

A NEAR-MISS ANALYSIS MODEL FOR IMPROVING THE FORENSIC INVESTIGATION OF SOFTWARE FAILURES

by

MADELEINE ADRIENNE BIHINA BELLA

submitted in fulfilment of the requirements for the degree

Doctor of Philosophy (COMPUTER SCIENCE)

in the

**FACULTY OF ENGINEERING, BUILT ENVIRONMENT AND
INFORMATION TECHNOLOGY**

at the

**UNIVERSITY OF PRETORIA
PRETORIA, SOUTH AFRICA**

SUPERVISOR: Prof. J.H.P. Eloff

Date of submission
22-12-2014

Abstract

The increasing complexity of software applications can lead to operational failures that have disastrous consequences. In order to prevent the recurrence of such failures, a thorough post-mortem investigation is required to identify the root causes involved. This root-cause analysis must be based on reliable digital evidence to ensure its objectivity and accuracy. However, current approaches to software failure analysis do not promote the collection of digital evidence for causal analysis. This leaves the system vulnerable to the reoccurrence of a similar failure.

A promising alternative is offered by the field of digital forensics. Digital forensics uses proven scientific methods and principles of law to determine the cause of an event based on forensically sound evidence. However, being a reactive process, digital forensics can only be applied after the occurrence of costly failures. This limits its effectiveness as volatile data that could serve as potential evidence may be destroyed or corrupted after a system crash.

In order to address this limitation of digital forensics, it is suggested that the evidence collection be started at an earlier stage, before the software failure actually unfolds, so as to detect the high-risk conditions that can lead to a major failure. These forerunners to failures are known as near misses. By alerting system users of an upcoming failure, the detection of near misses provides an opportunity to collect at runtime failure-related data that is complete and relevant.

The detection of near misses is usually performed through electronic near-miss management systems (NMS). An NMS that combines near-miss analysis and digital forensics can contribute significantly to the improvement of the accuracy of the failure analysis. However, such a system is not available yet and its design still presents several challenges due to the fact that neither digital forensics nor near-miss analysis is currently used to investigate software failures and their existing methodologies and processes are not directly applicable to failure analysis.

This research therefore presents the architecture of an NMS specifically designed to address the above challenges in order to facilitate the accurate forensic investigation of software failures. The NMS focuses on the detection of near misses at runtime with a view to maximising the collection of appropriate digital evidence of the failure. The detection process is based on a mathematical model that was developed to formally define a near miss and calculate its risk level. A prototype of the NMS has been implemented and is discussed in the thesis.

Summary

Title: A near-miss analysis model for improving the forensic investigation of software failures

Candidate: Madeleine Adrienne Bihina Bella

Supervisor: Prof J.H.P. Eloff

Department: Computer Science Department, Faculty of Engineering, Built Environment and Information Technology, University of Pretoria

Degree: Doctor of Philosophy in Computer Science

Keywords: software failure, failure analysis, digital forensics, near miss, near-miss analysis, forensic investigation, root-cause analysis

Acknowledgements

I would like to thank the following people for their contribution to the success of this work.

- Prof J.H.P Eloff, for his continuous guidance, support and thorough supervision throughout this study. He made the completion of this thesis possible and provided tremendous assistance and mentorship to enrich its content and to enrich my personal experience and exposure as a research student. He was a great mentor and an endless source of inspiration.
- SAP Innovation Center, which provided the infrastructure, the facilities and the research environment for this PhD work. They offered me financial support throughout the duration of this study, invaluable study time as well as exposure to world-class research and innovation to help me raise the level of my study to high international standards. I particularly would like to thank Danie Kok, the founder of the Innovation Center, and Dr E. Ngassam, Principal Researcher, for their support, encouragement and guidance throughout the various challenges I faced during my time as a PhD student in the center.
- The various institutions which offered me financial support through research awards and believed in the quality and value of my research. They also provided me with media coverage for my research work to raise my profile as a researcher. Their acknowledgements helped me build my confidence in my research abilities. These institutions and organisations are as follows: l'Oréal, the giant international cosmetics company, UNESCO (United Nations Educational, Scientific and Cultural Organisation), Google, provider of the well-known web-based search engine, and the Department of Science of Technology (DST).
- Last but not least, I would like to express my special gratitude to my parents for their constant support and encouragement.

Table of Contents

CHAPTER 1 INTRODUCTION	1
1.1 Introduction	1
1.2 Thesis statement	5
1.3 Problem statement	5
1.4 Research Questions	5
1.5 Scope and context of the study	6
1.6 Research methodology	7
1.7 Terminology used in the thesis	8
1.8 Defining the near-miss concept	10
1.8.1 Current definitions of near miss	10
1.8.2 Proposed definition of a near miss for software systems	14
1.9 Layout of thesis	15
CHAPTER 2 SOFTWARE FAILURES: OVERVIEW OF RECENT CASES	18
2.1 Introduction	18
2.2 Background on software failures	19
2.2.1 Definition of a software failure	19
2.2.2 Common causes of software failures	20
2.2.3 Manifestations of software failures	20
2.2.4 Consequences of software failures	21
2.3 Overview of recent major software failures	22
2.3.1 Lists of cases of major software failure available online	22
2.3.2 Overview of prominent cases of recent software failure	26
2.3.3 Software failures according to industry or sector	31
2.3.4 Lessons learnt	34
2.4 Case study of software-induced radiation overdoses: AECL Therac-25, Multidata RTP/2 and Varian IMRT	35
2.4.1 Accident description	39
2.4.2 How was the overdose detected?	40
2.4.3 How was the root cause identified?	41
2.4.4 Factors that facilitated the overdose	42
2.4.5 Factors that contributed to the negative impact of the accidents	43
2.4.6 Lessons learnt	43

2.5 Requirements for accurate failure investigation.....	43
2.5.1 Limitations in the investigation of software failures	44
2.5.2 Requirements for accurate software failure investigation.....	45
2.6 Conclusion.....	47
CHAPTER 3 USING DIGITAL FORENSICS FOR ACCURATE INVESTIGATION OF SOFTWARE FAILURES	49
3.1 Introduction	49
3.2 Overview of digital forensics	50
3.2.1 Introduction to digital forensics	50
3.2.2 Digital forensic applications	51
3.3 Motivation for using digital forensics for software failure investigations	52
3.3.1 Supporting literature	52
3.3.2 Definition of digital forensics	54
3.3.3 Lessons learnt from other forensic disciplines.....	57
3.4 The scientific foundation of digital forensics.....	60
3.4.1 The scientific method.....	60
3.4.2 Mathematical analysis.....	63
3.5 Best practices in digital forensics.....	66
3.5.1 Overview of best practices in digital forensics	66
3.5.2 How can digital forensic best practices help improve the accuracy of software failure investigations?	67
3.6 The digital forensic process	68
3.6.1 Overview of the digital forensic process.....	69
3.6.2 Standardising of the forensic investigation process	70
3.6.3 How can the digital forensic process help improve the accuracy of software failure investigations?.....	71
3.6.4 Suitability of digital forensics for accurate failure investigations	72
3.7 Conclusion.....	74
CHAPTER 4 THE ADAPTED DIGITAL FORENSIC PROCESS FOR FAILURE INVESTIGATIONS	75
4.1 Introduction	75
4.2 Challenges to the forensic investigation of software failures	76
4.2.1 The volatility of digital evidence	77

4.2.2	The lack of forensic tools and techniques for the root-cause analysis of software failures.....	77
4.2.3	The need to minimise downtime following a failure	78
4.2.4	The need for continuous system monitoring.....	78
4.3	Previous work on the forensic investigation of software failures	79
4.3.1	Previous work on operational forensics	80
4.3.2	Review of previous work on forensic software engineering.....	82
4.3.3	Critical assessment of previous work on the forensic investigation of software failures.....	83
4.4	The forensic failure investigation process.....	84
4.4.1	Phase 1: Evidence collection.....	84
4.4.2	Phase 2: System restoration	85
4.4.3	Phase 3: Root-cause analysis	85
4.4.4	Phase 4: Countermeasures specifications	86
4.5	Application of the forensic failure investigation process – Case study of Therac-25 accidents.....	87
4.5.1	Investigation of first Therac-25 accident	88
4.5.2	Investigation of second Therac-25 accident.....	88
4.6	Critical assessment of the failure investigation process.....	91
4.6.1	Advantages of the forensic failure investigation process.....	91
4.6.2	Limitations of the failure investigation process.....	92
4.7	Conclusion.....	92
CHAPTER 5 NEAR-MISS ANALYSIS: AN OVERVIEW.....		94
5.1	Introduction	94
5.2	Background on near-miss analysis.....	95
5.2.1	Overview of near-miss analysis	95
5.2.2	Tools and techniques used in near-miss analysis.....	96
5.2.3	History of near-miss analysis	102
5.3	Motivation for using near-miss analysis in failure investigation	104
5.3.1	Benefits of near miss-analysis over failure analysis	104
5.3.2	Benefits of analysing near misses instead of earlier precursors.....	105
5.3.3	Near-miss analysis success stories	106
5.4	Challenges to near-miss analysis in the software industry.....	106



5.4.1	Detection of near misses	107
5.4.2	High volume of near misses	107
5.4.3	Root-cause analysis of near misses	108
5.5	Conclusion.....	108
CHAPTER 6 THE NEAR-MISS DETECTION AND PRIORITISATION MODEL ...		110
6.1	Introduction	110
6.2	Formal definition of a Near Miss for software systems	111
6.3	Overview of reliability theory and failure probability formula for IT systems .	116
6.3.1	The reliability theory of redundant hardware components	116
6.3.2	Failure probability formula for hardware components	117
6.3.3	Proposed failure probability formula for software components	118
6.4	Mathematical modelling for near-miss failure probability	119
6.4.1	Loss of one spare.....	120
6.4.2	Loss of two spares.....	120
6.4.3	Loss of any number of spares	121
6.4.4	Illustration of the failure probability formula	121
6.5	Prioritisation of near misses	123
6.5.1	The near-miss prioritisation formula.....	123
6.5.2	Evidence collection for high-risk near misses	124
6.6	Conclusion.....	125
CHAPTER 7 THE NMS ARCHITECTURE		126
7.1	Introduction	126
7.2	Requirements and proposed solutions for the accurate investigation of software failures	127
7.2.1	Requirements	127
7.2.2	Proposed solutions	127
7.3	The NMS architecture	128
7.3.1	The overall near-miss and failure investigation process	128
7.3.2	The NMS architecture	129
7.4	Conclusion.....	135
CHAPTER 8 PROTOTYPING THE NMS – THE DESIGN PHASE.....		136
8.1	Introduction	136
8.2	The aims of the prototype	136

8.2.1 The original NMS architecture	137
8.2.2 Prototype goal and objectives	137
8.3 Setting up the lab environment	140
8.3.1 The logs of a software failure	140
8.3.2 The forensic investigation tool and techniques.....	144
8.3.3 The test plan	147
8.3.4 Conclusion	149
CHAPTER 9 PROTOTYPING THE NMS – THE DATA SET	150
9.1 Introduction	150
9.2 Technical platform used for developing the prototype	150
9.2.1 The prototype implementation plan	150
9.2.2 Technical set-up for prototype implementation.....	153
9.3 Experiment 1: Creating a suitable set of event logs	154
9.3.1 The crash file.....	155
9.3.2 The <code>iostat</code> output file.....	159
9.3.3 Summary of experiment and results.....	161
9.4 Conclusion.....	162
CHAPTER 10 PROTOTYPING THE NMS –DETECTING NEAR MISSES AT RUNTIME	
.....	164
10.1 Introduction	164
10.2 Prototype implementation plan	164
10.3 Experiment 2 – Part 1: Identifying near-miss indicators from the forensic analysis of the crash file	167
10.3.1 Goal.....	167
10.3.2 SOM analysis of the crash file	169
10.3.3 Statistical analysis of <i>Latency</i>	176
10.3.4 Conclusion based on forensic analysis of crash file	177
10.4 Experiment 2 – Part 2: Identifying near-miss indicators from the forensic analysis of <code>iostat</code> output file.....	177
10.4.1 SOM analysis of <code>iostat</code> output file	177
10.4.2 Statistical analysis of <code>iostat</code> output	181
10.4.3 Conclusion based on analysis of <code>iostat</code> output file.....	183
10.5 Summary of Experiment 2 – Overall near-miss indicators.....	183

10.6 Experiment 3: Defining a near-miss formula.....	184
10.6.1 Goal.....	184
10.6.2 Adapting C++ program to calculate near-miss indicators	185
10.6.3 Changing running conditions of C++ program.....	187
10.7 Step 4: Detecting near misses at runtime	191
10.7.1 Goal.....	191
10.7.2 Technical set-up	192
10.7.3 Results.....	192
10.8 Evaluation of prototype implementation.....	193
10.8.1 Benefits	193
10.8.2 Limitations	196
10.9 Conclusion.....	196
CHAPTER 11 CONCLUSION	198
11.1 Introduction	198
11.2 Revisiting the problem statement.....	198
11.2.1 Answering the main and secondary research questions	199
11.2.2 Achieving the goal of the research.....	202
11.3 Main contributions	203
11.3.1 Advancing the state of the art	203
11.3.2 Publications produced	204
11.4 Future research	206
APPENDIX 1 TECHNICAL DETAILS ON THE SOM ANALYSIS	208
SOM map creation process	208
How to read Viscovery SOMine output maps: Example of first 1000 records	208
Adding the number of running processes to the C++ program.....	210
APPENDIX 2 GLOSSARY OF TERMS	211
BIBLIOGRAPHY	217

List of figures

Figure 1.1: The proposed NMS in relation to the fields of failure analysis, digital forensics and near-miss analysis	5
Figure 1.2: Relation between a near miss, its preceding ASPs and the subsequent accident in the accident sequence.....	11
Figure 1.3: Relation between near-miss and failure in terms of risk level of event and loss incurred	15
Figure 1.4: Graphical depiction of layout of thesis.....	17
Figure 3.1: The various classes of digital investigation processes	70
Figure 4.1: The digital forensic process.....	76
Figure 4.2: Relationship between prosecutorial forensics and operational forensics	80
Figure 4.3: The adapted digital forensic process for software failures.....	86
Figure 5.1: The Safety Pyramid, adapted from Bird and Germain (1996)	96
Figure 5.2: Near-miss management process (Phimister et al., 2000)	97
Figure 5.3: Bird’s accident ratio triangle, adapted from Nichol (2012)	107
Figure 6.1: Classification of unsafe events based on their downtime duration	114
Figure 6.2: Failure probability graph	122
Figure 7.1: UML component diagram of NMS architecture	131
Figure 7.2: UML activity diagram of NMS	134
Figure 8.1: Adapted NMS component diagram for prototype implementation.....	139
Figure 8.2: Flowchart of failure simulation program to create the crash file	142
Figure 8.3: Prototype implementation plan	148
Figure 9.1: Prototype implementation plan	152
Figure 9.2: VirtualBox user interface and shared directory between Windows and Linux...	153
Figure 9.3: Diagram of the lab environment.....	154
Figure 9.4: Focus of experiment 1 – Near-Miss Monitor of adapted NMS architecture.....	155
Figure 9.5: Output of the free command in Linux	157
Figure 9.6: Code snippet for function to obtain the amount of free space on the flash disk .	157
Figure 9.7: Crash file at beginning of program.....	158
Figure 9.8: Crash file – point of failure	158
Figure 9.9: Crash file at record 31 042	159
Figure 9.10: iotop output file	160

Figure 9.11: iotop output file – point of failure of C++ program 161

Figure 10.1: Adapted NMS component diagram for prototype implementation..... 165

Figure 10.2: Prototype implementation plan 166

Figure 10.3: Focus of Experiment 2 – Event Investigation of adapted NMS architecture 167

Figure 10.4: Focus of Experiment 3 – Near-Miss Formula in adapted NMS architecture 184

Figure 10.5: Adapted crash file for near-miss detection..... 186

Figure 10.6: Crash file with bigger video and smaller flash disk 188

Figure 10.7: Filtering of *Processes* field to confirm fluctuation in the number of processes 188

Figure 10.8: iotop output file – records matching near-miss indicators in terms of *Disk read*
and *Disk write* 189

Figure 10.9: Records in crash file that match three near-miss indicators 189

Figure 10.10: FCM of factors in near-miss indicators 191

Figure 10.11: Near-miss formula 191

Figure 10.12: Focus of Experiment 4 - Near-Miss Classifier in adapted NMS architecture. 191

Figure 10.13: Program code for near-miss formula..... 192

Figure 10.14: First near-miss alerts in the crash file..... 193

Figure 10.15: Last near-miss alerts in the crash file 193

Figure 10.16: Proposed method for forensic analysis and identification of near-miss indicators
..... 195

Figure 12.1: SOM output maps for first 1000 records..... 209

Figure 12.2: Some component maps of first 1000 records – first record appears as an outlier
..... 210

List of tables

Table 2.1: Lists of real-life cases of major software failure collected by the researcher online over a period of time	24
Table 2.2: Prominent cases of recent software failures collected by the researcher.....	27
Table 2.3: Researcher’s summary of 3 cases of radiation overdose due to software errors ...	37
Table 3.1: Differences between digital forensics and troubleshooting	72
Table 3.2: Suitability of digital forensics for the requirements of an accurate software failure investigation.....	73
Table 4.1: Challenges to the forensic investigation of software failures	78
Table 6.1: Failure probability values	122
Table 10.1: SOM maps of selected program attributes over time	171
Table 10.2: SOM maps of <i>Latency</i> for various data sets close to the point of failure	175
Table 10.3: WMA of <i>Latency</i> across various data sets.....	176
Table 10.4: SOM maps of <i>iotop</i> output for various data sets.....	178
Table 10.5: SOM maps of running processes, <i>Disk read</i> , and <i>Disk write</i> close to the point of failure	180
Table 10.6: List of running I/O processes throughout the program’s execution before the point of failure.....	181
Table 10.7: WMA of <i>Disk write</i> across various data sets.....	182
Table 10.8: WMA of <i>Disk read</i> across various data sets.....	183

CHAPTER 1

INTRODUCTION

1.1 Introduction

IT systems are ubiquitous in today's interconnected society and play a vital role in a number of industries such as banking, telecommunications and aviation. Software, in particular, is embedded in most technical and electronic products, ranging from massive machines such as airplanes to lightweight devices such as mobile phones. Software applications are essential to the proper functioning of these products and their associated service offerings. Due to the reliance of modern living on these products and services, software failures that result in their unavailability or malfunctioning can cause disasters and may even be fatal. Unfortunately, such software failures have occurred since the beginning of the computer age, as is evidenced by the number of highly publicised IT accidents reported in the media.

One example of a crisis caused by a software failure is the system outage that occurred at the Royal Bank of Scotland (RBS), a major bank in the UK, in December 2013. Due to an unspecified technical glitch, the bank's various electronic channels were unavailable for a day and customers were unable to make payments or withdraw cash with their debit cards (Finnegan, 2013). This failure was not the first experienced by RBS. In June 2012, another major outage occurred and left millions of customers unable to access their bank accounts for four days, due to a failure in a piece of batch-scheduling software. As a result, deposits were not reflected in bank accounts, payrolls were delayed, credit ratings were downgraded and utility bills were not paid (Worstell, 2012). Recently, in November 2014, RBS was fined 56 million pounds by British regulators for the software failure that occurred in 2012 (BBC News, 2014).

Preventing the recurrence of catastrophes such as the examples quoted above is crucial and requires a thorough post-mortem investigation to determine and rectify the root cause. To ensure the validity of its results, such an investigation must be based on reliable digital evidence such as log files, database dumps and reports from system-monitoring tools. Sound evidence

of the software failure promotes the objectivity and comprehensiveness of the investigation, which implies greater accuracy of the results. Furthermore, reliable evidence is valuable in the event that the software failure leads to a product liability lawsuit.

However, current informal approaches to failure analysis do not promote the collection and preservation of digital evidence. Rather than depending on objective evidence analysis, failure analysis methods often rely on the investigator's experience with the system to identify the cause of the problem. Troubleshooting in particular, which is usually the first response to a system failure, focuses on restoring the system to its operational state as quickly as possible. This allows little time and resources to collect evidence of the failure. Besides, system restoration often requires rebooting, which destroys or tampers with valuable information that could pinpoint the root cause of the problem (Trigg & Doulis, 2008). Both these 'solutions' leave the system vulnerable to the recurrence of a similar failure.

In order to ensure that the failure analysis is based on reliable evidence, the investigation must follow a process that favours the collection and analysis of such evidence. The investigation must also follow a standard process that can be reproduced by independent investigators to ensure the objectivity and reliability of the results. The literature indicates that the scientific method is well suited for this purpose as it requires evidence to confirm a hypothesis made about the root cause of an investigated event (Young, 2007). It also requires independent verification to confirm the results of the investigation (Young, 2007). Indeed, the scientific method is a standard procedure used by scientists to investigate a problem, with the aim to reduce potential bias from the investigator and ensure repeatability of the results (Bernstein, 2009).

Using an investigation approach that applies the scientific method therefore seems a logical step to improve the accuracy of current approaches to failure analysis. A brief literature investigation points to the forensic approach, as forensic science applies the scientific method to reconstruct past events based on objective evidence (Vacca & Rudolph, 2010). The field of digital forensics, as the application of forensic science to digital systems, certainly appears to be a promising solution.

Digital forensics follows established procedures meticulously to ensure the accuracy and completeness of digital evidence and to interpret it objectively based on scientific analysis (Vacca & Rudolph, 2010). Despite the fact that it is currently limited to criminal and security events, digital forensics can very well provide an effective alternative for investigating major software failures. However, as it adopts a reactive approach, digital forensics can only be applied after the occurrence of a failure. This limits the effectiveness of the investigation, since volatile data that could serve as potential evidence may be lost or corrupted after a system crash.

In order to address this limitation of digital forensics, it is suggested that the forensic investigation be started at an earlier stage, *before* the software failure actually unfolds, so as to detect the high-risk conditions that can lead to a major failure. These forerunners to failures are known in the risk analysis literature as near misses (Jones, Kirchsteiger & Bjerke, 1999). By definition, a near miss is a hazardous situation where the sequence of events could have caused an accident had it not been interrupted (Jones et al., 1999). This interruption can be caused by chance or by human intervention. A simple example of a near miss in everyday life is the case of a driver crossing a red traffic light at a busy intersection at high speed without causing a collision.

As a near miss is very close to an entire accident sequence, it provides a fairly complete set of data about the potentially ensuing accident. Such data can be used as evidence to reconstruct the impending accident and to identify its root cause. In the case of software applications, the term ‘accident’ refers to a major failure. By alerting system users of an upcoming failure, the detection of near misses provides an opportunity to collect at runtime failure-related data that is complete and relevant. It eliminates the need to log all system activity, which can result in vast amounts of data presenting challenges for storage and analysis. Since current digital forensic tools are limited in their ability to handle and interpret large volumes of data (Nassif & Hruschka, 2011; Guarino, 2013) the detection and analysis of near misses can serve as an effective data reduction mechanism.

Near-miss analysis is usually performed through so-called near-miss management systems (NMS), which are software tools used to report, analyse and track near misses (Oktem, 2002). In many industries that are prone to high-risk accidents, these systems have been implemented

for decades with a view to improving safety. Examples of such industries include the aviation, the nuclear, the chemical, and the healthcare industries (Phimister, Vicki, Bier & Kunreuther, 2004). NMSs have a successful track record in organisations where they have been designed effectively, as they provide valuable additional information on accidents and their root cause. For instance, evidence shows that the use of such systems has contributed to the improvement of safety in the aviation industry (Phimister et al., 2004). It is therefore expected that an NMS can also be a valuable learning tool with regard to software failures.

An NMS that combines near-miss analysis (obtaining appropriate digital evidence of a failure) and digital forensics (performing an objective analysis of the evidence) can contribute significantly to the improvement of the accuracy of the failure analysis. However, such a system is not available yet and its design still presents several challenges due to the fact that neither digital forensics nor near-miss analysis is currently used to investigate software failures. Its existing processes and methodologies are not directly applicable to the specificity of the software industry (for near-miss analysis) or of software failures (for digital forensics).

For example, digital forensics, which is used to identify and prosecute the perpetrator of a computer crime, does not provide for quickly restoring a failed system to minimise downtime before starting the investigation. Regarding near-miss analysis – in many industries, near misses are obtained from observed physical events and conditions. However, in the software industry such an exercise is particularly challenging – in the case of software applications, some near misses might not be visible at all, as no system failure actually occurred.

The aim of this research is to design an NMS that can address the above challenges to facilitate the forensic investigation of software failures. As illustrated in Figure 1.1, the proposed NMS lies at the intersection of failure analysis, digital forensics and near-miss analysis. It will focus on the detection and prioritisation of near misses at runtime with a view to maximising the collection of appropriate digital evidence of the failure.

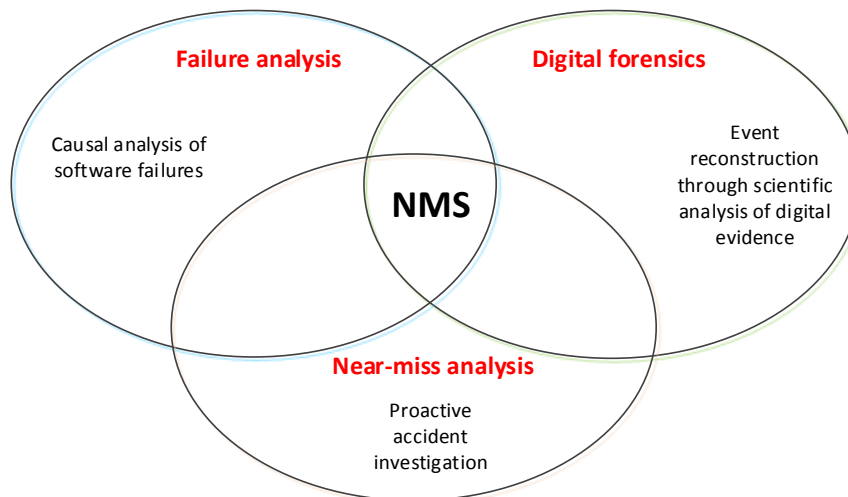


Figure 1.1: The proposed NMS in relation to the fields of failure analysis, digital forensics and near-miss analysis

1.2 Thesis statement

Against the background of the above discussion, the main claim of this research can be formulated as follows: *Near-miss analysis can help identify and collect more relevant and complete digital evidence of a software failure. This has the potential to improve the accuracy of the ensuing forensic analysis of the failure.*

Proving the above claim through the design of an appropriate NMS is the goal of this research.

1.3 Problem statement

The problem addressed by the current research can be formulated as follows: *Current failure analysis methods are informal and lack accuracy and objectivity, which can lead to the recurrence of disastrous software failures. The combination of the scientific approach of digital forensics and the sound evidence of the failure obtained through near-miss analysis has the potential to address this issue but is not available yet. The design of an NMS to fill this gap faces a number of challenges due to the specificities of software failure investigations that are not catered for by existing processes and methodologies of either digital forensics or near-miss analysis.*

1.4 Research Questions

This research will address the above problem by answering the following main research question: *What should the architecture for a near-miss management system look like such that*

it can improve the completeness and relevance of digital evidence of a software failure, thereby improving the accuracy of its forensic analysis?

In order to design such an architecture, research will need to be conducted to answer the following fundamental questions that arise as sub-problems:

- *How can the methodology of digital forensics be applied to the investigation of software failures?*

This requires reviewing techniques and procedures used in digital forensics and identifying the ones suitable for investigating software failures. It also entails adapting relevant but currently unsuitable procedures in order for them to satisfy the specificity of software failure investigations.

- *How can near-miss analysis be applied to the software industry effectively?*

This entails reviewing challenges to near-miss analysis across industries and identifying issues specific to the software industry. Assessing challenges to near-miss analysis requires a solid understanding of this discipline and its state-of-the-art. The literature on the topic indicates that, across industries, the two main challenges to near-miss analysis are the detection and prioritisation of near misses. These two challenges must therefore be reviewed from a software application perspective.

Addressing these challenges effectively entails reviewing previous work in near-miss analysis and identifying solutions, if any, applicable to the software industry. If necessary, these solutions can be modified according to the specific requirements of the software industry. Otherwise, new and suitable solutions must be provided.

1.5 Scope and context of the study

This research is limited to *operational failures*, in other words software failures that occur after the design, development and testing phases when a system is in production. Contrary to a pre-production system, an operational system is a finished product, which has been tested and is expected to work as intended. The margin for failures is therefore low due to the potentially severe impact of such occurrences..

Operational failures are the focus of this thesis as the reliability and performance level of a system can only be truly assessed when in production. Contrary to software failures experienced during software development and testing, operational failures occur in real-life affecting “real” users. They are harder to contain and predict than failures and defects that occur during system development, where the system is operating in a controlled and safe environment and potential recurrences of the failure are not detrimental to the intended end users. The cost of operational failures is therefore significantly higher compared to pre-production failures. The associated cost of a forensic investigation is therefore more valuable for an operational system.

1.6 Research methodology

The following steps will be taken to solve the problem stated in Section 1.3.

The *first step* is to conduct a literature study on failure analysis to understand current practices in this field. This is achieved through reviewing the literature on major software failures that occurred within the last five years. The cause and impact of the failures are first reviewed to assess the significance of the problem. Then follows an extensive study of the literature available on the investigation of such failures.

The *second step* is to critically assess the effectiveness of the documented failure investigation process and identify its limitations in terms of the accuracy and objectivity of the results. Requirements for improvement are subsequently formulated, which constitute the foundation for the design of the NMS proposed in this research.

The *third step* is to design a solution for the above requirements, in the form of a suitable NMS architecture. The first aspect of the architecture is the investigation process, which is based on digital forensics and therefore requires a review of the digital forensics process from a software failure perspective. The review aims to determine how digital forensic methodology can be applied to software failure investigations and what changes need to be made. An adapted digital forensic investigation process that meets the specified requirements is then designed and tested against the case study of a real-life software failure. The second aspect of the NMS architecture involves the near-miss analysis and therefore requires its review and critical examination from a software failure perspective. Challenges to near-miss analysis for investigating software

failures are examined. Solutions to the challenges are then proposed, in the form of a definition of near misses for software systems and a mathematical model to enable the detection of near misses based on this definition.

Afterwards an NMS architecture that integrates all the above partial solutions is designed. The NMS includes all the necessary phases of a digital forensic investigation, as well as the created mathematical model to detect near misses and enable the collection of evidence of the failure before the root-cause analysis is conducted.

The *fourth and final step* is to test the viability of the NMS architecture through the implementation of a prototype. Several experiments are conducted to demonstrate the detection of near misses and the forensic investigation of a failure based on the evidence collected from the near-miss detection.

1.7 Terminology used in the thesis

In order to avoid any misunderstanding, it is important to correctly interpret the terminology used in this thesis. Therefore, the researcher provides a brief definition of what is meant by the relevant terms used around the concepts of an accident, a failure and a near miss.

Event: a real-time factual occurrence that could seriously impact an operation (Jucan, 2005)

Condition: Any system state, whether precursor or resulting from an event, that may have adverse implications for the normal system's functionality (Jucan, 2005).

Accident: An undesirable event resulting in injury or damage (Jones et al., 1999).

Accident sequence: Sequence of events that result in an accident. The accident sequence starts with an initiating event such as a human error, and ends when the accident unfolds, also known as the accident end-state (Saleh et al., 2013).

Incident: Any undesirable event, including accidents and near misses (Jones et al., 1999).

Cause: A condition or an event that results in or participates in the occurrence of an effect.

Causes can be classified as:

- *Direct Cause*: A cause that resulted in the occurrence.
- *Contributing Cause* (also *Contributing Factor*): A cause that contributed to an occurrence but would not have caused it by itself.
- *Root Cause*: The cause that, if corrected, would prevent recurrence of this and similar occurrences (Jucan, 2005).

Failure: the inability of a system or component to perform its required functions within specified performance requirements (IEEE, 1999). This definition applies to both hardware and software system failures.

Digital forensics: the use of scientifically derived and proven methods towards the preservation, collection, validation, identification, analysis, interpretation and presentation of digital evidence derived from digital sources for the purposes of facilitating or furthering the reconstruction of events found to be criminal, or for anticipating the unauthorised actions shown to be disruptive to planned operations (Palmer, 2001).

Near miss: a hazardous situation, event or unsafe act where the sequence of events could have caused an accident if it had not been interrupted (Jones et al., 1999).

Near-miss management system: software system used to report, analyse and track near misses (Oktem, 2002).

Since the concept of a near miss is pivotal to this study, it requires a more specific definition as the generic one presented above. As it is not formally used in the software industry and has not yet been applied to digital forensics, there is no literature available on near misses with regard to software systems. The next section therefore provides an explanation of this concept as used in other industries and then formulates a definition of a near miss relevant for the purpose of this research.

1.8 Defining the near-miss concept

1.8.1 Current definitions of near miss

The concept of a near miss is primarily used in the domain of risk analysis and safety with regard to accident investigation and prevention (Saleh, Saltmarsh, Favarò, & Brevault, 2013). In the same way that what constitutes an accident differs from one industry to the next, what is considered a near miss also varies between industries. Even within an industry, a near miss may be defined differently from one organisation or field of practice to the next.

In order to fully comprehend the concept, this section presents an explanation of a near miss from three perspectives. Firstly, a general discussion of a near miss in everyday life is provided. Secondly, near misses are defined more formally from a risk analysis and safety perspective. Finally, industry-specific definitions of near misses are provided to illustrate how the formal concept is applied in practice.

1.8.1.1 General definition of a near miss

The expression ‘near miss’ can be interpreted incorrectly as ‘almost missing a set target’. The expression is better understood through its synonyms which are a ‘near accident’, ‘close call’, and ‘near hit’. A general and broad explanation of the expression ‘near miss’ is provided as follows:

A near miss is an unplanned event that did not result in injury, illness, or damage – but had the potential to do so. Only a fortunate break in the chain of events prevented an injury, fatality or damage; in other words, a *miss* that was nonetheless very *near* (MIC, 2014).

According to the American Heritage Dictionary of Idioms (Ammer, 2013), the expression ‘near miss’ originated during World War II, to refer to a bomb exploding in the water close enough to a ship to damage its hull. Soon afterward it acquired its present meanings.

1.8.1.2 Definition of near miss from a risk analysis and safety perspective

In the risk analysis and safety literature, a near miss is defined based on its relation to an accident or an accident sequence.

The concept near miss is used to identify high-risk conditions that can lead up to an accident. Such forerunners to failures are formally known as accident sequence precursors (ASP) or more simply as accident precursors (Saleh et al., 2013). ASPs are defined as “conditions, events and sequences that precede and lead up to accidents” (Phimister et al., 2004). They are also defined as “events that must occur for an accident to happen in a given scenario” (Carroll, 2004).

A near miss is a special type of ASP. It is a precursor whose elements differ only slightly from the potential accident or whose mitigating factors are unlikely or not robust (Phimister et al., 2004). In contrast to other ‘early’ precursors in the accident sequence, a near miss is the closest to the accident end-state (Saleh et al., 2013). It is very similar to the complete accident sequence, with only a few elements missing, either by chance or due to some human intervention. In simpler terms, a near miss is one step away from the accident. The researcher illustrates in Figure 1.2 this relation between a near miss, the preceding ASPs and the associated accident. In Figure 1.2, the accident is preceded by four ASPs. The fourth and last ASP is a near-miss event in case the accident does not unfold.

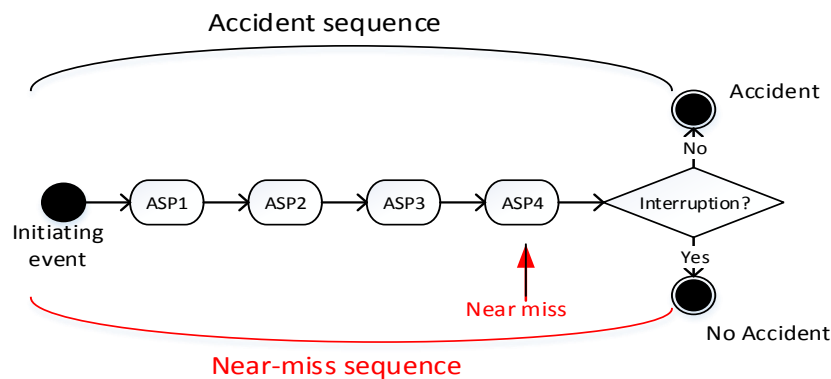


Figure 1.2: Relation between a near miss, its preceding ASPs and the subsequent accident in the accident sequence

Based on the above discussion, a near miss can be defined generally as “a hazardous condition where the accident sequence was interrupted” (Andriulo & Gnoni, 2014). This definition is discussed as follows based on the near-miss example mentioned in Chapter 1.

A simple fictitious example of a near miss in everyday life is a driver crossing a red traffic light at a busy intersection at high speed without causing a collision. In this example, there are two risky situations or ASPs: (1) crossing a red light, and (2) driving at high speed. Crossing a busy

intersection is a contributing factor to the likelihood of an accident but not an ASP, as it cannot by itself lead to an accident. The combination of the two ASPs and the contributing factor makes this event a high-risk situation conducive to an accident. The fact that no collision occurred makes this event a near miss. Various scenarios are possible to explain the lack of a collision:

- The driver's driving ability to avoid incoming cars
- The carefulness of the drivers of the incoming cars
- The speeding driver and the driver of an incoming car managed to stop right before crashing into each other
- Luck

The concept of near miss can be applied to almost any process in any industry. It is currently used in a number of industries including the chemical, aviation, nuclear, military and healthcare industries to learn about accident causes and prevent their occurrence and/or recurrence (Phimister, Oktem, Kleindorfer, & Kunreuther, 2003). Some interest has also been shown in its application in the construction industry (Wu, Yang, Chew, Yang, Gibb, & Li, 2010), oil and gas industry (Cooke, Ross, & Stern, 2011), financial industry (Mürmann & Oktem, 2002; Oktem, Wong, & Oktem, 2010), manufacturing industry (Gnoni, Andriulo, Nardone & Maggio, 2013) and outdoor activity sector (Goode, Salmon, Lenne & Finch, 2014). Various examples of near misses in some of these industries are provided in Kleindorfer, Oktem, Pariyani and Seider (2012). Examples of how near misses are defined in some of these industries are provided next in an attempt to determine how suitable these definitions are for the software industry.

1.8.1.3 Industry-specific definitions of near misses

This section presents definitions of near misses from three different industries: the aviation industry, which was the pioneer in the formal investigation of near misses (NASA, 2006); the medical field, which has a considerable amount of literature on the topic; and the occupational health and safety sector, which touches a number of industries. Examples are provided to illustrate the definitions, after which a critical assessment of the suitability of these definitions with regard to software systems is conducted.

In the aviation industry, more specifically in air-traffic control, a near miss is defined as “any circumstance in flight where the degree of separation between two aircraft is considered by either pilot to have constituted a hazardous situation involving potential risk of collision” (US Department of Defense, 2005). This definition limits near misses to the specific scenario whereby two aircraft passed one another too closely without causing a collision.

In the medical field, a near miss is defined as “an event that could have resulted in unwanted consequences, but did not because either by chance or through timely intervention the event did not reach the patient” (ISMP-Canada, 2014). For example, a hospital doctor mistakenly prescribes penicillin to a patient who is allergic to the drug. The error goes unnoticed by both the pharmacist and the nurse, but the patient mentions his allergic condition just before swallowing the tablets and the nurse stops him just in time (Nashef, 2003).

In the occupational health and safety domain, near misses are defined as “incidents where no property was damaged and no personal injury sustained, but where, given a slight shift in time or position, damage and/or injury easily could have occurred” (US Department of Labour, 2010). For example, a construction worker is walking on a designated path on a construction site and a wrench falls from scaffolding above, nearly hitting him (Pettinger, 2013).

Although near-miss definitions were provided for only three industries, they illustrate various points about an industry’s view on the near-miss concept.

Firstly, the above definitions all define near misses as observed physical events. Two factors are considered:

- The actual impact of the event (no collision in air-traffic control, patient not affected in medical field, or no injury or property damage in occupational health and safety)
- The potential impact of the event if the accident sequence had not been interrupted (aircraft collision, affected patient, property damage or injury)

Secondly, the definitions are narrow in the sense that they point to a specific type of event or situation, sometimes only applicable in the industry at hand (e.g. aircraft collision only applicable to aviation, affected patient only applicable to medicine). Therefore none of these definitions can be applied to the software industry for the following reasons:

- In the case of software applications, no physical near-miss event might be observed since no ‘accident’ occurred (the term ‘accident’ referring to a failure).
- The exact impact of a software ‘accident’ is not known beforehand. As illustrated earlier with the example of the RBS software failure, the consequences of software failures vary, based on the system involved. Since consequences can range from financial loss to loss of life, they cannot all be grouped under the expression ‘injury or property damage’ as is the case in occupational safety.

A broader and more relevant definition of a near miss with regard to software systems is therefore required. This is formulated in the next section.

1.8.2 Proposed definition of a near miss for software systems

The starting point for defining near misses in software systems is the following definition from Jones et al. (1999): a near miss is “a hazardous situation, event or unsafe act where the sequence of events could have caused an accident if it had not been interrupted”.

In terms of software systems, the term ‘accident’ may be substituted by the term ‘major failure’. Indeed, the software literature does not refer to adverse events as accidents but as failures. According to the definition provided earlier, accidents result in significant loss. Since a software failure may or may not result in loss, it can be argued that significant loss is incurred only when the failure is severe. Therefore the following definition roughly based on that of Jones et al. (1999) above is proposed for a near miss with regard to software systems:

A near miss is an unplanned high-risk event or system condition that could have caused a major software failure if it had not been interrupted either by chance or timely intervention.

The researcher illustrates the relation between a near miss and the associated software failure in terms of risk and loss in Figure 1.3. The early precursors in the accident sequence are also indicated in Figure 1.3.

The above definition is the general definition for near misses used throughout the rest of this thesis. In Chapter 6, a formal definition for near misses is provided to enable their automated detection by the proposed NMS.

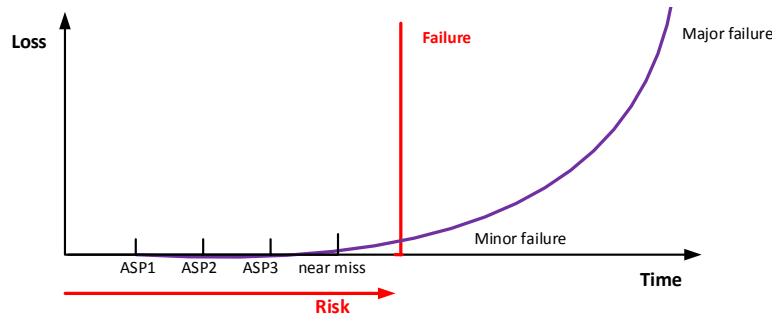


Figure 1.3: Relation between near-miss and failure in terms of risk level of event and loss incurred

Based on the above definition, examples of near misses in software systems can now be recognised. The researcher experienced a practical example of a near miss with a mobile procurement application developed in her research centre (Cashmore, 2012). Due to ineffective memory management caused by a programming error, one user unwittingly exceeded the specified limit of items in the product catalogue stored on his phone, and continuously added new items to the catalogue. Contrary to what would normally have been expected, this did not cause the application to crash. Investigations of this near miss revealed that the programming error overruled the code written to enforce the limit on the number of catalogue items.

1.9 Layout of thesis

This thesis consists of 11 chapters as depicted in Figure 1.4.

Chapter 1 provides an introduction to the thesis and indicates how the research is structured. It also provides a brief introduction to the concept of near miss, how it is defined across industries and its proposed definition for the software industry as used throughout the thesis. Since near-miss analysis is the central point of this research, a clear understanding of the concept of a near miss (new to digital forensics) is essential.

Chapter 2 is an overview of past software failures of significant magnitude, followed by a case study of their ineffective investigation. It provides a motivation for the significance of this study and a background for further explanations of the proposed NMS. Requirements to improve the investigation of software failures are provided.

Chapter 3 provides background information on digital forensics and motivates its potential application in the investigation of software failures, based on the requirements specified in Chapter 3.

Chapter 4 examines challenges to the application of digital forensics to software failure analysis and reviews previous work conducted on this topic. It then presents an adapted forensic investigation process suitable for software failures. This investigation process is based on solutions identified from the literature review of previous work and on the requirements specified in Chapter 2. The proposed process is validated with a case study of the real-life example of a major software failure. Limitations of the process are discussed, following the evaluation of the results of the case study.

Chapter 5 proposes near-miss analysis as a solution to the limitations of the forensic process described in the previous chapter. An overview of the field of near-miss analysis is provided. Challenges to near-miss analysis are then presented from a software perspective.

Chapter 6 presents proposed solutions to the challenges regarding near-miss analysis identified in the previous chapter. The solutions are presented as a mathematical model for defining, detecting and prioritising near misses.

Chapter 7 presents the NMS architecture. The architecture combines the mathematical model designed in Chapter 6 with the adapted forensic investigation process presented in Chapter 4.

Chapter 8 is the first chapter of a three-part series that describes the implementation of a prototype for the NMS architecture proposed in the preceding chapter. The prototype focuses on the detection of near misses. This chapter presents the design phase of the prototype implementation.

Chapter 9 describes the first experiment of the prototype implementation. The goal of this experiment is to obtain a data set suitable for the subsequent forensic analysis and near-miss detection.

Chapter 10 is the last of the three chapters describing the prototype implementation. It portrays the last experiments that were conducted to enable the detection of near misses at runtime.

Chapter 11 concludes the thesis.

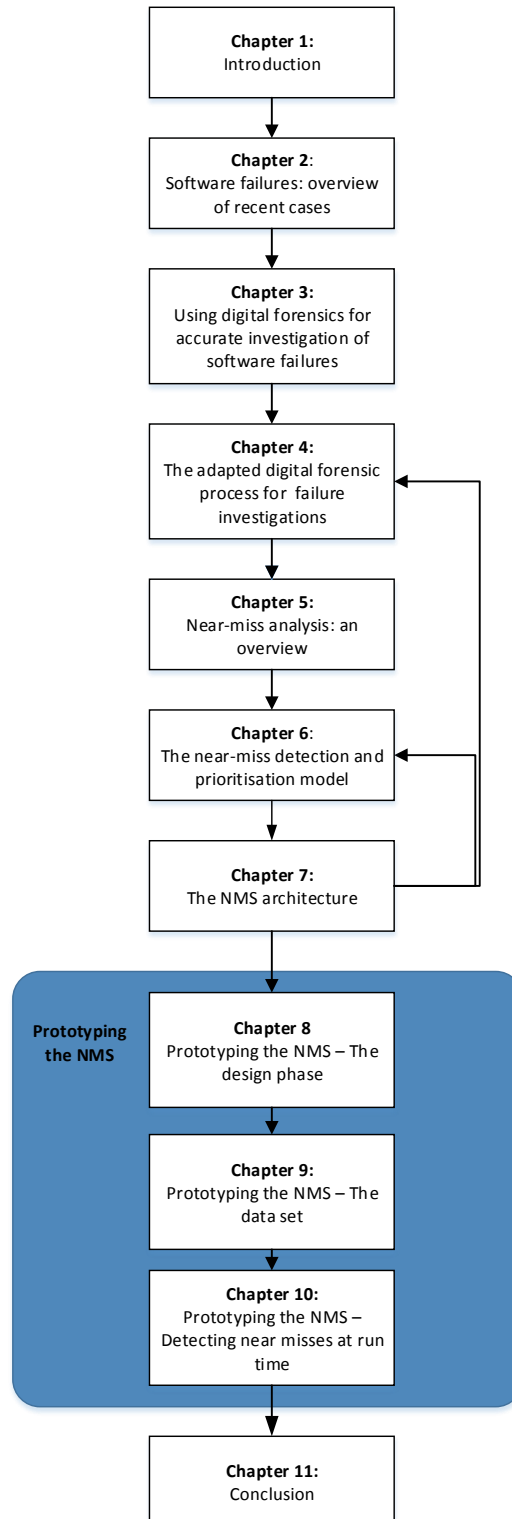


Figure 1.4: Graphical depiction of layout of thesis

CHAPTER 2

SOFTWARE FAILURES: OVERVIEW OF RECENT CASES

2.1 Introduction

Since major software failures often result in disasters ranging from financial loss to loss of lives, preventing their recurrence is absolutely necessary. As indicated in Chapter 1, a post-mortem investigation is required to identify their root cause and implement appropriate countermeasures. However, history shows that such an investigation is often conducted inefficiently and inaccurately, with no proper supporting evidence, which allows for the recurrence of severe accidents.

