functions of the closed source projects.

The more complex code is considered harmful for the security of software systems (Wurster and van Oorschot, 2008) and usually a lot of effort is made to encourage software developers to write a secure code. However, how the software developers' code writing skills impact on the complexity of software systems is an open question to answer, which is one of our future work directions.

Similarly, the author of this thesis is interested in comparing the complexity of functions for CVE bugs in the open and closed source projects. Above it is already discussed that how both types (open and closed source) of projects have very different development processes in place. Therefore, such an analysis will help to determine whether the different development processes also impact on the complexity of security relevant functions. Consequently increases the fix time of bugs at the maintenance stage.

Another future task in this direction is to analyse the developers' personal process of fixing security bugs in the open and closed source systems. Such an analysis will help to prepare general guidelines for the developers during the fixing of security bugs.

5.4 Future Vision

In response to malicious attacks, the security relevant bugs need to be resolved on the urgent basis to secure the software system. Therefore, resolving security bugs during the maintenance of software system is challenging for the software developers. To investigate the resolution of security bugs during software maintenance, at the start of this study, a literature survey has been conducted. Two key gaps relevant to triaging and fixing activities of security bug resolution has been found.

A gap that security bugs are triaged faster and reopens more often relevant to triaging activities has been found. Therefore, in this thesis, first it is investigated whether software developers rush to resolve security bugs on the triaging. A technique to measure rush the time management behaviour of software developers has been developed in this study. The technique has been applied to five case studies. Further, the rush values of security and non-security bugs are statistically tested to evaluate their significance. The results shows that security bugs are triaged in a rush in *Red Hat*, *FreeBSD* and *Mozilla* case studies. For the expansion of rush analysis, some plans has been discussed in the future work sections 5.3.2 and 5.3.3. The plans listed in above future work sections are more relevant to the verification of the technique to measure rush.

On the high level, one of the follow up work is to investigate that how incorporating the rush measurement technique can help to improve the development process. Software projects not using Bugzilla may follow somewhat different practices in the resolution of bugs, in such cases the proposed release-planning

based rush metric definition will have to be extended to fit for the purpose. The current calculation is based on the CVE bugs available from bug report references to the CVE entries. Different types of vulnerabilities in CVE records such as buffer over-flow, access validation error may also have different consequences on the security of system. Therefore, weighing in our rush metric with the classified features of CVE bugs developers could be more informed about what to do in the face of security attack.
Concretely, answering the following research question relevant to rush metric will give more insight about the triaging of security bugs and its role in the security maintenance.

> How can the 'rush' measurement technique be used to improve the bug triaging process?

A gap that security bugs are more complex to fix compared to the other types of bug relevant to fixing activities has been found. Therefore, in this thesis, it is investigated whether security bugs are more complex to fix. A technique to measure fan-in complexity of functions relevant to bug fixes has been developed in this study. The technique has been applied to two case studies *Samba* and *Mozilla Firefox*. For the expansion of fan-in analysis, some plans has been discussed in the future work section 5.3.4.The plans listed in above future work section are more relevant to the verification of the technique to measure fan-in complexity.
On the high level, one of the follow up work relevant to fan-in complexity technique is to measure the fixing time of security relevant bug fixes with the higher fan-in complexity. In this way, one can verify whether such bugs also takes the more time of developers for fixing. The bug fix complexity is key in determining the fixing time of bugs but the fixing complexity can only be measured after the resolution of bugs. However, such complexity measure can help to predict the fixing time of similar types of bug resolved in the past.

Concretely, answering the following research questions will give more insight about fixing of security bugs.

How much additional time developers takes to fix security bugs with higher fan-in complexity? How can the 'fan-in' complexity be used to predict the fixing time of security fixes?

Appendix A

Programming Scripts

A.1 Script to Obtain the Bug Entry Log log_B

The following shell script is used to retrieve the bug entry log log_B from all three case studies, namely *Samba*, *Mozilla Firefox*, and *Red Hat*. The bug entry log log_B has been used to get the time stamps when the bug is reported ($t_{REPORTED}$).

Before running the script, first the bug ids b from the `Bugzilla` database for each case study has been extracted. The advance search option at the `Bugzilla` database for retrieving the bug ids b for each case study has been used. In the advance search all the values of "product", "component", "version", "target", "status", "resolution", "severity", "priority", "hardware" and "OS" fields have been selected. Of course, different `Bugzilla` systems has different limits to show the data set. Therefore, the results of advance search have been downloaded in the Common Separated Files (csv) file format. The advance search process is repeatable, e.g. one can search all the bugs relevant to the *Samba* case study from their advance search web page [1]. The script starts by assigning the case study `Bugzilla` URL to the variable URL. For example, one can assign the URL *bugzilla.samba.org* for running

[1] https://bugzilla.samba.org/query.cgi

the script to get the bug logs of the *Samba* case study. The second line in script is assigning the downloaded list of bug ids b csv file to the variable $DATA$. The purpose of the third line is to store the output from the script in the folder name *tables*. Then, the condition that a file relevant to bug id b exists in the folder *reports* is checked. In cases when no file exists in the folder *reports* the file is downloaded from its web page using the bug identifier $show_bug.cgi?id$. The downloaded web is moved to the folder bug reports with the name $bugentry\_id$.

Finally, the script to obtain the time stamps when the bug is reported $t_{REPORTED}$ has been called. The details of script to obtain the time stamps when the bug is reported are given in the following subsection A.3.

```
1  \#\!/bin/bash
2  URL=bugzilla.case-study.org
3  DATA=bugs_case-study.csv
4  echo > tables/"$DATA"_bug_eventstamps.txt
5  if [ ! -e reports/bugentry_$id.html ]; then
6    curl -s -O "https://$URL/show_bug.cgi?id=$id"
7    mv show_bug.cgi\?id\=$id reports/bugentry_$id.html
8  fi
9  awk -f bug_init.awk -v id=$id reports/bugentry_$id.html >>
       tables/"$DATA"_bug_eventinitial.txt
10 done
```

A.2 Script to Obtain the Bug Activity Log log_A

The following shell script is used to retrieve all the bug activity $logs_A$ from the `Bugzilla` database for all the case studies. The bug activity logs log_A have been used to get the time stamps when the bug is assigned and resolved $t_{ASSIGNED}$ and $t_{RESOLVED}$ respectively. The same bug ids b csv file retrieved from the advanced bug search has been used as explained in the above section

to obtain the bug entry logs. The first two lines has been copied from the above script to obtain the bug entry logs because the variables "URL" and "DATA" are initialised in this script in the same ways as is done in the script to obtain the bug entry logs. The third line of script is used to redirect the output file of bug $t_{ASSIGNED}$ and $t_{RESOLVED}$ to the folder "tables". The procedure starts by checking the value of each bug id b in the "reports" folder. In cases where there is no file of bug activity log relevant to the bug id b found in the folder then the bug activity web page with the URL $show_activity.cgi?id$ relevant to the bug id b is obtained. The obtained bug log log_A is moved to the reports folder with the corresponding bug id b.
Finally, the script to obtain the values of time stamps when the bug is "Assigned" t_{ASSIGNED} and "Resolved" t_{RESOLVED} has been obtained and moved to the folder "tables".

```
1  #!/bin/bash
2  URL=bugzilla.case-study.org
3  DATA=bugs_case-study.csv
4  echo > tables/"$DATA"_bug_eventinitial.txt
5  for id in 'cut -d, -f1 bugzilla/$DATA'; do
6  if [ ! -e reports/$id.html ]; then
7      curl -s -O "https://$URL/show_activity.cgi?id=$id"
8      mv show_activity.cgi\?id\=$id reports/$id.html
9  fi
10 awk -f bugeventstamps.awk -v id=$id reports/$id.html >> tables/"
       $DATA"_bug_eventstamps.txt
11 done
```