This chapter reviews the problem of major software failures with the aim of determining how to improve the accuracy of their root-cause analysis so that major accidents do not reoccur. This is achieved through the following two steps. Firstly, a review of recent cases of major software failures is conducted to demonstrate the reality and seriousness of this issue. Secondly, an analysis of the investigation of those failures is performed to identify limitations and establish requirements for improvement. These requirements form the basis for the design of the NMS proposed in this thesis.

In order to demonstrate the diversity of software failures, the presented cases of failures cover a number of different industries. Although their public reports are obtained mainly from the software literature, some were encountered in the medical literature. Indeed, reports on recent cases of software failures in the healthcare industry are scarce in the computer science field, especially in the software literature. Therefore, it was deemed necessary to also review relevant literature from the medical domain to broaden the pool of software failure cases.

Medical software failures are particularly relevant for the purposes of this research for two reasons: they can be fatal and legislation requires an in-depth investigation of these accidents.

Hence, comprehensive reports on their investigation are sometimes available, in contrast to the lack of proper reporting on software glitches with lesser consequences in other industries.

This chapter is structured as follows. Section 2.2 provides background information on software failures by presenting a definition of a software failure and explaining the main causes and consequences of these events. Section 2.3 reviews recent cases of severe software failure. Section 2.4 examines the typical approach for investigating software failures through a case study of three fatal medical software failures and the investigations conducted subsequently. Finally, Section 2.5 presents the lessons learnt from the literature review of these software failures and develops requirements for improving their investigation.

2.2 Background on software failures

This section presents some general background information on the problem of software failures. The section first establishes a definition of a software failure as is relevant for this research. This is followed by a review of the causes, manifestations and consequences of software failures.

2.2.1 Definition of a software failure

According to Laprie (1992) “a system failure occurs when the delivered service no longer complies with the specifications, the latter being an agreed description of the system's expected function and/or service”. This applies to both hardware and software failures.

Similarly, the IEEE computer dictionary defines a failure as “the inability of a system or component to perform its required functions within specified performance requirements” (IEEE, 1990). This definition also applies to both hardware and software systems.

More commonly, the universal dictionary of the English language defines a failure as “non performance of action which was necessary, expected” (Wyld, 1961).

All three definitions above present the term failure in relation to some predefined specification(s). These specifications set the expected level of performance of a system, in other words what is considered ‘normal’ functioning.

Therefore, for the purposes of the research in hand, the definition of a software failure is *an unplanned cessation of a software system or component to function as specified*.

Software systems can fail for various reasons as will be discussed next.

2.2.2 Common causes of software failures

Common causes of software failures include resource exhaustion (e.g. memory leaks), system overload (e.g. network congestion), logic errors (e.g. incorrect formula that leads to miscalculations), misconfigurations (e.g. inappropriate settings due to changes) and security breaches (e.g. malicious software such as viruses and worms) (Pertet & Narasimhan, 2005).

Besides the above causes, human errors such as entering incorrect data, and glitches in routine maintenance operations, for instance failed software upgrade, also account for a significant number of failures. Industry research (Vision Solutions, 2004) shows that the latter accounts for about 15% of all software failures. Environmental problems (e.g. a power failure) can also cause a software system to stop functioning properly.

Failures can originate from the software programs of the server, the client or the network system (Marcus & Stern, 2003). In each case, they can manifest in either of several ways presented below.

2.2.3 Manifestations of software failures

Generally speaking, software failures result in downtime and poor system performance. The standard definition of downtime is “the period of time during which a system or component is not operational or has been taken out of service” (IEEE, 1990). The term ‘outage’ is also often used as a synonym of downtime. It is worth noting that a failure results in *unplanned downtime*, in contrast to *planned downtime*, which is the result of scheduled maintenance operations such as backups and upgrades (Pertet & Narasimhan, 2005).

Downtime and poor system performance can manifest in any one of the following ways as indicated by Pertet and Narasimhan (2005):

- **Partial or entire system unavailability:** Either the entire system or parts of it are down. The user may receive error messages such as “file not found” or “system unavailable” or his request may time out.
- **System exceptions and access violations:** The executing process may terminate abruptly, causing the application to hang or crash, with or without a warning message.
- **Incorrect results:** The system is operational but delivers wrong results such as some miscalculations or incorrect items (e.g. the wrong dosage of a requested medication).
- **Data loss or corruption:** The user is unable to access previously available data (e.g. a previously saved file) due to some accidental erasure or corruption.
- **Performance slowdown:** The system is unusually slow to respond to a user query.

In many cases, awareness of a software failure only occurs when the failure becomes visible to the end-users in one of the ways mentioned above. In most cases the above signs of software failure result in mere inconvenience, but they can in some instances have tragic consequences and cause a negative impact on both the service provider and the end-users. Consequences of software failures are reviewed next.

2.2.4 Consequences of software failures

A major software failure affects both the service provider and the service consumer, and it may have a negative impact on both company revenue and end-user well-being.

The service provider mainly suffers from a substantial loss of revenue. This can be due to a loss of productivity, repair costs (repairing and replacing damaged equipment, hiring external consultants to resolve the problem), product liability litigations or compensation paid to frustrated customers (Ponemon Institute, 2011). In addition, the service provider also suffers from a tarnished reputation, which can cause a fall in its stock price, affect staff morale and in turn result in increased customer churn (Mappic, 2013). Study reports indicate that unplanned downtime costs businesses across North America and Europe a total of 26.5 billion USD in lost revenue each year (Harris, 2011) and an average of 5 600 USD per minute (Ponemon Institute, 2011).

Furthermore, a system crash can open the door for unauthorised access to confidential information by weakening security barriers such as firewalls and intrusion detection systems (Bihina Bella, Eloff & Olivier, 2012). This can lead to a breach of privacy of the customers' records or of the company's trade secrets.

The service consumer can suffer from frustration because the service is unavailable, slow or delivers incorrect results. As software is embedded in a range of devices in a number of industries, a failed software application can affect any area of a consumer's day-to-day life, including the financial, transport, telecommunications, legal and healthcare areas. Some examples of the consequences of failed software in the above industries include the following: overcharged bills, cancellation of scheduled flights, unavailability of mobile communication services, wrongful arrests, and wrong medical procedures. Based on how widely the software system is used, these consequences can span entire countries and even cross continents. This is illustrated in the next section where real-life examples are offered of major software failures in the industries mentioned above.

2.3 Overview of recent major software failures

This section reviews cases of major software failure that occurred within the last six years to illustrate the prevalence of such events and their catastrophic impact. The review starts with a list of online sources of software failure cases in Section 3.3.1 and then discusses in more detail interesting cases selected from these sources in Section 3.3.2.

2.3.1 Lists of cases of major software failure available online

For the sake of protecting their public image, affected companies often underreport catastrophes that result from software malfunctions (Sommer, 2010), even more so when human lives were at risk (Bogdanich, 2010). However, various lists of worldwide software failures are publicly available on the web in an attempt to shed light on the extent of the problem and help avoid the recurrence of similar problems. Section 2.3.1.1 presents lists collected from the computer science literature by the author of this thesis, while Section 2.3.1.2 reviews lists collected by the author from the medical field. As mentioned in the Introduction (2.1), medical accidents are of particular interest to this research as detailed reports on their investigations are usually available. This is in contrast to the other failures listed in this section, where only

limited information about their investigation and technical details of the failures are provided. The discussion of these cases of software failures is therefore on a high level for the purpose of illustrating the variety and severe impact of major software failures.

2.3.1.1 Lists from the computer science field

Table 2.1 gives a summary of ten lists of software failure cases maintained by IT professionals. These lists were used to find the real-life examples of software failures that are discussed later in this chapter. Most of the lists provide a brief summary of the reported failures and focus on their impact, with little or no details on what caused the problem or how it was resolved. Additional research work is therefore required to find this information. For this reason, Table 2.1 has seven columns that help assess how detailed and complete the list is, since detailed lists require less additional research work. The seven columns are as follows:

- **Name or source of list:** The title of the list, if available, or its website
- **Author:** Name and affiliation of the main author of the list
- **Date of earliest failure:** Year and month (if available) of earliest failure recorded in the list
- **Date of latest failure:** Year and month (if available) of latest failure recorded in the list
- **Failure count:** Number of entries in the list
- **Source of reference material provided?** The entry is either a yes or a no, depending on whether the list provides references to documentation about the failure
- **Technical details provided?** The entry is either a yes or a no, depending on whether the list provides details about the cause of the failure and how it was investigated

In addition, Table 2.1 has an extra field called “**Selected cases**” that lists the failures that were selected from each online source for further discussion. These examples were selected by the author according to the following criteria: their severity, diversity, learning opportunity in terms of root-cause identification and failure resolution, as well as how much additional technical information could be found. The latter was limited for many of the examples.

The author of this thesis classified the lists as either dynamic or static, based on when last they were updated. Dynamic lists are updated as new events occur, while static lists are no longer updated. Several static lists rank events based on their severity.

The lists in Table 2.1 are sorted in descending order based on the date of the latest failure in each list. Table 2.1 shows that major software failures have been a source of concern for decades in various parts of the world, with cases having occurred as early as 1962. The table also shows that information on how the failure was investigated and resolved is rarely available, which may suggest that the investigation process was not documented or was kept confidential. Another observation from the table is the fact that some failures are reported in several lists, confirming their significance. In particular, the Therac-25 disaster is reported in every list whose earliest failure occurred before 1985. This disaster is therefore worth some attention and is discussed in detail in Section 2.4.

Table 2.1: Lists of real-life cases of major software failure collected by the researcher online over a period of time

Name or source of list	Author	Date of earliest failure	Date of latest failure	Failure count	Source of reference material provided?	Technical details provided?	Selected cases
Dynamic lists							
The Risks Digest, online newsletter	Peter Neumann, Computer Science Lab, SRI International, USA	Aug. 1985	Nov. 2014	Events reported every week since 1986	Yes	Yes (details on cause but not on investigation into failure)	Rent troubles at New York City public housing agency in 2009 (Fernandez, 2009); Blackberry outage in 2011 (Whittaker, 2011).
SoftwareQATest.com	Rick Hower, software testing consultant	1983	Apr. 2014	80	No	No	RBS failure in 2012 (Finnegan, 2013); Axa Rosenberg trading error in 2011 (Greene, 2011).
“Collection of Software Bugs”	Thomas Huckle, Institut für Informatik in Munich, Germany	1962	Apr. 2014	Over 50 cases in different industries	Yes	No (details in reference material only)	Therac-25 disaster in 1985-1987 (Leveson & Turner, 1993).
SQS.com, software quality testing company	SQS consultants	2010	Dec. 2013	10 per year (most are high profile)	No	No	United Airlines failure in 2012 (Karp, 2012); UK organ donor register error in 2010 (Roberts, 2010)
Static lists							



Name or source of list	Author	Date of earliest failure	Date of latest failure	Failure count	Source of reference material provided?	Technical details provided?	Interesting cases
ACM blog	Bertrand Meyer, Chair of Software Engineering, ETH Zurich, Switzerland	1985 (reference to The Risks Digest)	2011	Not available, cases spread out on various articles	Yes	No (details in reference material)	French National pension fund error in 2009 (Durand-Parenti, 2009)
“10 Seriously Epic Computer Software Bugs”, from listverse.com, a website that provides top 10 lists on various topics	Andrew Jones, no affiliation provided	1982	2005	10	No	Yes (details on cause but not on investigation into failure)	Therac-25 disaster in 1985-1987 (Leveson & Turner, 1993)
“20 Famous Software Disasters”, from DecTopics.com, a website on software development topics	Unspecified	1962	2005	20	Yes (many links are outdated)	Yes (details on cause but not on investigation into failure)	Multidata radiation accidents in 2001 (McCormick, 2004).
Software Horror Stories	Nachum Dershowitz, School of Computer Science, Tel Aviv University, Israel	1968	2004	107	Yes	Yes (details on cause but not on investigation into failure)	Therac-25 disaster in 1985-1987 (Leveson & Turner, 1993)
“History’s Worst Software Bugs”, from Wired News.com, technology news website	Simon Garfinkel, no affiliation provided	1962	2000	10	Yes	Yes (details on cause but not on investigation into failure)	Multidata radiation accidents in 2001 (McCormick, 2004); Therac-25 disaster in mid-1980s (Leveson & Turner, 1993)
“10 historical software bugs with extreme consequences”, from Pingdom.com, a website-monitoring company	Unspecified	1980	2000	10	Yes	No	Therac-25 disaster in mid-1980s (Leveson & Turner, 1993); Multidata radiation accidents in 2001 (McCormick, 2004);

It is worth noting that although the above lists present software failures in a wide range of industries, they do not provide recent examples of software failures in the medical field. Recent examples of such failures are therefore presented in the next section.

2.3.1.2 Online sources of software failures from the medical field

As software is embedded in a number of medical devices, a good place to start looking for medical software failures is the portal of the FDA, the U.S. Food and Drug Administration, where adverse events due to faulty medical devices have been reported since 1991 (FDA, 2013). A prominent case from the FDA portal is a fatal oxygen system failure in Minnesota, USA, in 2010 (Charette, 2010).

Another valuable source of information on medical software failures is the website of the IAEA (International Atomic Energy Agency) (IAEA, 2013a), which provides detailed reports on past radiation therapy accidents and related course material with a view to preventing their recurrence. Radiation accidents due to software errors gained prominence with the Therac-25 disaster mentioned earlier (Leveson, 2000) while the IAEA presents more recent cases. The New York Times newspaper (Bogdanich, 2011) also published a series of comprehensive articles on the topic in 2010, such as the fatal case that occurred in a New York City hospital in 2005 (discussed in detail in Section 2.4).

The next section reviews the cases of software failure selected from the above sources.

2.3.2 Overview of prominent cases of recent software failure

History shows ample examples of the devastating effect of major software failures. Table 2.2 presents a summary made by the author of this thesis of ten of the cases listed earlier in Table 2.1 and the examples mentioned in Section 2.3.1.2. The table only lists failures that occurred from 2009 onwards. Selected failures that occurred earlier are discussed in Section 2.4 under radiation therapy accidents.

Table 2.2 has nine columns that capture the essence of each failure, and the last three indicate how its root cause was identified, for how long the bug was present before its discovery and how it was fixed. The researcher specifically introduced these three fields to highlight the

inefficiency of the investigation process and its adverse effect on the improvement of the system. Besides their severity and diversity, the examples were selected based on the availability of information on the approach that was used to solve the failure. The examples are grouped according to the industry or sector affected. Five industries and sectors are represented: finance, airlines, mobile telecommunications, public services (government and law enforcement) and healthcare. Software failures in these industries have a direct effect on the daily life of service consumers. A brief description of each of the nine columns in Table 2.2 follows next:

- **Industry/sector:** The industry or sector affected
- **Company/institution:** The company or institution that experienced the failure
- **Failure description:** A brief description of the failure
- **Date:** Year of the failure (when available, the month is also specified)
- **Cause of failure:** The reported root cause of the failure, if available
- **Impact:** Consequences of the failure
- **How was the root cause identified?** Approach used to identify the source of the failure (usually it is done either by troubleshooting or through a comprehensive investigation)
- **How long was the error present before its discovery?** Time duration between the introduction of the error in the software and its discovery, if available
- **How was the error fixed?** Approach used to correct the faulty software, if available

Table 2.2: Prominent cases of recent software failures collected by the researcher

Industry / Sector	Company / Institution	Failure description	Date	Cause of failure	Impact	How was the root cause identified?	How long was the error present before its discovery?	How was the error fixed?
Finance	RBS (Royal Bank of Scotland)	Total outage: online and offline banking services unavailable for four days in 2012 and for one day in 2013 (Finnegan, 2013)	Dec. 2013 and Jun. 2012	Botched upgrade to batch-processing software	13 million affected users throughout the UK + GBP125 million in compensation and system recovery	Independent review conducted at the request of government (Financial Services Authority)	Error introduced during upgrade, date unspecified	Troubleshooting (patch from manufacturer) but problem not entirely fixed



Industry / Sector	Company / Institution	Failure description	Date	Cause of failure	Impact	How was the root cause identified?	How long was the error present before its discovery?	How was the error fixed?
	AXA Rosenberg group	Incorrect modelling of trading strategy in investment portfolio for two years (Greene, 2011)	Jun. 2009 (discovery of error) Apr. 2010 (disclosure of coding error)	Coding error in quantitative investment model, which minimised an important risk factor	Fraud charges + USD25 million fine + requirement to pay back investors who lost USD 217 million + customer churn	Root-cause analysis process unspecified, probably software review after investors lost money	Programming error was made two years before its discovery, and kept secret for 10 months after its discovery	New software release with coding error fixed 3 to 5 months after its discovery
	National Pension Fund, France	Overestimation of the pensions of one million people for 25 years (Durand-Parenti, 2009)	2009	Logic error in pension calculation algorithm	Cost of EUR300 million to taxpayers in pension overpayment	Root-cause analysis process unspecified	25-years-old bug, introduced at system's inception	Logic error was rectified
Airlines	United Airlines	Two-hour outage that disrupted flight scheduling worldwide (Karp, 2012)	Nov. 2012	Glitch in dispatch system software	636 flights were delayed and 10 cancelled	Root cause unknown	Origin of software problem unknown	Troubleshooting but problem not entirely fixed
Mobile telecoms	Blackberry	Communication services (call, text message, e-mail) unavailable for four days worldwide (Whittaker, 2011)	Oct. 2011	Core switch failure, faulty backup system, and server overload	Over ¾ of the 70 million users worldwide were affected	Troubleshooting	Unspecified, problem generated during operation	Troubleshooting but problem not entirely fixed
	Orange (France)	Services unavailable nationwide for 12 hours (Renault, 2012)	Jul. 2012	Bug in core network device	26 million subscribers affected nationwide	Detailed audit requested by government	Unspecified	Troubleshooting + new device-monitoring measures
Public services	Dallas county police force	About 2 dozens of prisoners were incorrectly released out of jail due to bug in the new record-keeping system (Hallman, 2014)	Jun. 2014	Software defects,	Criminals incorrectly freed from jail and still on the loose	Root-cause analysis process unspecified	1 week	Unspecified. Defects partially corrected



Industry / Sector	Company / Institution	Failure description	Date	Cause of failure	Impact	How was the root cause identified?	How long was the error present before its discovery?	How was the error fixed?
	New York City public housing agency	Millions of welfare families were overcharged for the rent of their public housing accommodation and taken to court for non-payment (Fernandez, 2009)	2009	Logic error, incorrect formula used to calculate rent	Families living in constant fear of eviction; contracted debt to pay extra rent amount	Root-cause analysis process unspecified, probably software review after complaints from tenants	8-month-old bug	Logic error rectified
Healthcare	UK organ donor register	Wrong organs removed from 25 donors' bodies (Roberts, 2010)	Feb. 2010	Errors in the data conversion software used during system upgrade	Donors' families deeply affected	Independent review requested by government following complaints from new donors	10-year-old bug, introduced at the inception of the system	Ordered a new improved system (based on availability of funds)
	Red Wing Ambulance, Minnesota, USA	Woman died in ambulance due to spontaneous shut off of oxygen delivery system (Charette, 2010)	Apr. 2010	Software glitch in oxygen system	Patient died + ambulance was placed under scrutiny	Independent investigations by both manufacturer and ambulance staff, but root cause was not found	Unknown	New improved oxygen system but the problem reoccurred. Ambulance now carries portable oxygen system

The above table shows the following facts about the failure cases that are presented in terms of the cause of the failure, as well as the approach used to fix it and to identify the failure's root cause.

2.3.2.1 Cause of the software failure

- **Failed upgrade:** two cases (RBS and UK organ donor register)
- **Logic error in software code:** three cases (Axa Rosenberg, France's national pension fund, New York City public housing agency)
- **Bug in core network device:** two cases (Blackberry and Orange)
- **Resource exhaustion:** 1 case (Blackberry)
- **Human error (incorrect configuration) :** one case (Dallas County police force)

The above causes are in line with the common causes of software failures presented in Section 3.2.2. However, two of the failure cases (United Airlines and Red Wing Ambulance) have an **unidentified root cause**, despite the investigations conducted. This is alarming as it implies that the problem cannot be solved and is likely to reoccur – which is exactly what happened in both cases.

2.3.2.2 Approach used to fix the failure

- **Troubleshooting:** RBS, United Airlines, Blackberry, and Orange. In each of these cases, troubleshooting was inadequate to solve the problem entirely and additional countermeasures had to be applied. This included conducting an in-depth independent investigation to accurately find the root cause or using a risk-mitigating solution (system-monitoring measures in the case of Orange).
- **New improved system:** AXA Rosenberg, UK organ donor register, and Red Wing Ambulance. In the latter case, the problem reoccurred with the new system as its root cause had not been identified. A risk-mitigating solution was then applied in the form of carrying portable oxygen systems.
- **Correction of coding errors and defects:** France's national pension fund, Dallas County police force, and New York City public housing agency.

2.3.2.3 Approach used to identify the root cause of the failure

- **Unspecified:** In many cases (AXA Rosenberg, France's national pension fund, Dallas County police force, and New York City public housing agency), information on the approach that had been followed to identify the root cause of the failure was not available. It can be assumed that the approach followed involved a software review after the wrong system output had been identified.
- **Comprehensive investigation:** This applies to the failures at RBS, Orange, the UK organ donor register and the Red Wing Ambulance. An investigation was only performed following a request from a higher authority, but this approach was more effective than troubleshooting.

A brief discussion of the above software failures follows next. The discussion is organised in five sections, one for each industry or sector affected.

2.3.3 Software failures according to industry or sector

2.3.3.1 Software failures in the financial industry

The 2012 SQS list of worst software failures indicates that the financial and banking sectors are the top industries affected by these problems. This is mostly due to their legacy systems not being upgraded due to economic constraints. The banking sector is the most error-prone as a result of new trends such as mobile banking, online retail shopping and cloud computing, since these new technologies are not always compatible with existing IT infrastructure (SQS.com, 2013).

One case in point is the system failure at RBS (Royal Bank of Scotland), a major bank in the UK, in December 2013. As indicated in Chapter 1, an unspecified technical glitch caused the bank's various electronic channels to be unavailable for several hours, leaving customers unable to make payments or withdraw cash (Finnegan, 2013). This failure occurred after another major outage in 2012, which left 13 million customers unable to access their bank accounts for four days due to a failure in a piece of batch-scheduling software (Worstell, 2012).

AXA Rosenberg Group, a global investment company, is another example of a financial company that was seriously affected by its legacy system. The equity investment firm was charged with fraud and fined USD 25 million in February 2011 for hiding a coding error in their quantitative investment model (Greene, 2011). The faulty program affected the Group's trading strategy and investment returns and caused investors a loss of USD 217 million.

Another costly error due to a legacy system was found in the French national pension fund system. A software design error caused the overestimation of the pensions of about a million people, which amounted to a cost of over EUR300 million to tax payers. Although it was only discovered in 2009, the error had been present since the inception of the IT system in 1984 (Durand-Parenti, 2009).

Other industries that often make the headlines for their software glitches are the airline industry and the mobile telecommunications industry, because of their high reliance on IT systems. Given the high usage of their services as part of our daily lives, these failures have an emotional impact on many people across geographical borders, as will be shown next.

2.3.3.2 Software failures in the airline industry

In November 2012, for the third time in that year, a glitch in the dispatch system software of United Airlines, the world's largest airline, caused havoc on a number of flight schedules at airports in the USA and around the world. Altogether 636 flights were delayed and ten cancelled due to the two-hour outage, leaving passengers stranded in airport lobbies (Karp, 2012).

2.3.3.3 Software failures in the mobile telecommunications industry

A well-known example of a mobile telecommunications failure is the Blackberry outage that occurred in October 2011. A core switch failure combined with a faulty backup system cut off over three quarters of the 70 million Blackberry smartphone users worldwide for almost four days (Whittaker, 2011). The outage started in Europe and the Middle East and spread to Africa, Latin America, the USA and Canada (Feldman, 2011). It is worth noting that shorter outages had already occurred in 2007 and 2008 for similar reasons (Horton, 2008). This implies that the root cause of the failure had not been properly addressed, hence the recurrence of the problem.

More recently, in July 2012, a software bug in a core network device of Orange, a major mobile operator in France, caused a 12-hour outage that affected 26 million subscribers nationwide. The magnitude of the event turned this national crisis literally into an affair of state and the government ordered an independent root-cause analysis (Renault, 2012).

Although it is not often reported, software errors also affect law enforcement and government agencies, which can result in wrongful criminal charges brought against citizens. Two failure examples from these public services resulted in civil court cases and are presented next.

2.3.3.4 Software failures in public services

In June 2011, more than 20 prisoners in Dallas County were incorrectly released out of jail due to a software glitch in the new record-tracking system. Due to heavy workload and some incorrect system configuration, some cases were not filed into the system within the prescribed timeframe. As a result, some criminals were mistakenly released and at the time of press, they were still on the loose although the police were looking for them. Police plan to find them and send back to jail and they have made changes to the record-tracking system to fix some of the defects and prevent further problems (Hallman, 2014).

Another case of wrongful incrimination happened in New York City in 2009, where hundreds of welfare families were overcharged in rent due to an error in the rent calculation system of the city's Housing Authority (Fernandez, 2009). Many families were taken to court and threatened with eviction for failing to pay the extra amount (up to USD 200). This computer error ran for nine months and left many tenants in a constant fear of being thrown out in the street, while it pushed some people to resort to debt to pay for the overcharge (Fernandez, 2009).

Extreme cases of software failures are a real threat to human well-being and can actually result in injury and loss of life. Examples from the healthcare industry where this is often the case are discussed next.

2.3.3.5 Software failures in the healthcare industry

Although they are chronically underreported (Bogdanich, 2010), software failures abound in medical devices. To support this argument, Roberts (2012) states that in 2011 software failures were responsible for 24% of all medical device recalls by the FDA. The review of the medical literature conducted on this topic highlights three principal areas of concern: radiation therapy machines, external infusion pumps and implantable medical devices such as pace makers.

Software design flaws in these machines cause problems such as incorrect dosages of medication, administration of incorrect treatment or abrupt system shutdown, which are often the result of operators' errors. Medical reports point to the following recurring causes for these faults: poor user interface design, unclear error messages and inadequate input validation.

Fatalities due to faulty software in these machines are often kept confidential but some are reported on the FDA and the IAEA portals. The cases discussed next represent some examples of such errors. The first two are not as recent as the examples listed in Table 3.2, but they were the latest that could be found to illustrate the above point.

In 2004 a patient with an implantable drug pump died from an overdose because the operator set the bolus interval to 20 minutes instead of 20 hours, thus at 60 times the prescribed rate. The operator's error was due to the poor user interface where the hour and minute fields for a bolus rate were ambiguously labelled on the computer screen (FDA, 2004a). One other death and seven serious injuries have been attributed to this data entry error (FDA, 2004b).

In 2007 a programming error unexpectedly shut down a patient's implantable infusion pump during use. The issue was caused by an overflow in the memory buffer that feeds the main processor. The underdosed patient's blood pressure dropped and he experienced increased intracranial pressure, followed by brain death (FDA, 2007). Interestingly, brain death is listed as the cause of the death (FDA, 2007), instead of the software failure.

More recently, in April 2010, a woman was killed by an oxygen software failure in an ambulance in Minnesota, USA. Unknown to the paramedic and for some unidentified reason, the oxygen delivery system spontaneously shut off for eight minutes (Charette, 2010).

A more morbid example is provided by the faulty software of the UK organ donor register. The wrong organs were taken from the bodies of 25 donors due to the software misreading their donor forms. Although the software error was introduced in 1999, the problem was only discovered in 2010 after new donors complained that their information was incorrect following a thank-you note from the organ donor agency (Roberts, 2010).

2.3.4 Lessons learnt

The overview of software failures presented earlier in the chapter proves that these events are an unfortunate reality. Although technical details on the investigations were not available, several issues emerged from this review:

- Many software errors are discovered accidentally rather than through a routine check or through system monitoring. For this reason, they can remain hidden for a long time and cause significant problems in the long run.
- Software failures are usually resolved through troubleshooting only, unless an in-depth investigation is ordered by a regulation authority.
- Troubleshooting alone is inadequate and cannot prevent the recurrence of failures. This suggests that troubleshooting may not identify the real source of the problem.
- A comprehensive investigation is often necessary to accurately identify the root cause of the failure.

The next section delves into the above issues in more detail by presenting a case study of three fatal software failures. These are cases of radiation overdoses whose technical reports are publicly available. Although these cases are not as recent as the examples described earlier, they indicate a recurring pattern in the way software failures are handled. This trend is not apparent from the above examples, due to the limited information available on their investigation.

2.4 Case study of software-induced radiation overdoses: AECL Therac-25, Multidata RTP/2 and Varian IMRT

This section presents the researcher's case study of three series of radiation overdose due to software malfunctions so as to demonstrate their similarities and infer valuable lessons in terms of failure investigation. Unlike other software failures whose technical details have been shielded from public view, comprehensive reports on these disasters are publicly available online, hence their selection. Although the three accidents occurred over a period of two decades and involved three different radiation therapy systems, they show striking similarities that suggest that history repeats itself. The same mistakes are still being made, which shows that software developers, manufacturers and operators do not learn from past failures. All three cases resulted in death and subsequent lawsuits and they clearly illustrate the devastating effect of software failures and of an inadequate post-mortem investigation.

The three cases in question are the Therac-25 disaster in the USA and Canada between 1985 and 1987, the radiation accidents at the Panama National Cancer Institute in 2001 and the

radiation overexposure at the St. Vincent Hospital in New York City in 2005. More recent cases of radiation overdose due to software glitches have been reported in 2007 (Bogdanich, 2010b), 2008 (Bogdanich, 2010b) and 2009 (Bogdanich & Rebelo, 2010). However, they could not be used as case studies due to the limited information published.

The first and third cases studied in this section involved the software component of a linear accelerator, which is a machine that generates beams of high-energy radiation to treat cancer patients (Bogdanich, 2010). The second case involved a treatment planning software, which is a component of a decision support system used to calculate recommended patients' treatment time and radiation dose (IAEA, 2013b). Table 3.3 presents a summary of each of the three cases.

Table 2.3 has twelve fields that provide specific information on each case. Each case is described in a separate column and each field is a row entry in that column. Although Table 2.3 contains the same fields as Table 2.2, it also displays the following differences:

- No field is provided for “Industry/Sector” as all cases are medical accidents.
- No field is provided for “For how long was the error present before its discovery?” as the software bugs responsible for each accident were all introduced during system development.

Table 2.3 contains the following five additional fields:

- **Impact on other parties.** Besides the patients who were directly involved, the software manufacturer, the hospital and the machine operators were all severely affected by the accidents and sometimes held accountable for it.
- **Factors that facilitated the overdose:** Various prior events and conditions contributed significantly to each overdose. If these had been addressed effectively early on, the administration of the overdose could have been prevented.
- **Factors that contributed to the negative impact of the accident:** A number of factors allowed the overdose and its negative health impact to persist for an unnecessary long period of time. If these had been addressed sooner, this could have limited the impact of the accident and may even have saved the patients' lives.
- **How was the overdose detected?** Unlike failures described in Section 2.3.2, which have symptoms that are immediately visible (e.g. system outage and incorrect

calculation), the radiation overdose was not visible immediately after the treatment had been administered.

- **Initial reaction from manufacturer:** This indicates how the system manufacturer reacted when hospital staff suspected and alleged that the software was the cause of the radiation overdose. This initial reaction played an important role in the subsequent investigation.

The above five fields were used to highlight the similarity between the accidents and the factors that had a negative impact on the investigation and subsequent system improvement.

Table 2.3: Researcher’s summary of 3 cases of radiation overdose due to software errors

Machine/Product	Therac-25 linear accelerator	Radiation treatment planning software RTP/2	Linear accelerator for IMRT (Intensity Modulated Radiation Therapy)
Software manufacturer	AECL (Atomic Energy of Canada Limited)	Multidata Systems International (US firm)	Varian Medical Systems (US firm)
Year of accident	1985-1987	2001	2005
Location	11 hospitals throughout the USA and Canada	Panama National Cancer Institute in Panama City	St. Vincent Hospital in New York City
Accident description	A series of six machine malfunctions occurred in various hospitals either due to operators entering incorrect data or the system crashing unexpectedly. Each time, the machine tripped and generated misleading error messages, but also delivered an extremely higher dose of radiation.	The software allowed the operators to enter input data in an incorrect format, which led to the miscalculation of patients’ treatment time and an overexposure to radiation for several months.	The computer crashed while the physician was trying to save the revised treatment plan. The instructions for the machine calibrations were mistakenly deleted and the machine delivered a higher level of radiation for three consecutive days of treatment.
Root cause of software failure	Race condition (programming error)	No validation of input data (logic error)	Non fail-safe termination (data corruption)
How was the overdose detected?	Continued symptoms of radiation overdose for several weeks	Continued unusual reactions in some patients for several months	Patient’s unusual reaction to treatment observed by his family
Initial reaction from manufacturer	Overconfident about software quality; rejected possibility that software was faulty; blamed patient’s symptoms on hardware faults or operator’s incorrect use of machine	Overconfident about software quality; rejected possibility that software was faulty; blamed patient’s symptoms on operator’s incorrect use of machine	Blamed accident on operator’s negligence



Machine/Product	Therac-25 linear accelerator	Radiation treatment planning software RTP/2	Linear accelerator for IMRT (Intensity Modulated Radiation Therapy)
How was the root cause identified?	Manufacturer investigation at the request of FDA after an informal troubleshooting approach	<ul style="list-style-type: none"> • Troubleshooting by independent experts • independent investigation from IAEA requested by Panama Government • FDA investigation of manufacturer's operations 	Manufacturer investigation submitted to the FDA. Details on the investigation were not found
How was the software bug corrected?	Manufacturer corrected bugs at the request of FDA.	The manufacturer conducted a recall and in-field correction of the software and provided a detailed description of the cause and circumstances of the incorrect data entry.	Manufacturer corrected bug and distributed an improved software with a fail-safe provision to its customers worldwide.
Factors that facilitated the overdose	<ul style="list-style-type: none"> • Unclear user manual • Poor system feedback • No investigation of prior harmless software malfunctions • No reporting of initial accidents to other users • Familiarity with similar and harmless malfunctions • Lack of a routine check 	<ul style="list-style-type: none"> • Unclear user manual • Poor system feedback • No investigation of prior harmless software malfunctions • No reporting of previous software malfunctions to other users • Lack of a routine check 	<ul style="list-style-type: none"> • Poor system feedback • No investigation of prior harmless software malfunctions • Familiarity with similar and harmless malfunctions • Lack of a routine check
Factors that contributed to negative impact of accident	<ul style="list-style-type: none"> • Late detection of overdose • Delayed root-cause analysis • Delayed software correction • Informal troubleshooting approach 	<ul style="list-style-type: none"> • Late detection of overdose • Delayed root-cause analysis • Delayed software correction 	<ul style="list-style-type: none"> • Late detection of overdose • Delayed root-cause analysis • Delayed software correction
Impact on cancer patients	Six patients were overdosed: three patients died and three were severely burnt.	28 patients were overdosed: 18 died, and the others developed serious health complications.	After two years of declining health, the patient died of his radiation injuries.
Impact on other parties	<ul style="list-style-type: none"> • The machine was recalled by the FDA in 1987. • FDA requested a corrective action plan (CAP) from AECL. • The AECL and hospital received lawsuits from affected patients and their families. 	<ul style="list-style-type: none"> • The three responsible physicians were trialled for murder. Two were sentenced to four years' imprisonment and banned from practising their profession for seven years. • The FDA banned the manufacturer from operating in the USA. 	<ul style="list-style-type: none"> • The FDA blamed the hospital for negligence and the manufacturer for the faulty system. • The city fined the hospital for USD1000. • Hospital paid financial settlement to victim's family.

Table 2.3 shows that the three medical accidents are very similar in terms of the following aspects that confirm the observations made in Section 2.3.4.

- **How was the overdose detected?** Patient's unusual reaction to treatment (and not planned software output verification)

- **How was the root cause identified?** Through troubleshooting, followed by thorough investigation (software review)
- **Factors that facilitated the overdose:** Lack of a routine check of the software operations
- **Factors that contributed to the negative impact of the accidents:** Late detection of the overdose

Table 2.3 also reveals the following aspects of the accidents that did not emerge from the software failures presented in Section 2.3.

- **Factors that facilitated the overdose:** No reporting of previous software failures and no investigation of prior harmless malfunctions
- **Factors that contributed to the negative impact of the accidents:** Delayed root-cause analysis, delayed software correction

Before discussing the above issues further, a brief chronological description of the accidents is presented next.

2.4.1 Accident description

2.4.1.1 Therac-25

Several bugs in this linear accelerator caused a series of six malfunctions in different hospitals between 1985 and 1987. In every accident, the machine was either unable to process instructions as they were given or it displayed misleading and unclear error messages. This led the operators into unknowingly administering a massive overdose of radiation that exceeded the prescribed dose by a hundred times (Leveson & Turner, 1993). Besides the fact that the accident caused the death of three patients and serious injury to three others, affected patients or their families filed several lawsuits against the various hospitals and the AECL. All lawsuits were settled out of court and the machine was recalled by the FDA in 1987 (McCormick, 2004).

2.4.1.2 Multidata RTP/2 Treatment Planning Software

The Multidata treatment planning software (TPS) enabled therapists to draw on a computer screen the placement of metal shields (called blocks) designed to protect healthy tissue from

radiation (McCormick, 2004). The TPS's normal operation only allowed up to four blocks, but one oncologist requested a fifth block to further protect the more sensitive tissue. The physicians tried a new data entry method to bypass the software constraint by drawing the five blocks as a single large block with a hole in the middle. The TPS accepted the invalid input data without giving a warning but, unknown to the physicians, calculated an incorrect treatment time that caused double the normal dose of radiation (IAEA, 2013b).

The impact of this failure was monumental. The modified treatment plan was administered to 28 patients treated for cancer of the prostate or the cervix (IAEA, 2001). Of them, 18 died, while the others developed serious health complications (Borras, 2006). The FDA issued an injunction against Multidata to prohibit the firm from operating in the USA until they fixed the bug and became fully compliant with the FDA safety standards (McCormick, 2004). In addition, the three physicians who inadvertently administered the radiation overdose were trialled for murder because they were legally required to verify the software calculation by hand (Garfinkel, 2005). One was acquitted, while the other two were sentenced to four years in prison and banned from practising their profession for seven years (Diaz, 2004).

2.4.1.3 Varian IMRT linear accelerator

In March 2005 a patient who was being treated for a tongue cancer was mistakenly administered the wrong treatment with a radiation seven times his prescribed dose. The problem occurred during the fifth radiation session, after the patient's reformulated treatment plan was accidentally deleted due to a system crash. The patient died from ensuing health complications two years later. The government investigators who conducted an enquiry found that both the hospital and the manufacturer were to blame for the accident. The city of New York levied a USD1000 fine against the hospital and ordered the hospital to pay a financial settlement to the victim's family.

2.4.2 How was the overdose detected?

In each of the above three radiation overdose cases, the patients developed obvious radiation burn symptoms shortly after the treatment. These were reddened and swollen skin in the case of the Therac-25, diarrhoea with the Multidata TPS, and swollen head and neck with the Varian

linear accelerator. However, because the symptoms were initially attributed to the disease, they were not followed up. The overdose was only discovered later in an unplanned way.

In the first case, the overdose was confirmed after the fourth and fifth accidents when the hospital technician managed to reproduce the conditions surrounding the accidents and measured the resulting dose. A day later, AECL followed the same procedure and confirmed the overdose. In the second case, a physician accidentally discovered the computer miscalculations many months after the overdose had been administered. When calculating the dosages for two patients with the same treatment, she suddenly noticed a mismatch between her results and the software's calculations (McCormick, 2004). The overdose was then confirmed through treatment simulation and the treatment was suspended (IAEA, 2013b). In the third case, the physician only conducted a verification test on the treatment plan after the third session and discovered the incorrect machine settings.

2.4.3 How was the root cause identified?

The root cause of the radiation accidents was only discovered through thorough software inspection, often after a number of unsuccessful troubleshooting attempts. Indeed, the investigators initially “diagnosed” the system failures based on their experience with the system and without supporting evidence, which is typical of troubleshooting. The basic investigation method was to try to reproduce the malfunction in order to find its origin.

This strategy is evident from the case of the Therac-25 where the AECL engineers initially suspected a hardware fault. They hardwired the conditions for this fault to occur and applied certain countermeasures, claiming afterwards that the problem had been solved. Their claim was proved incorrect as similar failures reoccurred after the suspected hardware problem had been fixed.

The same subjective diagnosis approach is also apparent in the case of the Panama accident. Some independent experts discovered the flaw in the software algorithm responsible for the overdose, based on their experience with a similar failure. Indeed, one of the radiotherapist experts said that the calculation error was a problem that had occurred in older similar treatment software. He remembered seeing a physician in the USA make this error ten years earlier and he consequently looked for it during the investigation. Fortunately in this case, independent

reviews from the IAEA and FDA confirmed his suspicion through testing of the software by using different data entry approaches (IAEA, 2001).

2.4.4 Factors that facilitated the overdose

In the case of all three accidents, no verification test was conducted right away to validate the output of the medical software. No routine check or system monitoring was in place to verify the proper functioning of the machine. This allowed the incorrect dose to be unwittingly administered several times to different patients. Two other factors that are recurrent in the three accidents also facilitated the overdose: the underreporting of system failures and the disregarding of harmless malfunctions.

As a matter of fact, the first and third malfunctions of the Therac-25 were not reported to other users of the machine until later accidents also occurred (Leveson, 1993). This gave users a false sense of confidence in the proper functioning of the machine. Besides, the previous numerous malfunctions of the machine were never investigated as they had been harmless. Indeed, since the installation of the machine (two years before the accidents), the operators had become accustomed to its frequent malfunctions – up to 40 per day – which had never affected any patient prior to the deadly accidents. In such cases, the operator would simply call a hospital technician to reset the machine and restore it to service (Leveson, 1993).

In the case of the Varian software, malfunctions of the software were equally common. Operators were used to its frequent but harmless crashes which they regularly reported to the manufacturer but which were never investigated (Bogdanich, 2010). Likewise, in the case of the Panama accident, the manufacturer neither reported nor investigated the miscalculations reported by previous customers close to a decade earlier.

Nevertheless, reports show that in all three cases these minor problems followed similar patterns as the deadly failures (e.g. machine abruptly stops and restarts, same unclear error messages or miscalculations). Against the background of the discussion on the near-miss concept in the previous chapter, these minor problems were clearly cases of near misses. They were, in hindsight, clearly worth some attention and could well have provided distinct clues about the design flaw that caused the accidents.

2.4.5 Factors that contributed to the negative impact of the accidents

In each of the three accidents, the overdose was only detected long after it had been administered, which implies that it was significantly harming the patients over a period of time.

The benefit of early failure detection is demonstrated with another case of the Varian software failure. Indeed, a similar problem with the Varian software occurred in a different hospital several months after the notorious accident mentioned above. Fortunately, the overdose was detected soon after the treatment and the patient was not injured (Bogdanisch, 2010). It is safe to say that the early detection of this software error saved the patient's life.

2.4.6 Lessons learnt

The three cases examined above clearly demonstrate that, in general, software failures are not handled efficiently. They reveal some clear limitations in the investigation of software failures and the resulting catastrophic consequences. More specifically, they confirm the observations made previously about the unplanned and inadvertent detection of software errors by using troubleshooting as an ineffective first reaction to a failure, as well as the problem posed by not reporting software failures and not investigating recurring near misses. Even more recent cases of radiation overdose caused by software failures (Bogdanich, 2010b) show similar patterns. This motivates the need for a more efficient and accurate failure investigation process that caters for these shortcomings. Designing a system to implement such an investigation process is the goal of this study. The shortcomings that have been identified are used as the basis for developing the key requirements that a near-miss management system should comply with. These requirements are established in the following section.

2.5 Requirements for accurate failure investigation

This section presents the key requirements for the proposed NMS inferred from the lessons learnt in the previous section. A brief review is given of the limitations identified in the existing approach towards failure investigation and then the requirements are established to address these limitations.

2.5.1 Limitations in the investigation of software failures

This section examines the limitations in the investigation process observed by the author of this thesis from all software failures reviewed in Sections 2.3 and 2.4. The limitations that are presented are those that the author found recurring in a number of cases and that directly affect the quality of the ensuing investigation.

2.5.1.1 Troubleshooting as the first response to a major software failure

Companies affected by failures are often reluctant to conduct a thorough investigation, as it is costly and time-consuming. They rather focus on quickly putting back the system into operation through troubleshooting. However, troubleshooting lacks objectivity and accuracy. It is a short-term solution and in-depth investigations are needed to find a long-term solution. Besides, by restoring and rebooting the system, troubleshooting tampers with digital evidence of the failure, which can have a negative effect on the subsequent investigation.

2.5.1.2 Lack of a standard investigation process

The above review of various failure cases indicates that each failure is handled differently. There is no common procedure to investigate failures to identify the root cause. This leads to some subjectivity in the procedure used and in the results obtained. Moreover, in many cases, details of the investigation process are not available, making it hard to assess its effectiveness or to reproduce the investigation to confirm its results.

2.5.1.3 No investigation of near misses

Near misses are not given any attention as they cause no harm. However, they clearly show similar patterns as the serious failures and can therefore provide valuable insight into software weaknesses and bigger threats.

Examples of near misses in software systems were provided by the reports of the three deadly medical accidents involving radiation therapy machines. Another example is the three-day Blackberry outage that occurred in 2011.

With regard to the Blackberry case, the outage was intermittent and comprised a succession of smaller failures, which indicates that different things went wrong at different times. Press

reports indicate that firstly, a central server went down, then the backup system failed; e-mail traffic was then rerouted to another main server, which soon became overloaded (Mashable, 2011). Each of these unsafe events was a precursor to the subsequent outage. This catastrophic event could have been a mere near miss if the faulty backup server had been replaced or repaired before the second main server became overloaded.

Regarding the radiation therapy accidents, all three cases were preceded by a number of harmless malfunctions following similar patterns as the deadly accidents. The correct handling of these near misses could have helped prevent the accidental death of a number of cancer patients.

2.5.1.4 No real-time detection of failures

Some software failures are only detected long after they occurred, due to the lack of a continuous monitoring of the system's operations. As a result, the impact of the failure can be significant and data that could help in the investigation may get lost as it gets overwritten by subsequent operations.

2.5.2 Requirements for accurate software failure investigation

From the previous discussion on the limitations in the failure investigation process, it is clear that a detailed investigation, rather than a quick fix, is required to prevent the recurrence of a similar failure. The investigation must have the following qualities:

- **Objectivity:** In order to provide reliable results, the investigation must be objectively based on an analysis of data about the failure. The results should be independently verifiable and not subject to the investigator's familiarity with the system.
- **Comprehensiveness:** The investigation must be comprehensive to cater for all possible causes of the failure, which must all be tested.
- **Reproducibility:** A different investigator should obtain the same results by following the same procedure as the initial investigators. This is best achieved through a standard investigation process, which eliminates the risk of subjectivity.
- **Admissibility in court:** Although it cannot be inferred from the limitations discussed earlier, it is important to realise that a number of the reviewed failures resulted in

lawsuits. It is therefore imperative that the investigation process and the results obtained be admissible in a court of law, in the event of litigations.

The failure investigation approach must also ensure the following:

- **Continuous system monitoring:** Monitoring of a system's operations and behaviour is required to avoid accidental or fortuitous discovery of failures, and to allow for the real-time detection of errors and risky conditions that are conducive to failures.
- **Investigation of near misses:** Identifying and addressing the root cause of near misses, even though they are harmless, can help prevent a more serious accident from developing and can also provide valuable information for conducting the root-cause analysis of the subsequent failure.

Several root-cause analysis methods have been proposed to address the above requirements and are commonly available today. Examples include application performance management (APM), business transaction management (BTM), change and configuration management (CCM) and the war room approach (Neebula.com, 2012). Except for the war room approach, these methods focus on maintaining and improving application performance through the continuous analysis of the end-user experience (APM), tracking the flow of transactions along a business transaction path (BTM) or detecting inappropriate changes and configurations (CCM) to prevent significant performance drop and related failures. The war room approach brings experts from different disciplines (e.g. server, network, application) into the same room so that they can analyse the problem together to quickly find its root cause (Neebula.com, 2012).

Despite its value, the war room is still vulnerable to the subjectivity of the participants and dependent on their knowledge of their specific discipline. The other methods, although valuable, focus on performance improvement and not on preventing the recurrence of failures; hence some manual guessing about the root cause of the failure is required (Neebula.com, 2012). Furthermore, none of the above methods simultaneously covers all the requirements identified above and none of them caters for the eventuality of a product liability lawsuit.

The solution to this problem suggested in this study is to use digital evidence of the failure as the basis of the investigation. Such a strategy has the potential to satisfy all the requirements

established previously. Indeed, rather than relying on the investigator's experience with the system, the investigation must follow a predefined process that collects and analyses evidence of the failure to find its origin. This will ensure the objectivity and reproducibility of the investigation. Besides, digital evidence is required for formal court proceedings. Compliance with the last two requirements (continuous system monitoring and investigation of near misses) can greatly assist with the process of obtaining reliable digital evidence. Continuous system monitoring can indeed detect failures as they occur and can therefore facilitate the collection of evidence before it gets overwritten by subsequent operations. Furthermore, the investigation of near misses provides an additional source of evidence about system malfunctions that can supplement the failure investigation.

Although digital evidence is primarily in the form of log files, it may also include any other data about the failure. Preserving the integrity of this evidence is necessary to ensure the reliability of the root-cause analysis. The informal troubleshooting approaches that are often adopted to conduct root-cause analysis do not promote the collection and preservation of digital evidence. In fact, in order to restore the system to its normal operational state, rebooting is often required, which completely destroys or at least tampers with potential evidence (Trigg & Doulis, 2008).

Before looking for alternative solutions, a logical step is to turn to the field of digital forensics, which uses objective evidence to provide clarity on the cause and circumstances of an event while adhering to principles of law (Vacca & Rudolph, 2011). It can therefore serve as an effective alternative to investigate software failures, although it is currently limited to the investigation of criminal events and security incidents. Digital forensics will be reviewed in the next chapter to determine its suitability with regard to the established requirements for a more accurate software failure investigation.

2.6 Conclusion

This chapter presented a number of recent severe software failures and the detailed case study of three fatal medical accidents caused by faulty software. The aim was to highlight the negative impact of software failures and to identify the limitations in current failure investigation practices. The literature review established that, despite their catastrophic consequences, major software failures are not given the timely and full attention they require.

They are often handled inefficiently and irresponsibly through informal troubleshooting, which enables their recurrence.

The limitations in current failure investigation practices create the need for a more accurate investigation approach. Requirements for such an approach have been established in this chapter, based on the identified shortcomings in failure investigations. Using digital evidence as the basis of the investigation was found necessary to meet these requirements, and digital forensics was identified as a promising solution to facilitate the collection and analysis of digital evidence. A review of the digital forensics process and its prospects as a solution to the lack of accuracy in software failure investigations are presented in the next chapter.

CHAPTER 3

USING DIGITAL FORENSICS FOR ACCURATE INVESTIGATION OF SOFTWARE FAILURES

3.1 Introduction

This chapter presents digital forensics as a promising solution to the limited accuracy of software failure investigations. As demonstrated in the previous chapter, inaccurate identification of the cause of the failure leads to the implementation of inappropriate countermeasures that are not suitable to prevent the failure from reoccurring. Although it is not currently used for failure analysis, the formal process of digital forensics has the potential to provide sound evidence of the root cause of the failure by making a scientific analysis of the digital evidence in this regard.

To ensure the reliability of the results of a digital forensic investigation, various steps are performed to enable the collection of the digital evidence and to preserve its integrity throughout the investigation. This process for collecting, preserving and analysing digital evidence was identified as favourable to satisfy the requirements established in the previous chapter for a more accurate failure investigation, namely *objectivity*, *comprehensiveness*, *reproducibility*, and *admissibility in court*. Supporting activities to foster these desired qualities of the investigation included the continuous monitoring of the system to detect failures early, and the investigation of near misses as an additional source of evidence of the failure.

Chapter 3 demonstrates the suitability of digital forensics to meet the above requirements. It first motivates the selection of digital forensics as a viable alternative for investigating software failures. It then discusses how digital forensics can be used to satisfy the requirements for more accurate failure analyses.

The chapter is structured as follows: Section 3.2 provides an overview of digital forensics and its current applications. Section 3.3 presents several arguments to support the suggestion to use

digital forensics to investigate software failures. Section 3.4 reviews the building blocks of digital forensics and their potential application in failure investigations to meet the above requirements. Finally, Section 3.5 uses these building blocks to determine how suitable digital forensics is to improve the accuracy of failure investigations.

3.2 Overview of digital forensics

This section starts with a brief introduction to the field of digital forensics and then presents an overview of its current applications.

3.2.1 Introduction to digital forensics

Initially called computer forensics (Vacca & Rudolph, 2010), digital forensics is also known under a number of different names, including computer forensic science (Noblett, Pollitt, & Presley, 2000), forensic computer science (IJoFCS, 2012), forensic computing (McKemmish, 2008), system forensics, as well as electronic or digital discovery (Vacca & Rudolph, 2010). It is a relatively new discipline in the established field of forensic science. Digital forensics was recognised as a forensic science by the American Academy of Forensic Sciences in 2008 (Kessler, 2009).

The term “forensic” means “suitable in a court of law” (Merriam-Webster, 2014). Forensic investigations are therefore conducted with a view to achieving that potential outcome. Forensic science, often shortened as forensics, is defined as the “application of scientific knowledge and methodology to legal problems and criminal investigations” (Free Online Law Dictionary, 2013). It deals with the scientific identification, analysis and evaluation of physical evidence (Free Online Law Dictionary, 2013). Digital forensics, as the application of forensics to computer science, also deals with the scientific handling of evidence, but such evidence is in an electronic form and resides on a digital device.

Digital forensics brings scientific rigor, combined with a solid legal foundation, to an investigation. This combination makes it an efficient approach to investigations, as it produces results that are reliable and legally acceptable in the eventuality of a lawsuit. For this reason, digital forensics is used for various legal and regulatory purposes as will be discussed next.

3.2.2 Digital forensic applications

Digital forensics is primarily used for the investigation of computer-related crimes. This includes crime cases where the computer is the target of the crime (e.g. hacking), the instrument of the crime (e.g. phishing) or a storage facility for evidence about the crime (e.g. money laundering) (Vacca & Rudolph, 2010). Note that computer in this context is a broad term that refers to any computing device on which information is stored in a binary form (Kessler, 2009). This includes laptops, desktops, servers, network devices such as routers and switches, as well as mobile devices such as mobile phones, tablets and digital cameras.

Digital forensics is a fast-growing field due to the increased occurrence of cybercrime and breaches of information security policies, which affect people's safety and companies' trade secrets. Digital forensics is used to identify the perpetrators of such malicious acts in order to prosecute or take disciplinary action against them. It also has non-prosecutorial applications in a number of professions. The following are some examples identified by Vacca and Rudolph (2010):

- The *military* uses digital forensics to obtain intelligence information from computers seized during military actions.
- *Law firms* use digital forensic professionals to find digital evidence in support of civil cases such as divorce and unfair labour practice.
- *Insurance companies* use digital evidence to investigate potential fraud based on, for instance, arson or workers' compensation claims.
- *Data recovery firms* use digital forensic techniques to recover data lost due to an accident or a hardware or software failure. Note that digital forensics is not used to identify the cause of the failure.

As these examples show, even when not used for prosecution, digital forensics is mostly limited to cases related to law, regulations or policies. This is understandable as the primary objective of a digital forensic investigation is to ensure that the findings can serve as valid evidence in a court of law. As in other branches of forensic science, its focus is on the integrity of the evidence that must have been collected and handled according to rigorous guidelines and in conformity with all applicable laws (Vacca & Rudolph, 2010).

It is precisely this emphasis on scientific rigor and standard procedures that distinguishes digital forensics from current failure analysis methods. Examples of such methods were discussed in the previous chapter and include troubleshooting, application performance management, business transaction management, change and configuration management, and the war room approach (Neebula.com, 2012). Scientific rigor is also the reason why digital forensics is deemed a promising candidate to address the limited accuracy in current practices of software failure analysis.

This argument is elaborated on in more detail in the next section. Various factors are discussed that indicate that digital forensics can be successfully used in non-criminal failure investigations while retaining its legal foundation.

3.3 Motivation for using digital forensics for software failure investigations

This section presents arguments to support the suggestion that digital forensics be used to improve the accuracy of software failure investigations. The argumentation is based on the following three pillars that are deemed sufficient and appropriate to prove the above point:

- Supporting software literature that recommends the forensic investigation of system failures
- Formal definition of digital forensics that allows for non-criminal investigations
- Analogy to other disciplines in forensic science that are also used to investigate non-criminal adverse events in their respective fields

These arguments are developed in Sections 3.3.1, 3.3.2 and 3.3.3 respectively.

3.3.1 Supporting literature

A review of the literature on software failure investigations indicates that a number of authors share the view expressed, namely that digital forensics has clear benefits over current failure analysis methods. Indeed, for over a decade, several authors (Grady, 1996; Corby, 2000; Johnson, 2002; Hatton, 2004; Stephenson, 2003; Jucan, 2010; Hodd, 2010; Meyer, 2011) have been advocating an in-depth root-cause analysis of software failures so as to prevent their reoccurrence and ultimately improve the faulty software. A number of them (Corby, 2000; Stephenson, 2003; Kent, Grance, Chevalier, & Dang, 2006; Turner, 2007; Meyer, 2011)

specifically recommended digital forensics to assist in this process. Although many of their publications focus on incident response cases (i.e. failures due to security incidents) (Kent et al., 2006; Turner, 2007), they indicate that the concept can be applied to other situations as well.

One of the first authors to suggest the use of digital forensics in computer failure investigations is Michael Corby. Already in 2000, Corby observed that the increased complexity of IT systems, combined with a focus on immediate system recovery, makes it challenging to establish the source of a failure and to ascertain whether it was intentional or accidental. As a solution, he proposed that digital forensic methodology be introduced to collect failure-related data before returning the system to operation, so as to prevent the loss of potential evidence in a post-mortem analysis. Corby (2007) argued that the proper collection of evidence significantly reduces the time needed for and complexity of the digital post-mortem.

Stephenson (2003) also suggested the use of digital forensics for failure investigations but focused on security incidents. He argues that investigations of adverse events in the IT industry lack structure and formal modelling, which can cast doubt on the credibility of the outcome. Digital forensics, due to its mathematical foundation, can add structure and rigor to an investigation, thereby providing confidence in the accuracy of the results. He consequently proposed an approach to digital post-mortems with a formal modelling of the investigation process and possible outcomes using colored Petri Nets (Girualt & Valk, 2003). Although this methodology was designed specifically for security incidents, it demonstrates the benefits of applying forensic techniques to the investigation of adverse events such as software failures.

In 2006, the American National Institute of Standards and Technology (NIST) published a guide on how to integrate digital forensic techniques with incident response (Kent et al., 2006). The guide clearly indicates that digital forensics can be used for a number of purposes, including troubleshooting operational problems and recovering from accidental system damage. Thus, NIST urged every organisation to acquire forensic capability to assist in the reconstruction of systems and network events. The guide explains how to establish such a forensic capability, as well as how to develop appropriate policies and procedures.

A year later, Turner (2007) also highlighted the value of combining the forensic approach with the procedure used for incident response, network investigation or system administration. He indicated that digital forensic tools and techniques can be used to address limitations of system administration procedures following an incident. Examples of such limitations are the loss of or tampering with potential evidence and the lack of recording of both the timestamp and the actions performed. He therefore recommended incorporating digital forensics as an integral part of the post-incident system administration process and proposed a “Digital Evidence Bag” to preserve digital evidence used in the investigation as a way to achieve this goal.

More recently, Meyer (2011) promoted the technical analysis of software failures for improving software quality and reliability. He even suggested the adoption of a law that systematically requests such an investigation for every large-scale software failure. Although Meyer (2011) does not specifically refer to the use of digital forensics in the investigation, he uses the analogy of airplanes crashes, which are legally required to be investigated thoroughly and thus have black boxes to record potential evidence. He believes that this formal detailed evidence-based analysis has significantly contributed to the increase in airline safety and argues that the IT industry should follow this approach to improve software quality.

In conclusion, the authors mentioned above are of the view that software failures – whether accidental or criminal – are not investigated efficiently. This does not only hamper the prevention of their recurrence, but also thwarts the correction of faulty software and obstructs the improvement of its quality and reliability. It also confirms the findings from the case studies of software failures in the previous chapter. As a solution, the authors referred to above recommend a formal evidence-based post-mortem analysis through the integration of digital forensics. As a matter of fact, using digital forensics for exactly such a purpose is catered for in its definition – as will be discussed next.

3.3.2 Definition of digital forensics

Various definitions of digital forensics are available in the literature, depending on the perspective used. It is usually defined from either a legal, a criminal, or a process perspective. This section examines some of these definitions to support the argument that digital forensics is suitable for failure investigations. It then presents a general definition suitable for the purpose of this study.

3.3.2.1 Existing definitions of digital forensics

The Digital Forensic Research Workshop (DFRWS) established one of the first formal definitions of digital forensics during their first meeting in 2001 (Palmer, 2001:22). It reads as follows:

Digital forensics is the use of scientifically derived and proven methods towards the preservation, collection, validation, identification, analysis, interpretation and presentation of digital evidence derived from digital sources for the purposes of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate the unauthorised actions shown to be disruptive to planned operations.

This definition focuses on the procedure followed in a digital forensic investigation as it lists the sequence of steps involved. It also specifies the domains of application of this process – reconstruction of criminal events and prevention of unauthorised actions – the latter referring to the breaching of policies. This definition reflects the standard application of digital forensics, as explained earlier. However, it clearly mentions event reconstruction as a goal, which is also the goal of a failure investigation.

Indeed, event reconstruction is the process of determining the underlying conditions and the chain of events that have led to an incident (Carrier & Spafford, 2004). It involves examining the evidence and proposing hypotheses about the events that occurred in the system and caused the incident (Jeyaraman & Atallah, 2006). The purpose of a failure investigation is also to identify the root cause of the failure, which is determined by the events and conditions that led to the failure.

In 2005 the above definition of digital forensics was revised by Willasen and Mjølunes (2005: page 1) to read as follows:

Digital forensics is the practice of scientifically derived and proven technical methods and tools towards the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of after-the-fact digital information derived from digital sources for the purpose of facilitating or furthering the reconstruction of the events as forensic evidence.

This definition is very similar to the previous one. However, one important difference is that it does not qualify the events analysed as criminal and therefore broadens the scope of digital forensics to the reconstruction of non-criminal events as well. This provides room for software failures as one type of non-criminal event that can be investigated with the digital forensic process.

A more recent and shorter definition of digital forensics is suggested by Vacca and Rudolph (2010:3):

Digital forensics is the process of methodically examining computer media as well as network components, software and memory for evidence.

This definition does not specify the type of events investigated but focuses on the source of the evidence, which can be any component of an IT system, either internal (software and memory) or external to the system (network component and storage media). Investigating a software failure may also require the examination of these different components of the system. Vacca and Rudolph's definition also refers to following a strict procedure for the investigation by qualifying the examination as methodical.

Next follows an analysis of the main aspects of the previous definitions that are relevant for the requirements for accurate software failure investigations.

3.3.2.2 How can the definitions of digital forensics satisfy the requirements for accurate failure investigation?

Based on the above review of digital forensic definitions, the following is a presentation of the aspects of digital forensics deemed suitable for the requirements established to improve the accuracy of software failure investigations. In every relevant aspect presented, the requirement that is satisfied is displayed in italics.

- The techniques and tools used are based on science, which implies the *objectivity* of the investigation.
- These techniques are applied to digital information obtained from digital sources. They accommodate any electronic device as a source of digital information, which allows the examination of all relevant components of the failed system and thus contributes to the *comprehensiveness* of the investigation.

- The investigation follows a standard predefined procedure, which contributes to its *objectivity* and *reproducibility*.
- Forensic evidence is the output of the investigation, in other words legal principles are followed so that the results are *admissible in court*. This also implies the *objectivity* of the findings as they are based on analysed evidence.

In summary, the above analysis demonstrates that digital forensics, by its mere definition, can be applied to the investigation of software failures and provide accurate results. The definition of digital forensics used in this thesis for this purpose is presented next.

3.3.2.3 Working definition of digital forensics

Köhn (2012:24) proposed the following definition of digital forensics, which encompasses all the aspects discussed above and is therefore adopted in this thesis:

Digital Forensics is a specific, predefined and accepted process applied to digitally stored data or digital media using scientific proven and derived methods, based on a solid legal foundation, to extract after-the-fact digital evidence with the goal of deriving the set of events or actions indicating a possible root cause, where reconstruction of possible events can be used to validate the scientifically derived conclusions.

The definition of digital forensics served as the second argument to motivate its suitability for software failure investigations. The third and last argument is derived from a review of other forensic disciplines that are also commonly used for the investigation of non-criminal failures and adverse events. They are reviewed in the next section.

3.3.3 Lessons learnt from other forensic disciplines

Although forensic science is not currently used for failure analysis in the software industry, this is not a new concept in other industries. Two industries that systematically make use of forensic science for such a purpose are the engineering and the healthcare industries. In the engineering industry, this field of practice is called forensic engineering (Noon, 2001), while it is known as forensic pathology in the healthcare industry (Dolinak, Matshes & Lew, 2005). In each of these disciplines, the goal of the investigation is to improve the quality and reliability

of the systems, products and procedures, and to prevent the recurrence of failures and adverse events. This is also the aim of the suggested forensic analysis of software failures. Forensic engineering and forensic pathology are briefly discussed in Sections 3.3.3.1 and 3.3.3.2 respectively.

3.3.3.1 Overview of forensic engineering

Forensic engineering applies various scientific examination tools and techniques, simulations and event reconstruction methods to identify the source of disastrous failures in engineering products (McDanel, 2006). The emergence of formal failure investigations as currently conducted in forensic engineering can be traced to the Industrial Revolution, during which many complex machines were introduced. The added complexity led to many accidents that required expert analysis to understand their causes and prevent their reoccurrence. The types of accidents evolved as engineering products developed from steamboats, railway trains and steel bridges in the 1800s to automobiles, home appliances and airplanes in the 1900s (Brown, Obenski & Osborn, 2003).

Forensic engineering has evolved from its initial focus on legal investigations in product liability cases to its current focus on failure analysis for product and system quality improvement purposes. Presently, most forensic engineering investigations never reach the courtroom and are conducted mainly with a view to preventing similar accidents in future (Carper, 2000).

One such example is the forensic investigation conducted into the collapse of the World Trade Center on 11 September 2001, which was undertaken to understand the impact of the fire on the collapse of the twin towers – despite the fact that the responsible parties were already known (Usmani, Chung & Torero, 2003). Various elements such as the construction design, fire properties of materials used, and their thermal expansion were examined through simulations and computer-based structural analysis. Based on this analysis it was determined that using reinforced concrete instead of lightweight steel, as well as providing an energy-absorbing structure could prevent the collapse of such tall buildings in the future (Zhou, 2004).

3.3.3.2 Overview of forensic pathology

Forensic pathology is a branch of medicine “that applies the principles and knowledge of the medical sciences to problems in the field of law” (Dimaio & Dimaio, 2001). It has been an integral part of medicine since the 19th century and was formally recognised in the United States in 1959 by the American Board of Pathology (ItsGov.com, 2011). Forensic pathology is primarily used to investigate the cause of death upon a legal request (Dimaio & Dimaio, 2001). However, it also has applications in public health and safety to prevent and control diseases and to prevent drivers’ injuries.

In public health, forensic pathologists track trends for a potential disease outbreak, the emergence of a new infectious disease, or a bioterrorism attack (Springboard, 2014). For instance, a forensic autopsy may uncover a previously undetected contagious disease, and this knowledge can be used to prevent an outbreak. It may also help identify a hereditary condition that will enable family members to proactively seek treatment or limit the effect on new-born babies and future offspring (Dolinak et al., 2005).

In public safety, forensic pathology is used particularly to promote driver safety (Springboard, 2014). As a matter of fact, the investigation of deaths and injuries in road accidents has led to policy changes in several countries. For instance, in the United States, such investigations led to the introduction of optional fitting and wearing of seat belts in cars in 1955. This legislation was later made mandatory in many countries, following its positive effect on road safety and reductions in fatalities (Onyiaorah IV, 2013).

The reviews of the above two disciplines show that, through the integration of scientific methods and legal principles, the forensic approach has ensured rigorous and comprehensive investigations and has created opportunities for improvement and for preventing the recurrence of failures and accidents. This effect has been demonstrated for over a century in the engineering and medical fields. In view of the above benefits in other fields of forensic science, similar benefits are expected in the software industry, based on the application of digital forensics to failure investigations. As was the case with forensic engineering, the increased complexity of software systems requires more formal and in-depth investigations than are currently available through using troubleshooting and other failure analysis methods.

Section 3.3 motivated the use of digital forensics for software failure analysis. It showed that digital forensics can be applied to non-criminal or legal cases and can be beneficial to failure analysis. The next step is to find out how digital forensics can improve its accuracy. This requires an understanding of the distinctive characteristics of a digital forensic investigation.

These characteristics are its use of scientific methods and techniques and its adherence to legal principles. These elements are reviewed in Sections 3.4 and 3.5 respectively. The investigation process is a product of these elements to ensure the legal acceptance of the results (discussed in Section 3.6). Examples are provided of how each of these characteristics of digital forensics can assist in the accurate investigation of software failures. The examples involve the three deadly radiation therapy accidents reviewed in the previous chapter

3.4 The scientific foundation of digital forensics

As a recognised science, digital forensics must adhere to accepted scientific methodologies. Two fundamental ones commonly used in digital forensics are the scientific method and mathematical analysis. The scientific method is discussed in Section 3.4.1 and mathematical analysis is reviewed in Section 3.4.2.

3.4.1 The scientific method

This section starts with a general overview of the scientific method. It then examines how the scientific method can help improve the accuracy of software failure investigations.

3.4.1.1 Overview of the scientific method

The scientific method is a process used by scientists to conduct an objective investigation of an event. Its aim is to minimise bias or prejudice from the experimenter and ensure the accuracy of the results (Bernstein, 2009). It generally consists of the following four steps (Young, 2007):

1. Observation of an event or problem related to the event
2. Formulation of a hypothesis (or hypotheses) to explain the event
3. Prediction of evidence for each hypothesis
4. Testing of hypothesis and predictions through controlled experimentations by several independent experimenters

This method is iterative as the steps are repeated until a conclusion can be reached. Findings are then reported.

One crucial element of the scientific method is falsification. The scientist must actively seek to disprove or falsify the hypothesis (Young, 2007). If experiments and observations discredit the initial hypothesis, this hypothesis can be discarded and a more valid one must be created and tested. Even if experiments confirm the hypothesis, alternative findings to disprove the hypothesis must be explored – this can be done by other scientists (Casey, 2010). In addition, new information can emerge during an investigation, and it must be reviewed and evaluated to ensure that the hypothesis still stands (Casey, 2010). Furthermore, results are peer reviewed before finalisation (Gogolin, 2013). Falsification promotes the objectivity and reproducibility of the results and is a key differentiator between a formal scientific investigation such as digital forensics and a non-scientific one such as troubleshooting.

3.4.1.2 How can the scientific method help improve the accuracy of software failure investigations?

The scientific method promotes the objectivity of the investigation

If applied to software failures, a digital forensic investigation based on the scientific method differs greatly from the troubleshooting approach. Throughout this chapter, troubleshooting is used as a reference to existing failure analysis methods as the review of software failure investigations in the previous chapter indicates that it is the most common first response to a failure. Details on the application and usage of other failure analysis methods were not found in the literature review on software failures.

A typical troubleshooting process consists of the following steps (Trigg & Doulis, 20018):

5. Recreate the problem. This means reproducing the actions that led to the failure.
6. Isolate the cause of the problem. This is a process of eliminating components that are not at fault.
7. Fix the problem.
8. Test the solution to see if it solves the problem.
9. If the problem is not solved, repeat the process, otherwise document the work that was performed.

It is evident from the above process description that troubleshooting neither uses digital evidence to identify the cause of the failure, nor provides digital evidence of the identified cause to ensure its reliability. In addition, no peer review of the results is required, so the investigator does not have to justify the selected solution to independent parties. The scientific method of digital forensics forces the investigator to take these steps to ensure that the results are objective and reliable.

The scientific method promotes the reproducibility of the investigation

The outcome of a troubleshooting process relies on the important first step, which is the recreation of the problem. No standard or controlled procedure is followed to perform this step, leading to potential bias from the investigator based on his experience. In addition, depending on the severity of the failure and the complexity of the system, recreating the problem is not always feasible or advisable, which can bring the investigation to a halt with no identified cause for the problem. Examples of such situations were provided in Chapter 2. A case in point is the death of a woman in an ambulance due to the spontaneous shut off of the oxygen delivery system (Charette, 2010). The investigators were unable to recreate the shut off and could not determine the reason for it. The ambulance now carries portable oxygen systems as a risk-mitigating solution.

The scientific method promotes the comprehensiveness of the investigation

As the above process shows, troubleshooting does not use falsification. As soon as a hypothesis (i.e. a potential cause of the failure) is confirmed, the hypothesis is deemed true and the system is restored to normal operations. As a result, the root cause may not be addressed and the problem may well reoccur at a later stage.

The case of the Therac-25 disaster presented in the previous chapter (Leveson & Turner, 2002) illustrates this situation. The engineers from AECL, the software vendor, were set on attributing the second radiation accident to a hardware fault and closed their mind to any other explanation for the failure. They hardwired the suspected fault, found some design errors that confirmed their hypothesis and stopped the investigation right after. As it turned out, this theory was soon proved wrong as other accidents occurred after they had fixed the hardware flaw.

Situations like these can be avoided by using falsification. Since other failure analysis methods do not follow the scientific method (Neebula.com, 2012), it is safe to infer that they too do not use falsification. However, unlike troubleshooting, details on their application and usage were not found in the literature review on software failures.

3.4.2 Mathematical analysis

In addition to applying the scientific method, digital forensics also uses other scientific methodologies to ensure the credibility of the results. These methodologies are based on rigorous mathematical techniques tested and accepted in the computer science community. Two main ones are discussed below: the use of a hash algorithm and neural networks.

3.4.2.1 Hash algorithms

One of the techniques commonly used is a hash algorithm. A hash algorithm generates a unique mathematical representation of a data set, drive or file, called a hash value, which can then serve as its fingerprint (Noblett et al., 2000). Popular hash functions are MD5, SHA-1 and SHA-256 and SHA-3 (Roussev, 2009; Breitingner, Stivaktakis & Baier, 2013). Hash values have a number of applications in digital forensic analysis. Some examples follow.

Validation of data authenticity and integrity

As in other forensic sciences, the evidence collected during an investigation must remain unchanged. When acquiring data from a suspect computer, the copied or imaged data must be an exact replica of the original data. A hash algorithm is applied to both the original data and its image. The resulting hash values of both data sets are then compared to ensure that the image has not been altered during acquisition. The hash value of the image is maintained in the case file and can be used at any moment during the investigation to verify that the integrity of the data has not been compromised (Mabuto & Venter, 2011).

File identification

Hash values are also used to remove irrelevant files or identify known files of interest to save time when searching for pieces of information through the collected data. Examples of irrelevant files are known and trusted files such as operating system files and application installation files. Examples of files of interest are illegal files such as pirated media or hacking scripts (Roussev, 2009). The list of hash values from the collected files is compared to a pre-

compiled hash set of known files such as the NIST National Software Reference Library (NIST, 2013), which has a collection of hash values of most common software applications.

Additional applications of hash values include discovering deleted files, processing corrupted or formatted computer media, identifying different versions of a file (e.g. a file that is saved as an HTML document and also as a text file or a new version of an executable file), and finding traces of a file across various data sources (e.g. a file system image and a memory dump) (Roussev, 2009). Roussev (2009) provides a detailed explanation of the techniques used for these various applications of hash values.

A number of forensic software tools are available to perform data analysis using hashing algorithms. Popular ones are FTK from AccessData (AccessData, 2013), Encase from Guidance Software (Guidance Software, 2013) and the open source Sleuth Kit maintained by Brian Carrier (sleuthkit.org, 2013). Mabuto and Venter (2011) provide a comprehensive list of digital forensic techniques and tools that are used in the industry.

3.4.2.2 Neural networks

One mathematical analysis technique used for data classification that has applications in digital forensics is neural networks. Indeed, one of the challenges in forensic investigations is the increasingly large volume of data to be analysed to produce reliable digital evidence (Quick & Choo, 2014). Digital forensic tools and techniques mentioned earlier are limited in their ability to analyse large data sets. Therefore more powerful mathematical techniques are used when appropriate. A particularly suitable technique adopted in the computer science community is neural networks.

Neural networks are formally defined as: a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs (Caudill, 1989). Neural networks are modelled on biological neural networks to identify patterns in data sets (Engelbrecht, 2003). For instance, supervised neural networks can be used for complex pattern recognition in network forensics. In Khan, Chatwin & Young (2007), the network is trained with consecutive snapshots of the file system to recognise the normal behaviour of a program. The trained network can then be used to automatically build an execution timeline on a forensic image of a file system to help identify

available evidence. Self-organizing maps (SOM's) have also been proposed for pattern classification in digital forensic investigations. A SOM is a model of unsupervised neural networks used for the analysis and visualisation of multi-dimensional data (Engelbrecht, 2003). SOM's can display a colored map showing identified clusters in the data set, thereby enabling the quick identification of outliers. A case study of the application of SOM's to digital forensic investigations was provided in Fei, Eloff, Venter & Olivier (2005).

3.4.2.3 How can mathematical analysis help improve the accuracy of software failure investigations?

Mathematical analysis promotes the legal acceptance of the results

Current failure analysis methods do not rely on mathematical analysis. Using mathematical analysis to authenticate evidence can therefore prove valuable in software failure investigations that result in court proceedings, more specifically in product liability litigations. It also helps provide sound evidence of the findings. An example is the case discussed in the previous chapter, namely of the radiation accident caused by Varian software in the St Vincent Hospital (Bogdanich, 2010).

The Varian software wrongfully indicated that the instructions for positioning the radiation beam were in place while the corresponding files were actually corrupted and inaccessible. A hash value of the image of the system's hard drive would confirm that this was indeed the case at the moment of the accident, and that the files did not get corrupted during the imaging process.

The Varian software wrongfully indicated that the instructions for positioning the radiation beam were in place while the corresponding files were actually corrupted and inaccessible. A hash value of the image of the system's hard drive would confirm that this was indeed the case at the moment of the accident, and that the files did not get corrupted during the imaging process. Processing the corrupted files would also show that the correct positioning of the radiation beam had been set prior to administering the treatment. This would help prove that the machine operator did not enter incorrect settings and that the faulty software was responsible for the radiation of the incorrect cells.

The Varian software indicated that the treatment plan was saved, while the file was in fact missing. Using data recovery techniques could also help recover the missing file and serve as further evidence that the correct treatment plan had been set prior to delivering the radiation dosage. This would prove that the poorly designed software that displayed a misleading “saved” message was to blame for the resulting overdose of radiation.

The above discussion is obviously a simplification of this complex failure as a more thorough analysis would be required to determine the source of the problem. However, it illustrates the potential application of a hash value for such a purpose.

Besides its scientific foundation, digital forensics relies on legal principles applicable to the handling of the digital evidence. A set of best practices has been developed to ensure adherence to these legal requirements and is reviewed in the next section.

3.5 Best practices in digital forensics

Results of a digital forensic investigation must be forensically sound to be admissible in court. Forensic soundness refers to the preservation of the integrity and completeness of the data throughout the investigation (McKemmish, 2008). Section 3.4.2 above presented analysis techniques used to verify the integrity and completeness of the evidence. These techniques, which support best practice developed to ensure forensic soundness of the digital evidence, are presented in Section 3.5.1 and their application to software failure investigations is discussed in Section 3.5.2.

3.5.1 Overview of best practices in digital forensics

Minimal handling of original data

The investigator should minimise manipulation of the original data to prevent its spoliation. He/She should rather make a copy of the relevant original data and work with that copy. Data duplication must not alter the original data and should provide a complete copy of the drive or device being copied (Vacca & Rudolph, 2010).

Keeping account of any change to the data

Whenever data is altered during the investigation, the investigator must record details of the change and be able to explain its impact on the investigation (Köhn, 2012).

Maintaining the chain of custody

The chain of custody is a legal term that refers to “the movement and location of evidence from the time it is obtained until the time it is presented in court” (Free Online Law Dictionary, 2013). It is a chronological documentation of the different persons having custody of the evidence and being responsible for its integrity as well as the different places where the evidence was stored (Free Online Law Dictionary, 2013). This requires recording the handling of the collected data. Collected data should therefore be kept by an independent neutral party in a secure storage protected from potential tampering (Corby, 2000).

Ensuring transparency of the investigation process

An audit trail of the process followed should be maintained and should be verifiable by an independent party. Reproducing the process should provide the same results. This requires the documentation of all the steps taken, tools and techniques used, and any problem or error encountered (McKemmish, 2008).

3.5.2 How can digital forensic best practices help improve the accuracy of software failure investigations?

Best practices promote the reproducibility of the investigation

A detailed documentation of the investigation makes it possible for independent parties to reproduce the results without unnecessary time delay. Transparency also provides a learning tool that can be used in the event of a similar problem. An example that illustrates this point is the case of the Therac-25 series of accidents (Leveson & Turner, 1993), which were unsuccessfully handled through troubleshooting.

The Therac-25 overdose of radiation was only acknowledged and confirmed by AECL, the software manufacturer, when they recreated the failure after the sixth accident. Indeed, the hospital physicist managed through numerous simulations to reproduce the condition that elicited the displayed error message and measured the resulting dose. AECL engineers had been unsuccessfully trying to reproduce the malfunction for a whole day following the accident

and wrongfully assumed that the machine was not at fault. After receiving instructions from the physicist, they followed the same procedure as him and finally obtained the same results.

Best practices promote the comprehensiveness of the investigation

Working on a copy of the data allows the investigator to carry on with the root-cause analysis after system restoration, thus minimising disruption to business operations. This gives him/her the necessary time to conduct a comprehensive investigation. In troubleshooting, once the system is restored, no further investigation is usually conducted. In rare cases where there is a subsequent root-cause analysis, causal data may already have been lost during system restoration activities, which can jeopardise the investigation.

Best practices promote the objectivity of the investigation

Following the above principles enables the investigation to be peer-reviewed more easily and quickly, as well as conclusions to be verified independently, which is not the case with existing failure analysis methods.

Best practices promote the legal acceptance of the results of the investigation

By observing legal principles to handle the collected data, the forensic soundness of the evidence is preserved throughout the investigation.

Applying the scientific method and adhering to applicable principles of law are the foundations of a digital forensic investigation. The investigation process is a by-product of these concepts and it is reviewed in the next section.

3.6 The digital forensic process

A digital forensic investigation must follow a structured procedure to ensure forensic soundness of the evidence. This structured procedure is referred to as the digital forensic process (Mabuto & Venter, 2011). A description of this process is provided in Section 4.4.3.1 and its potential application in software failure investigations is discussed in Section 4.4.3.2.

3.6.1 Overview of the digital forensic process

The digital forensic process is defined as a number of steps to be performed from the incident alert through to the reporting of findings (Casey, 2004). At its most basic level, this process has the following three steps: acquisition, analysis and reporting (Carrier, 2004). A description of the three phases follows.

3.6.1.1 Acquisition

The primary objective of this phase is to identify and collect all potential evidence for later analysis by using sound forensic procedures that include the following steps:

- Securing the electronic crime scene. The suspect computer and its surroundings are secured to avoid damage to or contamination of potential evidence.
- Documenting the crime scene. This includes taking handwritten notes of observations made and details of the suspect machine. Examples are its make, model, serial number, state (on or off) and peripherals such as external hard drives, speakers, webcams and cable modem. Documentation of the crime scene also includes taking photographs and video tapes of the room, the computer and the computer screen, and conducting preliminary interviews with witnesses and victims (Craig, 2006).
- Collecting and preserving the evidence. A typical digital forensic investigation makes use of both physical and digital evidence (Carrier, 2004). Collecting physical evidence involves seizing any suspect or relevant computer medium and computing device. Collecting digital evidence requires imaging the hard drive of the suspect device to a trusted device using software imaging tools. These tools use a write blocker to prevent any writing to the original drive (TechMediaNetwork, 2013). Preserving the evidence involves isolating it to avoid its contamination (Mukasey, Sedgwick, & Hagy, 2008).
- Packaging the evidence. Collected evidence should then be bagged and tagged.

3.6.1.2 Analysis

This phase examines the acquired data to find evidence that either validates or contradicts a hypothesis about the case or that shows that the system was tampered with to hide data. The scientific method is applied in this phase to reach conclusions based on the evidence found (Carrier, 2004). Mathematical analysis and other digital forensic techniques (e.g. string searching, production of time stamps, reconstruction of e-mail and web-browsing activity)

(Mabuto & Venter, 2011) are used to analyse the data and reconstruct the crime or incident that took place. This phase is typically performed in a digital forensic laboratory.

3.6.1.3 Reporting

Once the analysis is complete, the conclusions are drawn and the corresponding evidence is presented to the relevant party/parties. In a criminal case, this is typically a judge or jury, while in a corporate setting it is usually the executives and human resources of the employing company (Carrier, 2004). Presentation is usually in the form of a written report along with the electronic evidence on a computer medium, but it can also include expert testimony in a court of law (Casey, 2004).

3.6.2 Standardising of the forensic investigation process

The forensic investigation process was recently standardised by ISO/IEC 27043 (2015). This standard process is a harmonisation of the various idealised models for common incident investigation processes across a number of investigation scenarios where digital evidence is involved. The standard forensic investigation process is comprehensive in the sense that it includes the complete end-to-end processes from pre-incident preparation through to investigation closure as associated concurrent processes. This long list of investigation processes and sub-processes are classified into several high-level classes as shown in Figure 3.1, which encompass the three major phases described earlier.

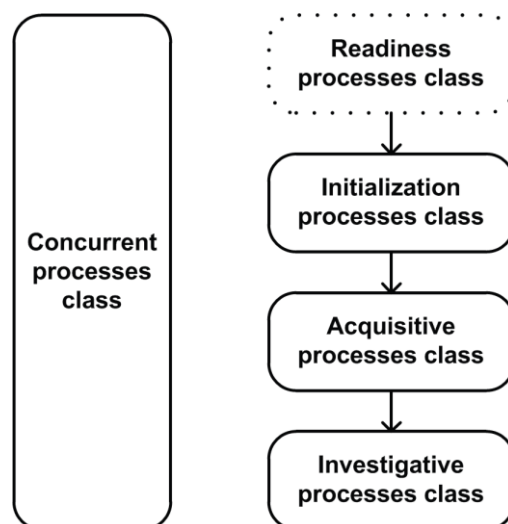


Figure 3.1: The various classes of digital investigation processes

ISO/IEC 27043 (2015) clearly requires that the first response to an incident does not negatively impact the possibility to perform a digital investigation, e.g. to avoid powering off the equipment, opening or changing files on a live system etc. This supports the argument made in the previous chapter that troubleshooting is an inadequate first response to a failure as it can tamper with potential digital evidence that may be required in the subsequent root-cause analysis. However, no detailed description of the first response sub-processes is provided in the Standard, which leaves the identification of an appropriate solution unclear.

Additionally, the Standard stipulates that “identifying potential digital evidence at the incident scene is of crucial importance for the remainder of the process, because if potential digital evidence is not identified at this point, it might not even exist at a later point during the process”. This further supports the argumentation made in this thesis of collecting volatile digital evidence at runtime to prevent the potential loss of the evidence after the failure.

Furthermore, the Standard indicates that incident detection procedures should be in place prior to the beginning of this process. This is in line with the suggestion to detect near misses at runtime to alert users of an upcoming failure so that appropriate actions might be taken prior the unfolding of the failure.

3.6.3 How can the digital forensic process help improve the accuracy of software failure investigations?

The digital forensic process promotes the reproducibility of the investigation

A standard process ensures reproducibility of the investigation and promotes its objectivity. The case of the Therac-25 disaster is a clear example of how following different procedures for root cause analysis can be detrimental. Each of the six incidents was handled differently, based on the given engineer’s experience with the system, and consequently produced different results each time. The failures were every time attributed to various elements, including the hardware, the electric circuit and the operator’s errors. The only common conclusion – which eventually proved to be erroneous – was that the software was not at fault.

The digital forensic process promotes the comprehensiveness of the investigation

The forensic process requires all potential evidence to be collected and analysed, as this promotes the comprehensiveness of the investigation. Following the same procedure for the

crime scene inspection would be equally valuable. Protecting, isolating and capturing the state of the scene of the failure could include taking snapshots of the error message and screenshots from the computing device, as well as capturing potential physical evidence such as the conditions of the treatment room and the patient’s reaction to treatment (e.g. skin redness) in the case of a medical device failure. This would help to reconstruct the chain of events that led to the failure and to understand the factors that contributed to the accidents (e.g. misleading “Saved” message in the case of the Varian software radiation accident; output diagram seemingly correct in the Panama case).

Section 3.6 reviewed the main attributes of digital forensics, their potential application in failure investigations and their benefits over current failure analysis in terms of improved accuracy. The discussion showed that digital forensics has the potential to provide results that are more accurate than current failure analysis methods. The next section provides a summary of this discussion to determine how well digital forensics is suited for this purpose.

3.6.4 Suitability of digital forensics for accurate failure investigations

Table 3.1 summarises the main differences in the investigation process between digital forensics and troubleshooting, based on the reviewed characteristics of digital forensics. Troubleshooting is used as the most common response mechanism to software failures.

Table 3.1: Differences between digital forensics and troubleshooting

Digital Forensics	Troubleshooting
Investigation based on standard reproducible process	Informal investigation process, not followed rigorously, not reproducible
Scientific analysis	Intuitive analysis, relies on experience with the system
Analysis based on obtained evidence	No evidence required
Review of all possible solutions to problem and selection of most convincing one	Find only the first satisfactory solution to problem
Falsification inherent to analysis process	No falsification
Peer review of findings	No peer review of findings
Documentation of actions taken and results obtained before, during and after investigation	Documentation only after investigation is complete
Adherence to applicable legal principles	Legal principles are not relevant for the investigation

Based on its main characteristics listed in the above table, digital forensics has the ability to satisfy the requirements for improved accuracy of failure investigations. Table 3.2 presents a summary of how this is achieved. The researcher organised the table in two columns: the first column lists each requirement, while the second column indicates the specific aspect of digital forensics that makes it suitable for the given requirement.

Table 3.2: Suitability of digital forensics for the requirements of an accurate software failure investigation

Requirement	Relevant digital forensic characteristic
Objectivity	<ul style="list-style-type: none"> • Scientific method • Mathematical analysis • Best practice for transparency – keeping an audit trail of process • Peer review
Reproducibility	<ul style="list-style-type: none"> • Standard digital forensic process • Best practice requiring documentation of process • Scientific method with peer review
Comprehensiveness	<ul style="list-style-type: none"> • Scientific method with falsification • Peer review • Digital forensic process requiring collection and analysis of all potential evidence • Best practice requiring to work on a copy of the evidence
Admissibility in court	<ul style="list-style-type: none"> • Analysis based on obtained evidence • Best practices for forensic soundness of evidence • Mathematical analysis to authenticate evidence

As Table 3.2 shows, digital forensics can satisfy all four requirements. Examples on how this could be achieved were provided throughout the chapter. However, despite these expected benefits, digital forensics is currently not used in failure investigations. Various challenges related to the nature of software failures have been cited to explain this gap. One of them is the volatility of the digital evidence, indicating that the evidence can be compromised following a failure. Another challenge is to limit downtime following a failure, which requires the forensic analysis to be conducted only after an initial system restoration. Applying digital forensics to the investigation of software failures needs to make provision for meeting these challenges. The digital forensic process might therefore need to be adapted to the above specific

requirements of software failures. The proposed solution to this issue is presented in the next chapter, following a review of previous work conducted on this subject.

3.7 Conclusion

Chapter 3 presented digital forensics as a potentially significant contribution to improving the accuracy of software failure investigations, and provided several motivating arguments for using digital forensics for such a purpose. The chapter also reviewed the main concepts of digital forensics and provided examples on how they can be applied to and benefit failure investigations, based on the requirements for accurate failure analysis that were established in the previous chapter. Despite being supported by a number of authors and having clear advantages over current failure analysis methods, digital forensics is not yet applied to failure investigations at this point in time. The next chapter therefore attempts to determine potential hindrances in the application of digital forensics to failure analysis and proposes an adapted digital forensic process to address such obstacles.

CHAPTER 4

THE ADAPTED DIGITAL FORENSIC PROCESS FOR FAILURE INVESTIGATIONS

4.1 Introduction

Chapter 3 presented the argument for using digital forensics to investigate software failures with greater accuracy. It discussed the supporting view of a number of authors on this topic and presented examples of various characteristics of a digital forensic investigation that could be applied to failure investigations and promote the requirements established for increased accuracy. However, despite this argumentation and although forensic science has been applied to failure analysis in some industries for over a century, the fact remains that digital forensics is currently not used in the software industry to investigate failures. This was clearly noticeable from the review of the investigation of recent cases of major software failures that was conducted in Chapter 2.