A.3 Obtaining $t_{REPORTED}$: the Event Stamp When the Bug is Reported

The following "AWK" script is used to get the time stamps when the bug is reported ($t_{REPORTED}$). The log_B is input of the script and the bug id

b and time stamps when the bug is reported $t_{REPORTED}$ is output of the script. An "AWK" script starts with the pattern matching string. Therefore, the first line of script is "Reported" indicating the start of script when the pattern "Reported" appears in the data. The screen-shot of example log_B indicates the pattern matching string in the data of log_B as shown in the Figure A.1. The second line of script is setting the variable *start* to true. The next statement in the script is matching a regular expression in the data of log_B to split the time stamp's data from the other information, which is irrelevant. Before splitting the string to get the regular expression, the value of *start* variable is checked to ensure that the string to split must be related to the pattern "reported". Then, the split statement is used with the three parameters to split the string containing the time stamp. The "AWK" function $split(\$1, a, >)$ splits the first field of the string using the delimiter " $>''$. In the next statement, the second element $a[2]$ of the array a is assigned to the *date* variable by adding the spaces between the second \$2 and the third \$3 fields. For example, the second element $a[2]$ of the array a in the example log_B data is $2003-09-0709:23UTC$ as shown in the Figure A.1. The $2003-09-07, 09:23, UTC$ are the first, second and the third elements of the second element $a[2]$ of array a in the given example log_B data. The *date* variable is printed with the variable *id*, which corresponds to the bug id b of the time stamp $t_{REPORTED}$. The variable *id* is the parameter provided to the script at the time of call. After printing the time stamp data, the variable *start* is rested to the zero.

Input: log_B

```
1 /Reported/ {
2   start = 1
3 }
4 /<td>[0-9]+-.*"/ {
5 if (start) {
6 split($1, a, />/)
7 date=a[2] " " $2 " " $3
```

```
8  ␣␣␣␣␣␣␣␣print␣id,␣date
9  ␣␣␣␣␣␣␣␣start␣=␣0
10 ␣␣}
11 }
```

Output: $(b, t_{REPORTED})$

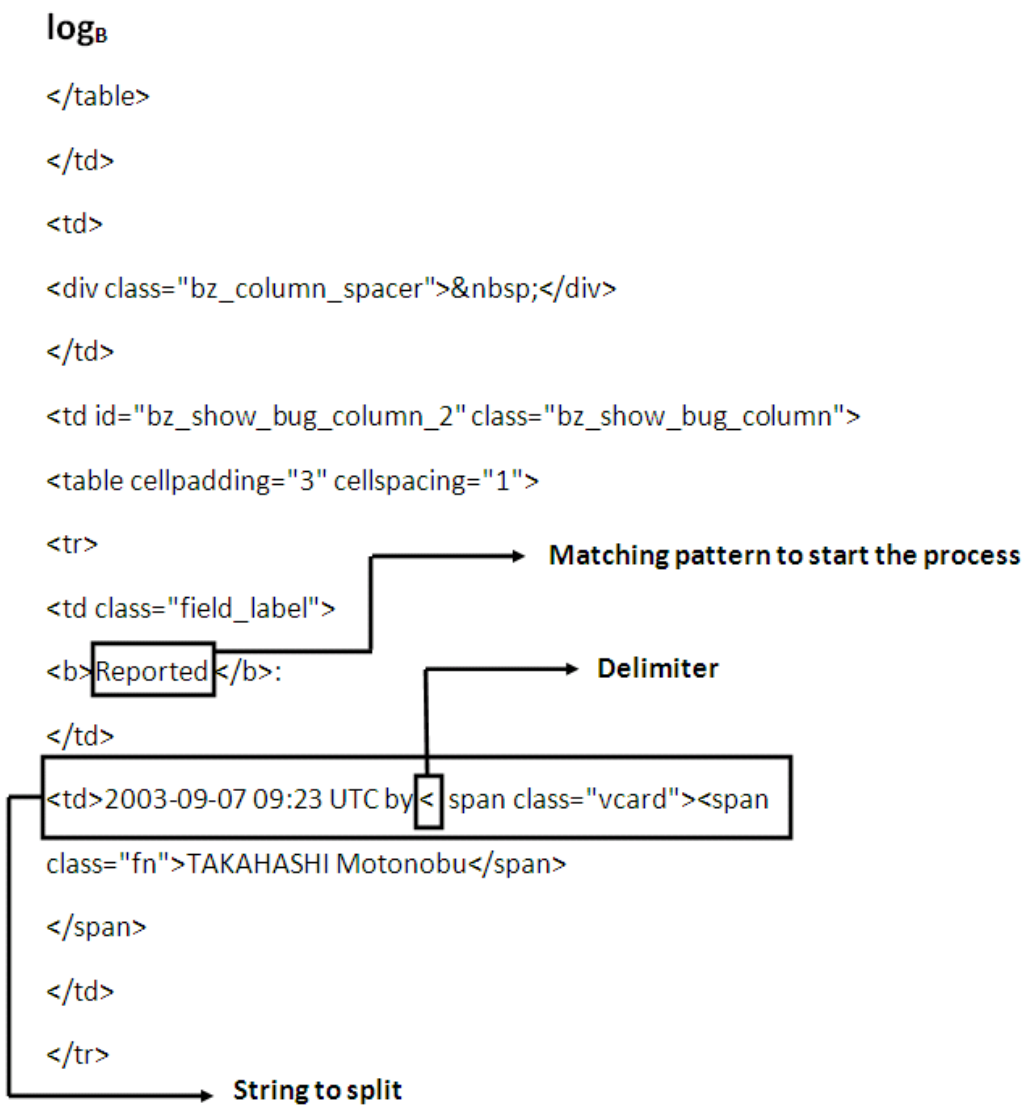

Figure A.1: The example of log_B data to retrieve the value of $t_{REPORTED}$.

A.4 Obtaining $t_{ASSIGNED}$ and $t_{RESOLVED}$: the Bug Event Stamps When the Bug is Assigned to Fix and When the Bug is Fixed

The following "AWK" script is used to get the time stamps when the bug is assigned ($t_{ASSIGNED}$) and resolved ($t_{RESOLVED}$). The log_A is input of the script and the bug id b and time stamps when the bug is assigned $t_{ASSIGNED}$ and resolved $t_{RESOLVED}$ are the outputs.