Chapter 4 therefore attempts to shed light on the factors that hamper the adoption of digital forensics in failure analysis and proposes a solution to these challenges. This solution is presented as an adapted digital forensic process that will be deemed more suitable for the specificities of software failures than the currently available process designed primarily for criminal cases. Some elements of this process have been inspired by the results of earlier research on the use of digital forensics for failure analysis, which are reviewed in the current chapter. The viability of this process is evaluated through the case study of a real-life software failure, more specifically the Therac-25 disaster presented in Chapter 2. The current chapter again presents this case study.

The chapter is structured as follows. Section 4.2 discusses specificities of software failure analysis that are not addressed by the current digital forensic process. Section 4.3 reviews previous work conducted on this subject to identify results that can help in the design of the proposed process for the forensic investigation of software failures. In Section 4.4 the proposed

process is described in detail, while in Section 4.5 it is applied to the Therac-25 accidents to examine its viability.

4.2 Challenges to the forensic investigation of software failures

The previous chapter demonstrated that the digital forensic approach can be applied to the investigation of software failures and can promote the accuracy of the investigation. This forensic approach involves three main phases (acquisition, analysis and reporting) associated with various activities that can be summarised as follows:

- Identify potential digital evidence of the event investigated
- Collect the digital evidence with a software imaging tool
- Store the imaged evidence to a trusted device
- Verify and maintain the authenticity and completeness of the stored evidence through mathematical analysis and best practices
- Analyse the stored evidence with the scientific method and forensic tools and techniques
- Draw conclusions about the event (e.g. author of crime and modus operandi) based on the scientific analysis of the evidence
- Report the conclusions with supporting evidence

The above process is illustrated in Figure 4.1.

The above process was designed for the investigation of computer crimes. A review of this process and the literature on forensic failure analysis indicates that the approach presents various challenges with regard to the investigation of software failures. These challenges are discussed next.

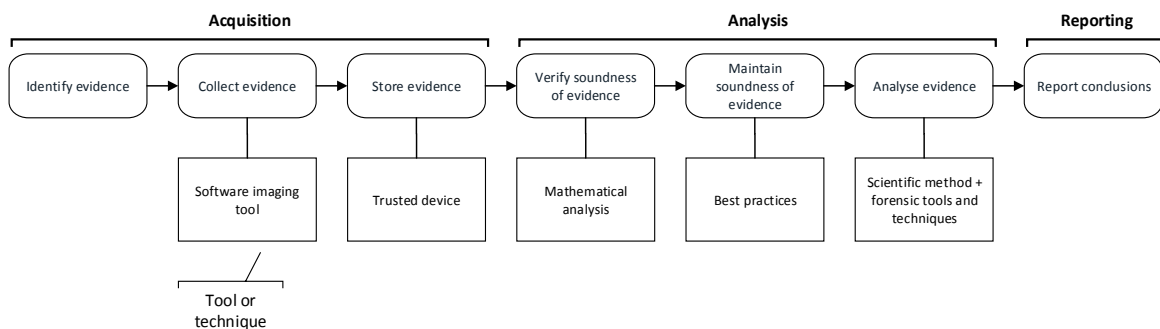


Figure 4.1: The digital forensic process

4.2.1 The volatility of digital evidence

As discussed in the first chapter, operational data pertaining to the failure that can be used as digital evidence may be lost or corrupted after a system crash. In addition, troubleshooting activities such as system rebooting can also tamper with available digital evidence (Trigg & Doulis, 2008). Collecting appropriate and reliable evidence may therefore be challenging. This is even more so in the case of embedded systems, as is the case for many of the systems mentioned in the real-life cases of failures in Chapter 2.

Embedded systems usually have no hard disc and the data is stored in memory chips, built internally in the embedded system (Breeuwsma, 2006). Imaging this memory is usually particularly challenging for a number of reasons. Firstly, imaging tools may not be authorised to run on these systems due to restrictions from the manufacturers. Secondly, the operating systems of embedded devices may not include features available on standard desktop operating systems for software-based memory imaging. Thirdly, the lack of standardised interfaces in embedded systems does not allow for an easy attachment of hardware memory imaging tools (Rabaiotti & Hargreaves, 2010). Some specialised tools and techniques for memory imaging in embedded systems, although limited, are available. Some examples, such as loading a small Linux operating system onto the device or using a test access port to directly access the memory chips, are discussed in Breeuwsma (2006) and Rabaiotti & Hargreaves (2010).

As evidence collection is one of the first steps in the forensic approach, it has repercussions for the entire forensic process. For this reason, the volatility of the evidence is considered to be the biggest challenge posed to a successful forensic investigation of software failures. This is even more so seeing that digital forensics relies on the completeness of the evidence to ensure the accuracy of the results. The researcher's proposed solution to this issue is the collection and analysis of data from near-miss events, which is not currently catered for in digital forensics. Near-miss analysis will be reviewed in detail in Chapter 5.

4.2.2 The lack of forensic tools and techniques for the root-cause analysis of software failures

Various forensic tools and techniques are available for the investigation of computer crimes. Although some of these techniques can be applied to failure analysis, they are either used to

authenticate evidence (e.g. mathematical analysis) or to find evidence of a known crime (e.g. string searching or reconstruction of web-browsing activity). They are not designed to find the (unknown) root cause of a failure. There is also no logical method available to pinpoint the root cause of a failure by using the scientific techniques that are currently available for data analysis (e.g. statistical analysis). Consequently, an accurate failure analysis method with suitable scientific techniques needs to be identified. Although the goal of this study is not to identify or provide forensic tools and techniques for failure analysis, the design of the prototype in Chapter 8 highlights some possible solutions.

4.2.3 The need to minimise downtime following a failure

System downtime can be very costly and minimising its duration is critical. This requires restoring the system to its normal functioning before a proper root-cause analysis can be performed or completed. It also requires the quick restoration of the system without losing potential evidence. Since restoration can disturb potential evidence, the evidence needs to be collected before restoring the system (Corby, 2000). This method differs from digital forensics, where the analysis can be started as soon as the evidence has been collected, regardless of the state (on or off) of the suspect machine. The digital forensic process clearly does not make provision for a two-phased approach to evidence collection (firstly collection of evidence and secondly system recovery). It is however valuable for the forensic investigation of a software failure.

4.2.4 The need for continuous system monitoring

Ensuring that operational data is captured and preserved after a failure requires continuous system monitoring and logging to ensure the availability of the data when needed (Corby, 2000). Continuous system monitoring also facilitates the early detection of failures, which can then be addressed timely. However, system monitoring is not a task performed in digital forensics and it is not provided for in its investigation process.

The above challenges are summarised in Table 4.1.

Table 4.1: Challenges to the forensic investigation of software failures

Challenge	Description
-----------	-------------

The volatility of digital evidence	Potential digital evidence may be destroyed during and after a failure
The lack of forensic tools and techniques for the root-cause analysis of software failures	Available forensic tools and techniques are not designed to find the root cause of a failure
The need to minimise downtime following a failure	In order to minimise downtime, the system must be restored before starting the digital forensic investigation. This process is not catered for in digital forensics.
The need for continuous system monitoring	Ensuring the availability of digital evidence requires continuous system monitoring but this task is not provided for in digital forensics

Proposed solutions for the first two challenges above are described in subsequent chapters, as they are novel techniques and methods. The proposed process focuses on the last two challenges, which can be addressed through adapting the digital forensic process and supplementing it with currently available methods. Despite the number of authors that recommend the forensic investigation of software failures, none of them has so far proposed a complete process that addresses the above challenges. Before proposing a potential solution to fill this gap, earlier work conducted on the forensic investigation of software failures is reviewed next.

4.3 Previous work on the forensic investigation of software failures

This section reviews publications on the forensic investigation of software failures. A review of these solutions lays the foundation for future development of a suitable forensic process for this purpose. Knowledge about previous work allows the researcher to incorporate and possibly improve the solutions suggested by previous researchers. It also facilitates the identification of the areas of this process design that have been overlooked and therefore need further investigation.

The literature review on the forensic investigation of software failures indicates that research on this topic has been conducted under two main disciplines: operational forensics and forensic software engineering. These disciplines are reviewed in Section 4.3.1 and 4.3.2 respectively.

4.3.1 Previous work on operational forensics

The term operational forensics was coined by Michael Corby in 2000 (Corby, 2000a). Corby defines operational forensics as “the application of computer forensic techniques to identify occurrences and underlying causes of observed computer based events”. Besides Corby, only one other author, Barry Hood (2010), conducted research on operational forensics. Previous work from Corby and from Hood is presented in Sections 4.3.1.1 and 4.3.1.2 respectively.

4.3.1.1 Previous work from Corby

As its definition suggests, operational forensics uses digital forensic techniques to analyse the cause of an event. According to Corby (2000b), digital forensics can be categorised into two branches based on the goal of the investigation: prosecutorial forensics and prosecutorial forensics. Traditionally, digital forensics has been prosecutorial by nature with the objective of collecting evidence for prosecution or disciplinary action. By contrast, the main goal of operational forensics is to gather evidence for system correction and improvement (Hodd, 2010). Unlike prosecutorial forensics, which only deals with computer crimes and security incidents, operational forensics handles any kind of computer event. It can be argued that in prosecutorial forensics, the stress is placed on the legal aspect of the investigation, while in operational forensics the emphasis is on the scientific approach of the analysis. The author of this thesis has summarised the distinctions between the two branches of digital forensics in Figure 4.2.

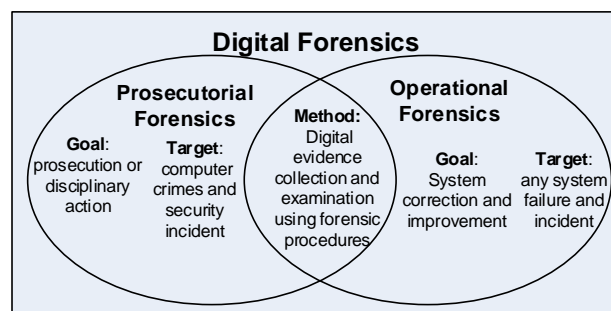


Figure 4.2: Relationship between prosecutorial forensics and operational forensics

Figure 4.2 shows the differences between the two branches and highlights their commonality, which is the collection and examination of digital evidence using forensic procedures.

The first publication on the forensic investigation of software failures is from Corby in 2000 (Corby, 2000a) under the term operational forensics. Corby's first paper defines the purpose and scope of this new discipline. It specifies that operational forensics is used to prevent future adverse system events, to define system performance benchmarks and to improve quality of service. It also makes suggestions on how to configure systems in a LAN environment to facilitate the collection of digital evidence following an incident. To this effect, Corby proposes a checklist of parameters and methods to maximise evidence collection (e.g. restricting network access points to active computers only, and preventing users from changing prescribed desktop settings).

Following this initial paper, Corby published three book chapters on operational forensics in 2000, 2007 and 2011 (Corby, 2000b; Corby, 2007; Corby, 2011) respectively. All three chapters focus on the specification of guidelines and procedures to build a pre-incident operational forensic program. This program ensures that potential evidence is collected, preserved and admissible in court and that the collection process is quick and effective.

The operational forensic pre-incident program (also known as forensic readiness) is built around four elements: a policy that specifies evidence retention as the first priority following an incident; guidelines on how to respond to an incident to preserve potential evidence (e.g. taking a photograph of the screen before rebooting the system); log procedures (e.g. activating system logs and keeping them safe on a protected device); and configuration planning (e.g. enabling disk mirroring for quick system recovery).

As can be seen from the above review, although operational forensics covers a complete forensic investigation, Corby's published work is mostly limited to how to prepare for potential evidence collection before a software failure occurs. Little information is provided on how to actually collect this evidence once a failure has occurred. It is worth noting that the second and third book chapters are mere updates on the first chapter with only a few changes. This attests to the limited progress that was made over the next decade.

4.3.1.2 Previous work from Hood

Hood, who references Corby's seminal work (Corby, 2000a), published two articles on operational forensics – both in 2010 (Hood, 2010a; Hood, 2010b). In both papers, he extends

the scope of operational forensics to include other forensic techniques deemed relevant for system improvement. Hence, the scope of the investigation is extended to physical, procedural, personnel and organisational areas. He subsequently investigates the use of modelling tools such as Petri Nets (Girault & Valk, 2003), which provide flexibility to cover all the above areas during an operational investigation. While Hood's papers focus on the formal modelling of a failure, they do not indicate the investigation process.

In addition to the limited work performed specifically on operational forensics, some other research has been conducted on the forensic investigation of software failures albeit under different names. Forensic software engineering is one such research area. It is reviewed in the next section (Johnson, 2002).

4.3.2 Review of previous work on forensic software engineering

The aim of forensic software engineering is to identify the systemic causes of a major software failure, including human issues (e.g. developer's fatigue or inadequate training), organisational issues (e.g. poor management leadership) and system engineering issues (e.g. poor communication between development teams) (Johnson, 2002). Therefore, it can be argued that while operational forensics focuses on the technical causes of the failure, forensic software engineering searches the "software engineering" causes of the failure, in other words the factors in the software development process that led or contributed to the malfunction (Hatton, 2004). This may include the development process or environment.

Like in the case of operational forensics, publications on forensic software engineering are few and remain on a conceptual level. Only two authors have so far published work specifically on this topic: Chris Johnson (Johnson, 2002) and Les Hatton (Hatton, 2004; Hatton, 2012). The Dependability Research Group at the University of Virginia, US, also did some research on this topic, which they refer to as *software forensics* (Dependability Research Group, 2007). The two previous researchers in this field (Johnson, 2002; Hatton, 2004) focus on the challenges that must be addressed by forensic software engineering, but they do not provide solutions. For instance, they focus on the difficulty in simulating conditions that have led to a failure (with a view to failure reconstruction), as well as on the lack of objective measures of software quality (with a view to identifying faulty software). They also do not consider the value of reliable evidence and how to obtain it.

4.3.3 Critical assessment of previous work on the forensic investigation of software failures

The above literature review clearly shows that limited work has been performed on the forensic investigation of software failures. No investigation process has been established and no forensic failure analysis method has been proposed. The field is essentially still at a conceptual level with no application in the industry. It is therefore the objective of this research to build on the limited work performed by previous researchers to further develop the field through the design of a forensic investigation model that can fill this gap. This model is aimed at addressing all the challenges discussed in Section 4.2. The first component of this model is a suitable end-to-end forensic investigation process that takes into consideration the specificities of failure analysis discussed previously.

To this effect, two significant contributions from previous research are used in the current thesis. They are the pre-incident operational forensic program and the two-phase approach of evidence collection, both proposed by Corby (2007) due to their relevance for this research. A pre-incident program is a pre-requisite for a successful forensic investigation as it ensures that when a problem occurs, information that can be used as evidence during the investigation is readily available and the responsible parties know how to collect it in a forensically sound manner. A forensic capability therefore needs to be created in the organisation before a forensic investigation can be conducted.

The organisation must be “forensic ready” by taking the following actions: (a) equip personnel with necessary forensic skills; (b) identify, acquire and maintain potential evidence such as log files and system-monitoring reports, and (c) develop supportive policies and procedures (Corby, 2000a; Kent et al., 2006). The organisation must also ensure that all system documentation is available and up to date. This includes system specifications, user manuals, licensing information, test plans, and a history of changes and reported incidents (Trigg & Doulis, 2008).

Once such forensic capability has been established, a forensic investigation can be conducted following a major software failure. The next section presents the investigation process proposed for this purpose.

4.4 The forensic failure investigation process

This section presents the forensic process proposed to investigate a software failure. The proposed investigative process consists of four basic stages. The first two occur immediately after the failure has been detected: firstly, collect evidence and secondly, restore the system. The third and fourth phases are the evidence analysis and the countermeasures specifications. They are conducted once the system has been restored. Phases 1, 3 and 4 are part of a standard digital forensic investigation, while Phase 2 is a troubleshooting task. These phases are described in Section 4.4.1 up to Section 4.4.4 respectively.

4.4.1 Phase 1: Evidence collection

This phase corresponds with the acquisition phase of a digital forensic investigation, as was described in Chapter 3. Shortly after a failure has been detected, all information that can assist in the investigation needs to be collected in a forensically sound manner by maintaining the chain of custody and preserving its integrity. The steps of the acquisition process that are applicable to this phase include securing and documenting the scene of the failure, followed by collecting, preserving and packaging the evidence.

Since the data may be acquired on a running system, the acquisition process is typical of a live forensic acquisition procedure (Grobler & von Solms, 2009). Therefore, in order to follow forensic best practice, two copies of the original data are made as the original data is released to be made operational again. So, the original data is imaged to a forensically sound storage device, and another copy of this image is made to another trusted device to be later used for analysis. The first image becomes the original data that is used to verify the integrity of the second image. A hash value of the original data should be made before it is released back into operation. Then a hash value of the first image and of the second image is made to be compared against the hash value of the original data throughout the investigation.

For the purpose of failure investigation, the evidence to be collected is classified as either primary or secondary. The primary evidence is the electronic data about recent system activity (e.g. event logs, network and system-monitoring reports). It is acquired while the system is still running, unless of course the failure caused it to shut down. It can also be collected from an external logging location, such as a syslog server. The secondary evidence corresponds with

the data obtained from documenting the scene of the failure (e.g. screenshot of the error message, interviews with the system administrator and the users who reported the failure), as well as system documentation as specified earlier in the pre-incident program. This documentation helps the investigator to understand the system's normal functioning and the circumstances of the failure.

It is important to note that, contrary to a digital forensic investigation, the failed computing device is not seized as physical evidence for analysis in a forensic laboratory. Instead, it is left on the scene to be fixed as quickly as possible during the restoration phase, which is described next.

4.4.2 Phase 2: System restoration

Once all the evidence has been acquired, the failure is fixed and the system is restored to its operational state as quickly as possible. Restoring the system does not intent to properly fix the problem but to quickly find some temporary solution using the troubleshooting approach. A restoration might be as simple as rebooting the system or it might necessitate some preliminary diagnostic of the failure to fix it. This will follow a typical troubleshooting process, which requires a recreation of the problem to isolate its cause (Trigg & Doulis, 2008). This system restoration is a temporary solution with temporary countermeasures (e.g. applying a software patch) until the root cause of the failure is identified in the subsequent analysis phase.

4.4.3 Phase 3: Root-cause analysis

Phase 3 corresponds with the analysis phase of a digital forensic investigation. The primary evidence collected in the first phase of the forensic failure investigation process is examined in a digital forensic laboratory in conjunction with the secondary evidence to identify the root cause of failure. Digital forensic and other scientific techniques are used to analyse the digital evidence. The investigation follows the scientific method. In the case of the responsibility for the software failure being attributed to a criminal or malicious intent, the investigation becomes a standard digital forensic case to identify and prosecute the perpetrator.

4.4.4 Phase 4: Countermeasures specifications

This is the reporting phase of a digital forensic investigation. It is a crucial segment of the forensic investigation of a software failure as it determines how to ensure that the failure does not reoccur. The conclusions reached in the previous phase on the basis of the examined evidence are documented and presented to the relevant parties along with recommendations for improvements. The timeframe for implementing the recommended changes should also be provided.

Ideally, required changes should be implemented immediately, but this is not always possible due to financial constraints. In this case, the suitability and durability of the temporary solution provided in Phase 2 should be determined. The system operations should be discontinued as soon as the temporary solution is no longer sufficient and until the specified corrections have been implemented to prevent a similar failure.

The complete investigation process is represented in the flowchart in Figure 4.2.

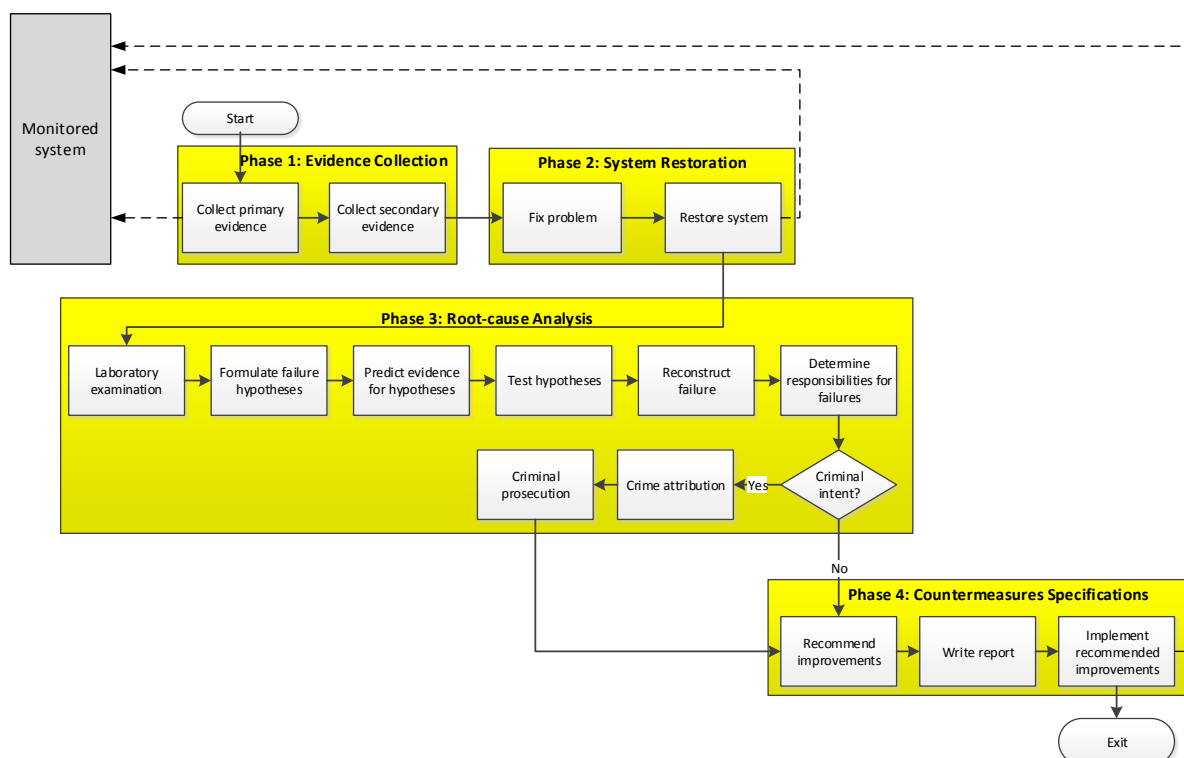


Figure 4.3: The adapted digital forensic process for software failures

The above process is generic as it does not indicate specific techniques to examine the collected evidence but rather steps that can lead to a thorough investigation of all possible causes of the failure and how to choose the most plausible one. The selection of the data analysis techniques will be based on the characteristics of the evidence collected (e.g. volume and format).

The main originality of this process is the inclusion of a System Restoration phase between the Evidence Collection and the Root-cause Analysis phases, which is not the case with a standard digital forensic investigation. Another key novelty is the Countermeasures Specifications phase, which forces the adoption of a more permanent solution than the one usually provided in the restoration phase. Other novel elements of this process include the systematic collection of secondary evidence in addition to the primary evidence, as well as the continuous system monitoring. As knowledge is gained about the software failures, information to be monitored is adjusted to be more relevant for the evidence analysis.

The benefits of using the digital forensic methodology for software failure investigations were discussed in Chapter 3. The advantages of adhering to the proposed process are demonstrated in the next section, based on the example of the Therac-25 disaster.

4.5 Application of the forensic failure investigation process – Case study of Therac-25 accidents

This section illustrates the application of the proposed investigative process in a real-life scenario. The example used is the infamous disaster of the Therac-25 radiation therapy machine, which was studied in detail in Chapter 2. Poorly designed software that was used to administer radiation treatment to cancer patients in the 1980s caused a series of six overdoses of radiation, which caused the death of three patients and serious injury to the remaining three (Leveson & Turner, 2002). Unlike some more recent software failures, a comprehensive report was compiled on the various accidents and investigations resulting from this disaster – hence, its selection as case study for the research in hand.

Section 4.5 demonstrates how the proposed process could have been used for the investigation using the first two radiation accidents as examples. For each accident, a description of the event and how it was handled by means of troubleshooting is provided first. The section then explains

how it could have been investigated with the forensic procedures of the proposed process. The first accident is presented and discussed in Section 4.5.1 and the second in Section 4.5.2.

4.5.1 Investigation of first Therac-25 accident

The first Therac-25 accident occurred at the Kennestone Oncology Center in the USA on 3 June 1985. The machine did not show any sign of unusual activity and did not generate an error message. However, the patient felt a high heat sensation after receiving treatment and accused the machine's operator of having burnt her. Shortly after returning home, the patient's skin reddened and swelled and she was in great pain. This was initially attributed to her disease. Weeks later, the patient's breast was removed, and her shoulder and arm were paralysed due to obvious radiation burn, but the doctors could not explain its cause. It was later estimated that 15 000 to 20 000 rads had been administered instead of the set 200 rads.

4.5.1.1 What was done in respect of troubleshooting?

No investigation was conducted for this accident as there was no information to indicate the machine had been responsible for the patient's condition.

4.5.1.2 What steps could have been taken with the proposed forensic investigation process?

The proposed failure investigation process would not have yielded any result either, as no primary or secondary evidence was available. Indeed, the system was not forensic ready as the logs were not activated due to memory constraints. There was no system documentation available and no previous case had been reported. What could have been done (but was not done), however, was to interview the patient and the machine operator and to file a report on the incident for future reference.

4.5.2 Investigation of second Therac-25 accident

The second Therac-25 accident occurred at the Ontario Cancer Foundation in Canada on 26 July 1985. The machine paused after five seconds of activation and displayed the following messages: HTILT, NO DOSE and TREATMENT PAUSE. As the machine indicated that no radiation had been administered, the operator retried four (4) times until the machine stopped. The patient complained of a burning electric sensation after the treatment. On 30 July, she was hospitalised as her skin was swollen and burnt and the machine was put out of service. She

died on 3 November 1985 from cancer but the autopsy revealed that the radiation burn would have necessitated a complete hip replacement had she survived. It was later estimated that she had received 13 000 to 17 000 rads.

4.5.2.1 What was done in respect of troubleshooting?

No information was collected. The machine was reset by the hospital's technician who did not find anything wrong. However, operation of the machine was discontinued five (5) days later, after the patient had been hospitalised.

In order to identify the cause of the problem, AECL first tried to recreate it with no success. They suspected a mechanical failure and hardwired its error conditions. They found some hardware design flaws and fixed them. They also modified the software to better control the positioning of the radiation beam. Based on these changes, AECL claimed a significant improvement of the machine, although they concluded that they could not ascertain the exact cause of the accident. The machine was put back into operation despite this uncertainty.

4.5.2.2 What steps could have been taken with the forensic investigation process?

4.5.2.2.1 Phase 1: Evidence Collection

- Collect primary evidence: No log files, but record error messages.
- Collect secondary evidence: No system specification and test plans, but obtain user manual and case history. Also interview the machine's operator and the patient.

4.5.2.2.2 Phase 2: System Restoration

- First reset the machine so that it can resume working.
- Discontinue usage of the machine as soon as the patient starts developing skin reddening and swelling after the treatment.
- Only put the machine back into service once the investigation has been completed and the implemented improvements have been tested.

4.5.2.2.3 Phase 3: Root-cause Analysis

Laboratory examination of collected data

- User manual: The user manual's description of many error messages was cryptic. The meaning of HTILT was unclear. NO DOSE indicates that no dose of radiation has been delivered.

- Report of first accident: Based on the two patients' testimony and symptoms, a correlation could have been established between the two events.

Formulation of hypotheses: three possible scenarios

- Electrical problem since patients experienced electrical shock.
- Hardware failure.
- Software error since the software controlled the machine.

Prediction and production of evidence to support hypotheses

- Some faulty wire, plug or internal electrical circuit, was expected in the case of the electrical problem. The electrical shock theory was ruled out after a thorough inspection by an independent engineering company that did not find any electrical problem in the machine.
- In terms of the hardware failure, some design flaw was expected or an incorrect positioning of the beam. AECL's test identified some hardware design flaws, which supported the hardware failure theory.
- Some logic errors in the code were expected to point to a software failure. AECL identified some weaknesses in the software, supportive of the software error theory.

Testing of the hypotheses

- Thorough testing of the improved machine after correction of the mechanical flaws would unfortunately not have prevented another overdose, as other accidents followed the second one despite this improvement. However, timeous testing of this hypothesis would have ruled out the mechanical failure theory and alerted the investigator to look for possible other causes.
- The only theory that remained involved a software error. Further examination of the software would be necessary to identify the bugs responsible for the failure.

The remaining steps of the Root-Cause Analysis phase (failure reconstruction and responsibility for failures) depend on the results of the thorough examination of the software to identify the bugs. The outcome of Phase 4 also depends on these results. As these results were obtained following the sixth and last Therac-25 accident and were explained in detail in Chapter 2, they are not covered in this case study.

The review of the above two accidents of the Therac-25 highlights the value of both a forensic readiness program and the failure investigation process. A well-established forensic readiness program would have ensured that appropriate steps were in place to collect evidence of the accidents. The failure investigation process would have ensured that the evidence was analysed effectively to identify the real source of the accidents. The next section examines in detail the benefits and limits of this process.

4.6 Critical assessment of the failure investigation process

This section provides a critical assessment of the proposed failure investigation process to identify areas that require improvement. The advantages of the process are presented in Section 4.6.1 and its limits in Section 4.6.2.

4.6.1 Advantages of the forensic failure investigation process

As the above case study demonstrates, the forensic failure investigation process offers many advantages over the troubleshooting method used in the case of the Therac-25. It could have located the source of the problem as a software error and not a hardware failure as suspected by AECL. More importantly, the proposed process demonstrates the need to minimise system downtime and to continuously monitor the system.

In essence, a forensic investigation based on this process would have provided more accurate results due to the following advantages of the process:

- Firstly, it would have ensured that the results of the investigation were reliable as they were based on objective scientific analysis.
- Secondly, it would have ensured that the root cause rather than a proximate cause for the failure was identified. Appropriate countermeasures could then have been implemented before the machine was restored to operation. This would have prevented further accidents.
- Thirdly, the failure process would have helped to improve the quality of the machine and AECL's procedures for failure analysis. AECL had no forensic capability and no standard mechanism to follow up on reported incidents, and valuable system documentation was missing, including software design specifications and a test plan.

It is quite easy to appreciate the value of a forensic pre-incident program in this case as it would have ensured that all relevant system documentation and logs were available. However, despite all the benefits indicated above, the proposed process also has some limitations that need to be addressed to make it viable. These limitations are presented next.

4.6.2 Limitations of the failure investigation process

As beneficial as this process is, it has the following limitations:

- It does not address the problem caused by the volatility of the data.
- The process is reactive as it awaits the occurrence of a failure. Learning from major operational failures implies that the high losses associated with these catastrophes should occur first. Preventing such events from happening altogether is therefore more desirable.

Addressing these—limitations requires continuous system monitoring, and looking for “operational markers” that indicate a potential failure. As explained in Chapter 1, such events are commonly known as near misses. Identifying near misses offers two main benefits: it can help prevent the failure from occurring, and it provides an opportunity to collect digital evidence of the potential failure before it is destroyed as part of the process of system restoration.

Indeed, it is generally agreed that near misses and related failures have common causes (Andriulo & Gnoni, 2014). Conducting a root-cause analysis of a near miss is therefore a valid method to identify the root cause of an impending failure. Detecting and investigating near misses is a well-established field that is successfully used to improve product reliability in a number of industries. It is however not yet in use in digital forensics. The detection of near misses is therefore included in the failure investigation process in order to obtain complete and relevant digital evidence of the failure. Near-miss analysis is a technique proposed for the evidence collection phase of this process and it is reviewed in the next chapter.

4.7 Conclusion

This chapter examined the challenges experienced in the successful application of digital forensics to the investigation of software failures. It reviewed previous work on this subject and proposed an adapted digital forensic process that is more suitable for addressing the

specified challenges. The process was applied to the case study of the Therac-25 accidents to demonstrate its benefits and its application in real life. However, it was found that this forensic failure investigation process did not meet all the challenges posed by the forensic investigation of software failures – more specifically, it did not deal successfully with the volatility of the digital evidence. If the digital evidence were to be lost after a major failure, it would seriously limit the accuracy of the forensic investigation.

The detection and analysis of near misses as precursors to major failures was therefore identified as a promising solution to this challenge. However, although it is well established in many engineering disciplines, near-miss analysis is new to digital forensics. For this reason, near-miss analysis, as it exists in the engineering disciplines, is presented in Chapter 5. The researcher's suggestion on how to apply near-miss analysis to digital forensics is discussed in Chapter 6.

CHAPTER 5

NEAR-MISS ANALYSIS: AN OVERVIEW

5.1 Introduction

The previous chapter presented the main challenges to using digital forensics for accurate failure analysis. It then proposed an adapted digital forensic investigation process that was designed to address these challenges. This forensic process for the investigation of software failures was limited to solving only two of the four challenges identified, namely the need to minimise system downtime following a failure and the need for continuous system monitoring. Chapter 5 now presents the proposed solution for the third challenge, while solutions for the remaining challenge are described in Chapter 10, as a result of the prototype implementation. The challenge addressed in this chapter, which is considered the biggest obstacle to the accurate forensic investigation of software failures, is the volatility of the digital evidence. The solution that is proposed to address this issue involves the use of near-miss analysis in the evidence collection phase of the failure investigation process.

Near misses, as immediate precursors to major failures, can address the above issue by providing complete and relevant evidence of the failure before it unfolds. The near-miss concept was defined in Chapter 1 and various examples of near misses in several industries were provided. Chapter 1 also indicated that near-miss analysis, which is an accident investigation technique used in a number of high-risk industries, is not yet formally in use in the software industry and not yet applied to digital forensics. This chapter therefore provides an overview of near-miss analysis and examines challenges to its application in investigating software failures. Proposed solutions to these challenges are developed in the next chapter.

The chapter is structured as follows: Section 5.2 provides some background information on near-miss analysis. Section 5.3 presents several arguments to motivate the suggestion to use near-miss analysis for the forensic investigation of software failures. Section 5.4 discusses

challenges to near-miss analysis, while Section 5.5 reviews previous work aimed at addressing such challenges and assesses the suitability of this previous work for software systems.

5.2 Background on near-miss analysis

This section provides background information on near-miss analysis. It starts with a description and a brief history of the field of near-miss analysis. It then presents the current applications, tools and techniques for near-miss analysis.

5.2.1 Overview of near-miss analysis

5.2.1.1 What is near-miss analysis?

As near misses are a special type of accident precursor, near-miss analysis is a specialised area of the broader field of accident precursor analysis. The American National Aeronautics and Space Administration (NASA), which was the first institution to formally investigate accident sequence precursors, defines Accident Precursor Analysis as “the process by which an organization evaluates observed anomalies and determines if the mechanism at the origin of that anomaly could recur with more severe results” (NASA, 2006). This definition refers to accident precursors as “observed anomalies”, which is not suitable for the software industry, since, as discussed in Chapter 1, accident precursors and near misses in software systems may not be visible at all in the absence of a failure.

No standard definition for near-miss analysis, which is also referred to as near-miss management, is available in the literature. Authors on this topic usually focus on defining the near-miss concept for the specific purpose of their research and in line with their particular industry, but they do not provide a definition for near-miss analysis. Near-miss analysis often refers to the process of identifying near misses and determining their root cause with a view to preventing and predicting accidents (Phimister et al., 2004).

Indeed, various accident investigations have revealed that almost all major accidents were preceded by a number of minor accidents and an even higher number of near misses as precursors (Mürmann & Oktem, 2002; Oktem, 2013). This is shown in Figure 5.1 in the popular safety pyramid (Bird & Germain, 1996). Recognising and handling these signals before an accident occurs has the potential to prevent an accident sequence from unfolding (Saleh et al.,

2013) and to improve safety by providing valuable information about potential accidents (Phimister et al., 2004).

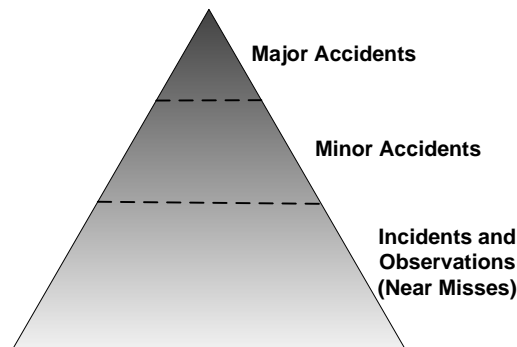


Figure 5.1: The Safety Pyramid, adapted from Bird and Germain (1996)

Based on a review of the literature, the researcher proposes the following definition of near-miss analysis with regard to software systems: *the root-cause analysis of near misses to prevent major software failures and understand their underlying causes.*

5.2.1.2 Why near-miss analysis?

Near-miss analysis is based on the observation that near misses and accidents have common causes but different outcomes (Andriulo & Gnoni, 2014). This is due to the fact that a near miss is an immediate precursor to the impending accident. It is literally one step away from an accident. Therefore, an accident and a related near miss have the same sequence of leading events – the only difference is that in the case of a near miss, the sequence was interrupted just before the accident occurred.

Due to the interruption in the accident sequence, near misses either result in no loss or the loss incurred is minimal, contrary to what happens in the case of accidents. Identifying the cause of a near miss is therefore a valid method of identifying the cause of the ensuing accident. It has the additional benefit that learning about the accident is conducted without first incurring the loss caused by an accident. Besides, if properly recorded, the data pertaining to the accident sequence is available and complete since it has not yet been affected by the potential accident.

5.2.2 Tools and techniques used in near-miss analysis

As a safety enhancement tool, near-miss analysis is often a component of a near-miss management program that is integrated with other safety management systems in an

organisation. Besides the identification and analysis of near misses, a near-miss management program also includes activities for disseminating information about near misses to decision makers and for implementing countermeasures (Phimister et al., 2000). This process varies from one industry or organisation to the next. It is often done manually but can be automated through an electronic system commonly referred to as a near-miss management system (NMS). Section 5.2.3.1 gives a brief overview of a typical NMS, while Section 5.2.3.2 to 5.2.3.4 reviews techniques used in NMSs to perform near-miss analysis.

5.2.2.1 Overview of near-miss management systems

Near-miss management system is an umbrella term used to refer to software systems used to record, analyse and track near misses (Oktem, 2002). They are sometimes referred to as near-miss systems. An effective NMS aims to quickly recognize near misses from the business operations in order to apply prevention measures (Gnoni et al., 2013).

In order to be effective, an ideal NMS is required to perform all activities of a near-miss management program. These activities are summarised in the following seven phases (Mürmann & Oktem, 2002; Phimister et al., 2004):

1. Identification (recognition) of a near miss
2. Disclosure (reporting) of the identified near miss to the relevant people
3. Distribution of the information to decision makers
4. Root-cause analysis (RCA) of the near miss
5. Solution identification (remedial actions)
6. Dissemination of actions to the implementers
7. Resolution of all open actions and completion of reports

The above seven-stage process is illustrated in Figure 5.2.

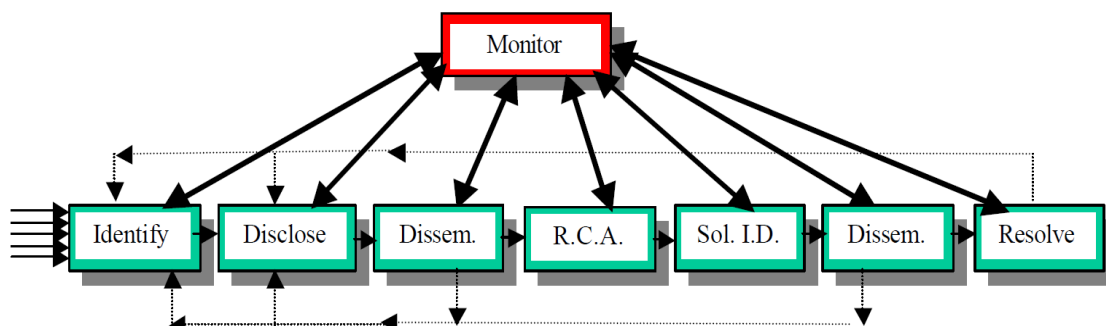


Figure 5.2: Near-miss management process (Phimister et al., 2000)

There are essentially two types of NMSs: single or dual. A single NMS only handles near misses, while a dual NMS handles both near misses and accidents (Phimister et al., 2000). A review of the literature on the design of industry-specific NMSs indicates that most NMSs are single. Significant research has been conducted on the design of effective single NMSs (Wu et al., 2010; Gnoni et al., 2013; Andriulo & Gnoni, 2014; Goode et al., 2014), especially in the healthcare industry for improved patient safety (Barach & Small, 2000; Callum, Kaplan, Merkley, Pinkerton, Rabin-Fastman, Romans, Coovadia & Reis, 2001; Aspden, Corrigan, Wolcott & Erickson, 2004; Fried, 2009).

Single NMSs usually place a strong emphasis on the identification and disclosure (reporting) phases of the near-miss management process described above. As such, they are often called *near-miss reporting systems* and are sometimes limited to that functionality of an NMS (Murff, Byrne, Harris, France, Hedstrom & Dittus, 2005; Goode et al., 2014). Barach and Small (2000) provide a comprehensive list of proprietary near-miss reporting systems in various industries, which provide a user interface where users can enter various details about an observed near miss. Near-miss reporting systems are sometimes called *incident reporting systems* (Macrae, 2007).

Apart from proprietary “private” near-miss reporting systems, some commercial NMSs are publicly available on the market. Commercial NMSs are mostly industry-specific. Examples include AlmostME, a near-miss reporting system (Napochi, 2013) for the medical field, and Dynamic Risk Predictor Suite (Near-miss Management LLC, 2014), a comprehensive NMS designed for manufacturing facilities.

Although an ideal NMS would perform the 7 steps described in Figure 5.2, most importantly, an NMS focuses on and performs the following three tasks:

- Identification of near misses
- Selection and prioritisation of near misses for analysis
- Root-cause analysis of the selected near misses

Techniques used for these activities are described in the following sections.

5.2.2.2 Techniques for near-miss identification

The identification of near misses is often done manually by means of observation. Recognising an observed event or condition as a near miss requires a clear definition of what constitutes a near miss with various supporting examples. Organisations therefore spend considerable effort to formulate a simple and all-encompassing definition of near misses that is relevant for their respective business operations (Ritwik, 2002; Phimister et al., 2003). This definition can differ significantly from one industry to the next as was discussed in Chapter 1.

Some effort has also been made at the intelligent detection of near misses through the NMS by defining metrics to characterise and quantify near misses.

Much of the industrial work on automated near-miss detection is based on study reports from the US Nuclear Regulatory Commission (NRC) and involves the use of Bayesian statistics to determine the risk of a severe accident based on operational data of observed unsafe events (Belles et al., 2000). Examples of such events include the degradation of plant conditions and failures of safety equipment (Belles et al., 2000).

Significant research has also been conducted in other industries to find generic metrics or signs of an upcoming accident, such as equipment failure rates, or failures of system components (Leveson, 2015). Probabilistic risk analysis (PRA), a recurring suggestion, also consists of estimating the risk of failure of a complex system by breaking it down into its various components and determining potential failure sequences (Phimister et al., 2004).

More recent research has proposed the use of location tracking information and sensors for environment surveillance to detect near misses in dynamic and uncontrolled environments such as on construction sites (Wu et al., 2010). In all the above work, near misses are usually identified as those events that exceed a predefined level of severity.

5.2.2.3 Techniques for near-miss prioritisation

Various quantitative and qualitative approaches are used to prioritise near misses across industries. Quantitative analysis is reviewed in Section 5.5.1.1 and qualitative analysis in Section 5.5.1.2.

5.2.2.4 Quantitative analysis

On the quantitative side, the two main approaches used to prioritise near misses are risk-based classification and statistical analysis.

Risk-based classification of near misses

Risk-based classification ranks near misses based on the severity level of their potential consequences or their frequency.

Ritwik (2002) determined the severity of potential consequences with a risk decision matrix that assigns a weight to the near-miss “relevancy” or “learning” impact on determining the potential worst-case scenario. According to Kleindorfer et al (2012), the risk level of a near miss is proportional to the amount of time that the event caused the system to cross predefined safety and quality limits, in other words, the amount of time that the system was in an unsafe or low-performing state. This time measurement is used to determine the risk of profit losses by calculating the actual loss that would be incurred for that unsafe period of time.

Probabilistic risk analysis (PRA) can also be used to determine the risk level of a near miss. PRA involves estimating the risk of failure of a complex system by breaking it down into its various components and determining potential failure sequences (Vesely, 2011). Using near-miss data, PRA allows for the severity of potential accidents to be determined (Phimister et al., 2004).

The frequency of a near miss can be obtained from historical data on reported near misses and be used to determine trends in the occurrence of certain events (Phimister et al., 2004). For instance, in Greenwell, Knight & Strunk (2003), the increase in the number of reported near misses is used to indicate that the process is heading towards a shutdown or an accident.

Statistical analysis to prioritise near misses

Statistical analysis has also been used to study the significance of near misses. In Bier and Mosleh (1990) and in Johnson and Rasmuson (1996), Bayes statistics is proposed to estimate the frequency of severe accidents based on the frequency of observed near misses. Bier (1993), on the other hand, provides a critical review of previous statistical methods developed for this purpose in the nuclear industry. Cooke and Goossens (1990) propose changes to the

methodology developed for nuclear power plants to make it suitable for the chemical process industry. Various factors such as the existence of initiating events and the probability of successful recovery are examined to classify near misses. In the finance industry, regression analysis is used to estimate the loss distribution of a near miss – hence the likelihood of a failure and its losses within a specific timeframe –so as to assess its level of severity (Mürmann & Oktem, 2002).

5.2.2.5 Qualitative analysis

Qualitative methods are also used to classify near misses based on their closeness to a potential accident. Some of these methods include the simulation of potential accidents and modelling of new accident scenarios to identify factors and system elements that are most likely to contribute to the occurrence or avoidance of an actual accident (March, Sproull & Tamuz, 2011). Case studies of how this is implemented in the process and transportation industries are provided in Van der Schaaf (1991). Another qualitative approach to understand the significance of a near miss is the Delphi method. This method is a group decision-making tool by means of which information on the probability of an accident can be gained from a panel of experts (Linstone & Turoff, 2002).

5.2.2.6 Techniques for near-miss root-cause analysis

Causal analysis of near misses can be performed with investigation techniques taken from engineering disciplines, such as fishbone diagrams, event and causal factor diagrams, event tree analysis, fault tree analysis, failure mode and effects analysis (Phimister et al., 2003; Jucan, 2005; Hecht, 2007; RealityCharting, 2013). The investigation consists of answering a series of questions that give insight into the factors that led to the near miss, the possible adverse consequences of the near-miss, and the factors that prevented or limited those consequences. The investigation can be assisted by various tools such as a comparative timeline to organise data and various matrices such as the missed-opportunity matrix and the barrier-analysis matrix (Corcoran, 2004).

Statistical analysis has also been proposed for learning from near misses. Some examples are using estimation techniques, simulations and regression analysis (Mürmann & Oktem, 2002). Historical near-miss data can be used to estimate the loss distribution, i.e. the likelihood of a failure and its losses within a specific timeframe. Regression analysis can help determine

exacerbating factors such as the frequency of certain operations. This information can then be used for simulating possible accident scenarios (Mürmann & Oktem, 2002).

As valuable as the above root-cause analysis techniques are, they do not adhere to forensic principles. Thus they are not suitable for the forensic investigation of software failures. To this end, it is suggested that the methods and techniques of digital forensics be used as specified in the failure investigation process to analyse near misses – the same way as software failures will be analysed.

Although new to digital forensics, the value of analysing near misses has been recognised in various engineering disciplines, and near-miss analysis has been conducted for over three decades. This is shown in the next section, which provides a brief overview of the history of near-miss analysis.

5.2.3 History of near-miss analysis

Although the learning opportunity offered by the analysis of near misses is not used in the IT industry, its application to the investigation of accidents in other industries is not a new concept at all. The emergence of formal near-miss analysis as currently conducted in a number of scientific disciplines can be traced back to the mid-1970s in the United States. This section reviews the origin and evolution of near-miss analysis. As a near miss is a type of ASP, the history of near-miss analysis is intertwined with the history of ASP analysis.

The analysis of ASPs emerged from the need to improve safety in industries that are prone to catastrophic industrial accidents. As is the case with many safety improvement initiatives, ASP analysis was initiated formally following tragic events – in particular two accidents that occurred in the United States in the 1970s (NASA, 2006). In both cases, the investigation that followed found some leading events and conditions that could have prevented the disasters had they been identified timely and handled appropriately (Phimister et al., 2004). Hence organisational programs, systems and methodologies were created to detect these precursors and learn from them. The events in question are the following:

- The crash of the American TWA Flight 514 that killed all 85 passengers and seven crew members in 1974 (NASA, 2006). Following the investigation of this accident, NASA

established the Aviation Safety Reporting System in 1976 to report and analyse observed ASPs in the aerospace industry (NASA, 2006).

- The nuclear accident at Three Mile Island that caused the release of toxic gas into the environment in March 1979 (Minarick, 1982). Shortly after that accident, the Nuclear Regulatory Commission (NRC) initiated an ASP program to identify, analyse and document ASPs (including near misses) (Phimister et al., 2004).

NASA and the NRC established quantitative and qualitative analysis techniques to determine the risk of a severe accident, based on operational data of observed unsafe events (Belles, Cletcher, Copinger, Dolan, Minarick, Muhlheim, O'Reilly, Weerakkody, & Hamzehee, 2000; Kirwan, Gibson & Hickling, 2007; NASA, 2011). Examples of such events included the degradation of plant conditions and failures of safety equipment (Belles et al., 2000).

The ASP methodology was subsequently adapted for use with other types of industrial accidents and adopted by the respective industries. The latter included the chemical industry (Ritwik, 2002; Phimister et al., 2003), the oil and gas industry (Cooke et al., 2011; Vinnem, Hestad, Kvaløy & Skogdalen, 2010; Skogdalen & Vinnem, 2011), the healthcare industry (Barach & Small, 2000; Sujana, 2012) and the finance industry (Mürmann & Oktem, 2002). To provide a complete history of ASP analysis falls beyond the scope of this research. However, a fairly comprehensive summary was written by Jones et al. (1999).

Nowadays, the analysis of ASPs and near misses has spread to a wide range of subjects. Saleh et al. (2013) indicate that over 58 000 articles listed in the Web of Science database have the term “precursor” in their title and this concept is used by around hundred different fields of science. A number of these articles also have the term “near miss” in the title or as a keyword. A keyword search for the term “near miss” in the Science Direct database results in over 83 000 articles.

In addition, two major research projects on near-miss analysis provide a significant number of papers on the topic. The first one is the Near Miss Project at the Risk Management and Decision Processes Center at the Wharton School, University of Pennsylvania, which has been ongoing since 2000 (Phimister et al., 2000). The researchers conducted more than 100 interviews in several plants in five Fortune 500 companies to assess the near-miss programs managed by

their Environmental, Health and Safety departments. The second project is the Accident Precursor Project which was conducted in 2003 by the US National Academy of Engineering. The report of the resulting workshop that extensively reviewed near-miss analysis across industries to promote cross-industry knowledge sharing is available in an online book (Phimister et al., 2004).

Additionally, several study reports have been produced by the ASP program of the NRC (Belles et al., 2000). NASA also published a handbook on precursor analysis in 2011 (NASA, 2011). Other research work has been published in workshop proceedings (Bier, 1998; Van der Schaaf, 1991).

Clearly, near-miss analysis is worth some attention in a variety of disciplines. The next section motivates the selection of near-miss analyses for the forensic investigation of failures in the software industry, more specifically in the adapted digital forensic process that was presented in Chapter 4.

5.3 Motivation for using near-miss analysis in failure investigation

This section presents arguments to support the suggestion to use near-miss analysis for the forensic investigation of software failures. Section 5.3.1 presents benefits of proactively investigating near misses compared to reactively investigating failures. Section 5.3.2 presents benefits of analysing near misses in comparison to earlier precursors. Section 5.3.3 presents cases of the successful application of a near-miss analysis in various industries.

5.3.1 Benefits of near miss-analysis over failure analysis

The two main reasons for suggesting the use of near-miss analysis to complement the failure investigation process are discussed below. The first reason was mentioned in Chapter 1 and the second reason was inferred from the review of major software failures in Chapter 2.

- Near-miss analysis provides an opportunity to proactively collect evidence of the failure before it actually unfolds. This limits the risk of having evidence destroyed due to the failure.

- In contrast to severe failures which can be scarce, near misses can be numerous, thus they offer ample opportunity to learn more from their richer data sets. More cases of near misses also provide more evidence of a particular weakness in a system.

Moreover, it is generally agreed that in many cases accident precursors can be analysed more effectively than can accidents – for the following reasons (Oktem et al., 2010; Ritwik, 2002):

- Investigation of a severe failure is costly and time consuming. Hence, financial constraints and resource limitations can severely limit the depth of the investigation. Near misses are also smaller in size and easier to deal with than serious accidents.
- Legal concerns may affect the investigation adversely. For instance, in product liability litigations, organisations may well withhold information that could penalise them.

Reporting near misses has also been a legislative recommendation in the European Union since 1997 under the Seveso II Directive (Seveso II, 1997). These events are to be reported in MARS (Major Accident Reporting System), the mandatory reporting system for major industrial accidents within the European Union (Jones et al., 1999). In its Annex VI, the Seveso II Directive (Seveso II, 1997:33) makes the following recommendation:

Accidents or “near misses” which Member States regard as being of particular technical interest for preventing major accidents and limiting their consequences (...) should be notified to the Commission.

5.3.2 Benefits of analysing near misses instead of earlier precursors

For the purpose of this research it is argued that investigating near misses is more valuable than investigating other early accident precursors for the following reasons:

- Various studies show that the number of precursors to an accident can be considerable (Borg, 2002; Bird & Germain, 2006). Selecting only the near misses reduces the number of precursors to be investigated, which can save resources required for the investigation.
- As a near miss is closer to the complete accident sequence, investigating a near miss provides the most complete pre-emptive evidence about the associated accident. It can therefore be used to identify the most accurate root cause of that accident.
- Early precursors can result in a number of false alarms, as they are further away from the unfolding of the accident. As they are closer to the accident, near misses provide

the highest level of confidence about the imminence of an accident, which can lead to the implementation of the most relevant countermeasures.

- Early precursors are generally events and conditions that have been observed in the past and are easily identifiable. Near misses, on the other hand, are not predefined as they can vary from one accident scenario to the other. They are therefore best suited to identify new failure modes and possibly prevent them.

5.3.3 Near-miss analysis success stories

Near-miss analysis is used to help improve the reliability of a system, product or process by reducing its risk exposure to a potential disaster. It has a successful track record in organisations where it has been effectively implemented. For instance, evidence shows that near-miss analysis contributed significantly to the improvement of safety in the aviation industry (Phimister et al., 2004). Other examples with measurable benefits are discussed below.

- Studies from Norsk Hydro, a Norwegian aluminium company, show that when near-miss analysis was introduced in the organisation in 1985, the number of near misses reported went from 0 to 1800 within 13 years. This resulted in a reduction of lost-time injuries by around 75% (Jones et al., 1999).
- In Canada, a petroleum company reduced injury by 80% over a year and by 100% over four years after implementing a near-miss reporting program in 1986 (Borg, 2002).
- In Malaysia, an oil company experienced a reduction in the monthly average cost of equipment-related accidents from \$675 000 to \$80 000 within a year after a near-miss reporting program was introduced in 1994 (Borg, 2002).

As beneficial as it is, near-miss analysis also has challenges that need to be addressed before its expected benefits can be reaped in the software industry. These challenges are discussed in the next section.

5.4 Challenges to near-miss analysis in the software industry

Across industries, the successful application of near-miss analysis faces three main challenges: (1) the detection of events and conditions that can be classified as near misses, (2) the high volume of near misses and (3) the root-cause analysis of near misses. A discussion of these challenges follows next.

5.4.1 Detection of near misses

Identifying near misses through observed physical events and conditions, as is done in many industries, is especially challenging in the software industry. Indeed, in the case of software applications, near misses might not even be visible as no system failure occurs and the events are virtual rather than physical. A near-miss might occur in the backend of the system (e.g. near exhaustion of memory) with no visible sign on the user interface. In the absence of specific near misses to refer to, providing a definition that clearly describes near misses in software systems is also a challenge.

An automated intelligent near-miss detection process is therefore required. However, although existing techniques can provide useful results, they are generally specific to the industry concerned and often require prior knowledge about near misses from historical data. Regrettably such data is not yet available in the software industry, where the concept of near miss is still largely unexplored.

5.4.2 High volume of near misses

Near misses can be frequent. In actual fact, they can be as much as 7-100 times more frequent than accidents (Aspden et al., 2004). In the hydrocarbon process industry, the accepted ratio of severe injury to near miss is between 15 and 25 (Ritwik, 2002). More impressively, an extensive study of industrial accidents conducted in 1969 indicates that a severe injury can have up to 600 near misses as precursors (Nichol, 2012). This is shown in the popular accident ratio triangle in Figure 5.3. A more recent study in 2003 suggests that this number could be even higher (Nichol, 2012).



Figure 5.3: Bird's accident ratio triangle, adapted from Nichol (2012)

A high volume of near misses is also expected in the software industry, as shown by reports of various major software failures. An example is the case of the Therac-25 disaster that was

discussed earlier, where up to forty near misses per day had been reported prior to the fatal accidents. This high volume of near misses can become unmanageable due to limited investigative resources. Therefore, it is necessary to select and prioritise near misses that are passed on for root-cause analysis.

Although they all have some merit, both the quantitative and qualitative approaches that are used to classify and prioritise near misses have disadvantages that limit their application to the software industry. For instance, the validity of the quantitative analysis techniques depends heavily on the risk threshold set for near misses. A high threshold may overlook significant events that were not anticipated, especially in new or immature software systems, while a low threshold will likely result in many false alarms (Phimister et al, 2004). Besides, generic metrics of near misses might not be applicable to all types of systems and all types of failures.

As the above review shows, some work is still required to detect, classify and prioritise near misses from a software system perspective. The researcher's vision on how this can be done is explained in the next chapter.

5.4.3 Root-cause analysis of near misses

As valuable as the available techniques for root-cause analysis of near misses are, they do not follow sound forensic principles and do not rely on sound digital evidence. Thus they are not suitable for the NMS proposed in this paper, which aims to apply the digital forensic methodology to analyse near misses in the same way that digital forensics is proposed to investigate software failures. The use of digital forensics to investigate near misses and software failures also faces specific challenges which were discussed in the previous chapter.

5.5 Conclusion

Chapter 5 presented near-miss analysis as a promising technique for dealing with the volatility of the digital evidence required to conduct a forensic investigation into software failures. As near-miss analysis is not yet used in digital forensics, the chapter emphasised its application and benefits in other industries to motivate its application in digital forensics. Challenges to near-miss analysis for such a purpose (i.e. the high volume of near misses and their difficult detection due to the fact that they are not easily observable in a software system) were also

presented. Chapter 6 next presents the proposed solution to address these challenges. The solution comprises a formal definition of a near miss suitable for the software industry, and a mathematical model to detect and prioritise near misses. The architecture of an NMS to automate this detection and prioritisation process is presented in Chapter 7.

CHAPTER 6

THE NEAR-MISS DETECTION AND PRIORITISATION MODEL

6.1 Introduction

The previous chapter presented near-miss analysis as a promising technique for dealing with the main challenge to the successful implementation of the failure investigation process designed in Chapter 5. This challenge was the potential loss of digital evidence following a system crash. As near miss-analysis is new to digital forensics, benefits of this field as used in other industries were presented to motivate the selection of this approach.

Near-miss analysis was proposed to proactively collect complete and relevant digital evidence of a potential failure *before the failure occurs*. However, challenges to reap the expected benefits of near miss-analysis were also identified, specifically the high volume of near misses (which can become unmanageable), and the limited visibility of near misses (which makes their detection challenging).

Chapter 7 presents the proposed solution to these problems, namely a mathematical model developed to detect and prioritise near misses as they occur on a running system. The design of the mathematical model is based on the review of previous work on near-miss analysis that was conducted in Chapter 6. This review identified two concepts used across industries that were deemed applicable to the software industry: using near-miss analysis to improve the reliability of a system and determining the risk level of a near miss based on its failure probability, in other words the probability that it will cause the system to fail.

Therefore, although this research does not use near-miss analysis to improve software reliability but rather to improve the accuracy of failure analysis, reliability concepts specific to the IT industry were used to develop methods to define, detect and prioritise near misses in software systems. These concepts are the following: the service level agreement (SLA), which

defines the contractually agreed level of reliability of a system; and the reliability theory of IT systems, which provides a formula to calculate the failure probability of a system.

The remainder of this chapter is organised as follows: Section 7.2 proposes a definition of the concept ‘near miss’ based on the SLA concept, which is suitable for the software industry. Section 7.2 subsequently provides a mathematical formula to express the definition in a formal manner and to detect near misses. Section 7.3 presents the reliability theory of IT systems as a suitable basis for prioritising near misses. The mathematical modelling for a near-miss failure probability based on this theory is presented in Section 7.4. Finally, Section 7.5 defines a prioritisation method for near misses according to the definition and failure probability formula as proposed in this chapter.

6.2 Formal definition of a Near Miss for software systems

In Chapter 2, a generic definition of a near miss with regard to software systems was proposed as follows:

A near miss is an unplanned high-risk event or system condition that could have caused a major software failure if it had not been interrupted either by chance or timely intervention.

It was decided that this generic definition needs to be fine-tuned and formalised to enable the detection of near misses.

With regard to software systems, a failure is “the inability of a system or component to perform its required functions within specified performance requirements” (IEEE, 1990). Since no system is immune to a malfunction, no system vendor can guarantee that the system will work perfectly and continuously at all time. In other words, some periods of unplanned downtime are expected. As an illustration, even the public switched telephone network (PSTN), the traditional telephone network that is considered to be the most reliable communications network, has some margin for failure with a designed reliability of 99.999% which translates to a margin of 5 min and 15 s of downtime in a year (Horton, 2008).

For the above reason, the performance requirements of a typical software system make provision for a downtime “allowance”. This allowed downtime can be indicated informally in

the system's specifications document, but it is usually specified formally in a contract between the service provider and the receiver of the service (customer). This contract is referred to as the service level agreement (SLA) (Sevcik, 2008).

SLA's are service management contracts that are processed by real-time monitoring and measuring of the provided service levels at runtime. SLA's specify mandatory service provisioning terms such as Quality of Service (QoS) attributes and functional service properties. They may also include several technical and business service level objectives (SLOs) with their metrics used for evaluation of the service level. Thus, SLAs assist with the calculation and measurement of service parameters, which in turn indicate the provider's adherence to the promised service levels (Stamou, Kantere, Morin, Longo & Bochicchio, 2013).

SLA's typically include the responsibilities of both the customer and the service provider in terms of the provision of the service. This includes customer's requirements to be met by the service provider, the fee paid if the requirements are met, the penalty incurred by the service provider if they are not satisfied, and the period of time the agreement holds. A critical QoS specified in the SLA is the customer's service availability (Das, 2012).

For instance, the SLA for a website may specify that the site will be operational and available to the customer at least 99.9% of the time in any calendar month. This indicates that the website should not be down for more than 0.1% of the time in a month. For a 30-day month, this corresponds to a downtime limit of 0.03 day or 43 min and 12 s. If the website is down for more than this amount of time in a month, it does not meet the customer's expectation in terms of the SLA. Thus it violates the SLA and is considered to have failed.

Therefore, for the purposes of the research in hand an event is considered a failure if its resulting downtime exceeds the downtime allowance specified in the SLA. Similarly, an event is considered a near miss if it can lead to the exceeding of that allowance. The researcher therefore proposes the following specific definition of a near-miss for the purpose of facilitating its detection:

A near miss is an unsafe event or condition that causes a downtime whose duration is close to exceeding the downtime allowance specified in the SLA.

Note that the SLA concept used in the above definition does not necessarily refer to a formal contract between the service provider and the customer. It is rather a concept that refers to any objective predefined performance level specified for a given system.

This definition of a near miss is illustrated by the earlier example of the SLA of the website that specifies a downtime allowance of 43 min and 12 s per month. This allowed downtime provides a means to classify an event as one of three possibilities:

- A failure: SLA violation; monthly downtime exceeds 43 min and 12 s.
- An acceptable failure: Not close to SLA violation; monthly downtime is significantly less than 43 min and 12 s. This downtime is part of the expected system behaviour.
- A near miss: Close to SLA violation; monthly downtime is close to or equals 43 min and 12 s, but not more than that.

Using the SLA as a measurable characteristic in the definition of a near miss provides a way of quantifying the severity of an unsafe condition and prioritising near misses. The SLA performance metric used for that purpose is the downtime (as opposed to expected availability), as it has a direct effect on a system's reliability. Indeed, reliability is often expressed in terms of the Mean Time Between Failures (MTBF), which is "the average operating time (uptime) between failures of a system" (MTL Instruments, 2010). As system failures result in downtime, the MTBF is the average operating time between periods of downtime.

The above concept is proposed as the basis to formally define a near miss. This requires determining how close the experienced downtime should be from exceeding the allowed downtime to be considered a near miss. Specifying a near-miss threshold is suggested for this purpose. This threshold will vary from one organisation to the next, depending on its risk tolerance. For instance, Organisation A might be comfortable with a 95% threshold (95% of the downtime allowed), while Organisation B will limit its risk tolerance level to 75%. This would correspond to a total monthly downtime of 41 min and 2 s for Organisation A and 32 min and 24 s for Organisation B. This threshold-based definition of a near miss can be mathematically expressed as follows:

$D_{\text{experienced}}$ is the experienced downtime

D_{allowed} is the SLA downtime allowance

α is the near-miss threshold in percentage; $\alpha < 1$

$\alpha \times D_{\text{allowed}}$ is the near-miss threshold in time value

If $\alpha \times D_{\text{allowed}} \leq D_{\text{experienced}} \leq D_{\text{allowed}}$ then $D_{\text{experienced}}$ is a near miss

Figure 7.1 illustrates the downtime-based classification of events explained above.



Figure 6.1: Classification of unsafe events based on their downtime duration

As an illustration: with reference to the earlier example of Organisation A, a near miss is any monthly downtime of between 41 min and 2 s (the 95% threshold) and 43 min and 12 s (the SLA downtime allowance). Thus:

$$\alpha = 0.95$$

$$D_{\text{allowed}} = 43 \text{ min and } 12 \text{ s}$$

$$\alpha \times D_{\text{allowed}} = 41 \text{ min and } 2 \text{ s}$$

$$\text{If } 41 \text{ min and } 2 \text{ s} \leq D_{\text{experienced}} \leq 43 \text{ min and } 12 \text{ s}$$

then $D_{\text{experienced}}$ is a near miss.

Note that the downtime allowed is not a static but rather a dynamic value as it diminishes with every downtime experienced previously in the same SLA measurement period. D_{allowed} is thus the limit left after experiencing previous downtimes, if any. This needs to be accounted for and changes the above formula as explained below.

For the purposes of the research in hand, the formal definition for a near miss is as follows:

Let T be the current measurement period, usually the current calendar month or the current year.

Let d be the current day, such as day number 10 of the current month or year.

T spans the period from the first day of the month or year until the current day. Thus:

$$T = [1; d]$$

Let n be the number of previous acceptable downtimes in T .

$$n \geq 0$$

D_{previous} is the total downtime experienced previously in T .

D_k is the duration of the downtime number k in T . Thus D_1 is the duration of the first downtime experienced and D_n is the duration of the latest downtime. D_0 indicates that no downtime occurred

previously.

$$D_{\text{previous}} = \sum_{k=0}^n D_k$$

D_{now} is the downtime experienced currently.

$D_{\text{remaining}}$ is the SLA downtime limit left after experiencing previous downtimes in T.

$$D_{\text{remaining}} = D_{\text{allowed}} - D_{\text{previous}}$$

$$\text{If } \alpha \times D_{\text{remaining}} \leq D_{\text{now}} \leq D_{\text{remaining}} \rightarrow \text{near miss} \quad (1)$$

Since the downtime allowed is decreasing over time with every new downtime experienced in T, the value of D_{now} should also reflect the impact of the previous downtimes experienced in T. This is achieved by using a weighted moving average (WMA) of all the previous downtimes in T to calculate D_{now} . A WMA gives more weight to the most recent data in a time series and attaches less importance to older data. It is therefore used for trend forecasting (Holt, 2004). This is particularly relevant for the research in hand, which aims to predict likely failures based on near misses. The WMA of the previous downtimes shows the trends in the system downtimes and can indicate whether the system is heading towards a large downtime (for instance, through a series of short downtimes).

The WMA of the downtimes is calculated by multiplying each downtime by its position in T and dividing the sum of these values by the total of the multipliers. Using the previously defined parameters of n (the number of previous acceptable downtimes in T) and D_k (the duration of the downtime at position k in T), the WMA for D_{now} is formally expressed as follows:

$$D_{\text{now}} = \frac{n(D_n) + (n-1)(D_{n-1}) + (n-2)(D_{n-2}) + \dots + 2(D_2) + 1(D_1)}{n + (n-1) + (n-2) + \dots + 2 + 1}$$

Equation (1) above enables the identification of near misses among periods of downtime. This identification is reactive, i.e. it is performed after the near miss caused a downtime. The identified near misses can then be logged so that they can be investigated at a later stage. This reactive detection still provides a valuable opportunity to learn about the system weakness that has been responsible for the downtime. However, ideally, near misses should be identified before they result in downtime so that a system outage may be prevented. A method for detecting near misses proactively is provided in Section 7.5.

Once near misses have been defined and recognised, they need to be prioritised for possible further investigation. The latter can be accomplished by employing methods from prioritisation schemes that have been developed for other industries. They were reviewed in the previous

chapter to evaluate their suitability for software systems. This review indicated that previous work in near-miss prioritisation is industry-specific and not much of it is applicable to the software industry, except for the concept of prioritising near misses based on their failure probability. An approach to determine this failure probability is provided by the reliability theory that will be reviewed in the next section. This theory provides a formula to calculate the failure probability of a system. It can thus serve as a basis to determine the risk level of a near-miss event, based on the likelihood of a system failure due to that event.

6.3 Overview of reliability theory and failure probability formula for IT systems

Near-miss analysis is commonly performed to improve a system's reliability. One common approach in this regard is to add redundancy. With the advance of cloud computing and virtualisation, redundancy is inherent in most enterprise systems. Therefore, while near misses can occur in any system, the proposed mathematical model only considers near misses in redundant systems. Redundancy is usually associated with hardware, but the argumentation is that its basic concepts and its corresponding reliability theory are transferable to software components. Examples of such components are memory slots, virtual servers, databases and data. The following is an overview of the concepts of redundancy and the failure probability formula as currently applied to hardware components.

6.3.1 The reliability theory of redundant hardware components

Redundant systems have a number of equivalent spare components that work in parallel. If one of the redundant components fails, it is removed from the system and another operational one takes over its functionality. This failover process allows the system to continue its operations (Highleyman, 2008). The recovery time, also known as the 'mean time to recover' (MTTR), is equal to the failover time for parallel components. It is the time it takes the system to detect and disable the failed node and transfer its operations to another node in working condition. If this failover time is short enough, users will not experience any interruption and will not even realise that a fault has occurred (Highleyman, 2008).

Based on the concept of redundancy, the author argues that a redundant system is at a high risk of failing when no spare resources are available to take over the operations of the active units,

if need be. At this point, the failure of any one unit will bring the entire system down. In other words, the failure of one active unit can cause a system outage only if all spares are already down. Stated in formal terms, for a redundant system with n number of units of which there is s number of spares, it takes the failure of $s+1$ units to bring the system down.

As an illustration, let us use the case of a redundant system constituted of five servers and two spares. Thus n is equal to 7 (the total number of servers) and s is equal to 2. The system requires five active servers at all time to be operational. If one server is lost, one of the spares takes over. The system keeps on running and is left with only one spare. If a second server is lost before recovering the failed one, the second spare takes over and the system is now left with five active servers and no spare. The system will keep on working as long as these five servers are working. If a third server is lost at this point, the system fails, since only four active servers are left. Thus, the entire system goes down when $s+1$ units (i.e. 3 units) fail.

The above implies that the risk of a system failure increases as the number of available spares decreases. It can thus be argued that in the context of redundant systems, near misses can be caused by the likely exhaustion of critical redundant resources. The failure probability of a redundant system can therefore be determined based on the number of spares lost as will be described next.

6.3.2 Failure probability formula for hardware components

As mentioned earlier, when all units are up and running, the failure probability for a redundant system with s number of spares is the probability of losing $s+1$ units. The reliability theory (Holenstein, Highleyman & Holenstein, 2003) demonstrates that the formula to calculate this probability is:

$$F = f(1-a)^{s+1}$$

This formula can be explained as follows:

F is the probability that the system will be down.

a is the probability that a unit will be up (its expected availability).

$(1-a)$ is the probability that a unit will be down.

s is the sparing level (i.e. $s+1$ units must fail in order for the system to fail).

f is the number of failure modes or the number of ways that $s+1$ unit failures will cause a system outage.

The formula assumes that the system can be restored to service as soon as a failed unit is repaired (parallel repair). The rationale is as follows: since there are s number of spares, it will take the failure of $s+1$ units to bring the system down. Since the probability of losing one unit is $(1-a)$, the probability of losing $s+1$ units is $(1-a)^{s+1}$. However, there are several ways in which $s+1$ units out of n units can fail. The number of failure modes f is equal to $C(n, s+1)$ (read “ n combination $s+1$ ” or “ n choose $s+1$ ”). Therefore, the probability that the system will fail is $f(1-a)^{s+1}$. Thus:

$$F = f(1-a)^{s+1} = C(n, s+1) \times (1-a)^{s+1} = \frac{n!}{(s+1)! (n-s-1)!} (1-a)^{s+1}$$

This formula is designed for redundant hardware resources that are usually identical in terms of functionality, electronics and hardware design. Therefore they all have the same failure probability (probability that the unit will be down). In addition, they are designed to be independent of each other, so that the failure of one unit does not affect the functioning of the other units. The above situation is not always applicable to software units and the above formula needs to be revised accordingly as will be described next.

6.3.3 Proposed failure probability formula for software components

In cases where software reliability is critical, the redundant software units are often designed to be identical only in terms of functionality to avoid common sources of vulnerability and identical failure modes. Therefore they do not have the same software and hardware design. The concept of design diversity is used to refer to this approach of designing independent software with similar functionality (Pullum, 2001).

Design diversity allows the independent creation of multiple versions of the software based on diverse but equivalent specifications (Pullum, 2001). The various versions of the software are written by independent programmers, with a different logic and possibly different languages and environments. Jain and Gupta (2011) conducted a survey of design diversity techniques for software redundancy.

Due to design diversity, redundant software units are likely not to have the same failure probability. Therefore the researcher revised the above formula as follows:

$$x = 1 \text{ to } s+1$$

a_x is the probability that unit x will be up (its expected availability).

$(1 - a_x)$ is the probability that unit x will be down.

The probability of losing $s+1$ units is $(1-a_1) \cdot (1-a_2) \cdot (1-a_3) \dots (1-a_s) (1-a_{s+1})$. This is equal to

$$\prod_{x=1}^{s+1} (1 - a_x)$$

Thus:

$$F = f \times \prod_{x=1}^{s+1} (1 - a_x) = C(n, s+1) \times \prod_{x=1}^{s+1} (1 - a_x)$$

$$F = \frac{n!}{(s+1)! (n-s-1)!} \prod_{x=1}^{s+1} (1 - a_x) \quad (2)$$

Equation (2) also caters for the case where all redundant units are identical with the same failure probability (e.g. with virtualisation and cloud computing). In that case the equation will turn out to equivalent to the original equation from the reliability theory of hardware components since $\prod_{x=1}^{s+1} (1 - a_x)$ will now be equal to $(1-a)^{s+1}$.

Equation (2) establishes the foundation for calculating the failure probability of a redundant system. However, it is limited in the sense that it caters only for the case when all spares are working. Hence it needs to be adapted to accommodate the case of a near miss, in other words the loss of a number of spares. The mathematical model for the near-miss failure probability is developed and discussed in the next section.

6.4 Mathematical modelling for near-miss failure probability

For the purpose of this research, and in the context of redundant systems, the failure probability of a near miss is the probability of a system failure, given the loss of any number of critical spare units. This is a conditional probability that can be expressed as $P(F|D)$ (probability of F given D), where D is the event that d number of spare resources are down. Thus $P(F|D1)$ is the probability of failure given that one (1) spare resource is down. Bayesian statistics (Devore & Berk, 2012) are generally used to calculate conditional probabilities using the following formula:

$$P(F | D) = \frac{P(F \cap D)}{P(D)} = \frac{P(D | F) \times P(F)}{P(D)}$$

Since the elements of this formula are not all available, it is preferable to use a logical deduction to determine $P(F|D)$. An appropriate starting point is the failure probability formula of the reliability theory presented in the previous section. As explained earlier, this formula enables

the calculation of the probability of failure of a redundant system when all the redundant units are up. Therefore, there is a need to determine how this formula is affected when one or more spare units are down. For simplicity's sake, this process is explained incrementally, starting with the loss of one spare, then two, and finally generalised to any number of spares.

6.4.1 Loss of one spare

$P(F|D0)$ is the probability of failure given that no (0) spare is down (i.e. when all s spares are up). The system fails when $s+1$ units are down. Thus, referring to Equation (2) in Section 7.3:

$$\begin{aligned} P(F | D0) &= F = f \prod_{x=1}^{s-1} (1 - a_x) \\ &= C(n, s + 1) \times \prod_{x=1}^{s-1} (1 - a_x) \\ P(F | D0) &= \frac{n!}{(s + 1)!(n - s - 1)!} \prod_{x=1}^{s-1} (1 - a_x) \end{aligned}$$

If one spare goes down, the system is left with $n-1$ units and $s-1$ spares. Following the same logic used to obtain the above equation, the system will fail if all spares plus 1 unit fail, thus if $(s-1)+1$ units fail. In other words, it now takes the failure of s units to bring the system down. Since the probability of losing any unit x is $(1 - a_x)$, the probability of losing s units is $(1-a_1)$. $(1-a_2)$. $(1-a_3)$... $(1-a_s)$. This is equal to $\prod_{x=1}^s (1 - a_x)$. The number of failure modes f_1 , which indicates the number of ways that s units can fail out of $n-1$ units, is equal to $C(n-1, s)$. Therefore, the probability that the system will go down is $f_1 \times \prod_{x=1}^s (1 - a_x)$ or $C(n-1, s) \times \prod_{x=1}^s (1 - a_x)$. Thus:

$$\begin{aligned} P(F | D1) &= f_1 \prod_{x=1}^s (1 - a_x) = C(n-1, s) \times \prod_{x=1}^s (1 - a_x) \\ P(F | D1) &= \frac{(n-1)!}{s!(n-1-s)!} \prod_{x=1}^s (1 - a_x) \end{aligned}$$

6.4.2 Loss of two spares

Similarly, if two spares go down, the system is left with $n-2$ resources and $s-2$ spares. The system will fail if $(s-2)+1$ units fail, thus if $s-1$ units fail. The probability of losing $s-1$ units is $\prod_{x=1}^{s-1} (1 - a_x)$ and the number of failure modes f_2 is equal to $C(n-2, s-1)$. Thus:

$$\begin{aligned} P(F | D2) &= f_2 \prod_{x=1}^{s-1} (1 - a_x) = C(n-2, s-1) \times \prod_{x=1}^{s-1} (1 - a_x) \\ P(F | D2) &= \frac{(n-2)!}{(s-1)!(n-2-(s-1))!} \prod_{x=1}^{s-1} (1 - a_x) \end{aligned}$$

$$P(F | D2) = \frac{(n-2)!}{(s-1)!(n-1-s)!} \prod_{x=1}^{s-1} (1-a_x)$$

6.4.3 Loss of any number of spares

Using the same argumentation as above, the formula is generalised for any d number of spares that go down. If d number of spares is lost, there are $n-d$ units and $s-d$ spares left. It thus takes the failure of $(s-d)+1$ units to bring the system down. Since the probability of losing any one unit x is $(1-a_x)$, the probability of losing $s-d+1$ units is $\prod_{x=1}^{s-d+1} (1-a_x)$. The number of failure modes f_d is thus $C(n-d, s-d+1)$. Hence:

$$\begin{aligned} P(F | Dd) &= (n-d, s-d+1) \times \prod_{x=1}^{s-d+1} (1-a_x) \\ &= \frac{(n-d)!}{(s-d+1)!(n-d-(s-d+1))!} \prod_{x=1}^{s-d+1} (1-a_x) \\ &= \frac{(n-d)!}{(s-d+1)!(n-s-1)!} \prod_{x=1}^{s-d+1} (1-a_x) \\ P(F | Dd) &= \frac{(n-d)!}{(s-d+1)!(n-1-s)!} \prod_{x=1}^{s-d+1} (1-a_x) \\ P(F | Dd) &= \frac{(n-d)!}{(s-d+1)!(n-1-s)!} \prod_{x=1}^{s-d+1} (1-a_x) \end{aligned} \quad (3)$$

Determining the failure probability of a system in production based on the number of spares lost, can thus provide a way to quantify the risk level of near misses. Such quantification can be used to prioritise the near misses for investigation.

6.4.4 Illustration of the failure probability formula

This section illustrates the application of the Equation (3) established above with the simple example of a system composed of several redundant servers. The near miss is the loss of spare servers. The formula is used to calculate the failure probability (F) and how it increases as spare servers go down. The results are then represented by means of a graph.

The example of a system with five servers and two spares is reused. For the sake of simplicity, it is assumed that all servers have the same failure probability. Assuming the failure probability of one server at any given time is 0.033 (3.3%), the following applies:

$$n = 5; s = 2; 1-a = 0.033$$

Equation (3) is used to calculate the system failure probability for various numbers of failed servers. Results are provided in Table 6.1 and graphically represented in Figure 6.2.

Table 6.1: Failure probability values

Number of failed servers	Failure probability (%)
0	0.035937
1	0.6534
2	9.9
3	100

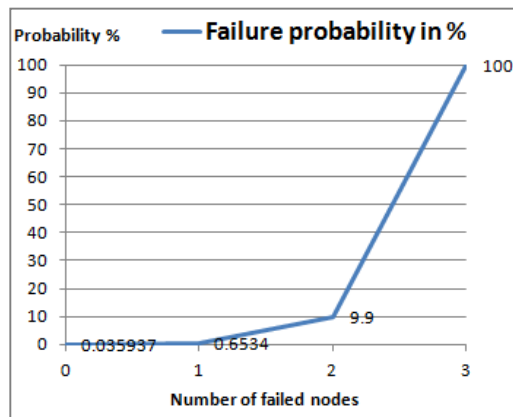


Figure 6.2: Failure probability graph

This graph enables a quick and easy visualisation of a near miss and can facilitate its prioritisation. For instance, the graph shows how the failure probability significantly increases when the second spare is lost, since no more spares are available.

However, the prioritisation of near misses based on the formula in Equation (3) has some limitations as it does not take into consideration the concept of the SLA used in the proposed definition of a near miss. Indeed, this formula determines the risk of a system outage following the loss of some spare resources. Recalling from the discussion in Section 6.2 that a failure is a breach of the SLA, a system outage is only a failure (from a business perspective) when the duration of the downtime exceeds the allowance specified in the SLA. Otherwise it is an acceptable failure, as depicted in Figure 6.1. The prioritisation scheme consequently needs to be refined to take into account the duration of the system downtime. This prioritisation method is established in the next section.

6.5 Prioritisation of near misses

This section presents the researcher's near-miss prioritisation method based on the SLA concept and formally expressed in a mathematical formula.

6.5.1 The near-miss prioritisation formula

A near miss was previously defined as a potential failure, more specifically as an event that can lead to the violation of the SLA. The violation of the SLA was also defined in terms of the system downtime. According to the same logic that was used to measure the severity of a failure based on the downtime experienced, the severity of a potential failure or near miss can be assessed based on its expected downtime. In other words, determining for how long the system will be down in the eventuality of an outage caused by this near-miss event.

To this effect, two parameters are needed: the failure probability of the near miss and the expected recovery time for the outage or MTTR (mean time to repair). The expected downtime is then calculated as the product of the failure probability and the MTTR. The failure probability is provided by Equation (3) established earlier and the MTTR can be obtained from the system vendor specifications or through historical observations. The system enters a "critical zone" when the expected downtime is greater than the SLA downtime allowance. This can be expressed as the following formula (Equation (4)):

D_{expected} is the expected downtime, thus the expected loss of productivity due to a failure.

$D_{\text{remaining}}$ is the SLA downtime limit remaining after previous downtimes in the current measurement period.

$P(F|Dd)$ is the probability of failure, given the current unsafe situation (loss of spares).

MTTR is the expected recovery time following an outage.

$$D_{\text{expected}} = P(F|Dd) \times \text{MTTR}$$

$$\text{If } D_{\text{expected}} > D_{\text{remaining}} \rightarrow \text{critical zone} \quad (4)$$

If a successful recovery is performed and the outage is prevented when the system is in the critical zone, then this event is classified as a near miss. Equation (4) can thus allow the proactive detection of a near miss, i.e. before it causes an outage. On the other hand, if the system recovery is not successful, the event is classified as a serious failure in the sense that

the SLA has been breached. Both cases need to be investigated to identify their root cause and prevent their reoccurrence.

In the case of a near miss, the closeness between the expected downtime and the SLA downtime allowance can be used to assign a risk level to the event. The risk level will determine how important it is to conduct a thorough forensic analysis of this near miss and how much of the limited resources available can be allocated to this task. However, as explained in Section 7.2, different organisations have different risk tolerance levels and may prefer a larger margin of safety when detecting a near miss. Instead of using the whole SLA downtime allowance to define a near miss, they may specify a portion of that downtime as their near-miss threshold. Their system will thus enter a critical zone earlier, which will give them more time for remedial action. The near-miss threshold can be adjusted over time as more experience is acquired in detecting and handling near misses. When this threshold is included, Equation (4) is adjusted as follows:

α is the near-miss threshold; $\alpha \leq 1$

$$\text{If } D_{\text{expected}} \geq \alpha \times D_{\text{remaining}} \rightarrow \text{critical zone} \quad (5)$$

6.5.2 Evidence collection for high-risk near misses

Only events in the critical zone are passed on for analysis. These events are prioritised based on the value of their expected downtime as per Equation (5) above. As soon as the system is in the critical zone, the following two actions need to be initiated by the system administrator:

- Automatic collection of data related to the event
- Recovery of the lost spares to prevent the outage

Event-related data will serve as digital evidence for the root-cause analysis of the near miss. When collecting the data, it is imperative to ensure that the process does not exhaust the remaining available resources, as this could harm the system and accelerate its crash. Ideally, event data should be collected before corrective actions are implemented so as to capture the unsafe state of the system and avoid any tampering with potential evidence. However, due to time constraints, this may not be possible. Some mitigating action can then be taken, such as logging all recovery steps to facilitate the reconstruction of the near-miss condition at a later stage.

The suggestion on how to automate the detection and classification process explained above is presented in Chapter 8 together with the design of an NMS that implements that process.

6.6 Conclusion

This chapter proposed a mathematical model specific to the software industry to detect and prioritise near misses at runtime so as to address the issue caused by their high volume and low visibility. Digital evidence of the near misses with the highest risk level was subsequently collected for root-cause analysis. The near-miss detection method was based on a mathematical formula created to define near misses using the SLA of the monitored system. The near-miss prioritisation process was based on a mathematical formula established to calculate the failure probability of a near miss. The proposed mathematical model was designed to be integrated with the Evidence Collection phase of the process designed for the forensic investigation of software failures in Chapter 5. The architecture of an NMS that combines this investigation process with the near-miss detection and prioritisation model is presented in Chapter 8.

CHAPTER 7

THE NMS ARCHITECTURE

7.1 Introduction

Chapter 4 presented the post-mortem forensic process that was proposed to investigate software failures based on reliable digital evidence. Chapter 6 addressed the main limitation of this process by suggesting the proactive approach of near-miss analysis. A mathematical model was proposed to detect and prioritise near misses in order to proactively collect complete and relevant evidence about likely software failures. Chapter 7 now presents the NMS architecture that combines both the forensic investigation process and the near-miss detection and prioritisation model. This original NMS architecture is proposed to promote the usage of sound forensic evidence to conduct an accurate root-cause analysis of software failures. Designing such a model to address the lack of forensic principles and sound evidence in existing approaches towards failure analysis constitutes the main goal of this thesis.

The NMS architecture is designed to satisfy the requirements for accurate failure investigation established in Chapter 2. These requirements ensure that software failures and near misses are dealt with in a forensically sound manner as they arise. The NMS architecture facilitates the collection and preservation of digital evidence about these events, to ensure that the subsequent root-cause analysis provides objective and reliable results. These results are then used to implement effective countermeasures so that the same failures do not reoccur in the future.

Chapter 7 is structured as follows: Section 7.2 revisits the requirements identified in Chapter 2 for the accurate investigation of software failures. It also reviews the proactive and reactive solutions proposed earlier to address the requirements as mentioned above. Section 7.3 presents the NMS architecture that combines both partial solutions.

7.2 Requirements and proposed solutions for the accurate investigation of software failures

7.2.1 Requirements

The following list of requirements established for an accurate evidence-based investigation of software failures was discussed in detail in Chapter 2:

- Objectivity
- Comprehensiveness
- Reproducibility
- Admissibility in court

In addition, two additional requirements were established in Chapter 4 for the successful forensic investigation of software failures:

- Quick system restoration to minimise downtime
- Continuous system monitoring

7.2.2 Proposed solutions

Two processes were proposed to satisfy the requirements listed above. The first one is a forensic investigation process based on digital forensics. This process is an adaptation of the digital forensic process designed to accommodate challenges specific to failure analysis. The second process is a near-miss detection and prioritisation process for the analysis of near misses before a failure occurs. This process is tailor-made for software systems as no such process is as yet available in the software industry. For the sake of clarity, the two processes are reproduced in the sections below.

7.2.2.1 The forensic investigation process for software failures

A representation of this investigation process was provided in the flowchart in Figure 4.1. The process adheres to digital forensic principles. It also comprises some elements of troubleshooting. Digital forensics ensures that the process and its results are reliable and admissible in court, while troubleshooting facilitates the timely restoration of the failed system to limit its downtime.

7.2.2.2 The near-miss detection and prioritisation process

The following process was proposed to effectively manage near misses and use them as tools to improve the accuracy of the root-cause identification of software failures: detection, prioritisation, data collection for high-risk near misses, failure prevention and root-cause analysis. In Chapter 6, mathematical formulas were developed to formally define a near miss, as well as enable its detection and prioritisation based on its risk level. This risk level was defined as the conditional probability that the system will fail, given that the near miss has occurred. Only near misses with a risk level above a predefined threshold were selected for evidence collection and root-cause analysis.

This near-miss analysis process is designed to complement the failure investigation process that was presented in the previous section. It is achieved by enabling the safe collection of failure-related data before a failure occurs and increasing the pool of failure-like events that can point to weaknesses in the software system. This process is therefore integrated with the Evidence Collection phase of the forensic investigation process, more specifically for the collection of primary (i.e. digital) evidence. This process requires the continuous monitoring of the system to identify in real-time unsafe events and conditions that can be classified as near misses. The NMS architecture that integrates both of the above processes is described next.

7.3 The NMS architecture

This section presents the NMS architecture that was designed to satisfy all requirements for accurate software failure investigations listed previously. It combines both the post-mortem forensic investigation process and the pre-emptive near-miss analysis process described above. The section starts with an overview of the overall near-miss and failure investigation process. It is followed by a detailed description of the architecture designed to automate this process.

7.3.1 The overall near-miss and failure investigation process

To detect near misses, the system needs to be monitored with a view to recording and reviewing event logs. It is suggested that system events be logged in a central repository such as a Syslog server. Event logs that match the near-miss formula established in Equation (1) in the previous chapter are flagged as potential near misses and are then sent to another module for

prioritisation. This module calculates the system's failure probability and expected downtime based on each potential near miss.

Afterwards, events identified as being in the critical zone are passed on to another component for data collection. The Simple Network Management Protocol (SNMP) is proposed for this purpose. This Internet-standard protocol enables information exchange between a manager (central unit) and its agents (the other system units) (Presuhn, 2002). In this case, the SNMP manager is the data collection module and it requests additional information about the event from the relevant units through their SNMP agents. Corrective steps are subsequently taken to prevent a system failure, if possible. Finally the collected data is used for root-cause analysis of the event. This root-cause analysis follows the forensic investigation process specified earlier. Upon identification of the root cause of the event, recommendations are made to correct the system flaw.

The architecture of an NMS that was designed to implement the above process is described in the next section.

7.3.2 The NMS architecture

The proposed NMS architecture is shown as a UML component diagram in Figure 7.1. The architecture consists of the five main components below, listed in their logical sequence:

- The Near-Miss Monitor
- The Near-Miss Classifier
- The Near-Miss Data Collector
- The Failure Prevention
- The Event Investigation

Some components are made up of several sub-components that all have a type: a document file, an executable file or a database table. Dashed arrows indicate a component's dependency from the source component to the target component. For instance, the Near-Miss Classifier requires high-risk event logs from the Near-Miss Monitor. Some dependencies are subject to conditions, for example an expected downtime must occur in the critical zone to activate a data request from the Near-Miss Data Collector.

Each of the main components processes the event logs from the redundant units of the system that is being monitored. The five main components of the system are used to perform a multi-stage filtering process that progressively discards “irrelevant” events and only retains near misses with the highest risk factor. The key component of the architecture is the module used to prioritise near misses. This module is named the Near-Miss Classifier. It uses Equation (5) that was established in the previous chapter to classify near misses based on their expected downtime.

A detailed description of the main components of the architecture follows.