In the bug activity log log_A, the assigned, resolved and reopened status are added at the time when the bug status changes to any of these states. Therefore, the script starts with the pattern matching string *Added* as indicated on the first line to begin the process. The data to match the pattern "Added" is indicated in the Figure A.2 of log_A example screen-shot.

The values of variables *start* and *last* are set to true by assigning 1 to both of them. The purpose of *start* variable is to process the time stamps at the time of bug assignment. On the other hand, the purpose of the *last* variable is to only get the latest value of time stamps when the bug is resolved, ignoring all the reopening states.

The script is divided into two parts. The purpose of the first part of the script is to process the regular expression by matching the pattern starting with " $< tdrowspan''$. Once the script finds the pattern, it splits the third field of the split string using the split function and stores in the array a using the delimiter " $>''$. The second element of array a is stored in the variable *date* by adding the spaces between the fourth and fifth fields. Then the value of the *start* variable is checked to ensure that the time stamps belong to the time stamp when the bug was assigned. The value of the *date* variable is then assigned to the $firstdate$ variable and the value of the *start* variable is set to zero.

The purpose of the second part of the script is to check to which status of bug the obtained time stamps from the data of log_A belong. To check

that the obtained time stamps belong to the status of the bug when it is assigned, the pattern $< td > ASSIGNED$ is used. In cases where the pattern matches, the value of the $b1$ variable is set to $firstdate$. Similarly, the patterns matching to $< td > RESOLVED$ and $< td > REOPENED$ are matched to check whether the time stamp is related to "resolved" and "reopened" states of the bug. The example of patterns matching the status of the bug are shown in the Figure A.2 of example log_A. In the log_A when the pattern is matched to "Resolved", the variable *last* is set to true by assigning the value 1. However, the value of the *last* variable is set to zero when the pattern is matched to "reopened" state of bug.

In the last part of the script, the value of the *last* variable is checked to be true with the help of condition $last == 1$. If the value is true then the bug id b with the value of assigned $b1$ and resolved $b2$ are printed in the file. Otherwise, the script prints only the value of time stamp when the bug is assigned $b1$.

Input: log_A

```
1  /<th>Added<\/th>/ {
2    start = 1
3    last=-1
4  }
5  /<td rowspan="[0-9]+" valign="top">[0-9]+-[0-9]+/ {
6    split( $3,a,/>/)
7    date=a[2] " " $4 " " $5
8    if (start) {
9      firstdate = date
10     start = 0
11   }
12 }
13 /<td>ASSIGNED/ {
14         b1="ASSIGNED: " firstdate
15 }
16 /<td>RESOLVED/ {
17         b2="RESOLVED: " date
```

```
18          last=1
19  }
20  /<td>REOPENED/ {
21          b3="REOPENED: " date
22          last=0
23  }
24  END {
25          if (last == 1)
26                  print id, b1, b2
27          else
28                  print id, b1
29  }
```

Output: (b, $t_{ASSIGNED}$, $t_{RESOLVED}$)

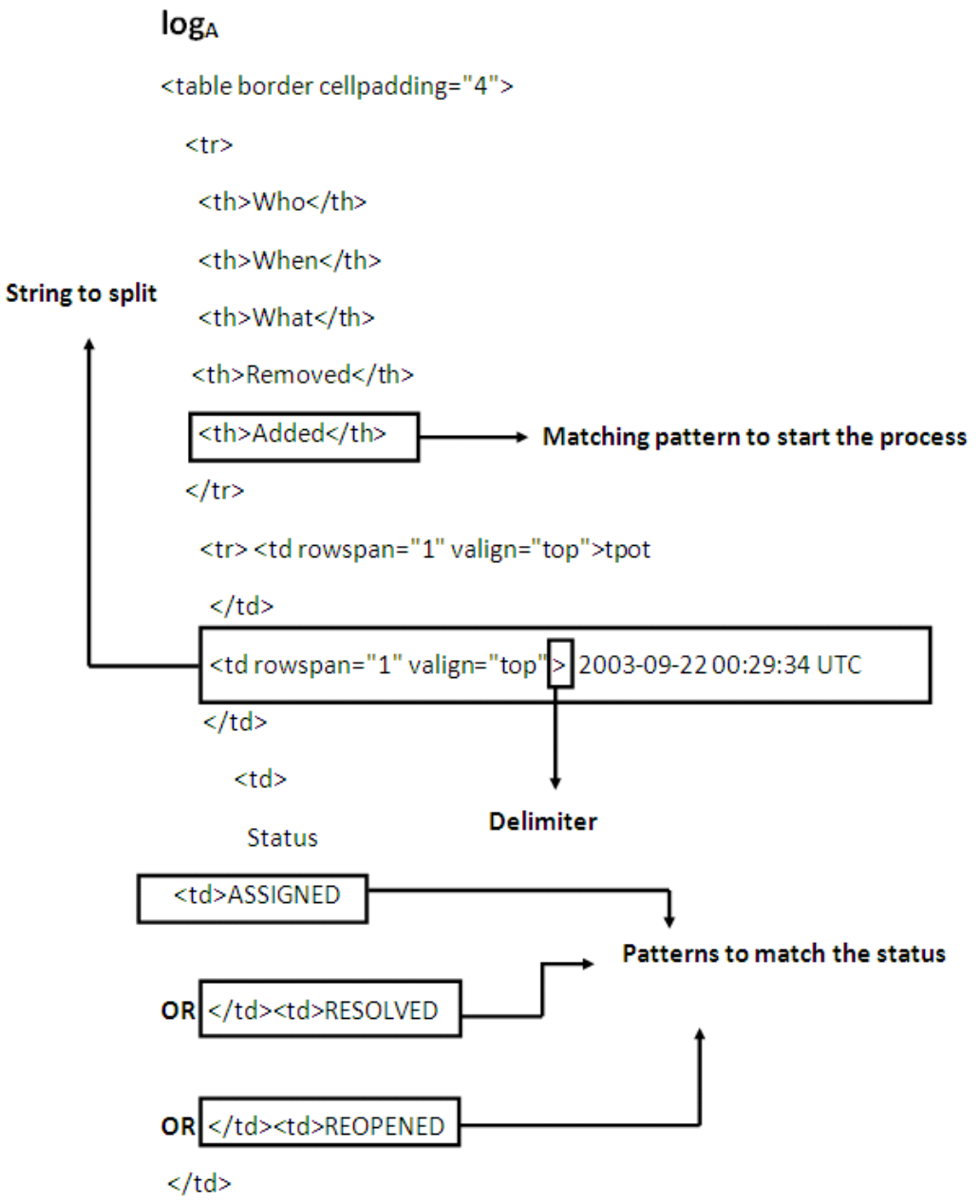

Figure A.2: The example of log_A data to obtain the bug $t_{ASSIGNED}$ and $t_{RESOLVED}$ time stamps.

A.5 Obtaining Call Graphs

To use `Egypt` for generating a call graph, it is important to install the *GCC* version 4.7, which is compatible for the "fdump-rtl-expand" option. This option is not available in the default version 4.2. Through this option *GCC* dumps the given source code file to its intermediate representation with an extension of *.expand*. The following is a statement by statement description of the code used to obtain the call graphs. In the first statement, the folder path to *Samba* in shell has been changed to do operations on the *Samba* code. The second statement is used to install the gcc compiler version4.7. After the installation of the compiler, the command "CC" standing for "C-compiler" is used to generate the expand files from the gcc compiler using the option "fdump-rtl-expand". Before doing this the *Samba* code first needs to be configured. This is done using the *waf* configure command. After the configuration, in the next statement the *Samba* "make file" is generated to build executable *Samba* code. The purpose of "make file" is to organise the code compilations. Then, the expand file output is redirected to the *callgraph.dot* file in the *tmp* folder. Finally, the output is refined by removing the first and last lines *digrpah* and } through searching the files with the *.expand* extension only. The purpose of refining the output is to make sure that it only contains the calling relationship.

```
1 cd ~/samba
2 sudo port install gcc47 // for mac users
3 CC="gcc-mp-4.7 -fdump-rtl-expand" ./buildtools/bin/waf configure
4 CC="gcc-mp-4.7 -fdump-rtl-expand" make
5 echo > /tmp/callgraph.dot /echo commands redirect the output to
     the callgraph.dot file\
6 find . -name "*.expand" | while read filename; do
7   egypt \$filename | grep -v "digraph" | grep -v "\}" >> /tmp/
       callgraph.dot
8 done
```

A.6 Program to Compute the Fan-ins of a Call Graph

```
1  public static HashMap<String, Integer> getFanin() throws Exception {
2      HashMap<String, Set<String>> g;// input
3      HashMap<String, Integer> f = null; // output
4      g = new HashMap<String, Set<String>>();
5      Scanner scanner = new Scanner(new FileReader("callgraph-3.6.12.dot"));
6      while (scanner.hasNextLine()) {
7        String line = scanner.nextLine();
8        String[] labels = (String[]) line.split("\"");
9        if ((labels.length > 3)) {
10         String v_prime = labels[1];
11         String v = labels[3];
12         if (!(g.containsKey(v))) {
13           Set<String> c = new HashSet<String>();
14           c.add((v_prime));
15           g.put(v, c);
16         } else {
17           Set<String> c = g.get(v);
18           c.add(v_prime);
19         }
20       }
21     }
22     f = new HashMap<String, Integer>();
23     for (String v : g.keySet())
24       f.put(v, g.get(v).size());
25     return f;
26   }
```

A.7 Program to Get List of the Functions from the Change Set

```
1  public static HashSet<String> getChangedFunctions(String p)
2    throws Exception
```

```
3  {
4    HashSet<String> v_p = new HashSet<String>();
5    Process proc = Runtime.getRuntime().exec(new String[] { "/bin/
       bash", "-c", "/usr/bin/git␣diff␣" + p + ".." + p + "^" });
6    proc.waitFor();
7    Scanner scanner = new Scanner(new InputStreamReader(proc.
       getInputStream()));
8    while (scanner.hasNextLine()) {
9      String line = scanner.nextLine();
10     String[] labels = (String[]) line.split("\@\@");
11     if ((labels.length > 2) && labels[2].contains("(")) {
12       String[] namesWithoutBrackets = labels[2].split("\\(");
13       String[] names = namesWithoutBrackets[0].split("\\W");
14       String v_delta = names[names.length - 1];
15       v_p.add((v_delta));
16     }
17   }
18   return v_p;
19 }
```

A.8 Program to Obtain the List of Patches

```
1  public static HashSet<String> getPatches()
2    throws Exception
3  {
4    HashSet<String> P = new HashSet<String>();
5    Process proc = Runtime.getRuntime().exec(new String[] { "/bin/
       bash", "-c", "/usr/bin/git␣--no-pager␣log␣>␣t.log"});
6    proc.waitFor();
7    Scanner scanner = new Scanner(new File("t.log"));
8    while (scanner.hasNextLine()) {
9      String line = scanner.nextLine();
10     if (line.startsWith("commit")) {
11       String[] labels = (String[]) line.split("commit");
12       if ((labels.length > 1)) {
13         String p_ID = labels[1];
```

```
14            P.add((p_ID));
15          }
16        }
17      }
18      return P;
19  }
```

A.9 Program to Compute Accumulative Fan-ins of all the Patches

```
1  public static float getAccumulativeFanin(HashSet<String> P,
       HashMap <String, Integer> F_v) throws Exception {
2       HashMap<String, Integer> F_p = new HashMap<String, Integer
            >();
3       for (String p : P) {
4         HashSet<String> V_p = getChangedFunctions(p);
5         int f_p =0;
6         for(String v : V_p)
7           if (F_v.get(v)!=null)
8             f_p= f_p + F_v.get(v);
9         F_p.put(p, f_p);
10      }
```

A.10 Program to Compute Average Fan-ins of all the Patches

```
1  public static float getAverageFanin(HashSet<String> P, HashMap <
       String, Integer> F_p) throws Exception {
2       int sum_f_p = 0;
3       for (String p : P)
4         sum_f_p = sum_f_p + F_p.get(p);
5       avg_f_p = (float) sum_f_p/P.size();
6       return avg_f_p;
7     }
```

A.11 Program to Obtain the List of Bugs and their Corresponding Patches

```
1  // precondition: t.log always starts with "commit"
2  public static HashMap<String, HashSet<String>> getBugsPatches()
3  throws Exception {
4   HashMap<String, HashSet<String>> B_P = new HashMap<String,
        HashSet<String >>();
5   Scanner scanner = new Scanner(new File("t.log"));
6   String patchID = null;
7   String matchingString = null;
8   String firstLine = null;
9   while (scanner.hasNext()) {
10  if(firstLine==null){
11    firstLine = scanner.nextLine();
12  }
13  else if (firstLine.startsWith(("commit"))) {
14    String[] commitLine = (String[]) firstLine.split("commit");
15    patchID=commitLine[1];
16    do {
17          Pattern pattern = Pattern.compile(".*[^a-z](bug|fix|
                correct)\\D*(\\d+).*",
18             Pattern.CASE_INSENSITIVE);
19      matchingString = scanner.nextLine();
20          Matcher matcher = pattern.matcher(matchingString);
21          if (matcher.find()) {
22            String bugID = matcher.group(1);
23            if(!(B_P.containsKey(bugID))){
24                  HashSet<String> P = new HashSet<String>();
25                  P.add(patchID);
26                  B_P.put(bugID, P);
27            }
28        else{
29                  HashSet<String> P = B_P.get(bugID);
30                  P.add(patchID);
31                  B_P.put(bugID, P);
```

```
32              }
33            }
34        firstLine=matchingString;
35      } while (!(firstLine.startsWith("commit"))&& scanner.hasNext
          ());
36    }
37   }
38   return B_P;
39 }
```