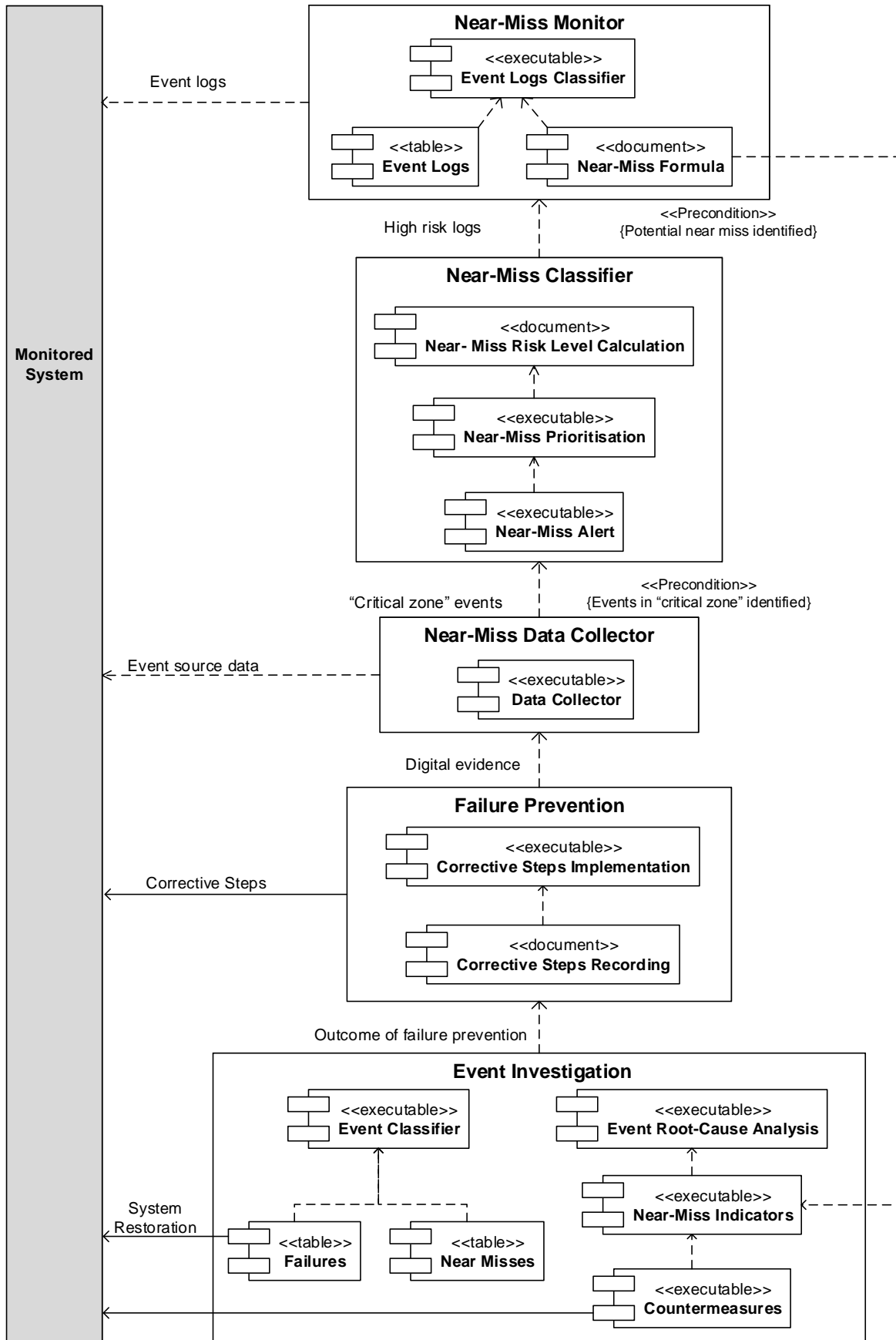


Figure 7.1: UML component diagram of NMS architecture

7.3.2.1 The Near-Miss Monitor

The Near-Miss Monitor monitors the redundant units of the system to identify potential near misses based on the near-miss definition formula. Events from the monitored system are logged to provide information relevant for near-miss detection in line with the near-miss formula. The logged information must include, among others, the status of the unit (up or down) and the duration of the downtime, if applicable. The Near-Miss Monitor keeps track of previous downtime experienced by the system, as well as the remaining downtime allowed in the SLA. If the system goes down and a match is found between these parameters and the near-miss definition formula, the downtime experienced is classified as a potential near miss and sent to the Near-Miss Classifier for prioritisation.

7.3.2.2 The Near-Miss Classifier

The Near-Miss Classifier calculates the risk level of the potential near misses based on their failure probability and expected downtime. It uses and prioritises events accordingly. Logs of events identified as being in the “critical zone” are sent to the Near-Miss Data Collector and an alarm is raised to notify the system administrator.

7.3.2.3 The Near-Miss Data Collector

This module is implemented as an SNMP Manager. The SNMP Manager requests data from the units in the critical zone. Such data may include the source identifier (e.g. IP address), running processes, system settings and error messages. This data is then stored in the Event Data table and transferred to the Failure Prevention module.

7.3.2.4 The Failure Prevention

With this module, the system administrator uses the collected data to identify and implement appropriate corrective steps in an attempt to prevent – or at least mitigate the impact of – system failure. This might include ending some active but unused processes or deleting some stored but unnecessary data to free up memory. The administrator records the steps implemented in a log file for future reference. He then sends the outcome of the recovery attempt (successful or unsuccessful) to the Event Investigation module.

7.3.2.5 The Event Investigation

Based on the outcome of the recovery process in the previous component, the Event Investigation module classifies events as either near misses or failures and stores the event details in the appropriate table for future reference. If the event is a failure, a system restoration is first conducted to limit the experienced downtime. The administrator then conducts a root-cause analysis of the event based on the data stored. The root-cause analysis enables the identification of near-miss indicators that can be used to adjust the formula used in the Near-Miss Monitor.

Afterwards, recommendations for improvement are made and implemented either immediately or at a later scheduled time. The recommendations are stored along with the event details in the relevant table. These steps allow for the creation of an event history that can be looked up in the event of a similar event occurring in the future.

This overall process is summarised in the UML activity diagram in Figure 7.3.

This architecture meets the objectives for incorporating near-miss analysis in the digital forensic investigation of software failures as stated in Section 8.2. It enables the automatic detection of near misses based on objective performance measures specified in the organisation concerned. The detection process is flexible enough to accommodate changing performance requirements and to suit requirements specific to an organisation. The architecture also enables the automatic classification of potential near misses and the prioritisation of near misses to facilitate their investigation. Once near misses have been selected for investigation, the architecture enables the safe and proactive collection of data related to the potential failure for root-cause analysis. This evidence is likely to be more complete and have more integrity than the data collected following the failure. Additionally, the designed NMS enables the improvement of the software once the cause of the outage has been established and recommendations for improvement have been implemented. This prevents the recurrence of a similar failure in the future. An additional benefit of the architecture is that it enables the prevention of an impending failure if appropriate corrective actions are executed timely.

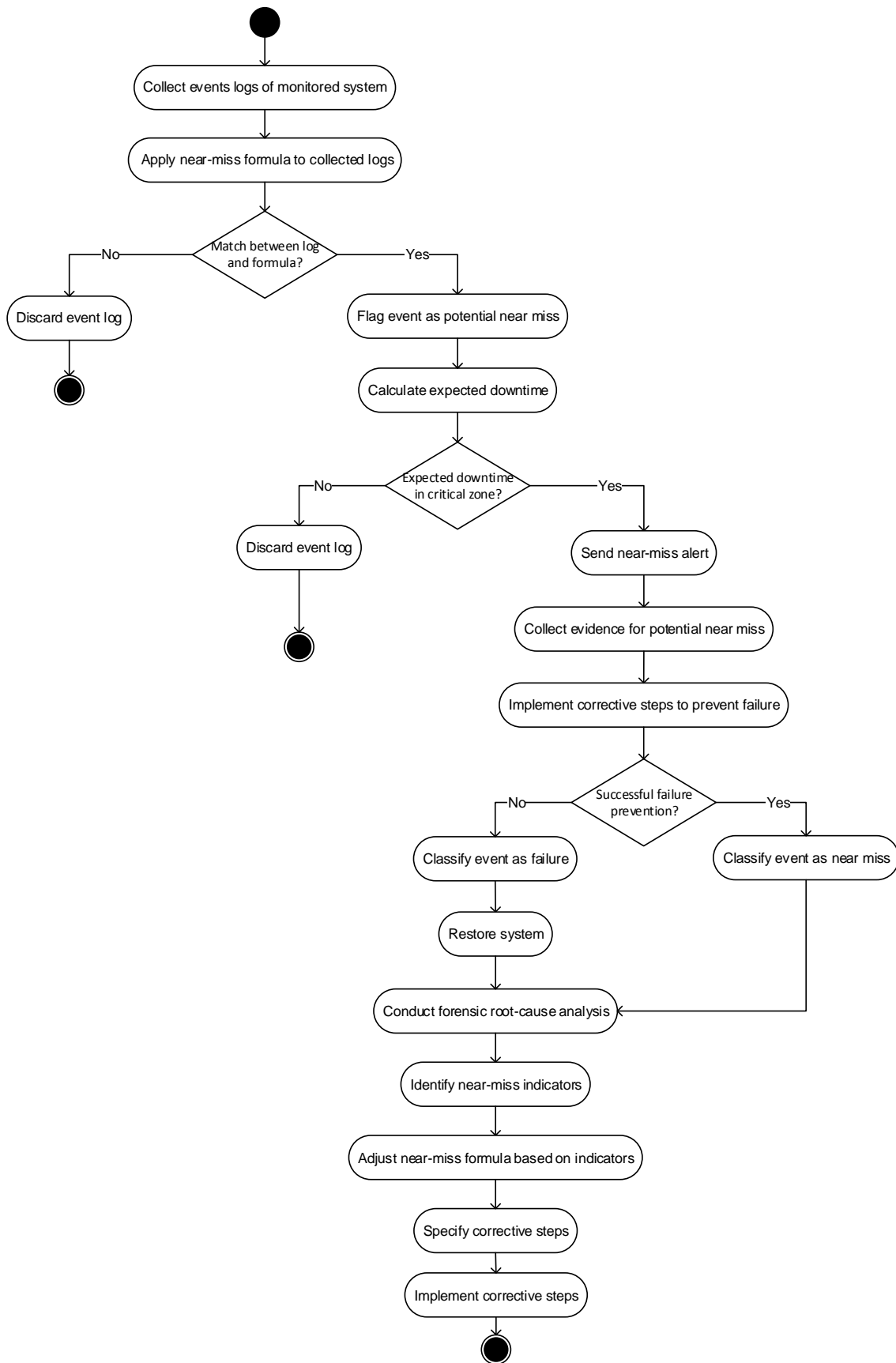


Figure 7.2: UML activity diagram of NMS

The NMS architecture is purposefully designed to be generic as it is on a conceptual level. However, in practice, its implementation will be dependent on the architecture of the target system.

In the case of standard desktop systems and computing-based architectures, it is recommended to design the NMS as a stand-alone system that monitors the target system. This will prevent memory constraints on the target system.

In the case of embedded systems, due to the potentially technical barriers to access data from the system, the recommendation would be to embed the code for the NMS in the design of the embedded system. An example of this design was provided in the prototype implementation where the code to collect and monitor attributes used for near-miss detection was inserted in the C++ program that implemented the file copying application used as the target system for near-miss analysis.

7.4 Conclusion

This chapter provided a high-level description of the original NMS architecture that has been proposed to overcome the limitations of existing approaches to software failure investigations. The proposed architecture has the potential to satisfy all the requirements established for accurate software failure investigations. Original features of the architecture include the provision of a scientific and legal foundation through its alignment with the digital forensic process, the addition of a system restoration step before the failure analysis and, most importantly, the detection, classification and investigation of near misses to obtain complete and relevant digital evidence of the failure for accurate root-cause analysis. A description of the prototype that was developed to test the viability of the NMS architecture is provided in the next chapter.

CHAPTER 8

PROTOTYPING THE NMS – THE DESIGN PHASE

8.1 Introduction

Chapter 7 presented the architecture of an NMS that was proposed to detect, prioritise and investigate near misses. The detection of near misses is based on a mathematical formula developed in Chapter 6 to define near misses formally. Near-miss analysis is proposed as a novel approach to optimise the collection of sound and relevant digital evidence of a failure for accurate root-cause analysis. By alerting system users of an upcoming failure, the detection and prioritisation of near misses provides an opportunity to maximise the collection of appropriate system logs at runtime and reduce the collection of irrelevant data.

This chapter is the first of a three-part series describing the prototype implementation of the NMS architecture. The prototype was designed to demonstrate the viability of the architecture, more specifically the detection of near misses from the analysis of event logs. The chapter also documents the design phase of the prototype. The next two chapters respectively describe the creation of suitable event logs for the prototype implementation and the forensic analysis of these logs to detect near misses. The process to conduct such an analysis was presented in Chapter 4.

The remainder of Chapter 8 is structured as follows. Section 8.2 presents the objectives of the prototype. An overview of the preliminary work that was performed to set up the lab environment is next described in Section 8.3. The prototype implementation plan is then presented in Section 8.4.

8.2 The aims of the prototype

This section starts with a brief review of the NMS architecture, which is the basis of the prototype implementation. The objectives of the prototype implementation are presented in relation to this original architecture.

8.2.1 The original NMS architecture

The prototype aims to demonstrate a subset of the NMS architecture. A UML component diagram (OMG, 2007) of the NMS architecture was provided in Figure 7.2 in Chapter 7.

The NMS architecture is made up of five components that work in sequence. A detailed description of each component was provided in Chapter 8. The following is a summary of the components' respective main functions:

1. Near-Miss Monitor: Monitor target system to identify logs of high-risk events that are potential near misses. Potential near misses are identified based on the near-miss formula defined in Chapter 6.
2. Near-Miss Classifier: Classify the potential near misses based on their risk level and send an alert for the events with the highest risk level. A mathematical model to calculate this risk level was developed in Chapter 6.
3. Near-Miss Data Collector: Collect digital evidence of the potential near misses for which an alert has been sent.
4. Failure Prevention: Apply corrective measures in an attempt to prevent the upcoming failure.
5. Event Investigation: Use the evidence collected to conduct a root-cause analysis of the events using the scientific method and digital forensic tools and techniques. These events are classified as near misses in case the failure does not unfold. The root-cause analysis enables the identification of appropriate countermeasures to be applied, should the unsafe events reoccur in the future. It also enables the identification of near-miss indicators that can be used to adjust the formula used in the Near-Miss Monitor.

The prototype focuses on the following three components of the architecture: Near-Miss Monitor, Near-Miss Classifier, and Event Investigation. These components perform the key functions of the NMS and are sufficient to meet the objectives of the prototype implementation, as will be discussed in the next section. Some functionality of the Failure Prevention as well as of the Data Collector are present in the prototype but a full implementation of these components falls outside the scope of this research.

8.2.2 Prototype goal and objectives

The goal of the prototype implementation is twofold:

- Demonstrate the viability of the digital forensic process formulated in Chapter 4 to conduct a root-cause analysis of a software failure. In the absence of an SLA, identifying the root cause of a failure is necessary in order to identify near-miss indicators for that particular type of failure. Each indicator is a unique system condition and the combination of all indicators and their interdependencies provides a pattern in the system behaviour that indicates that the system might be heading towards a failure.
- Demonstrate the viability of detecting near misses at runtime. This also demonstrates that a near miss is a viable and relevant concept for the software industry.

The above-mentioned goal is accomplished by means of the following:

- The use of collected digital evidence of the failure as the basis for the root-cause analysis. Digital evidence should be sufficient to identify the source of the failure and no prior experience with the system should be required.
- The identification of near-miss indicators to define a near miss.
- The development of a near-miss formula. This formula is a mathematical expression of all the indicators with their respective interdependencies.
- The identification of potential near misses using a set of event logs.
- The detection of near misses at runtime.

Figure 8.1 shows the adapted diagram of the NMS architecture that was used to implement the prototype in accordance with the above objectives.

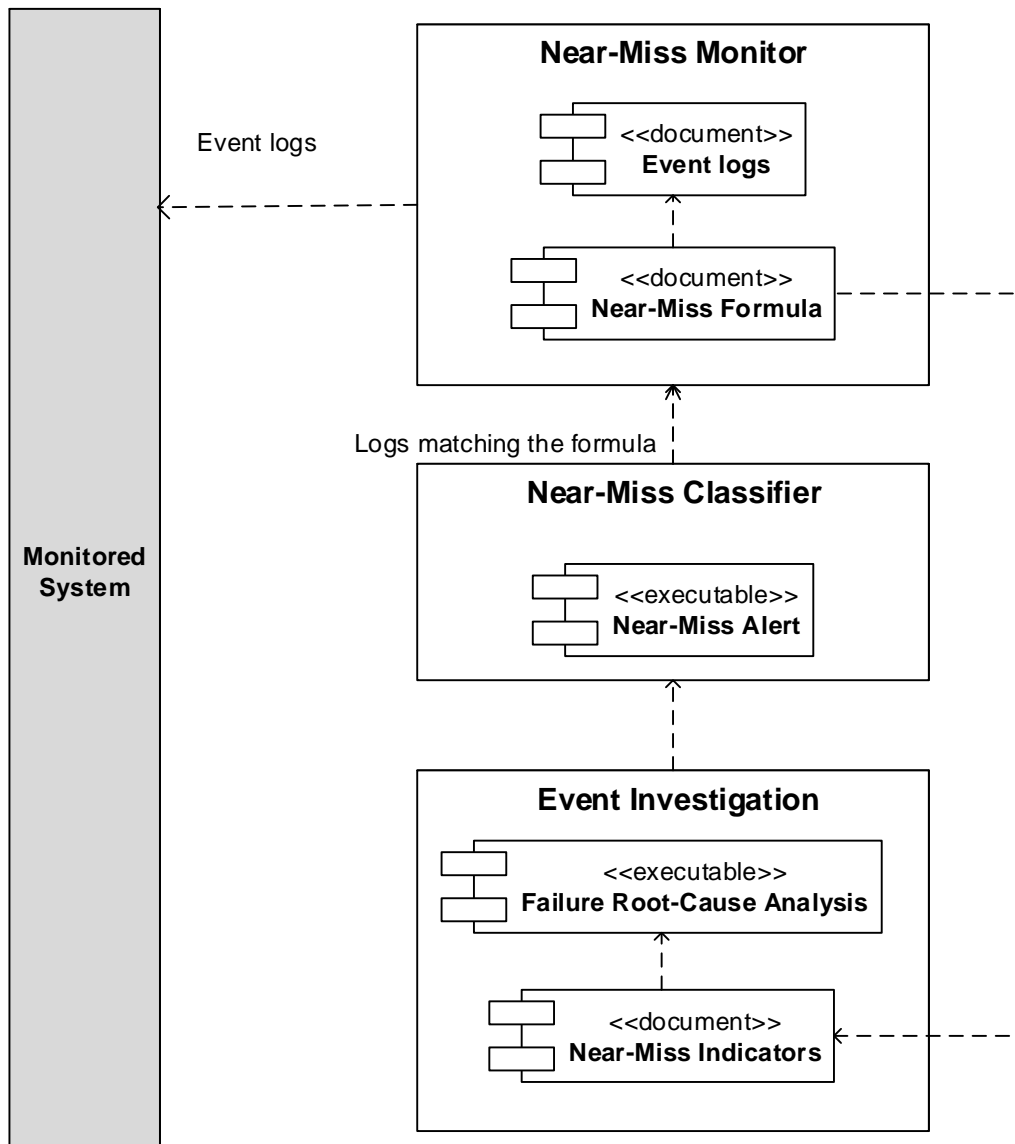


Figure 8.1: Adapted NMS component diagram for prototype implementation

The components in Figure 8.1 are implemented as follows:

- Analyse a software failure to identify its root cause. (Event Investigation)
- Use the root cause to identify near-miss indicators. (Event Investigation)
- Use the near-miss indicators to define a near-miss formula. The formula is used in the subsequent monitoring of the system. (Near-Miss Monitor)
- Send an alert for potential near misses detected at runtime. (Near-Miss Classifier)

The prioritisation of potential near misses is not part of the prototype implementation as it is not required for meeting the goal and objectives specified earlier.

8.3 Setting up the lab environment

Conducting a root-cause analysis of a software failure was the basis for the prototype's goal and objectives. Conducting such an analysis required three necessary elements: the logs of a software failure, a forensic investigation tool with suitable data analysis techniques and a test plan. These elements are discussed in Sections 8.3.1, 8.3.2 and 8.3.3 respectively.

8.3.1 The logs of a software failure

Obtaining logs of a past software failure that would be suitable for the prototype proved challenging. The researcher therefore opted to simulate a failure and generate logs of the event. Two types of logs were deemed relevant for the root-cause analysis: logs created by the researcher and logs generated by the computer system used for the failure simulation. The process that was followed to obtain each of the types of log is discussed in the next two sections.

8.3.1.1 Logs created by the researcher

A software failure was simulated by writing a program with some deliberate weaknesses that would result in a failure. While running, the program would write its output to a file along with some statistics of its running environment. This information would serve as logs of the failure. The output file would therefore be the log file created by the researcher. It is subsequently referred to as the crash file.

The software failure to be investigated had to be caused by the exhaustion of resources, in line with the near-miss failure probability formula established in Chapter 6. The argumentation for this choice was as follows. Unlike many other unpredictable sources of failures, resource exhaustion could be predicted through monitoring. The pattern of an upcoming failure could therefore be observed and used to define a potential near miss. For the purpose of this prototype implementation, memory was selected as the resource to be exhausted. For this reason, the program had to fail due to a lack of available memory and the environment statistics to be recorded would be memory-related.

The researcher decided for the program to exhaust the memory of an external drive. This allows more control over the drive's available free space than is possible with a computer's internal hard drive, as the latter is controlled by the operating system. The program was designed to run

as a loop that copies a video clip to a flash disk repetitively beyond the flash disk free space. The choice of a video clip was due to its usually larger file size than other file formats. This enabled the file operation (copy to flash disk and reading copy to check its integrity) to take enough time for some relevant parameters (e.g. duration) to be observed and recorded. A representation of the program's structure is provided in a high-level flowchart in Figure 8.2.

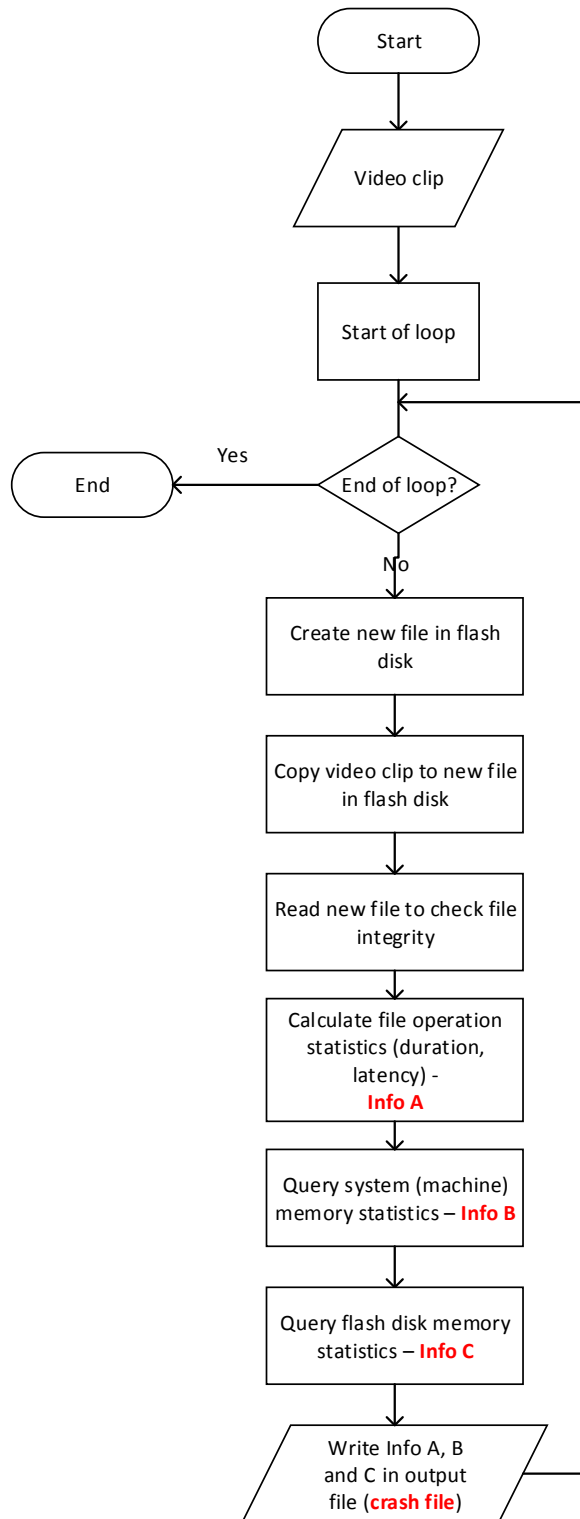


Figure 8.2: Flowchart of failure simulation program to create the crash file

As shown in Figure 8.2, in every iteration of the loop, the program writes to the crash file three types of information:

- File operation statistics: e.g. duration (time to copy the video clip and read the copy) and latency (time delay between two consecutive file operations)
- Memory statistics (e.g. free space and used space) from the computer system
- Memory statistics from the flash disk

In order to obtain the above memory statistics, the program was written in C++ as this language provides built-in functions to access information about any mounted file system.

The aim of generating a crash file was to find some indicators of the upcoming failure that could be used as potential near-miss indicators. The above program was therefore purposefully designed to be simple so as to focus on the near-miss indicators instead of on the complexity of the computer system. The following assumptions were made regarding the identification of the indicators:

- They would be found in the crash file in the last few entries before the failure.
- They would reflect a pattern in the system's behaviour that is significantly different from the expected behaviour of the system.
- The crash file needed to be big enough for such a pattern to be visible. This required having a high number of output records.

A spreadsheet format was used for the crash file as it is a format used by a number of forensic tools to display digital evidence (Fei et al., 2005). In a typical forensic investigation, each file in the forensic data set is represented as a record with various fields describing the file, such as file name, creation date and size. The same approach was followed when generating the crash file, using the creation of every new copy of the video file as a record.

The researcher also assumed that the running of the above program on any operating system (OS) would be recorded in that OS's own log files. Information in these log files could therefore be used to corroborate information in the crash file and could provide some additional near-miss indicators. For this reason, the computer's log files were also explored to find suitable ones for the above-mentioned purpose. This exercise is discussed in the next section.

8.3.1.2 System logs

The crash file was generated on a machine running on a Linux operating system (OS). Various Linux log files and utilities were therefore explored to find relevant information that could be linked to the crash file. Since the failure simulating program was performing significant input (reading copy of video clip) and output (copying video clip to new file) operations, it was deemed most appropriate to use the `iostat` monitoring utility to show input and output (I/O) usage on the Linux disk. The `iostat` command continuously displays a table of I/O usage by processes and threads and refreshes the information every second (Linux.die.net, 2014). It provides various I/O statistics such as disk-reading bandwidth and disk-writing bandwidth.

Using `iostat` was preferred to other similar commands like `top` and `htop` as these commands provide memory consumption information, which was already available from the crash file. The researcher planned on running the `iostat` command concurrently with the failure simulating program, so that I/O usage of this program would be displayed. The output of `iostat` would then be redirected to a file, and would serve as a system-generated log file. To find near-miss indicators, both the crash file and the `iostat` output file would be analysed using the tools and techniques described next.

8.3.2 The forensic investigation tool and techniques

8.3.2.1 The investigation tool

Ideally, one should use a digital forensic tool to conduct a forensic (root-cause) analysis of the log files. Popular digital forensic tools were listed in Chapter 4. However, these tools are limited in their ability to handle and interpret large volumes of data, as well as in their visualisation capability (Nassif & Hruschka, 2011; Guarino, 2013). Since one of the goals for the prototype implementation was to observe a pattern in the system's behaviour, a tool with powerful visualisation capability that could handle large data sets efficiently was required. For this reason, a tool with a Self-Organising Map (SOM) analysis capability (Engelbrecht, 2003) was selected. The SOM is a powerful data classification technique optimised for large data sets (Engelbrecht, 2003), as will be explained in the next section.

To the best of the researcher's knowledge, popular digital forensic tools are not equipped with a SOM capability. Therefore, a SOM tool was used instead of a digital forensic tool. A

commercial SOM tool called Viscovery SOMine (viscovery.net, 2014) was used. The trial version of this tool was used as it was freely available and it provided all the functionality needed for this prototype. An overview of the SOM is provided in the next section.

8.3.2.2 The investigation techniques

Two techniques were used to analyse the failure: the SOM analysis and statistical analysis. Both these techniques were used due to their scientific foundation (mathematics) and their ability to identify trends in the data set. They are discussed next.

8.3.2.2.1 Overview of the SOM

The SOM is a model of unsupervised neural networks used for the analysis and visualisation of multi-dimensional data (Engelbrecht, 2003). Like other unsupervised neural network algorithms, the SOM classifies input data based on the similarity of the input vectors (records in the data set). Similar vectors are grouped in the same cluster. However, the distinguishing feature of a SOM is that its neurons (or nodes) represent a topological system (usually a two-dimensional rectangular or hexagonal map) and they are arranged in an ordered and structured way, based on their weights (Hollmén, 2000). Neurons with similar weight vectors are situated close to one another while neurons with very different weights are physically far apart (Kohonen, 1990). Like other neural network algorithms, the SOM has two successive operating modes (Kohonen & Honkela, 2007):

1. The training process where the map is constructed through competitive learning, which means that the learning process is data driven (Fei, Eloff, Venter & Olivier, 2006). This phase requires a very large number of input vectors to accurately represent all or most of the patterns to be identified.
2. The mapping process where a new input vector is quickly and automatically assigned a location on the map, based on its feature.

Usually, a SOM can be graphically visualised by displaying a unified distance matrix (U-matrix) that shows the different clusters identified in the input data. The U-matrix calculates the Euclidian distance between the map units and a colour is assigned to each unit based on this distance. Close units have similar colours (Hollmén, 2000). In case the identified clusters are labelled, their label can also be displayed on the associated map unit.

The researcher aimed to use these clusters to identify a change in the system's behaviour from expected to unexpected. Her previous experience with the SOM demonstrated the feasibility of this approach (Bihina Bella, Eloff & Olivier, 2009). Using a SOM can offer several benefits as the algorithm is very fast and highly visual. It quickly reduces the complexity of a large data set to a few patterns that are quickly identifiable (Hsu, 2006).

It is worth mentioning that the SOM is not a forensic technique as such. However, it is based on science (mathematics), which is a primary requirement for forensic techniques. This scientific foundation enables the results of the SOM analysis to be objective and reliable. Furthermore, its suitability and efficiency for forensic investigations was demonstrated before by a number of earlier researchers (Fei et al., 2006; Palomo, North, Elizondo, Luque & Watson, 2012).

8.3.2.2.2 *Statistical analysis*

In order to identify trends in the data set, a statistical measure called a weighted moving average (WMA) was used. Since A WMA gives more weight to the most recent data in a time series and less importance to older data, it is used for trend forecasting (Holt, 2004). This is particularly relevant for the research at hand, which aims to predict likely failures based on near misses. The WMA of the previous values of a parameter shows the trends in that parameter and can indicate a change in the system's behaviour, which can potentially be used to detect an upcoming failure.

The WMA of a parameter is calculated by multiplying each value (D) by its position (n) in the time series, and dividing the sum of these values by the total of the multipliers (positions). Its formula is as follows:

$$\text{WMA} = \frac{n(D_n) + (n-1)(D_{n-1}) + (n-2)(D_{n-2}) + \dots + 2(D_2) + 1(D_1)}{n + (n-1) + (n-2) + \dots + 2 + 1}$$

The normal average was also used and compared to the WMA to determine deviation from expected behaviour.

8.3.3 The test plan

As discussed in Chapter 2, a forensic investigation needs to be reproducible and thus has to adhere to a pre-defined procedure. Therefore, the following high-level plan was established for the forensic analysis and near-miss detection:

1. Run the C++ (failure simulation) program so that it creates a crash file in a spreadsheet format.
2. Concurrently run the `iostat` utility to create a log file of the program I/O usage.
3. Conduct a forensic root-cause analysis of both the crash file and the `iostat` output file.
4. Use the root cause to identify indicators of the upcoming failure.
5. Use the interdependencies between indicators to define potential near misses through a formula.
6. Use the near-miss formula to detect potential near misses while the C++ program is running.

A flowchart of the above process is shown in Figure 8.3. The six steps listed above are grouped into four main experiments as shown in the flowchart. Figure 8.3 also shows that the root-cause analysis follows the scientific method. As explained in Chapter 4, the scientific method has three main steps: formulating a hypothesis; predicting evidence for the hypothesis; and testing the hypothesis with an experiment. Both the SOM analysis and the statistical analysis are used to test the hypothesis.

In the last step, the near-miss formula is inserted into the C++ program to detect potential near misses. When a potential near miss is detected, an alert is sent prompting for some corrective actions. Implementing these corrective actions falls outside the scope of this research and is consequently not included in the prototype implementation.

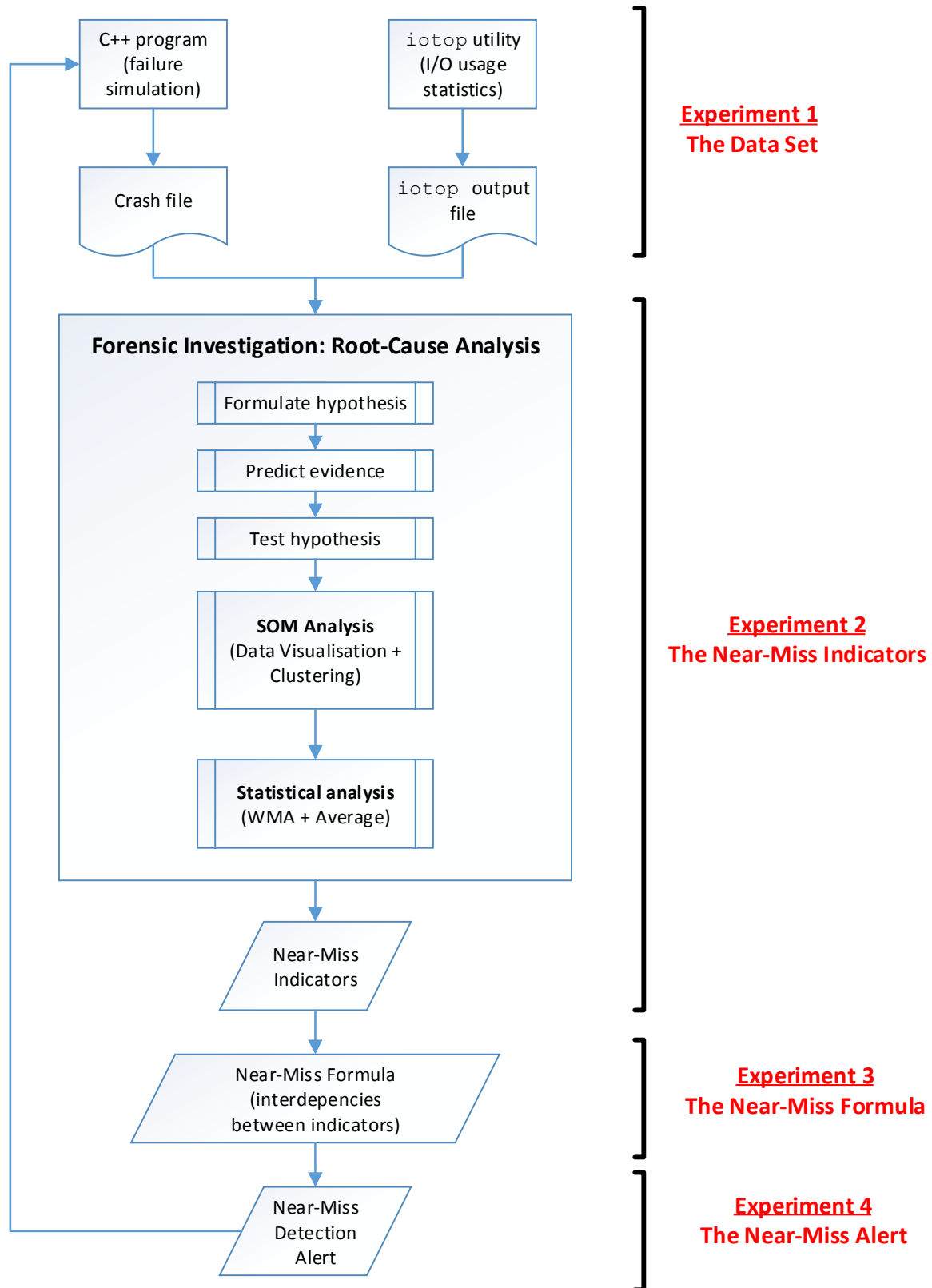


Figure 8.3: Prototype implementation plan

To ensure the forensic soundness of the results throughout the process of prototype implementation, attention was paid to adherence to best practice in digital forensics. This includes minimal handling of original data, keeping account of any change to the data (change was made only to the format of the data, not its value), and maintaining the chain of custody (data was not moved from its source system).

The implementation of the above plan is documented in Chapters 9 and 10.

8.3.4 Conclusion

This chapter discussed the design phase of the prototype implementation of the NMS architecture, which was presented in Chapter 7. The chapter first presented a diagram of the subset of the NMS architecture implemented in the prototype. This subset focused on the root-cause analysis of a software failure to identify near-miss indicators and detect potential near misses based on these indicators. Details of the prototype goal and objectives, as well as of the preparatory work to set up the lab were provided. The preparatory work included identifying the required data set, selecting the data analysis tool and techniques, and creating a test plan. The next chapter describes the implementation of the first experiment of the prototype, which involves the creation of a suitable data set. The other three experiments are described in Chapter 10.

CHAPTER 9

PROTOTYPING THE NMS – THE DATA SET

9.1 Introduction

Chapter 8 described the design phase for the prototyping of the NMS architecture presented in Chapter 7. The prototype was designed to demonstrate the viability of the architecture. The design phase clearly specified the scope of the prototype as the runtime detection of near misses from event logs. The design phase also outlined the plan to implement the prototype.

This chapter describes the first step of the implementation plan, which is the production of suitable event logs to detect near misses. Indeed, since suitable event logs were not readily available, they had to be created by simulating a software failure. Chapter 9 describes the experiment that was conducted to simulate a failure due to memory exhaustion and to generate logs suitable for the subsequent forensic analysis and near-miss detection. Due to their lengthy documentation, these subsequent phases are described in the next chapter.

Chapter 9 is further structured as follows: Section 9.2 describes the technical platform used for developing the prototype, while the log creation experiment is described in Section 9.3.

9.2 Technical platform used for developing the prototype

9.2.1 The prototype implementation plan

The prototype implementation plan was presented in detail in the previous chapter. This plan was implemented over a series of four experiments, each one building on the previous one to obtain more relevant information and more usable results. The four experiments are depicted in Figure 9.1.

Chapter 9 focuses on the first experiment, which is the creation of a set of failure logs. In the previous chapter, this data set was designed to consist of two complementary types of event logs: logs from a failure simulation program created by the researcher and logs from the

program's host machine produced by the operating system. The former, referred to as a crash file, was created from a C++ program that fails due to external memory exhaustion on a Linux machine. The latter was selected to be generated from the Linux `iostat` utility, which monitors the disk I/O usage. This process is shown in the highlighted areas in Figure 9.1.

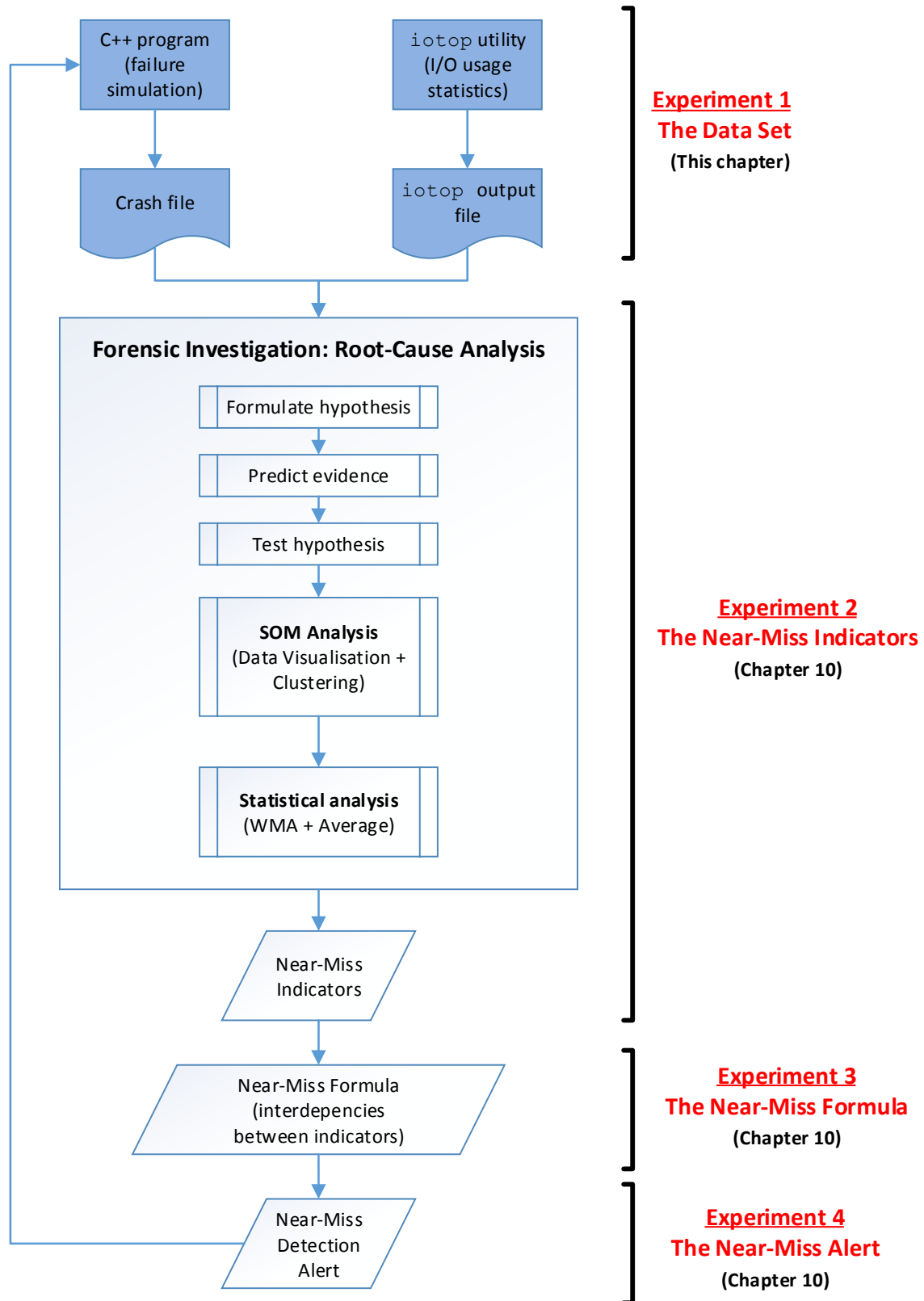


Figure 9.1: Prototype implementation plan

9.2.2 Technical set-up for prototype implementation

The prototype was implemented as a failure simulating program running on a Linux operating system (OS). As only one machine was available for conducting the experiment as well as documenting its results, the program was executed on a virtual machine to avoid any potential crash of the host machine. The host machine was running on a Windows 7 OS, while the virtual machine was running on a Linux Debian OS, both using a 64 bit architecture. The Linux machine was allocated 1 GB of virtual hard drive. Oracle VM VirtualBox was used as the virtualisation software product (virtualbox.org, 2014) as it is freely available online as open source software.

A shared directory was created between the host and the guest OS so that files could be shared between them. This was necessary in order to process in Windows (i.e. make a forensic analysis of) the crash file generated in Linux. Figure 9.2 shows a screenshot of the user interface of VirtualBox, as well as the shared directory – named VirtualBoxShare – as displayed on the Windows host machine and on the Linux virtual machine. The screenshot shows that the content of the shared directory is the same on both machines.

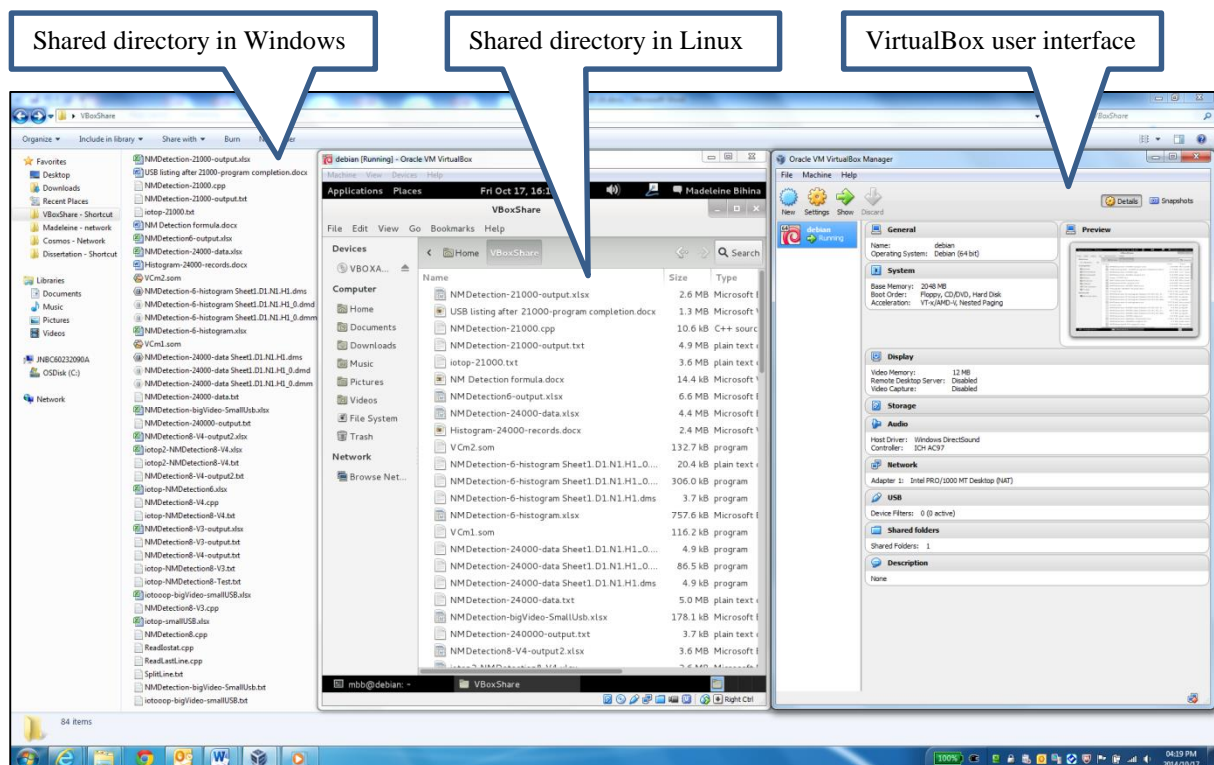


Figure 9.2: VirtualBox user interface and shared directory between Windows and Linux

A diagram of the lab environment is provided in Figure 9.3. In addition to the guest and host machines and the shared directory between them, the lab environment includes the SOM analysis tool (Viscovery SOMine) and the `iostat` utility that was running in parallel to the C++ program. The C++ program was written to simulate a failure by repetitively copying a video clip to a flash disk until the flash disk's free space was exhausted.