A.12 Program to Get Bugs' Fan-ins

```
1  public static HashMap<String, Integer> getBugsFanin(HashMap<
       String,
2    HashSet<String>> B_P,
3          HashMap<String, Integer> F_p)throws Exception {
4          HashMap<String, Integer> F_B = new HashMap<String,
               Integer>();
5          for(String  b : B_P.keySet()){
6            int f_b=0;
7            for(String p : B_P.get(b))
8                  f_b= f_b+ F_p.get(p);
9                  if(f_b!=0)
10         F_B.put(b, f_b);
11       }
12   return  F_B;
13 }
```

A.13 Program to Get Average Fan-ins of all the Bugs

```
1  public static float getAverageFaninBugs(HashMap<String, Integer>
       F_B)
2    throws Exception {
3    float avg_f_B=0;
```

```
4     int sum_f_b=0;
5     for(String  b : F_B.keySet())
6         if(F_B.get(b)!=null)
7         sum_f_b= sum_f_b + F_B.get(b);
8     avg_f_B=sum_f_b/F_B.size();
9     return  avg_f_B;
10  }
```

A.14 Program to Get Average Fan-ins of CVE and Non-CVE Bugs

```
1  // precondition: first line starts with the bug id
2  // output: the program will output fan-in of CVE bugs if the
       provided to it contains CVE relevant bug. On the other hand,
       if the file provided to it contains the Non-CVE bugs then the
        program will output the fan-ins of Non-CVE bugs
3  public static float getAverageFaninBugs(File file, HashMap<
       String, Integer>  F_B) throws Exception {
4     HashSet<String> B_withtype = new HashSet<String>();
5     Scanner scanner = new Scanner(file);
6     while (scanner.hasNext()) {
7             String bugID = scanner.nextLine();
8             B_withtype.add(bugID);
9     }
10    int sum_f_b=0;
11    for(String b : B_withtype){
12      if(F_B.get(b)!=null)
13          sum_f_b = sum_f_b + F_B.get(b);
14    }
15    float avg_f_B= (float) sum_f_b/B_withtype.size();
16    return avg_f_B;
17 }
```

Appendix B

Definitions and Calculations

In this appendix section, the first subsection explains the symbols and definitions used in Chapter 3 and Chapter 4 of the thesis. The second subsection describes the *p-value* and the screen-shots of *p-values* of all the case studies derived from the Statistical Package for the Social Sciences (`SPSS`).

B.1 Definitions of Symbols used in the Chapter 3 and Chapter 4

In this section, the symbols and definitions used in Chapter 3 and Chapter 4 have been provided respectively.
In the table B.1, each row introduces a symbol and its definitions used in the 3, and the instance of the symbol in the *Samba* case study.

Table B.1: Symbols and definitions used in Chapter 3.

Symbol	Description	Definition	E.g. *Samba*
$LOGS$	A set of bug entries log_B and bug activities log_A	Definition 1	6171-items
B	A set of bugs	Definition 1	6171-items
B_s	The set of security bugs B_s is the subset of bugs B	Definition 1	9-items
C	A set of derived bug life cycle from L	Definition 2	Omitted
E	A set of bug fixing event stamps	Definition 3	Omitted
τ_1	Duration of bugs triage time set	Definition 4	22369.05-days
τ_2	Duration of bugs fixing time set	Definition 4	121396.35-days
R_τ	A set of weekly release dates	Definition 5	Omitted
${\tau'}_1$	A set of precise triage duration based on weekly release fix	Definition 6	Omitted
$\lambda(B)$	The planned rush in triaging of bugs	Definition 7	Omitted

Similarly, in the table B.2, each row introduces a symbol and its definitions used in the 4, and the instances of symbol in the *Samba* case study.

Table B.2: Symbols and definitions used in Chapter 4.

Symbol	Description	Definition	E.g. *Samba*
$G =< V, E >$	A directed graph of function calls (i.e., call graph)	Definition 8	1.00
F_v	The set of function fan-ins of a call graph	Definition 9	2553.00
C_v	The set of callers of a function	Definition 9	Omitted
L	A set of commit logs	Definition 10	Omitted
P	A set of all the patches	Definition 10	156694.00
p	A set of changes as a patch	Definition 11	Omitted
Δ	A change to the code	Definition 11	Omitted
V_p	A subset of functions being changed by a patch p	Definition 11	Omitted
F_P	The set of patch fan-ins	Definition 12	458958
f_P	The accumulative fan-in of a patch	Definition 12	2.92
B	The set of bugs	Definition 13	2540
B_P	The set of bugs and their corresponding patches	Definition 13	2540
F_B	A set of bugs fan-ins	Definition 14	Omitted
f_b	The sum of bugs fan-ins	Definition 14	21306.00
$\bar{f}_B$	The average fan-in of a set of bugs	Definition 14	8.39
B_S	The set of given security bugs	Definition 15	9
f_{b_s}	The sum of security bug fan-ins	Definition 15	317.00
$\bar{f}_{B_s}$	The average fan-in of security bugs	Definition 15	26.42

5.2 Calculating *p-value* to Evaluate the Statistical Significance of Rush

The *p-value* is the probability statistical value to test the significance of sample population data. The *Mann Whitney U-test* to statistically evaluate the significance of rush in all the three case studies has been chosen based on the *p-value* using predetermined threshold of 0.05. For applying the *Mann*

Whitney U-test, The standard SPSS [1] statistical package has been used for the statistical analysis of data.

[1] http://en.wikipedia.org/wiki/SPSS

Bibliography

Abandah, H. and Alsmadi, I. (2013), 'Call graph based metrics to evaluate software design quality', *The International Journal of Software Engineering and Its Applications,* **7**(1), pp. 1–12.

Ahmed, M. F. and Gokhale, S. S. (2009), 'Linux bugs: Life cycle, resolution and architectural analysis', *The Journal of Information and Software Technology,* **51**(11), pp. 1618–1627.

Ahmed, M. and Gokhale, S. (2008), Linux bugs: Life cycle and resolution analysis, *in* 'Proceeding of the 8th International Conference on Quality Software', QSIC '08, pp. 396–401.

Albrecht, A. and Gaffney, J. E. (1983), 'Software function, source lines of code, and development effort prediction: A software science validation', *The Journal of IEEE Transactions on Software Engineering,* **SE-9**(6), pp. 639–648.

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

Anvik, J., Hiew, L. and Murphy, G. C. (2005), Coping with an open bug repository, *in* 'Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange', ACM, pp. 35–39.

Anvik, J., Hiew, L. and Murphy, G. C. (2006), Who should fix this bug?, *in* 'Proceedings of the 28th International Conference on Software Engineering', ICSE '06, ACM, pp. 361–370.

Arce, I. (2002), 'Bug hunting: the seven ways of the security samurai', *The Computer Journal,* **35**(4), pp. 11–15.

Banker, R. D., Datar, S. M. and Zweig, D. (1989), Software complexity and maintainability, *in* 'Proceedings of the 10th International Conference on Information Systems', ICIS '89, ACM, pp. 247–255.

Baysal, O., Holmes, R. and Godfrey, M. (2012), Revisiting bug triage and resolution practices, *in* 'Proceeding of the 1st International Workshop on User Evaluation for Software Engineering Researchers', USER '12, pp. 29 –30.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R. and Zimmermann, T. (2008), What makes a good bug report?, *in* 'Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering', SIGSOFT '08/FSE-16, ACM, pp. 308–318.

Bhattacharya, P., Ulanova, L., Neamtiu, I. and Koduru, S. (2013), An empirical analysis of bug reports and bug fixing in open source android apps, *in* 'Proceeding of the 17th European Conference on Software Maintenance and Reengineering', CSMR '13, pp. 133–143.

Caglayan, B., Misirli, A. T., Miranskyy, A., Turhan, B. and Bener, A. (2012), Factors characterizing reopened issues: a case study, *in* 'Proceedings of the 8th International Conference on Predictive Models in Software Engineering', PROMISE '12, ACM, pp. 1–10.

Chen, K., Schach, S. R., Yu, L., Offutt, J. and Heller, G. Z. (2004), 'Open-source change logs', *The Journal of Empirical Software Engineering,* **9**(3), pp. 197–210.

Cheney, J. S. (2010), 'Heartland payement systems: Lesson learned from a data breach', http://philadelphiafed.org/consumer-credit-and-payments/payment-cards-center/publications/discussion-papers/2010/D-2010-January-Heartland-Payment-Systems.pdf. [Online; accessed 04-April-2014].

Crowston, K. and Scozzi, B. (2008), 'Bug fixing practices within Free/Libre open source software development teams', *The Journal of Database Management,* **19**(2), pp. 1–30.

Cubranic, D. and Murphy, G. (2003), Hipikat: recommending pertinent software development artifacts, *in* 'Proceeding of the 25th International Conference on Software Engineering', ICSE '03, pp. 408–418.

do Rego, R., Ribeiro, M., Aleixo, E. and De Souza, R. M. C. R. (2008), Bug reports retrieval using self-organizing map, *in* 'Proceeding of the 3rd International Conference on Digital Information Management, ICDIM 08', pp. 320–325.

Donohue, B. (2014), 'Hackers milk ie zero day before patch', http://threatpost.com/hackers-milk-ie-zero-day-before-patch/104713. [Online; accessed 07-April-2014].

Easterbrook, S., Singer, J., Storey, M.-A. and Damian, D. (2008), Selecting empirical methods for software engineering research, *in* 'Guide to advanced empirical software engineering', Springer London, pp. 285–311.

Finkle, J. (2013), 'Oracle corp to fix java security flaw shortly', http://www.reuters.com/article/2013/01/12/us-usa-java-security-idUSBRE90B0EX20130112. [Online; accessed 04-April-2014].

Fischer, M., Pinzger, M. and Gall, H. (2003), Populating a release history database from version control and bug tracking systems, *in* 'Proceeding

of the 19th International Conference on Software Maintenance, ICSM 03', pp. 23–32.

Florian, C. (2014), 'Report: Most vulnerable operating systems and applications in 2013', http://www.gfi.com/blog/report-most-vulnerable-operating-systems-and-applications-in-2013/. [Online; accessed 07-April-2014].

Francalanci, C. and Merlo, F. (2008), Empirical analysis of the bug fixing process in open source projects, *in* B. Russo, E. Damiani, S. Hissam, B. Lundell and G. Succi, eds, 'Open Source Development, Communities and Quality', Vol. 275 of *IFIP International Federation for Information Processing*, Springer, pp. 187–196.

Gegick, M., Rotella, P. and Xie, T. (2010), Identifying security bug reports via text mining: An industrial case study, *in* 'Proceeding of the 7th IEEE Working Conference on Mining Software Repositories', MSR '10, pp. 11–20.

Giger, E., Pinzger, M. and Gall, H. (2010), Predicting the fix time of bugs, *in* 'Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering', ACM, pp. 52–56.

Grubb, P. and Takang, A. A. (2003), *Software maintenance: concepts and practice*, World Scientific Publishing Company Incorporated.

Guo, P. J., Zimmermann, T., Nagappan, N. and Murphy, B. (2011), Not my bug! and other reasons for software bug report reassignments, *in* 'Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work', ACM, pp. 395–404.

Guo, P., Zimmermann, T., Nagappan, N. and Murphy, B. (2010), Characterizing and predicting which bugs get fixed: an empirical study of microsoft

windows, *in* 'Proceeding of the ACM/IEEE 32nd International Conference on Software Engineering', pp. 495 –504.

Haley, C., Laney, R., Moffett, J. and Nuseibeh, B. (2008), 'Security requirements engineering: A framework for representation and analysis', *The Journal of IEEE Transactions on Software Engineering* **34**(1), pp. 133–153.

Henry, S. and Selig, C. (1990), 'Predicting source-code complexity at the design stage', *The Journal of IEEE Software,* **7**(2), pp. 36–44.

Hooimeijer, P. and Weimer, W. (2007), Modeling bug report quality, *in* 'Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering', ASE '07, ACM, pp. 34–43.

Huang, J. Gotel, O. and Zisman, A. (2012), *Software and Systems Traceability*, Springer Books.

Hui Yang, de Roeck, A., Gervasi, V., Willis, A. and Nuseibeh, B. (2011), 'Analysing anaphoric ambiguity in natural language requirements', *The Journal of Requirements Engineering* **16**(3), pp. 163–189.

Höst, M., Regnell, B. and Wohlin, C. (2000), 'Using students as subjects -—A comparative study of students and professionals in lead-time impact assessment', *The Journal of Empirical Software Engineering,* **5**(3), pp. 201–214.

In Orbit, L. R. (2011), 'Tools to get a pictorial function call graph of code', http://stackoverflow.com/questions/517589/tools-to-get-a-pictorial-function-call-graph-of-code. [Online; accessed 03-mayl-2014].

Jain, V., Rath, A. and Ramaswamy, S. (2012), Field weighting for automatic bug triaging systems, *in* 'Proceeding of the IEEE International Conference on Systems, Man, and Cybernetics', SMC '12, pp. 2845–2848.

Jeong, G., Kim, S. and Zimmermann, T. (2009), Improving bug triage with bug tossing graphs, *in* 'Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering', ESEC/FSE '09, ACM, pp. 111–120.

Johnson, P. M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., Zhen, S. and Doane, W. E. J. (2003), Beyond the personal software process: Metrics collection and analysis for the differently disciplined, *in* 'Proceedings of the 25th International Conference on Software Engineering', ICSE '03, IEEE Computer Society, pp. 641–646.

Jongyindee, A., Ohira, M., Ihara, A. and Matsumoto, K.-i. (2012), 'Good or bad committers? — a case study of committer's activities on the eclipse's bug fixing process', *The Journal of Transactions on Information and Systems,* **E95-D**(9), pp. 2202–2210.

Kasuya, E. (2001), 'Mann–Whitney u test when variances are unequal', *The Journal of Animal Behaviour,* **61**(6), pp. 1247–1249.