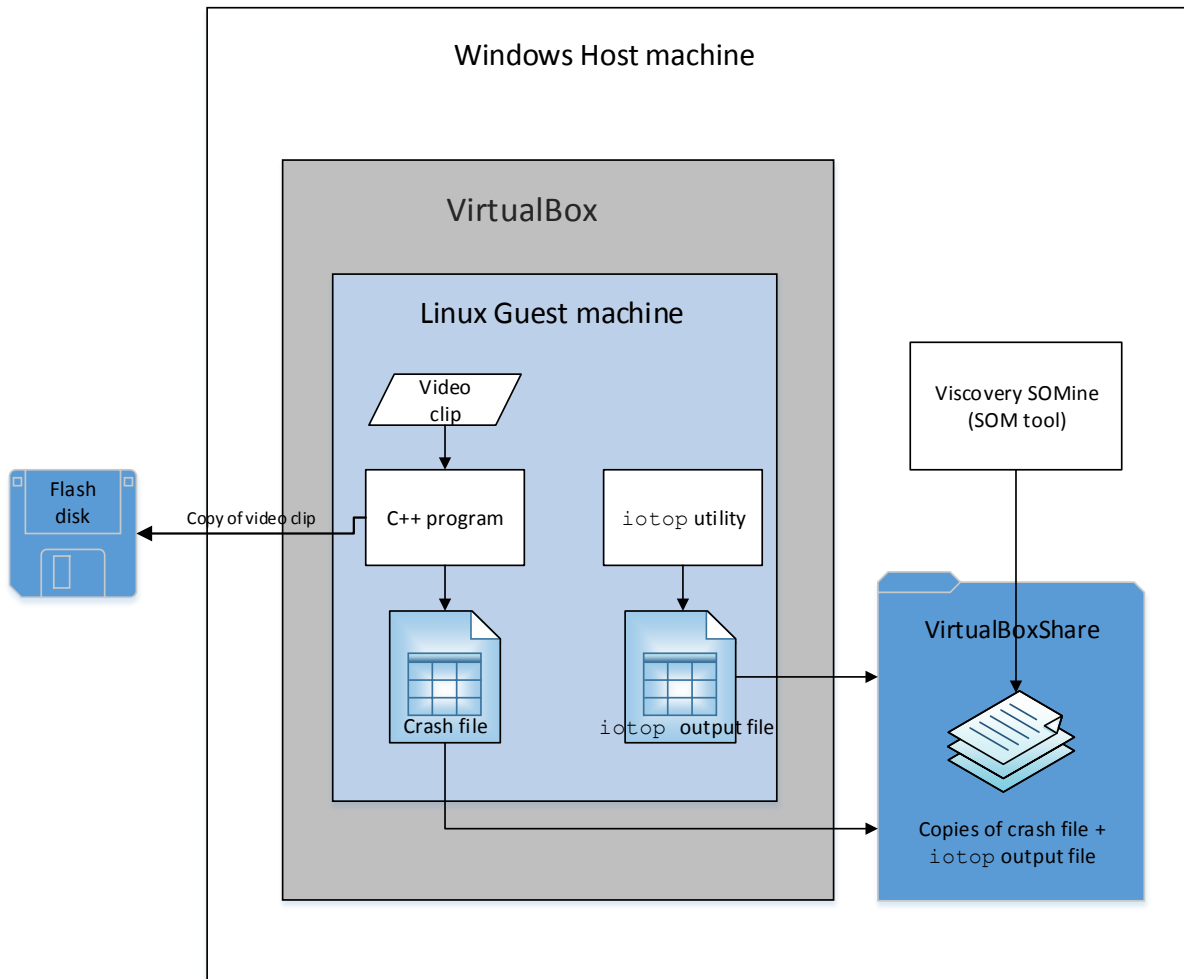


Figure 9.3: Diagram of the lab environment

9.3 Experiment 1: Creating a suitable set of event logs

The goal of this first step was to create a crash file and an output file of the `iostat` utility that would be suitable for the forensic analysis to be performed next. The suitability of the files was defined in terms of the following characteristics:

- Providing relevant information about the failure
- Containing a large number of output records, preferably several thousands of them
- Having a spreadsheet format to facilitate their forensic analysis

The focus of step one was therefore on the NM Monitor of the NMS adapted architecture presented in the previous chapter. This component is shown in Figure 9.4, where the sub-component relevant for this phase of the prototype implementation is highlighted.

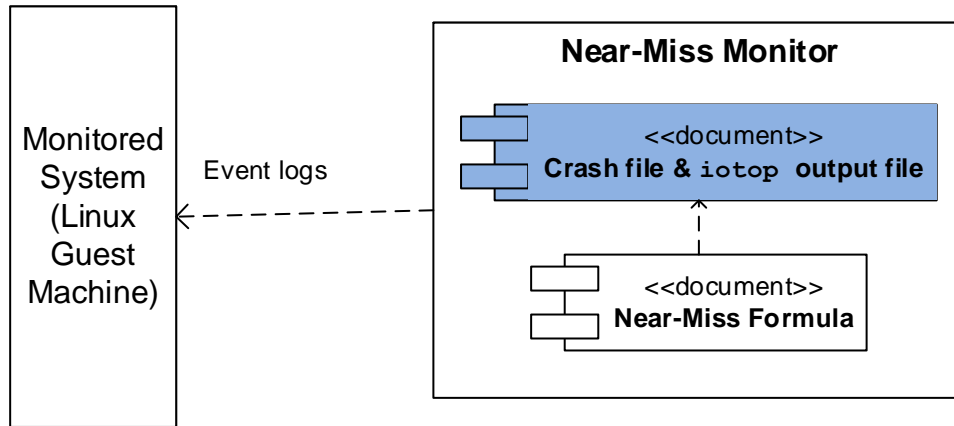


Figure 9.4: Focus of experiment 1 – Near-Miss Monitor of adapted NMS architecture

9.3.1 The crash file

9.3.1.1 Technical set-up for crash file

The C++ program was designed to repeatedly copy a video clip to a flash disk. Every time a new copy of the video clip was made, various statistics about the C++ program, the Linux machine and the flash disk were displayed. As the failure was caused by memory exhaustion, these statistics were selected to be related to memory usage, in order to help identify relevant near-miss indicators.

The literature indicates that software failures due to resource exhaustion often manifest through a performance slowdown, with unusually slow response time (Pertet & Narasimhan, 2005). Memory statistics that could indicate symptoms of the above were therefore recorded.

Information logged about the C++ program

In the case of the C++ program, a slow response time can be the result of either (or a combination) of the following conditions:

- A longer time duration to complete a file operation (the latter refers to copying the video clip and reading the new copy to verify its integrity)
- A longer time delay (latency) between two successive file operations

These two statistics are recorded while the program runs. In order to calculate the duration of a file operation, the start time and the end time are required. It is also necessary that the file status (successful or unsuccessful copy) be recorded to detect the failure. The list of attributes recorded from the C++ program therefore looks as follows:

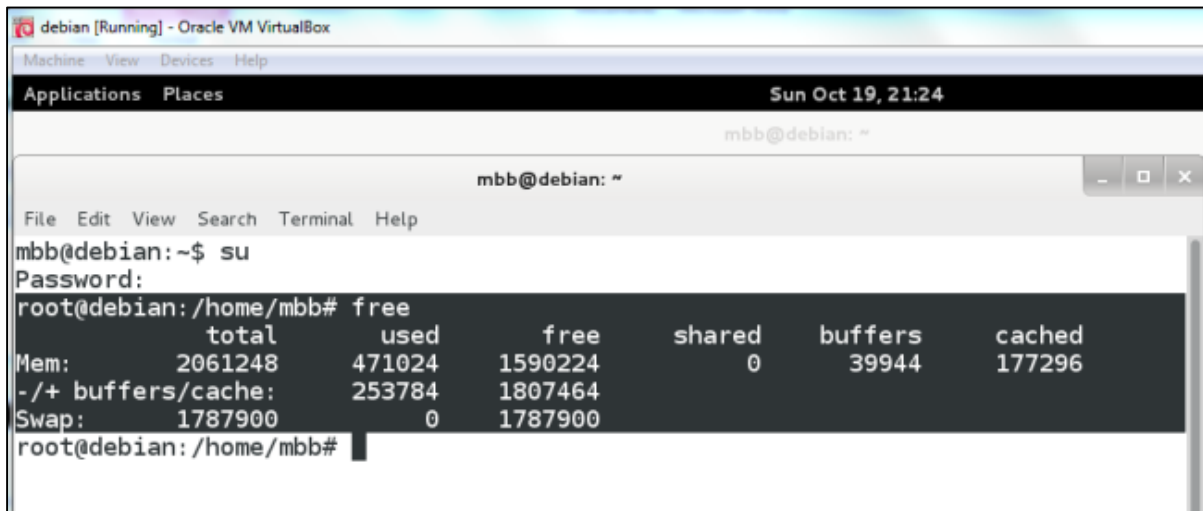
- **File Nr:** Number of the new copy of the video file. This corresponds with the number of the current iteration of the program's loop and is also the record number.
- **Creation time:** Time when the new file was created to copy the video clip.
- **File status:** The program displays "OK" if the video file was copied successfully and "not_OK" otherwise.
- **End time:** Time when the file operation completed.
- **Duration:** Duration of file operation. This is the time difference between the "creation time" and the "end time". For greater accuracy, it was expressed in milliseconds. Hence the creation time and end time were displayed with millisecond precision in the format hh:mm:ss.000.
- **Latency:** Time delay between the end of one file operation and the beginning of the next one.

Information logged about the Linux machine

Regarding the C++ program's host system, the literature indicates that a heavy load and memory usage on a Linux machine often manifest through random access memory (RAM) exhaustion and high swapping activity (Santosa, 2006; Bytemark, 2014). Since a large amount of input (reading new video copy; reading time of file operation), output (writing video clip to new file) and temporary storage of data (to convert time values to desired format; to store the machine's memory statistics into variables) is performed on the C++ program, high activity on memory buffering and caching is also expected. These attributes are therefore recorded as follows:

- **Mem Used:** Amount of RAM used
- **Mem Free:** Amount of RAM available
- **Buffers:** Amount of RAM buffered
- **Cached:** Amount of RAM used for caching of data
- **Swap Used:** Amount of swap space used
- **Swap Free:** Amount of free swap space

The above memory statistics are provided in kB. The Linux `free` command was invoked from the C++ program to display these statistics. A typical output of the `free` command is displayed in Figure 9.5. The command name and its output are highlighted. The standard output had to be parsed to display only the above-selected statistics and to display them in the crash file format.



```

mbb@debian:~$ su
Password:
root@debian:/home/mbb# free
              total        used        free       shared    buffers     cached
Mem:           2061248      471024     1590224           0       39944     177296
-/+ buffers/cache:      253784     1807464
Swap:          1787900           0       1787900
root@debian:/home/mbb#

```

Figure 9.5: Output of the free command in Linux

Information logged about the flash disk

Regarding the flash disk, it is expected that increasing memory consumption manifests through a decrease in the amount of free space. Therefore, the amount of free space is recorded as the program runs. The Linux built-in `statvfs()` function was used in the C++ program to display this statistic. This function provides a structure that contains various details about a mounted file system. The function was written to return only the USB free space, calculated as the number of free memory blocks multiplied by the size of a block, in kB. The corresponding code snippet is shown in Figure 9.6.

```

//function to get usb memory statistics
void getUsbMemInfo()
{
const char *path = "/mnt/flashdisk/test.txt";
struct statvfs usb;
if (statvfs(path, &usb)==0)
{ unsigned long long freeSpace = ((unsigned long long)usb.f_bsize * (unsigned long long)usb.f_bfree)/1024;
cout << freeSpace <<"\t";
else {cout <<"Unable to get free space in usb\t";}
}
}

```

Figure 9.6: Code snippet for function to obtain the amount of free space on the flash disk

Program's running conditions

Since a large data set was required for the subsequent SOM analysis, the crash file was designed to maximise the number of records. This was achieved by running the program with the largest flash disk and the smallest video file at hand, which resulted in the following:

- A flash disk with a capacity of 128 GB
- A video clip of 3.91 MB
- A maximum of 31 001 potential records in the crash file (128 GB/3.91 MB)

In order to force a failure, the size of the program's loop was deliberately set to be higher than the maximum number of potential records. It was set to 31 150. The resulting crash file is presented next.

9.3.1.2 Results

Screenshots of the crash file are provided in Figure 9.7 (beginning of file) and 10.8 (point of failure). The highlighted row in Figure 9.8 (file number 31 002) indicates the point of failure, after the last successful copy of the video clip was made to the flash disk.

File Nr	Creation time	Mem Used	Mem free	Buffers	Cached	Swap Used	Swap Free	USB free space (kB)	File Status	End time	Duration	Latency
1	2014/08/30 02:35:57.904	136076	898512	35444	64756	0	647164	124996320	OK	2014/08/30 02:36:03.754	5850	
2	2014/08/30 02:36:03.768	161268	873320	50316	72772	0	647164	124992288	OK	2014/08/30 02:36:03.963	195	14
3	2014/08/30 02:36:03.978	165768	868820	50316	76780	0	647164	124988256	OK	2014/08/30 02:36:04.187	209	15
4	2014/08/30 02:36:04.201	170268	864320	50316	80788	0	647164	124984224	OK	2014/08/30 02:36:04.392	191	14
5	2014/08/30 02:36:04.406	174712	859876	50316	84796	0	647164	124980192	OK	2014/08/30 02:36:04.602	196	14
6	2014/08/30 02:36:04.616	179240	855348	50316	88804	0	647164	124976160	OK	2014/08/30 02:36:04.808	192	14
7	2014/08/30 02:36:04.824	183768	850820	50316	92812	0	647164	124972128	OK	2014/08/30 02:36:05.011	187	16
8	2014/08/30 02:36:05.026	188240	846348	50320	96820	0	647164	124968096	OK	2014/08/30 02:36:05.236	211	15
9	2014/08/30 02:36:05.252	192768	841820	50320	100828	0	647164	124964064	OK	2014/08/30 02:36:05.440	188	16
10	2014/08/30 02:36:05.455	197296	837292	50320	104836	0	647164	124960032	OK	2014/08/30 02:36:05.644	188	15

Figure 9.7: Crash file at beginning of program

File Nr	Creation time	Mem Used	Mem free	Buffers	Cached	Swap Used	Swap Free	USB free space (kB)	File Status	End time	Duration	Latency
31001	2014/08/30 14:04:16.803	1014216	20372	2204	875804	9212	637952	8192	OK	2014/08/30 14:04:18.004	1201	64
31002	2014/08/30 14:04:18.040	1013472	21116	2204	875048	9212	637952	4160	OK	2014/08/30 14:04:19.362	1322	36
31003	2014/08/30 14:04:19.401	1012604	21984	2204	874152	9212	637952	128	Not_OK	2014/08/30 14:04:19.793	392	39
31004	2014/08/30 14:04:19.814	1012860	21728	2220	874508	9212	637952	0	Not_OK	2014/08/30 14:04:19.857	44	21
31005	2014/08/30 14:04:19.878	1012860	21728	2220	874508	9212	637952	0	Not_OK	2014/08/30 14:04:19.921	42	21
31006	2014/08/30 14:04:19.942	1012860	21728	2220	874508	9212	637952	0	Not_OK	2014/08/30 14:04:19.987	45	21
31007	2014/08/30 14:04:20.007	1012860	21728	2220	874508	9212	637952	0	Not_OK	2014/08/30 14:04:20.050	43	20
31008	2014/08/30 14:04:20.071	1012860	21728	2220	874508	9212	637952	0	Not_OK	2014/08/30 14:04:20.115	44	21
31009	2014/08/30 14:04:20.135	1012860	21728	2220	874508	9212	637952	0	Not_OK	2014/08/30 14:04:20.179	43	20
31010	2014/08/30 14:04:20.200	1012860	21728	2220	874508	9212	637952	0	Not_OK	2014/08/30 14:04:20.243	43	21
31011	2014/08/30 14:04:20.264	1012860	21728	2220	874512	9212	637952	0	Not_OK	2014/08/30 14:04:20.318	54	21
31012	2014/08/30 14:04:20.339	1012860	21728	2220	874512	9212	637952	0	Not_OK	2014/08/30 14:04:20.381	42	21

Figure 9.8: Crash file – point of failure

The following observations were made from the crash file:

- The 31 001 successful records were generated in 11h 28 min 21s 458 ms.
- The average values for Duration and Latency were 1295 ms and 36 ms respectively.

- Contrary to what was expected, the C++ program neither generated an error message nor terminated once the flash disk's free space was used up.
- The program continued to run past the failure point until the loop was terminated.
- For some unclear reason, the program stopped displaying values for the last four parameters (File Status, End time, Duration and Latency) only after record 31 042 (see Figure 9.9).

	A	B	C	D	E	F	H	I	J	K	L	M	N
31042	31041	2014/08/30 14:04:22.321	1012860	21728	2220	874516	9212	637952	0	Not_OK	2014/08/30 14:04:22.376	55	21
31043	31042	2014/08/30 14:04:22.397	1012860	21728	2220	874516	9212	637952	0	Not_OK	2014/08/30 14:04:22.442	44	21
31044	31043	2014/08/30 14:04:22.463	1012860	21728	2220	874516	9212	637952	0				
31045	31044	2014/08/30 14:04:22.527	1012860	21728	2220	874516	9212	637952	0				
31046	31045	2014/08/30 14:04:22.601	1012860	21728	2220	874516	9212	637952	0				
31047	31046	2014/08/30 14:04:22.669	1012860	21728	2220	874516	9212	637952	0				
31048	31047	2014/08/30 14:04:22.735	1012860	21728	2220	874516	9212	637952	0				
31049	31048	2014/08/30 14:04:22.800	1012860	21728	2220	874516	9212	637952	0				
31050	31049	2014/08/30 14:04:22.874	1012860	21728	2220	874516	9212	637952	0				
31051	31050	2014/08/30 14:04:22.940	1012860	21728	2220	874516	9212	637952	0				

Figure 9.9: Crash file at record 31 042

The next section describes the process followed to generate a suitable output file from the `iostat` utility.

9.3.2 The `iostat` output file

9.3.2.1 Technical set-up for `iostat` output file

As discussed previously, the `iostat` utility was used to show I/O usage on the Linux machine and potentially provide additional near-miss indicators. `iostat` displayed the following information:

- **Time:** Time information is displayed
- **TID:** Process or thread ID
- **PRIO:** Process I/O priority (class/level)
- **User:** Username running the process or thread
- **Disk read:** I/O bandwidth read by each process or thread
- **Disk write:** I/O bandwidth written by each process or thread
- **Swpin:** Percentage of time spent while swapping in
- **I/O:** Percentage of time spent while waiting on I/O
- **Command:** Name of process or thread

`iotop` was executed with the following command: `iotop -ktoqqq -d .5`. The command had the following arguments to generate a suitable output file for the forensic analysis:

- **k**: Display I/O bandwidth in kB.
- **t**: Display time. Time is used to correlate entries in the `iotop` output file with records in the crash file.
- **o**: Display only processes actually doing I/O, instead of showing all processes.
- **qqq**: Remove all headers. Headers are displayed every time the information is refreshed. By default, this happens every second.
- **-d .5**: Change refreshing time interval to 0.5 sec. This allows correlation with the crash file at a higher level of precision.

The `iotop` command and the C++ program were executed concurrently in separate terminals. As it runs continuously, `iotop` was stopped manually after the termination of the C++ program. The output of `iotop` was redirected to a file stored in the shared directory between the host and the guest machine. This output file, which was also stored in a spreadsheet format, is discussed next.

9.3.2.2 Results

Figure 9.10 shows a screenshot of the first entries in the `iotop` output file.

	A	B	C	D	E	F	G	I	J
1	Time	TID	PRIO	User	Disk read (kB/s)	Disk write (kB/s)	Swapin (%)	I/O (%)	Command
2	02:35:54	131	be/3	root	0	281.83	0.00	6.51	[jbd2/sda1-8]
3	02:35:55	3945	be/4	root	0	7.58	0.00	0.00	python /usr/sbin/iotop -ktoqqq -d .5
4	02:35:57	3950	be/4	root	1129.04	0	0.00	99.99	./videoCrashTwoFolders-V2
5	02:35:58	3950	be/4	root	1036.41	7.63	0.00	80.37	./videoCrashTwoFolders-V2
6	02:35:58	3950	be/4	root	2754.27	0	0.00	96.62	./videoCrashTwoFolders-V2
7	02:35:59	3950	be/4	root	2753.8	0	0.00	95.51	./videoCrashTwoFolders-V2
8	02:35:59	3950	be/4	root	3004.32	0	0.00	97.65	./videoCrashTwoFolders-V2
9	02:36:00	3950	be/4	root	2744.16	0	0.00	95.94	./videoCrashTwoFolders-V2
10	02:36:00	3950	be/4	root	2237.68	0	0.00	96.92	./videoCrashTwoFolders-V2
11	02:36:00	131	be/3	root	0	54.72	0.00	7.59	[jbd2/sda1-8]
12	02:36:00	3945	be/4	root	0	7.82	0.00	0.00	python /usr/sbin/iotop -ktoqqq -d .5
13	02:36:01	3950	be/4	root	2703.35	0	0.00	93.62	./videoCrashTwoFolders-V2

Figure 9.10: iotop output file

The following observations were made based on the output of `iotop`:

- As expected, the C++ program (called `videoCrashTwoFolders-V2` in Figure 9.10) was the process performing the most I/O activity. It was the most recurring one.
- A number of other processes were performing I/O activities on the Linux disk. The most recurring ones were the following:

- jbd2/sda1-8 – a journaling block device that records file system operations and runs continuously on the Linux machine (Sovani, 2013)
- flush-8:16 – a process that is used for garbage collection (Rodrigues, 2009)
- kswapd0 – a process that manages swap space (Rusling, 1999)
- The value of Swapin was 0 throughout the entire file, which corresponds with the entire execution of the C++ program. No explanation was available for this pattern.
- No particular sign of the failure was visible from the file. At the time of the failure, the values of the various attributes of `iotop` did not seem much different than throughout the rest of the file, as is clear from the highlighted area in the `iotop` output file in Figure 9.11. The highlighted area corresponds with the time frame for the point of failure in the crash file. In the crash file in Figure 9.8, the point of failure corresponds with record 31 002, which starts at 14:04:19.401 and ends at 14:04:19.793.

	A	B	C	D	E	F	G	I	J
208865	14:04:18	17 be/4	root		0	0	0.00	77.99	[kswapd0]
208866	14:04:18	3950 be/4	root		0	2903.98	0.00	74.30	./videoCrashTwoFolders-V2
208867	14:04:18	17 be/4	root		0	0	0.00	99.99	[kswapd0]
208868	14:04:18	3950 be/4	root		1.95	3124.78	0.00	96.94	./videoCrashTwoFolders-V2
208869	14:04:19	4191 be/4	root		0	0	0.00	99.99	[flush-8:16]
208870	14:04:19	3950 be/4	root		0	3315.1	0.00	94.10	./videoCrashTwoFolders-V2
208871	14:04:19	17 be/4	root		0	0	0.00	93.00	[kswapd0]
208872	14:04:19	17 be/4	root		0	0	0.00	74.77	[kswapd0]
208873	14:04:19	3950 be/4	root		650.26	459.89	0.00	61.67	./videoCrashTwoFolders-V2
208874	14:04:19	131 be/3	root		0	22.62	0.00	11.40	[jbd2/sda1-8]
208875	14:04:20	3950 be/4	root		211.08	7.82	0.00	6.46	./videoCrashTwoFolders-V2
208876	14:04:20	19110 be/4	root		0	7.82	0.00	0.00	python /usr/sbin/iotop -ktoqqq -d .5
208877	14:04:20	4191 be/4	root		0	0	0.00	99.99	[flush-8:16]

Figure 9.11: iotop output file – point of failure of C++ program

9.3.3 Summary of experiment and results

Obtaining a suitable crash file and a corresponding `iotop` output file was the goal of the experiment described in this chapter. The acquisition of the data was conducted as follows:

- It was done at runtime so it was a live forensic acquisition. As the program was running and writing data to the standard output, this data was automatically being copied to a file (crash file)
- A copy of the original crash file was made and stored on the machine used for the forensic analysis. The original data was created on a Linux machine and copied to and analysed on a Windows machine.
- Since the Linux machine was a virtual machine on the Windows machine, no transport of data was made and the image was transferred electronically from Linux to Windows.
- The forensic soundness of the image cannot be ascertained since no hash value was calculated, since the focus of the experiment was on the detection of near misses and

not on the forensic soundness of the data acquisition. This lack of a sound forensic procedure was also motivated by the fact that the experiment was conducted in a closed controlled environment, with limited risk for unauthorized access and tampering of data. Nevertheless, throughout the investigation, we made sure that we did not modify the value of the copied fields in the data set.

- Any changes to the data (layout and format) was carefully documented.
- The same procedure was conducted for acquiring data from the `iotop` command

The suitability of the crash file was defined as follows:

- It must have a large number of records.
- The records should be both successful and unsuccessful to identify the failure.
- The records should contain various relevant memory-related statistics about the C++ program and its host machine.
- The file had to be in a spreadsheet format.

The suitability of the `iotop` output file was defined as follows:

- It must provide relevant details about the I/O activity of the C++ program and of the Linux disk.
- It should be possible to correlate its entries with the records in the crash file based on time.
- The file had to be in a spreadsheet format.

As both files met all the above requirements, it is safe to say that the goal of this first experiment was achieved. These characteristics were selected to facilitate the forensic analysis of the failure and to enable the identification of near-miss indicators. The subsequent experiments conducted for this purpose are described in Chapter 10.

9.4 Conclusion

This chapter described the first experiment of the prototype implementation, the design of which was presented in Chapter 9. This experiment was conducted to obtain a suitable set of event logs of a software failure caused by memory exhaustion. The experiment involved programmatically provoking a software failure from a lack of memory and recording relevant

memory-related statistics during the program's execution. The statistics that were obtained were used as event logs.

The researcher required the event logs to conduct a forensic analysis of the software failure that would allow her to identify near-miss indicators and detect potential near misses before the failure had the opportunity to reoccur. The log files resulting from the experiment satisfied all the requirements that had been specified to facilitate the previously mentioned objectives. They were therefore deemed appropriate for conducting the remaining three experiments of the prototype implementation, as will be described in the next chapter.

CHAPTER 10

PROTOTYPING THE NMS –DETECTING NEAR MISSES AT RUNTIME

10.1 Introduction

Chapter 9 presented the four-phase plan of the NMS prototype implementation, and described the first phase of this plan. The first phase consisted of an experiment aimed at creating log files of a simulated software failure. Two large sets of supplementary logs were obtained from the experiment. Chapter 10 now describes the following three phases of the prototype implementation plan. The phases consist of a series of experiments aimed at identifying near-miss indicators from the forensic analysis of the logs and detecting near misses at runtime, based on the indicators.

The rest of the chapter is structured as follows: The prototype implementation plan is briefly reviewed in Section 10.2. The implementation of the last three phases of the plan is then documented in Section 10.3 through to Section 10.7. An evaluation of the final results is provided in Section 10.8.

10.2 Prototype implementation plan

The prototype was developed to test the viability of the adapted NMS architecture represented in Figure 8.2 in Chapter 8. The plan designed for this purpose was outlined in Figure 8.4, also in Chapter 8. For clarity, both these figures are reproduced in this chapter – in Figures 10.1 and 10.2 respectively.

Figure 10.2 shows the creation of a set of event logs in the first experiment as described in the previous chapter. This experiment corresponds with the sub-component named “Event Logs” of the Near-Miss Monitor in the architecture diagram in Figure 10.1. The three subsequent experiments that make use of these logs are described in this chapter. They focus on the

remaining components and sub-components of the NMS architecture diagram. Each experiment builds on the previous one to obtain more relevant information and ensure more usable results. These experiments involve the following actions:

- Conduct a root-cause analysis of both the crash file and the `iotop` output file and identify near-miss indicators.
- Define a near-miss formula. Although a generic near-miss formula was proposed in Chapter 7, this formula was based on a predefined performance level for the monitored system, also referred to as an SLA (Service Level Agreement). In the case of this prototype implementation, such an SLA was not available; hence the need to define a near-miss formula relevant for the software failure at hand.
- Detect potential near misses at runtime using the created formula.

The three experiments are described sequentially in the next three sections.

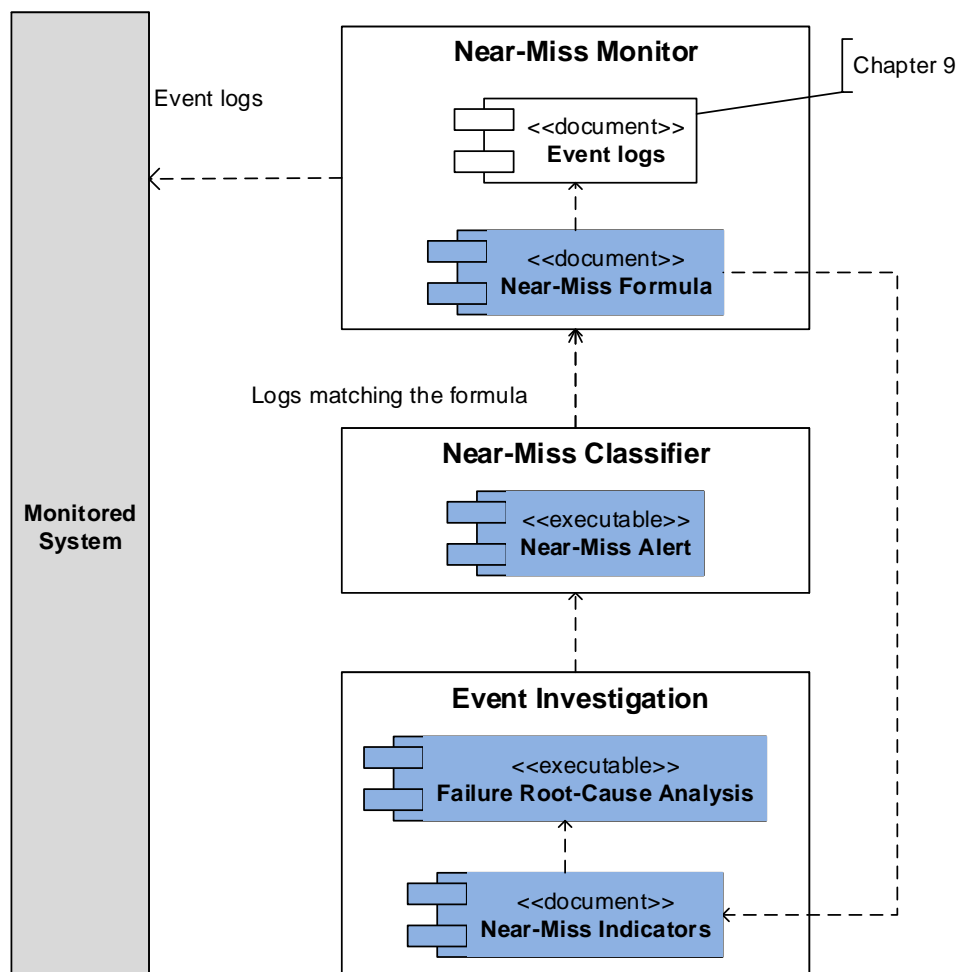


Figure 10.1: Adapted NMS component diagram for prototype implementation

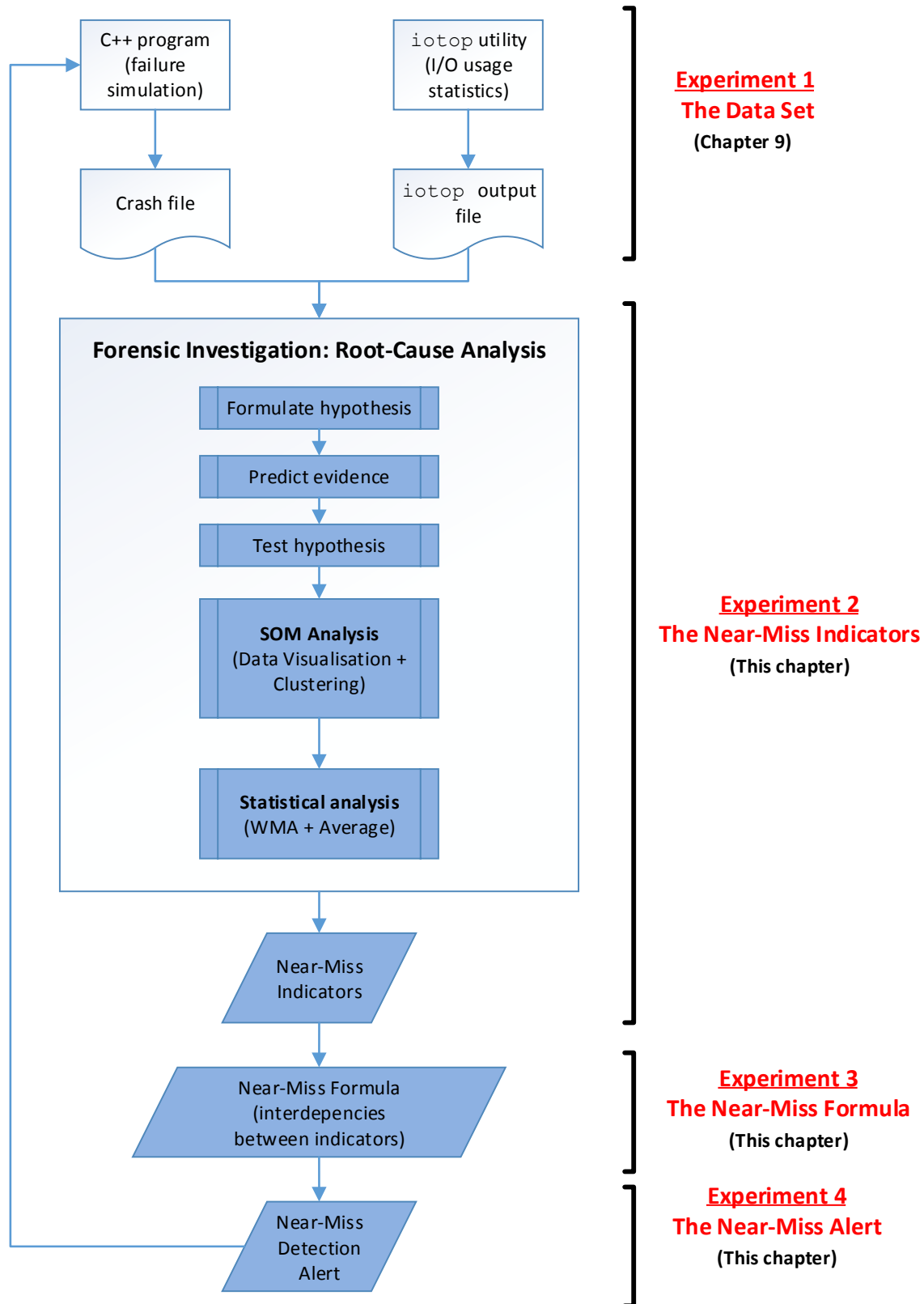


Figure 10.2: Prototype implementation plan

10.3 Experiment 2 – Part 1: Identifying near-miss indicators from the forensic analysis of the crash file

10.3.1 Goal

The goal of this step was to identify near-miss indicators that could be used to define and detect near misses from the crash file. Identifying near-miss indicators first requires the investigator to determine the root cause of the failure, and then to identify system conditions pointing to that root cause before the reoccurrence of the failure. Both the crash file and the `iostat` output file were analysed for this purpose. The focus of this phase was therefore on the Event Investigation of the adapted NMS architecture (see Figure 10.3). As the documentation of this phase is lengthy, it has been broken down into 3 parts. Part 1 (Section 10.3) describes the root-cause analysis and the near-miss identification process of the crash file, while Part 2 (Section 10.4) describes the same analysis conducted with the `iostat` output file. A summary of the overall near-miss indicators identified is provided in Part 3 (Section 10.5)

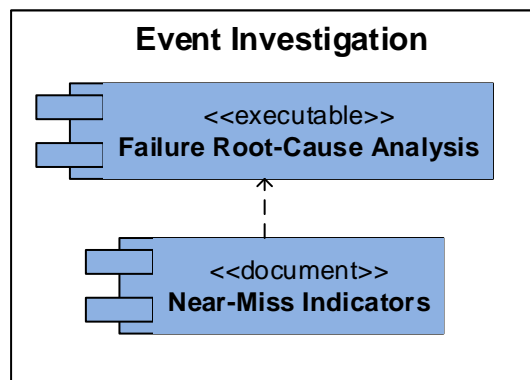


Figure 10.3: Focus of Experiment 2 – Event Investigation of adapted NMS architecture

Since the root-cause analysis was conducted with a view to identifying near-miss indicators, the whole experiment was oriented towards that purpose.

Identifying near-miss indicators was based on the assumption that it was possible to see the failure coming by monitoring the relevant memory usage statistics provided in the crash file and the `iostat` output file. Indeed, it was expected that the C++ program would have a stable operating mode under normal conditions (when enough memory was available) and that this normal behaviour would be disrupted when memory became insufficient.

Therefore, for the purpose of this experiment, the root cause was expected to effect some unusual changes in the monitored statistics close to the point of failure. These changes could be expected to indicate an upcoming failure and would be used to define near-miss indicators. The scientific method was used to pinpoint the unusual changes as documented thereafter.

Formulate hypothesis

Ideally, one would conduct a root-cause analysis without any biased opinion regarding the source of the failure. However, due to the nature of the prototype design, the source of the failure was already known to be memory exhaustion. As discussed in the previous chapter, memory exhaustion usually manifests through a performance slowdown. The analysis of the crash file therefore aimed to find evidence of this trend.

Predict evidence for the hypothesis

Symptoms of a performance slowdown in the execution of the C++ program were expected from the crash file. In addition, as memory was depleting, it was expected that activity would be observed on the Linux disk, aimed at managing a shortage in memory. The following symptoms were therefore expected:

- A longer time duration to complete a file operation
- A longer latency between two successive file operations
- An increased level of caching, buffering and swapping

These changes were expected in the last records before the failure. Based on the average duration of 1.295s to create a record, it was assumed these changes would occur in the last couple of seconds before the failure.

Test hypothesis with experiment

It was assumed that the above trend in the memory statistics would be visible from a trend analysis of the behaviour of the system (Linux machine) as the program was running. The experiment was therefore aimed at outlining the trends in the system's behaviour, both from a SOM analysis and a statistical analysis of the crash file.

10.3.2 SOM analysis of the crash file

10.3.2.1 The map creation process

The SOM analysis was performed with the commercial tool Viscovery SOMine. Little pre-processing was required as the crash file was stored as an Excel spreadsheet, which is an input file format handled by Viscovery SOMine. Details about the pre-processing and the map training and creation process are provided in Appendix 1.

Profiling the system's behaviour was performed in three steps, namely the overall system's behaviour before the failure was outlined, the shift in focus to the system's behaviour close to the point of failure, and finally a comparison between these two profiles.

10.3.2.2 Behaviour of the system before the failure

Technical set-up

In order to observe trends in the system's behaviour, the researcher created SOM maps for several random sets of 1000 records throughout the crash file, among the records marked as "OK". The argument for this strategy was that creating one single map of all the "OK" records would not provide a detailed view of the variations in the system's behaviour. Four sets of records were selected: first 1000, 10 000 to 11 000, 20 000 to 21 000 and last 1000 before the failure. The resulting output maps are shown in Table 10.1. For increased visibility, the table is spread over three pages.

In line with the expected evidence for memory exhaustion discussed earlier, the focus of the SOM analysis was on the following attributes: *Creation Time*, *Buffers*, *Cached*, *Swap Used*, *Duration* and *Latency*. Table 10.1 therefore only shows the component maps for the attributes mentioned. A brief explanation of how to read the maps is provided next.

The component maps show the distribution of the values in the data set over time for each attribute. The scale of the values is displayed on a bar below each map. Values range from lowest on the left to highest on the right of the bar. Values on the map are differentiated by their colour on the scale. So, lowest values are in blue and highest values are in red.

The map of the attribute *Creation Time* was used as the basis for understanding the distribution of values over time. On this map, the first records are in the top right corner of

the map, where the time is the earliest (lowest value, dark blue colour), and the last records are on the bottom left corner, where time is the highest (red colour). So, the values move from right to left. This topology is applicable to all component maps.

Results

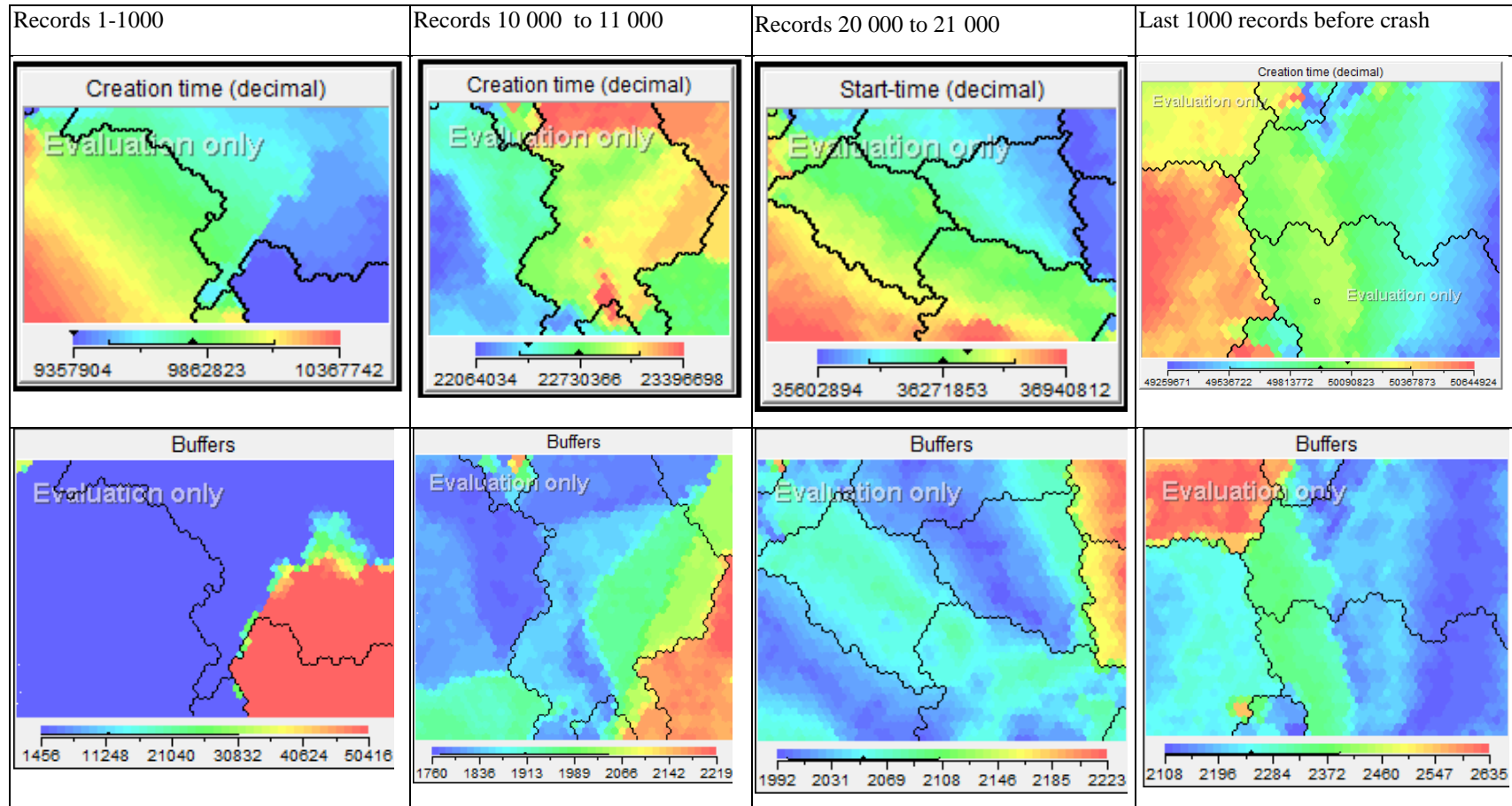
A study of Table 10.1 shows the following trends:

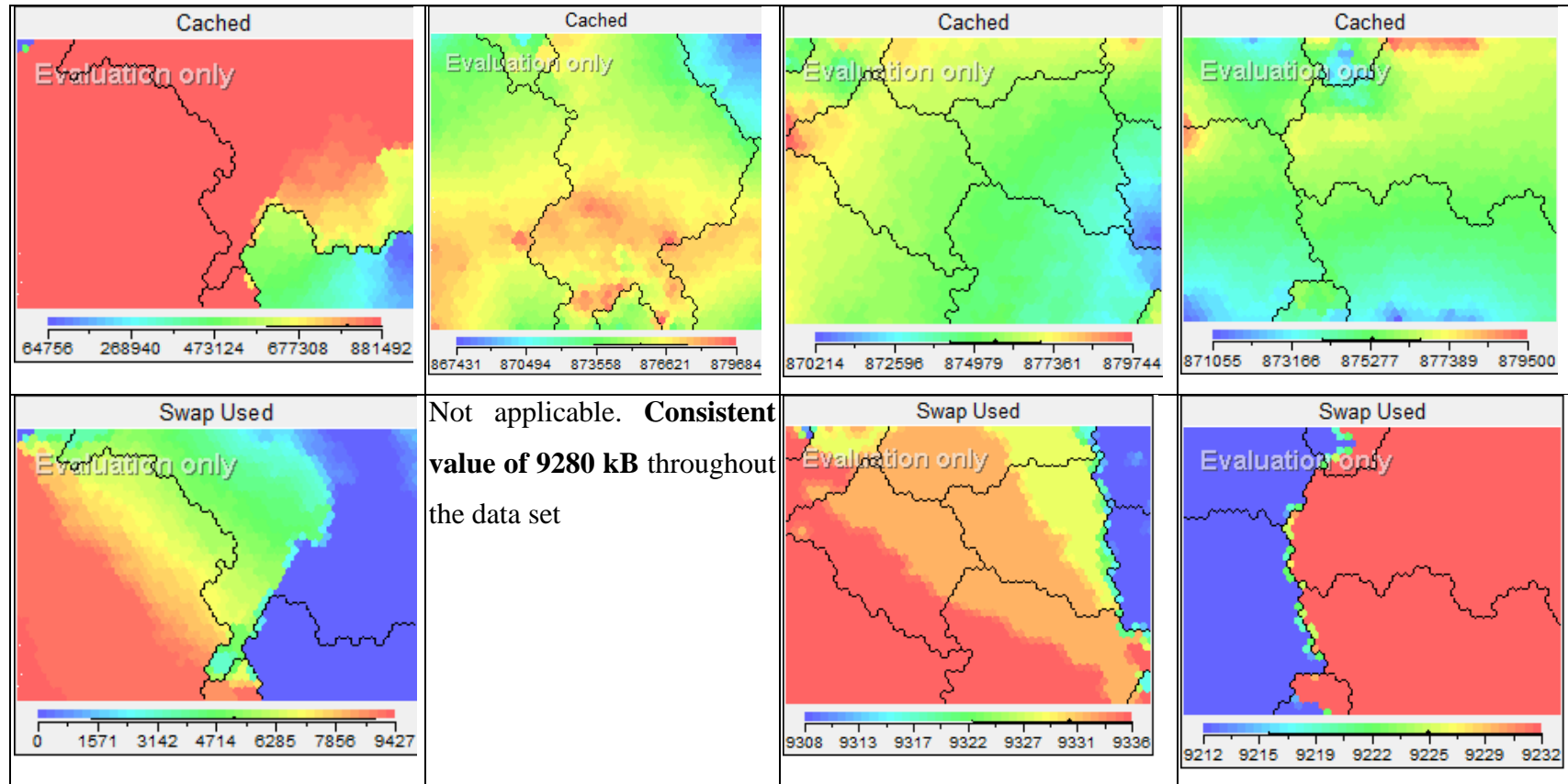
- *Latency* increases over time. The minimal value goes from 13 ms to 20 ms and finally to 33 ms. There are occasional big increases (outliers displayed in red), but the biggest increase occurs in the last data set, closer to the failure (3890 ms).
- *Duration* remains around 1000 ms, close to the average of 1295 ms, with occasional big jumps throughout the various data sets.
- *Swap Used* starts at 0, increases steadily up to 9400 kB in the first 1000 records and then remains close to that value throughout the program's execution. This stair-stepping pattern of swap usage is typical of a system under high memory pressure (Splunk wiki, 2012).
- *The value of Cached* rapidly increases from 647 000 kB to 881 000 kB and remains fairly constant around this value throughout the program's execution.
- *The value of Buffers* start high at 51 416 kB, quickly decrease up to 1456 kB in the first 1000 records and then remain between 2000 kB and 2600 kB throughout the program's execution. The maps show that the values change frequently throughout each data set, although the range of values remains very small.