Kim, S. and Ernst, M. D. (2007), Which warnings should i fix first?, *in* 'Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering', ESEC-FSE '07, ACM, pp. 45–54.

Kim, S., Pan, K. and Whitehead Jr, E. E. (2006), Memories of bug fixes, *in* 'Proceedings of the 14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering', ACM, pp. 35–45.

Kim, S. and Whitehead, Jr., E. (2006), How long did it take to fix bugs?, *in* 'Proceedings of the 2006 International Workshop on Mining Software Repositories', MSR '06, ACM, pp. 173–174.

Ko, A., Myers, B., Coblenz, M. and Aung, H. (2006), 'An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks', *The Journal of IEEE Transactions on Software Engineering,* **32**(12), pp. 971–987.

König, C. and Kleinmann, M. (2005), 'Deadline rush: A time management phenomenon and its mathematical description', *The Journal of Psychology,* **139**(1), pp. 33–45.

Kuan, J. (2004), Is open source software" better" than closed source software? using bug-fix rates to compare software quality, *in* 'Industry Studies Association Working Papers', Industry Studies Association.

Kula, R. G., Fushida, K., Kawaguchi, S. and Iida, H. (2010), Analysis of bug fixing processes using program slicing metrics, *in* M. A. Babar, M. Vierimaa and M. Oivo, eds, 'Product-Focused Software Process Improvement', number 6156 *in* 'Lecture Notes in Computer Science', Springer, pp. 32–46.

Lamkanfi, A. and Demeyer, S. (2012), Filtering bug reports for fix-time analysis, *in* 'Proceeding of the 16th European Conference on Software Maintenance and Reengineering', CSMR'12, IEEE, pp. 379–384.

Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y. and Zhai, C. (2006), Have things changed now?: an empirical study of bug characteristics in modern open source software, *in* 'Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability', ASID '06, ACM, pp. 25–33.

Loeliger, J. and McCullough, M. (2012), *Version Control with Git: Powerful tools and techniques for collaborative software development*, " O'Reilly Media, Inc.".

MacKenzie, D., Eggert, P. and Stallman, R. (2003), *Comparing and Merging Files with GNU diff and patch*, Network Theory Ltd.

Mani, S., Nagar, S., Mukherjee, D., Narayanam, R., Sinha, V. S. and Nanavati, A. A. (2013), Bug resolution catalysts: Identifying essential non-committers from bug repositories, *in* 'Proceedings of the 10th Working Conference on Mining Software Repositories', MSR '13, IEEE Press, pp. 193–202.

Margaret, R. (2007), 'What is buffer overflow', http://searchsecurity.techtarget.com/definition/buffer-overflow. [Online; accessed 04-December-2014].

Marin, M., Deursen, A. V. and Moonen, L. (2007), 'Identifying crosscutting concerns using fan-in analysis', *The Journal of ACM Transactions on Software Engineering and Methodology,* **17**(1), p. 3.

Marin, M., Van Deursen, A. and Moonen, L. (2004), Identifying aspects using fan-in analysis, *in* 'Proceeding of the 11th Working Conference on Reverse Engineering', WCRE '04, pp. 132–141.

Marks, L., Zou, Y. and Hassan, A. E. (2011), Studying the fix-time for bugs in large open source projects, *in* 'Proceedings of the 7th International Conference on Predictive Models in Software Engineering', Promise '11, ACM, pp. 11:1–11:8.

McCabe, T. J. (1976), 'A complexity measure', *The Journal of IEEE Transactions on Software Engineering,* **SE-2**(4), pp. 308–320.

Mitropoulos, D., Gousios, G. and Spinellis, D. (2012), Measuring the occurrence of security-related bugs through software evolution, *in* 'Proceeding of the 16th Panhellenic Conference on Informatics', PCI '12, pp. 117 –122.

Mockus, A., Fielding, R. T. and Herbsleb, J. D. (2002), 'Two case studies of open source software development: Apache and mozilla', *The Journal of ACM Transactions on Software Engineering and Methodology,* **11**(3), pp. 309–346.

Mozilla (2014), 'Handling mozilla security bugs', https://www.mozilla.org/en-US/about/governance/policies/security-group/bugs/. [Online; accessed 24-April-2014].

Mubarak, A., Counsell, S. and Hierons, R. (2010), An evolutionary study of fan-in and fan-out metrics in OSS, *in* 'Proceeding of the 4th International Conference on Research Challenges in Information Science', RCIS '10, pp. 473–482.

Murphy-Hill, E., Zimmermann, T., Bird, C. and Nagappan, N. (2013), The design of bug fixes, *in* 'Proceedings of the 35th International Conference on Software Engineering', ICSE '13, IEEE Press, pp. 332–341.

Nagaraju, S. S., Craioveanu, C., Florio, E. and Matt, M. (2013), 'Software vulnerability exploitation trends: Exploring the impact of software mitigations on patterns of vulnerability exploitation', http://www.microsoft.com/en-gb/download/details.aspx?id=39680. [Online; accessed 07-April-2014].

Nistor, A., Jiang, T. and Tan, L. (2013), Discovering, reporting, and fixing performance bugs, *in* 'Proceedings of the 10th Working Conference on Mining Software Repositories', MSR '13, IEEE Press, pp. 237–246.

Paulson, J., Succi, G. and Eberlein, A. (2004), 'An empirical study of open-source and closed-source software products', *The Journal of IEEE Transactions on Software Engineering,* **30**(4), pp. 246–256.

Peralta, A., Romero, F., Olivas, J. and Polo, M. (2010), Knowledge extraction of the behaviour of software developers by the analysis of time recording logs, *in* 'Proceeding of the 19th IEEE International Conference on Fuzzy Systems', FUZZ '10, pp. 1–8.

Pfleeger, C. P. and Pfleeger, S. L. (2006), *Security in Computing (4th Edition)*, Prentice Hall PTR.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J. and Wang, B. (2003), Automated support for classifying software failure reports, *in* 'Proceeding of the 25th International Conference on Software Engineering', ICSE '13, pp. 465–475.

Prechelt, L. and Unger, B. (2001), 'An experiment measuring the effects of personal software process (PSP) training', *The Journal of IEEE Transactions on Software Engineering,* **27**(5), pp. 465–472.

Rasch, R. H. and Tosi, H. L. (1992), 'Factors affecting software developers' performance: An integrated approach', *The Journal MIS Quarterly,* **16**(3), pp. 395–413.

Ryder, B. (1979), 'Constructing the call graph of a program', *The Journal of IEEE Transactions on Software Engineering* **SE-5**(3), pp. 216–226.

Saleem, S. B., Montrieux, L., Yu, Y., Tun, T. T. and Nuseibeh, B. (2013), Maintaining security requirements of software systems using evolving crosscutting dependencies, *in* 'Aspect-Oriented Requirements Engineering', Springer, pp. 167–181.

Serrano, N. and Ciordia, I. (2005), 'Bugzilla, ITracker, and other bug trackers', *The Journal of IEEE Software,* **22**(2), pp. 11–13.

Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W., Ohira, M., Adams, B., Hassan, A. and Matsumoto, K. (2010), Predicting re-opened bugs: A case study on the eclipse project, *in* 'Proceeding of the 17th Working Conference on Reverse Engineering', WCRE '10, pp. 249 –258.

Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W., Ohira, M., Adams, B., Hassan, A. and Matsumoto, K.-i. (2012), 'Studying re-opened bugs in open source software', *The Journal of Empirical Software Engineering,* **18**(5), pp. 1005–1042.

Shin, Y. and Williams, L. (2008), An empirical model to predict security vulnerabilities using code complexity metrics, *in* 'Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement', ESEM '08, ACM, pp. 315–317.

Sillitti, A., Janes, A., Succi, G. and Vernazza, T. (2003), Collecting, integrating and analyzing software metrics and personal software process data, *in* 'Proceeding of the 29th Euromicro Conference', pp. 336–342.

Sliwerski, J., Zimmermann, T. and Zeller, A. (2005), When do changes induce fixes?, *in* 'Proceedings of the 2005 International Workshop on Mining Software Repositories', MSR '05, ACM, pp. 1–5.

Stamelos, I., Angelis, L., Oikonomou, A. and Bleris, G. L. (2002), 'Code quality analysis in open source software development', *The Journal of Information Systems,* **12**(1), pp. 43–60.

Svahnberg, M., Gorschek, T., Feldt, R., Torkar, R., Saleem, S. B. and Shafique, M. U. (2010), 'A systematic review on strategic release planning models', *The Journal of Information and Software Technology* **52**(3), pp. 237–248.

The Bugzilla Team (2012), 'The bugzilla guide - 4.2.3 release', http://www.bugzilla.org/docs/4.2/en.

Thung, F., Lo, D. and Jiang, L. (2013), Automatic recovery of root causes from bug-fixing changes, *in* 'Proceeding of the 20th Working Conference on Reverse Engineering', WCRE '13, IEEE, pp. 92–101.

Tian, Y., Lawall, J. and Lo, D. (2012), Identifying linux bug fixing patches, *in* 'Proceeding of the 34th International Conference on Software Engineering', ICSE '12, pp. 386–396.

Viega, J. and McGraw, G. (2011), *Building Secure Software: How to Avoid Security Problems the Right Way (Paperback) (Addison-Wesley Professional Computing Series)*, 1st edn, Addison-Wesley Professional.

Vijayan, J. (2010), 'Heartland breach expenses pegged at $140m – so far', http://www.computerworld.com/s/article/9176507/Heartland_breach_expenses_pegged_at_140M_so_far. [Online; accessed 04-April-2014].

Wang, D., Zhang, H., Liu, R., Lin, M. and Wu, W. (2012), 'Predicting bugs' components via mining bug reports.', *The Journal of Software,* **7**(5), pp. 1149–1154.

Weiss, C., Premraj, R., Zimmermann, T. and Zeller, A. (2007), How long will it take to fix this bug?, *in* 'Proceedings of the 4th International Workshop on Mining Software Repositories', MSR '07, IEEE Computer Society, p. 1.

Wu, L. L., Xie, B., Kaiser, G. E. and Passonneau, R. (2011), BUGMINER: software reliability analysis via data mining of bug reports, Technical Report cucs-024-11, Department of Computer Science, Columbia University.

Wurster, G. and van Oorschot, P. C. (2008), The developer is the enemy, *in* 'Proceedings of the 2008 Workshop on New Security Paradigms', ACM, pp. 89–97.

Wynekoop, J. L. and Walz, D. B. (2000), 'Investigating traits of top performing software developers', *The Journal of Information Technology and People,* **13**(3), pp. 186–195.

Xia, X., Lo, D., Wang, X. and Zhou, B. (2013), Accurate developer recommendation for bug resolution, *in* 'Proceeding of the 20th Working Conference on Reverse Engineering', WCRE '13, IEEE, pp. 72–81.

Xiao, J. and Afzal, W. (2010), Search-based resource scheduling for bug fixing tasks, *in* 'Proceeding of the 2nd International Symposium on Search Based Software Engineering', SSBSE '10, pp. 133 –142.

Yanyan, Z. and Renzuo, X. (2008), The basic research of human factor analysis based on knowledge in software engineering, *in* 'Proceeding of the 2008 International Conference on Computer Science and Software Engineering', Vol. 5, pp. 1302–1305.

Ye, Y. and Kishida, K. (2003), Toward an understanding of the motivation of open source software developers, *in* 'Proceeding of the 25th International Conference on Software Engineering', pp. 419–429.

Yin, R. K. (2014), *Case study research: Design and methods*, Sage publications.

Yu, Y., Jurjens, J. and Mylopoulos, J. (2008), Traceability for the maintenance of secure software, *in* 'Proceeding of the IEEE International Conference on Software Maintenance', ICSM '08, pp. 297–306.

Yu, Y., Leite, J. C. S. d. P. and Mylopoulos, J. (2004), From goals to aspects: Discovering aspects from requirements goal models, *in* 'Proceedings of the 12t IEEE International Requirements Engineering Conference', RE '04, IEEE Computer Society, pp. 38–47.

Zaman, S., Adams, B. and Hassan, A. E. (2011), Security versus performance bugs: a case study on firefox, *in* 'Proceedings of the 8th Working Conference on Mining Software Repositories', MSR '11, ACM, pp. 93–102.

Zeller, A. (2002), Isolating cause-effect chains from computer programs, *in* 'Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering', SIGSOFT '02/FSE-10, ACM, pp. 1–10.

Zeller, A. (2005), *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann Publishers Inc.

Zhang, D., Guo, Y. and Chen, X. (2008), Automated aspect recommendation through clustering-based fan-in analysis, *in* 'Proceeding of the 23rd

IEEE/ACM International Conference on Automated Software Engineering', ASE '08, IEEE, pp. 278–287.

Zhang, F., Khomh, F., Zou, Y. and Hassan, A. (2012), An empirical study on factors impacting bug fixing time, *in* 'Proceeding of the 19th Working Conference on Reverse Engineering', WCRE '12, pp. 225–234.

Zhang, H., Gong, L. and Versteeg, S. (2013), Predicting bug-fixing time: An empirical study of commercial software projects, *in* 'Proceedings of the 35th International Conference on Software Engineering', ICSE '13, IEEE Press, pp. 1042–1051.

Zhang, H., Zhang, X. and Gu, M. (2007), Predicting defective software components from code complexity measures, *in* 'Proceeding of the 13th Pacific Rim International Symposium on Dependable Computing', PRDC '07, pp. 93–96.

Zhong, X., Madhavji, N. and El Emam, K. (2000), 'Critical factors affecting personal software processes', *The Journal of IEEE Software,* **17**(6), pp. 76–83.

Zhou, J., Zhang, H. and Lo, D. (2012), Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports, *in* 'Proceeding of the 34th International Conference on Software Engineering', ICSE '12, pp. 14–24.