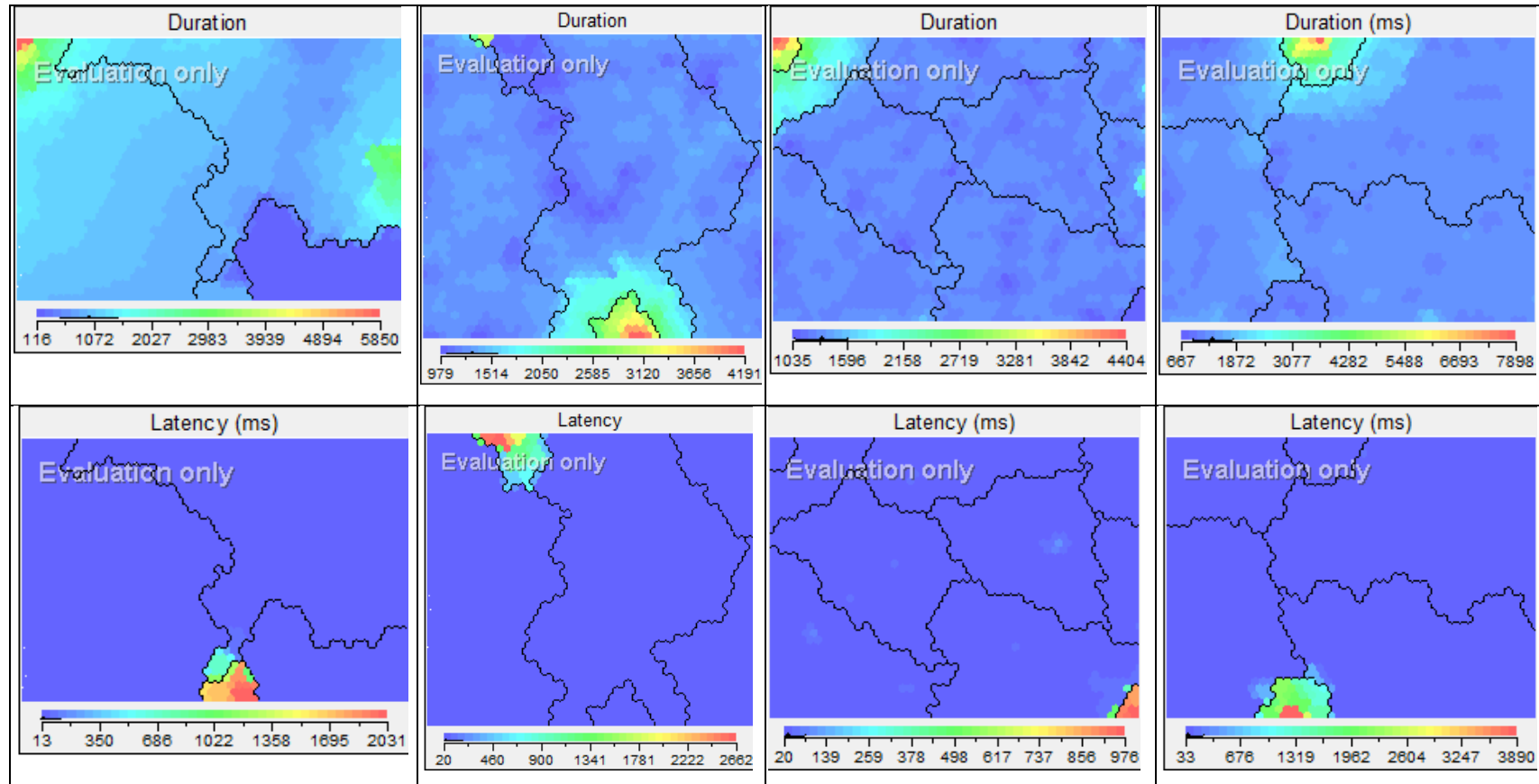
Of all the above attributes, the one that shows a distinctive change throughout the program as well as close to the failure is *Latency*. Interestingly, the assumption that big changes would be observed in the other attributes was not confirmed. Besides, no correlation between *Latency* and the other attributes was observed. For instance, an increase in *Latency* does not correspond with an increase in *Duration*.

In order to find more detailed and usable information about the observed pattern in *Latency*, the researcher conducted a SOM analysis and a statistical analysis of *Latency*. The analysis was performed with values close to the failure and is described in the next section.

Table 10.1: SOM maps of selected program attributes over time







10.3.2.3 Behaviour of Latency close to the point of failure

SOM analysis of *Latency* – Technical set-up

Various component maps of *Latency* were created for various data sets in proximity of the point of failure, before and after the first “not OK” record. For this purpose, only the first 50 “not OK” records were retained in the crash file. The following data sets were selected: last 1000 (including “not OK”), last 100 before failure, last 100 (including “not OK”) and last 50 before failure. The file numbers were added as labels on the maps to understand the distribution of values over time. The resulting SOM maps are discussed next.

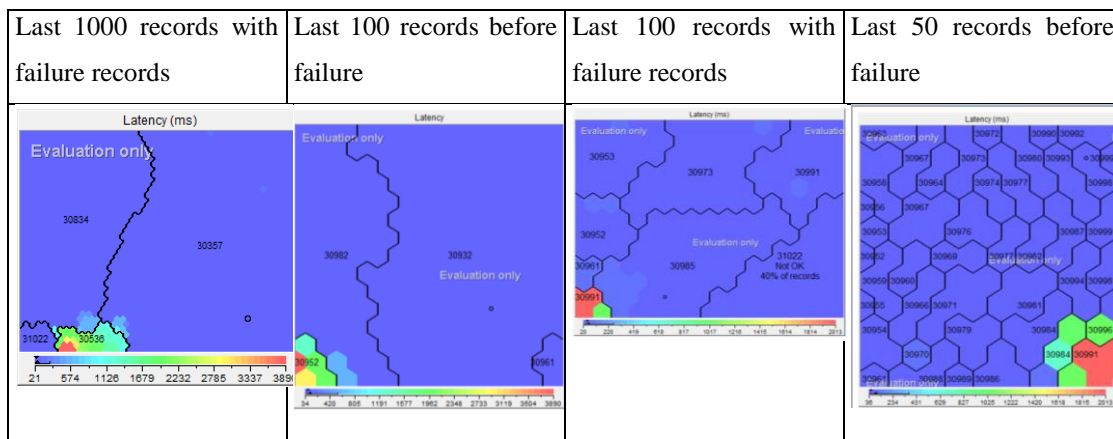
SOM analysis of *Latency* – Result

The SOM maps are shown in Table 10.2. The following observations were made:

- The lack of homogeneity close to the point of failure. The number of clusters in the last data set is considerably higher than in the previous ones. This indicates that these records are erratic in terms of the other attributes used to train the maps.
- The high value of *Latency* throughout the last 100 records before the failure. *Latency* remains mostly around 40 ms, which is much higher than the values of 13 ms to 20 ms in the first 21 000 records.

In order to model the increase in *Latency* as a potential near-miss indicator, this trend needed to be expressed in terms of some statistical parameters. A statistical analysis of *Latency* was therefore conducted next.

Table 10.2: SOM maps of *Latency* for various data sets close to the point of failure



10.3.3 Statistical analysis of *Latency*

Technical set-up

The statistical analysis was conducted as follows:

- Calculation of overall average from all the “OK” records
- Calculation of the WMA of *Latency* throughout the various data sets in Table 11.1 (first 1000, 10 000 to 11 000, 20 000 to 21 000 and last 1000 before failure)
- Comparison of overall average to WMA

In each of the above data sets, the WMA was calculated for three subsets (last 150, last 100 and last 50). This was done for this dual purpose:

- Establish the trend (increasing or decreasing) in each data set
- Establish the trend across the data sets

Result

Table 11.3 shows the results of the calculation of the WMA across the various data sets. The calculated value of the overall average was 36 ms.

Table 10.3: WMA of *Latency* across various data sets

	Records 1-1000	Records 10 000-11 000	Records 20 000 - 21 000	Last 1000 records before failure
Last 150 records	17.39	35.85	26.23	96.31
Last 100 records	17.5	31.79	26.63	96.15
Last 50 records	17.7	23.03	27.58	113.589

Table 10.3 shows that the last data set (last 1000 records before failure) differs significantly from the rest in the following two ways:

- The values of WMA are more than twice the values in the other data sets.
- The trend of values differs throughout the three subsets. In the first three data sets, the WMA moves in one direction throughout the subsets: up, down and up respectively. However, in the last data set, the WMA goes down from the first subset to the second and then up from the second subset to the third. This confirms that the records in that data set are erratic.

10.3.4 Conclusion based on forensic analysis of crash file

The conclusion reached from the above analysis of the crash file and of *Latency* was that the system did indeed slow down towards the end of the C++ program's execution. This slowdown was due to a significant increase in *Latency*. The near-miss indicator identified from this analysis was as follows: in the last 150 records before the failure, the WMA of *Latency* is more than twice its average.

After establishing a near-miss indicator from the crash file, the researcher proceeded to do the same with the `iostat` output file.

10.4 Experiment 2 – Part 2: Identifying near-miss indicators from the forensic analysis of `iostat` output file

The above analysis of the crash file confirmed the hypothesis of a system's slowdown before the failure, as well as the expectation of unusual changes in some of the recorded statistics in line with the slowdown. For this reason, some unusual I/O usage pattern was also expected from the output of `iostat`. The expected evidence for this trend was an increase in garbage collection, the termination of some processes and a drop in the C++ program's writing bandwidth (Santosa, 2006; Bytemark, 2014). This was based on the assumption that the program would take up most of the I/O bandwidth and that other running processes would compete for this resource.

The forensic analysis of the `iostat` output file followed the same process as with the crash file. Firstly, some SOM maps were created for the four data sets of 1000 records presented previously. This was followed by a statistical analysis of attributes that showed some behavioural change close to the failure.

10.4.1 SOM analysis of `iostat` output file

Technical set-up

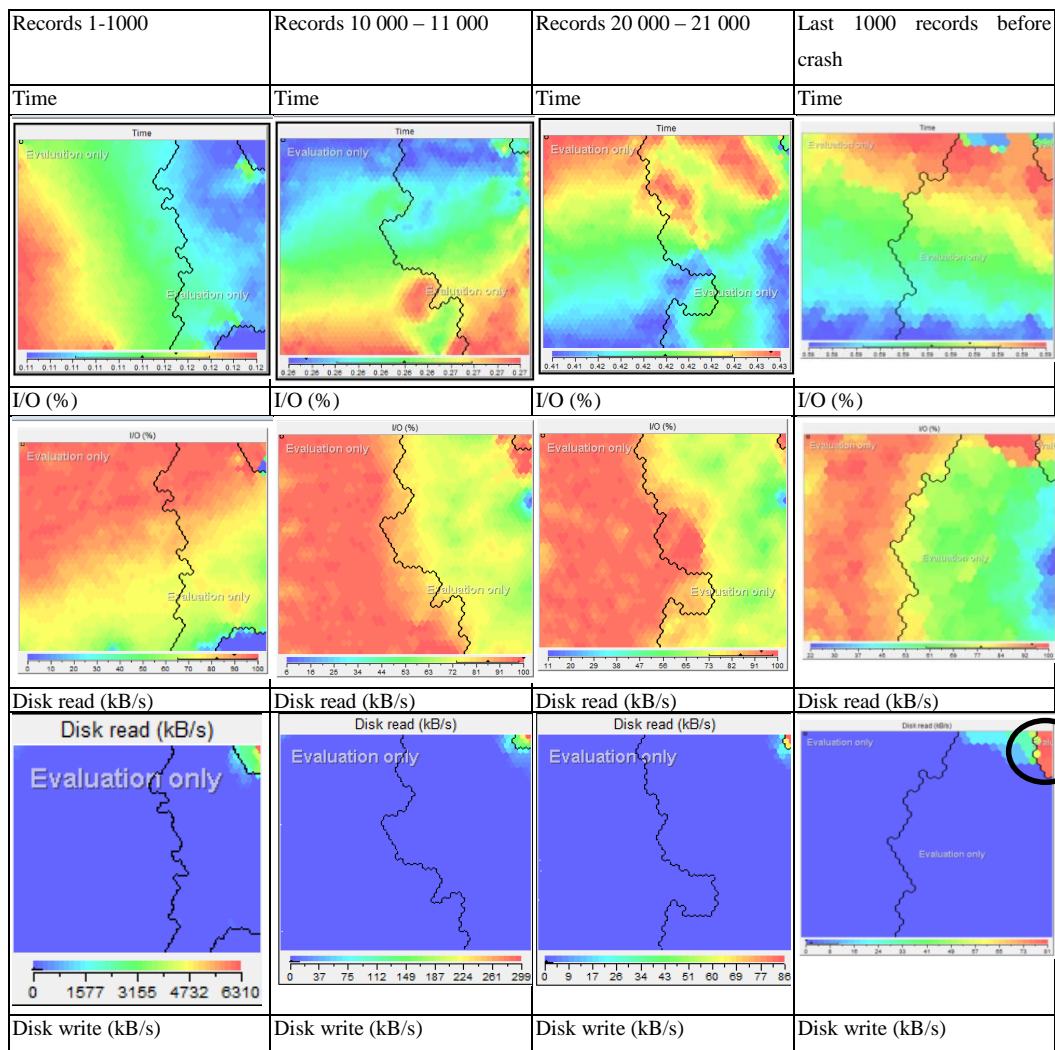
Based on the expected evidence, SOM maps were generated for the following attributes in the `iostat` output file: *Time*, *Disk read*, *Disk write*, *I/O* and *Command* (process name). As indicated in the previous chapter, no map was generated for *Swapin* as it had a constant value of 0 throughout the entire file. Records for each data set were selected based on

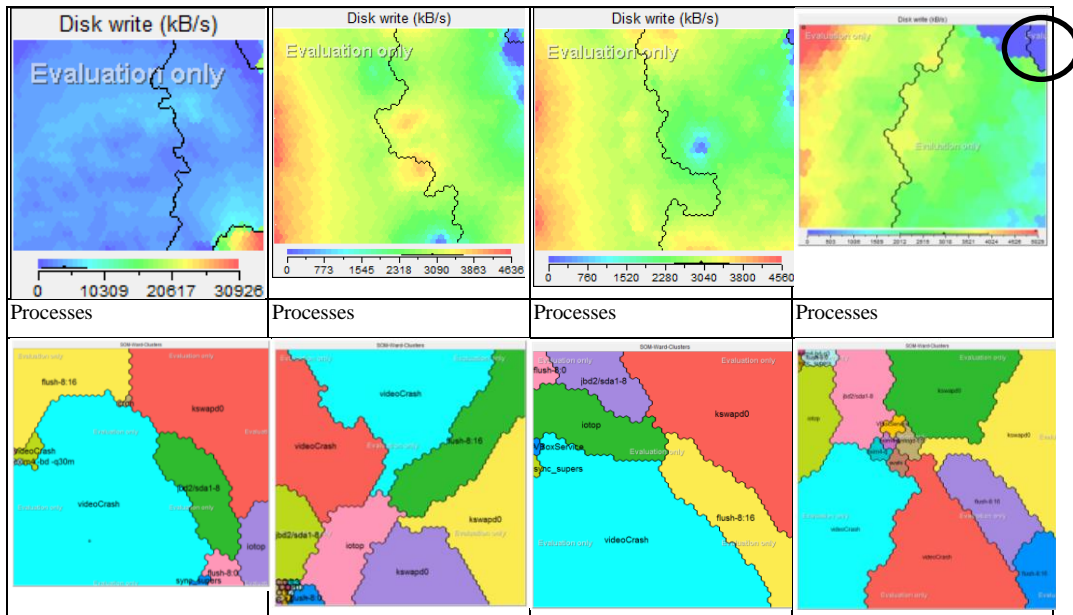
their corresponding start time in the crash file. In order to show the distribution of the various processes and threads in the file, a map was first generated with the *Command* attribute and process names were added as labels in each cluster. Then, the file was filtered to only retain records of the C++ program and a component map for each of the other attributes was generated. The result is discussed next.

Result

Table 10.4 shows the SOM maps of *iotop* for the various data sets. The table is spread over two pages.

Table 10.4: SOM maps of *iotop* output for various data sets





The following clear changes close to the point of failure are observed in all attributes, except for I/O:

- The number of processes fluctuates throughout the program's execution.
- Although *Disk read* is very homogeneous and very low throughout the program's execution, a sharp increase appears towards the end.
- *Disk write* is quite high throughout the program's execution, but a large decrease appears close to the end.
- There seems to be a correlation between the changes in *Disk read* and *Disk write* as they appear on the same area on both maps (this area is circled on both maps).

As she did with the crash file, the researcher decided to zoom in on the last 100 records before the failure to get a clearer picture of the changes observed above. The *Disk read* (DR) and *Disk write* (DW) maps were labelled with their average value in each cluster. The resulting maps are shown in Table 11.5. The following observations were made:

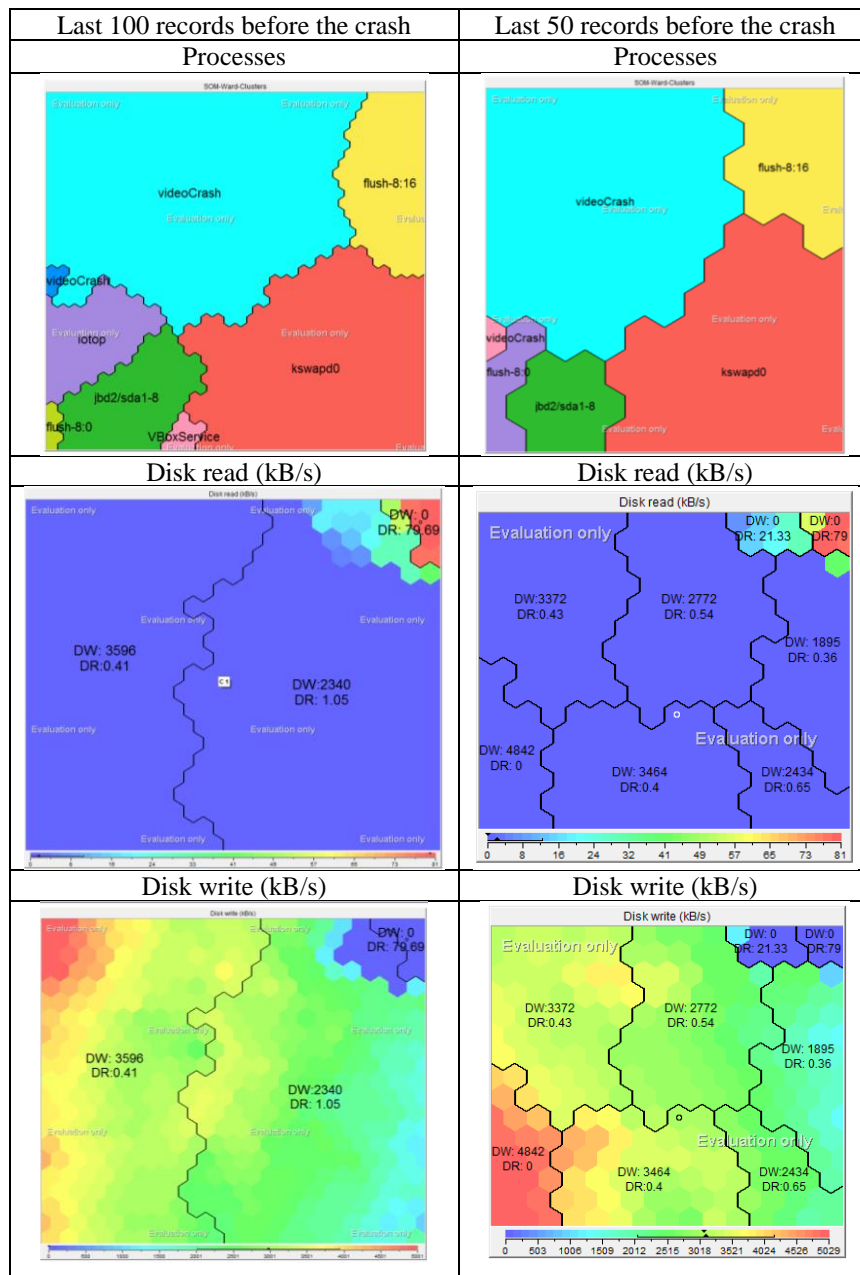
- The number of processes actually decreases in the last 50 records before the point of failure.
- *Disk read*, which is mostly around 0.5, increases sharply to 21 and 79 in the last 100 records before the failure.
- *Disk write*, which is mostly around 3000, decreases sharply to 0 in the last 100 records before the failure. A decrease in *Disk write* can explain an increase in

Latency observed in the crash file. As the disk-writing bandwidth drops, it takes longer for the system to create a new file for the next file operation.

- The number of clusters in *Disk read* and *Disk write* increases significantly in the last 50 records before the failure, showing how erratic those records are compared to the previous ones in the file.

A statistical analysis was subsequently performed to model the above behaviour formally.

Table 10.5: SOM maps of running processes, *Disk read*, and *Disk write* close to the point of failure



10.4.2 Statistical analysis of iotop output

Technical set-up

Statistical analysis was first performed on the *Command* attribute. A simple filtering on the corresponding column in Excel provided a list of the processes per data set. Then statistical analysis was performed on the other attributes (*Disk read* and *Disk write*) by calculating their average and WMA, as was done previously with the crash file. The results are discussed next.

Result

Table 11.6 shows the list of processes per data set. The table clearly shows the fluctuations in the number of processes throughout the execution of the program. As was expected, some processes terminate close to the failure. An example is VBoxService, a non-system process that launches at start up from the VBox application (virtualisation software). It is terminated in the last 50 records before the failure.

Table 10.6: List of running I/O processes throughout the program’s execution before the point of failure

Records	1-1000	10 000 – 11 000	20 000 -21 000	Last 1000 before failure	Last 100 before failure	Last 50 before failure
Processes	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> [flush-8:0] <input checked="" type="checkbox"/> [flush-8:16] <input checked="" type="checkbox"/> [iotop -ktoqqq -d .5] <input checked="" type="checkbox"/> [jbd2/sda1-8] <input checked="" type="checkbox"/> [kswapd0] <input checked="" type="checkbox"/> [sync_supers] <input checked="" type="checkbox"/> cron <input checked="" type="checkbox"/> exim4 -bd -q30m <input checked="" type="checkbox"/> exim4 -q <input checked="" type="checkbox"/> VBoxService <input checked="" type="checkbox"/> videoCrash 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> [flush-8:0] <input checked="" type="checkbox"/> [flush-8:16] <input checked="" type="checkbox"/> [jbd2/sda1-8] <input checked="" type="checkbox"/> [kswapd0] <input checked="" type="checkbox"/> [sshd: mbb [priv]] <input checked="" type="checkbox"/> [sshd: mbb@pts/0] <input checked="" type="checkbox"/> [sync_supers] <input checked="" type="checkbox"/> atd <input checked="" type="checkbox"/> cron <input checked="" type="checkbox"/> exim4 -bd -q30m <input checked="" type="checkbox"/> exim4 -q <input checked="" type="checkbox"/> iotop <input checked="" type="checkbox"/> rsyslogd -c5 <input checked="" type="checkbox"/> VBoxService <input checked="" type="checkbox"/> videoCrash 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> [flush-8:0] <input checked="" type="checkbox"/> [flush-8:16] <input checked="" type="checkbox"/> [jbd2/sda1-8] <input checked="" type="checkbox"/> [kswapd0] <input checked="" type="checkbox"/> [sync_supers] <input checked="" type="checkbox"/> iotop <input checked="" type="checkbox"/> VBoxService <input checked="" type="checkbox"/> videoCrash 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> [flush-8:0] <input checked="" type="checkbox"/> [flush-8:16] <input checked="" type="checkbox"/> [jbd2/sda1-8] <input checked="" type="checkbox"/> [kswapd0] <input checked="" type="checkbox"/> [sync_supers] <input checked="" type="checkbox"/> acpid <input checked="" type="checkbox"/> avahi-daemon: running [phd.local] <input checked="" type="checkbox"/> exim4 -bd -q30m <input checked="" type="checkbox"/> exim4 -q <input checked="" type="checkbox"/> iotop <input checked="" type="checkbox"/> rsyslogd -c5 <input checked="" type="checkbox"/> VBoxService <input checked="" type="checkbox"/> videoCrash <input checked="" type="checkbox"/> (Blanks) 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> [flush-8:0] <input checked="" type="checkbox"/> [flush-8:16] <input checked="" type="checkbox"/> [jbd2/sda1-8] <input checked="" type="checkbox"/> [kswapd0] <input checked="" type="checkbox"/> iotop <input checked="" type="checkbox"/> VBoxService <input checked="" type="checkbox"/> videoCrash 	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> [flush-8:0] <input checked="" type="checkbox"/> [flush-8:16] <input checked="" type="checkbox"/> [jbd2/sda1-8] <input checked="" type="checkbox"/> [kswapd0] <input checked="" type="checkbox"/> iotop <input checked="" type="checkbox"/> videoCrash
Number of processes	11	15	8	13	7	6

Table 10.7 shows the WMA for *Disk write* across various data sets. The overall average for *Disk write* was 3044.89 kB/s. The following observations were made from the table:

- The WMA does not decrease significantly close to the point of failure. This is contrary to the observed drop in values in the SOM maps, which indicates that the

values of 0 observed from the maps are not continuous throughout the last 1000 records but are rather scattered at some instances in the data set.

- Similarly to the observation made with *Latency* in the crash file, the last 1000 records present a different pattern than the previous data sets. Contrary to the first three data sets whose values move in one direction across their subsets (up, down, then up), the values in the last data set first move up and then stay down. This indicates a clear change in behaviour close to the point of failure, where values frequently go up and down, reaching 0 in some instances.
- In the last data set, the WMA is slightly lower than the overall average for *Disk write* (3044.89), but this trend is also applicable to the second data set (records 10 000 to 11 000), so it is not considered a reliable near-miss indicator.

Table 10.7: WMA of *Disk write* across various data sets

	Records 1 - 1000	Records 10 000 – 11 000	Records 20 000 – 21 000	Last 1000 records before failure
Last 150 records	3470.30	2970.77	3034.89	3007.70
Last 100 records	3491.87	2928.20	3047.62	3038.33
Last 50 records	3574.70	2876.71	3051.77	3020.36

Table 11.8 next shows the WMA for *Disk read* across various data sets. The overall average for *Disk read* was 1.519 kB/s. The following observations were made from the table:

- The values in the last data set are considerably higher than in the previous ones, indicating that the values of *Disk read* are increasing significantly in that data set, in line with the observations from the SOM maps.
- In the last data set, the WMA is higher than the overall average (1.519). However, as the difference between the average and the WMA is relatively small, using the difference between the average and the values of *Disk read* close to the failure (21 to 79) is a preferred potential near-miss indicator.

Table 10.8: WMA of *Disk read* across various data sets

	Records 1 - 1000	Records 10 000 – 11 000	Records 20 000 – 21 000	Last 1000 records before failure
Last 150 records	0.421588	0.633928	0.392432	1.641302
Last 100 records	0.421847	0.543243	0.3867	1.95798
Last 50 records	0.424449	0.367411	0.380561	2.89212

10.4.3 Conclusion based on analysis of *iotop* output file

From the above forensic analysis of the *iotop* output file, the following near-miss indicators were identified:

- The number of running processes declines towards the point of failure.
- The values of *Disk read* are much higher than (i.e. more than double) the overall average.
- In the last records before the failure, the value of *Disk write* drops to 0 at various instances.

10.5 Summary of Experiment 2 – Overall near-miss indicators

The current section has presented the lengthy forensic analysis conducted to identify near-miss indicators for the provoked software failure. The forensic analysis, which comprised a SOM analysis and a statistical analysis of the failure’s logs, indicated a slowdown in the execution of the program close to the point of failure. As a summary, the near-miss indicators pointing to that reduced performance were identified as follows:

- The WMA of *Latency* is greater than the overall average of *Latency*.
- The number of processes running at the beginning of the program declines close to the point of failure.
- The disk-reading bandwidth is more than twice the overall average.
- The disk-writing bandwidth drops to 0 at various instances. This pattern is not continuous until the failure, so it is only expected to appear for a few records close to the failure, but not up to the failure.

Due to the nature of the statistical analysis performed, the above near-miss indicators were mostly observed in the last 100 records before the failure. They may however have

emerged a few records earlier. It is important to note the following regarding these indicators:

- They are specific to the software failure at hand, the conditions of its occurrence (lab experiment) and forensic analysis (`iotop` used for correlation to program's logs). Therefore, they cannot be generalised to other types of software failure or to other failures due to memory exhaustion.
- The near-miss indicators cannot be used in isolation as some of the above patterns also occur during the normal functioning of the C++ program. These indicators can only point to an upcoming failure when they all occur simultaneously. This interdependency between the indicators is necessary to define a near-miss formula, which is the purpose of the next experiment.

10.6 Experiment 3: Defining a near-miss formula

10.6.1 Goal

The goal of this experiment was to define a near-miss formula based on the interdependencies among the near-miss indicators identified in the previous experiment. The near-miss formula was subsequently required to detect near misses from event logs during the execution of the C++ failure simulation program. The focus of the present phase of the prototype implementation was therefore on the Near-Miss Formula, a sub-component in the Near-Miss Monitor of the adapted NMS architecture. It is shown in Figure 10.4.

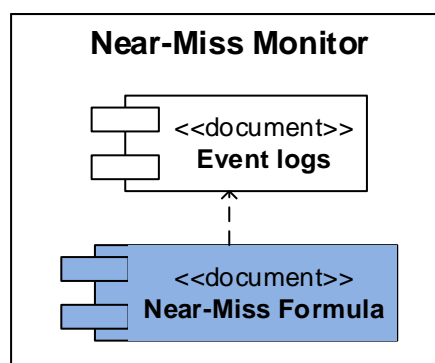


Figure 10.4: Focus of Experiment 3 – Near-Miss Formula in adapted NMS architecture

In order to provide reliable results, the near-miss formula had to be not only accurate but also relevant when executing the program with different variables. These variables were

the size of the video clip to be copied, the amount of free space on the flash disk, and the number of processes running in parallel to the C++ program.

Indeed, in the first experiment conducted to generate suitable event logs, the values for these variables had been carefully selected for the purpose of generating a maximum number of logs. This was performed to facilitate their subsequent forensic analysis. The near-miss indicators identified in the previous experiment were therefore closely tied to these values. Consequently, their accuracy and predictability had to be tested by running the program with different values for the above variables. To this end, the near-miss formula was created through a series of tests that included the following tasks:

- Modify the program to add the near-miss indicators (number of running processes, average and WMA of *Latency*) as attributes in the crash file.
- Run the program a number of times with various values for the above-mentioned variables (size of video clip, flash disk free space and number of concurrent processes).
- In every new execution of the program, verify the validity of the previously identified near-miss indicators.

A description of the above tests is provided next.

10.6.2 Adapting C++ program to calculate near-miss indicators

For the identified near-miss indicators to be observed (excluding those from `iostat`), they had to be displayed as attributes in the crash file. To this end, the C++ program was adapted to calculate them.

10.6.2.1 Adding the number of running processes

The first added attribute was the number of running processes. To confirm that this number was fluctuating as the program was running, it was first displayed at the beginning of the program before the loop. It was then queried and displayed in the loop for every record. The Linux command `ps -eLf | wc -l` was used to obtain the total number of processes and threads running on the system. Details about this command are provided in Appendix 1.

10.6.2.2 Adding the average and WMA of latency

In order to observe the trends in the values of *Latency*, the WMA of the last 200 records was displayed continuously. This was based on the results of the previous statistical analysis. For the first 200 records of the program, the “standard” WMA was calculated using all the previous values of *Latency*. Then, from record number 201, only the previous 200 records were retained to calculate the WMA.

Regarding the average of *Latency*, the overall average was calculated for every new record, using all the previous values. The motivation for this process was the fact that the average of *Latency* was not known beforehand every time the program was run. So, it was calculated as the program was running with the assumption that closer to the end of the program’s execution (before the failure), the average would stabilise to its overall final value. A potential near-miss indicator (‘WMA is higher than average’) that was observed when the final average had been reached would be less likely to be a false alarm, as the other near-miss indicators would also be applicable.

10.6.2.3 Results

Figure 10.5 shows the resulting crash file with the above additional attributes.

	A	B	H	I	J	K	L	M	N
1									
2	Total processes: 272								
3									
4	File Nr	Creation time	File Status	End time	Duration	Nr of processes	Latency	Avg-latency	WMA-latency
5	1	2014/09/30 03:19:43.338	OK	2014/09/30 03:19:43.458	120	Processes: 272			
6	2	2014/09/30 03:19:43.510	OK	2014/09/30 03:19:43.555	45	Processes: 272	latency: 51	avg: 51	WMA: 51
7	3	2014/09/30 03:19:43.569	OK	2014/09/30 03:19:43.691	122	Processes: 272	latency: 15	avg: 33	WMA: 27
8	4	2014/09/30 03:19:43.742	OK	2014/09/30 03:19:45.564	1822	Processes: 272	latency: 51	avg: 39	WMA: 39
9	5	2014/09/30 03:19:45.596	OK	2014/09/30 03:19:47.584	1987	Processes: 272	latency: 32	avg: 37	WMA: 36
10	6	2014/09/30 03:19:47.624	OK	2014/09/30 03:19:49.602	1978	Processes: 272	latency: 40	avg: 37	WMA: 37
11	7	2014/09/30 03:19:49.634	OK	2014/09/30 03:19:51.141	1507	Processes: 272	latency: 32	avg: 36	WMA: 35
12	8	2014/09/30 03:19:51.185	OK	2014/09/30 03:19:52.919	1734	Processes: 272	latency: 44	avg: 37	WMA: 37
13	9	2014/09/30 03:19:52.936	OK	2014/09/30 03:19:54.774	1838	Processes: 272	latency: 17	avg: 35	WMA: 33
14	10	2014/09/30 03:19:54.806	OK	2014/09/30 03:19:56.504	1698	Processes: 272	latency: 32	avg: 34	WMA: 33

Figure 10.5: Adapted crash file for near-miss detection

To increase visibility, the researcher hid the columns showing the memory statistics as they proved irrelevant for detecting near misses. The initial number of running processes is displayed in the top left corner of the file, before the new records get generated.

Once the attributes for the near-miss indicators had been added to the crash file, the program was executed under different conditions (variables) to verify whether the near-miss indicators would still apply.

10.6.3 Changing running conditions of C++ program

Firstly, the size of the video clip was changed. As the original video clip was the smallest at hand (3.91 MB), a slightly bigger one was used next (5.97 MB). Then, the size of the flash disk was changed. Since the initial flash disk was the biggest available (128 GB), a smaller one (8 GB) was used. The flash disk was therefore capable of holding a maximum of 1371 (8GB/5.97MB) copies of the video clip. The C++ program was executed with a loop size of 1800, to cause a deliberate failure.

In order to verify the validity of the near-miss indicators in terms of the disk-reading and -writing bandwidth, the `iostat` command was executed in parallel to the C++ program. (As a reminder, these indicators are as follows: the values of *Disk read* are more than twice the average and *Disk write* is equal to 0.) The results are discussed next.

10.6.3.1 Results from the crash file

The failure point of the resulting crash file is displayed in Figure 11.6, in the area highlighted in red at the bottom of the screen. The failure occurred at record 1110. Figure 11.6 also shows that in the last 33 records before the failure (from record 1077, highlighted in yellow at the top of the screen), the WMA of *Latency* is indeed greater than the average. Another observation is the fact that the average has stabilised to its overall final value (40), as expected.

10.6.3.2 Results from the `iostat` output file

The calculated overall average of *Disk read* was 0.91. Figure 11.8 shows the records in the `iostat` output file that match both near-miss indicators in terms of *Disk read* and *Disk write*. Only eight records matched the indicators.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	03:19:44	3427 be/4	mbb	.X	K/s	11.56 K/s	0 %	0 %	gnome-terminal							
199	03:29:46	25576 be/4	root	1.91 K/s	0 K/s	0 %	99.99 %	./NMDetection-bigVideo-SmallUsb								
263	03:33:37	25576 be/4	root	7.65 K/s	0 K/s	0 %	99.99 %	./NMDetection-bigVideo-SmallUsb								
329	03:37:35	25576 be/4	root	26.9 K/s	0 K/s	0 %	99.99 %	./NMDetection-bigVideo-SmallUsb								
465	03:45:02	25576 be/4	root	30.99 K/s	0 K/s	0 %	99.99 %	./NMDetection-bigVideo-SmallUsb								
604	03:53:37	25576 be/4	root	34.74 K/s	0 K/s	0 %	97.62 %	./NMDetection-bigVideo-SmallUsb								
605	03:53:38	25576 be/4	root	35.49 K/s	0 K/s	0 %	99.99 %	./NMDetection-bigVideo-SmallUsb								
606	03:53:39	25576 be/4	root	32.47 K/s	0 K/s	0 %	99.99 %	./NMDetection-bigVideo-SmallUsb								
607	03:53:40	25576 be/4	root	32.31 K/s	0 K/s	0 %	99.24 %	./NMDetection-bigVideo-SmallUsb								
630																

Figure 10.8: `iostat` output file – records matching near-miss indicators in terms of *Disk read* and *Disk write*

Using the time of the entries in the `iostat` output file, the above records were manually correlated to the records in the crash file matching the near-miss indicator for *Latency*. The result of this correlation is shown in Figure 11.9. Only three records actually matched the three near-miss indicators. For these records the number of running processes is the same as the initial number. Out of these three records, only the last one – record 1078 – is close to the point of failure (32 records or 1min 7s 625 ms before the failure). Although this makes the other two records false alarms, this low number of false alarms is satisfactory and proves the validity of the near-miss indicators.

	A	B	E	F	G	H	I	J	K	L	M	N
1	331	2014/09/30 03:29:46.578	18880	4777360	298585	OK	2014/09/30 03:29:48.222	1644	Processes: 272	latency: 509	avg: 40	WMA: 43
2	455	2014/09/30 03:33:37.215	18880	4017488	251093	OK	2014/09/30 03:33:39.089	1874	Processes: 272	latency: 108	avg: 39	WMA: 41
3	1078	2014/09/30 03:53:35.712	17299	199744	12484	OK	2014/09/30 03:53:42.017	6304	Processes: 272	latency: 38	avg: 40	WMA: 41
4												

Figure 10.9: Records in crash file that match three near-miss indicators

The above test was conducted several more times, each time changing the size of the video clip or the amount of free space on the flash disk. The test was also conducted without running the `iostat` command, to see whether this would affect the fluctuations in the number of processes running. Results similar to the one described above were obtained. The near-miss indicators for *Latency*, *Disk read* and *Disk write* would match a few records close to the failure but the number of processes would either be smaller than or equal to the initial number. The conclusion was that this pattern was indeed the proper near-miss indicator for that attribute.

Consequently, the final near-miss indicators used to define the near-miss formula were as follows:

- The WMA of *Latency* is greater than the average of *Latency*.
- The number of running processes before the failure is less than or equal to the initial number at the beginning of the program's execution.
- The disk-reading bandwidth is more than twice its overall average.
- The disk-writing bandwidth drops to 0 at various instances.

A fuzzy cognitive map (FCM) of the above near-miss indicators is provided in Figure 10.10. FCMs are fuzzy graph structures that depict perceived relationships between attributes of a complex system (Coetzee & Eloff, 2006). Knowledge about the relationships is based on human common sense and intuition (Smith & Eloff, 2002). An FCM consists of nodes connected by annotated arrows (Heydebreck, Klofsten & Krüger, 2011).

Figure 10.10 shows the causal relationship between memory availability and the four factors mentioned above that affect the system's performance and are used to define near-miss indicators: latency, number of running processes, disk-reading bandwidth (DR), and disk-writing bandwidth (DW). The type of relationship is indicated by a sign on the arrow as follows:

- A plus (+) sign indicates a positive relationship, where a higher value in one factor prompts a higher value in the connected factor (or a lower value in A prompts a lower value in B). Example: the higher the free memory, the higher the number of processes that can run simultaneously; or the lower the free memory, the lower the disk-writing bandwidth.
- A minus (-) sign indicates a negative relationship, as opposed to a positive relationship. Example: the lower the disk-writing bandwidth, the higher the latency; or the lower the number of processes, the higher the disk-reading bandwidth.

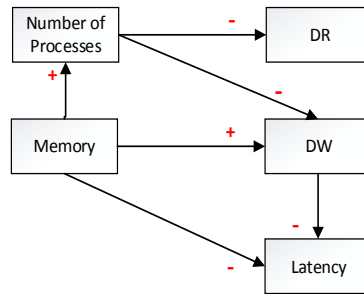


Figure 10.10: FCM of factors in near-miss indicators

Based on the above tests, the formula to detect potential near misses was defined as follows:

If Nr-Processes \leq Initial-Nr-Processes AND
 WMA-latency $>$ Avg-latency AND
 DR $>$ (Avg-DR \times 2) AND
 DW == 0
 \Rightarrow Near Miss

Figure 10.11: Near-miss formula

The above formula was used in the next and last experiment of the prototype implementation to detect near misses during the program's execution. This process is documented in the next section.

10.7 Step 4: Detecting near misses at runtime

10.7.1 Goal

The goal of this experiment was to verify whether near misses could be detected during the execution of the program by using the formula developed in the previous experiment. Once potential near misses had been detected, an alert would be sent. The focus of this phase was therefore on the Near-Miss Classifier of the adapted NMS architecture, as is shown in Figure 10.12.

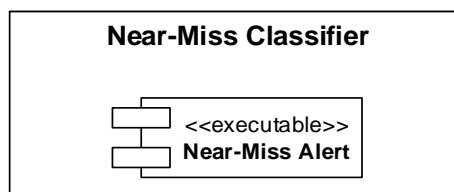


Figure 10.12: Focus of Experiment 4 - Near-Miss Classifier in adapted NMS architecture

10.7.2 Technical set-up

The near-miss formula was inserted in the program's loop after the calculations of all the necessary attributes. So, in every iteration of the loop, the program checked whether the values of these attributes matched the formula. If a match was found, the program displayed a notification message with some suggestion to prevent the failure. As discussed in Chapter 8, preventing the failure was outside the scope of this research, which means that no attempt was made to implement the suggested countermeasures.

Implementing the complete formula proved challenging as the attributes *Disk read* and *Disk write* were obtained from a source external to the program, namely the `iostat` command. Due to technical constraints in the output of `iostat`, the researcher was not able to introduce these two parameters in the formula into the program. The near-miss formula was therefore implemented in the program without the `iostat` attributes, with the researcher being fully aware that this would affect the output of this formula. The program's code is shown in Figure 10.13.

```
/******NEAR-MISS DETECTION FORMULA*****  
if ((WMA > avg) && (pCountLoop <= pCountInitial))  
{cout << "Near-miss alert! consider adding external memory to prevent the crash." << "\t";}  
}
```

Figure 10.13: Program code for near-miss formula

This formula uses the following variable names:

- WMA: WMA of *Latency*
- avg: average of *Latency*
- pCountLoop: count (number) of running processes for the current record
- pCountInitial: count of processes at the beginning of the program

10.7.3 Results

The crash file was generated with 21 829 “OK” records, a loop size of 22 000, and the original video clip. As expected, this reduced formula matched a number of records (10 660) at various instances in the crash file, not all close to the failure, as the *Disk read* and *Disk write* attributes were not included in the formula. However, a manual correlation with the near-miss indicators in the `iostat` output file reduced the number of near-miss

alerts to 162. If the complete formula had been applied, near-miss alerts would have been generated only for those 162 records.

The first of these alerts starts at record 5 742, indicating that the near-miss formula does not apply to the early records in the file, as was expected (see Figure 10.14). Only 9% of the near-miss alerts (13) appear in the first half of the program’s execution and are highlighted in Figure 10.14. The remaining 91% of the near-miss alerts are generated in the second half of the program’s execution before the failure. This again confirms that the near-miss indicators mostly emerge close to the failure. Figure 10.15 shows the last alerts in the crash file and indicates that the last alert (highlighted in red) was generated 6s 872 ms before the failure. This confirms the validity of the formula and demonstrates the feasibility of detecting near misses at runtime.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	5742	2014/10/25 02:08:48.463	OK	2014/10/25 02:08:50.342	1878	Processes:277	latency:29	avg:44	WMA:46	Near-miss alert! Consider adding external memory to prevent the crash.						
2	6837	2014/10/25 02:29:53.829	OK	2014/10/25 02:29:55.242	1413	Processes:277	latency:979	avg:44	WMA:45	Near-miss alert! Consider adding external memory to prevent the crash.						
3	7727	2014/10/25 02:47:17.389	OK	2014/10/25 02:47:18.891	1501	Processes:277	latency:943	avg:44	WMA:51	Near-miss alert! Consider adding external memory to prevent the crash.						
4	7759	2014/10/25 02:47:54.718	OK	2014/10/25 02:47:56.663	1945	Processes:277	latency:469	avg:44	WMA:53	Near-miss alert! Consider adding external memory to prevent the crash.						
5	7906	2014/10/25 02:50:44.964	OK	2014/10/25 02:50:47.034	2070	Processes:277	latency:51	avg:44	WMA:49	Near-miss alert! Consider adding external memory to prevent the crash.						
6	8159	2014/10/25 02:55:45.445	OK	2014/10/25 02:55:47.335	1889	Processes:277	latency:40	avg:45	WMA:48	Near-miss alert! Consider adding external memory to prevent the crash.						
7	8748	2014/10/25 03:27:46.016	OK	2014/10/25 03:27:47.173	2021	Processes:277	latency:27	avg:45	WMA:49	Near-miss alert! Consider adding external memory to prevent the crash.						
8	9784	2014/10/25 03:27:46.016	OK	2014/10/25 03:27:47.173	1156	Processes:277	latency:28	avg:45	WMA:47	Near-miss alert! Consider adding external memory to prevent the crash.						
9	10339	2014/10/25 03:39:00.356	OK	2014/10/25 03:39:01.710	1354	Processes:277	latency:897	avg:45	WMA:49	Near-miss alert! Consider adding external memory to prevent the crash.						
10	10695	2014/10/25 03:46:03.994	OK	2014/10/25 03:46:04.218	224	Processes:277	latency:1028	avg:45	WMA:53	Near-miss alert! Consider adding external memory to prevent the crash.						
11	10703	2014/10/25 03:46:11.787	OK	2014/10/25 03:46:13.706	1918	Processes:277	latency:30	avg:45	WMA:54	Near-miss alert! Consider adding external memory to prevent the crash.						
12	10728	2014/10/25 03:46:44.472	OK	2014/10/25 03:46:57.263	1292	Processes:277	latency:48	avg:45	WMA:52	Near-miss alert! Consider adding external memory to prevent the crash.						
13	10858	2014/10/25 03:49:34.720	OK	2014/10/25 03:49:36.331	1611	Processes:277	latency:802	avg:45	WMA:53	Near-miss alert! Consider adding external memory to prevent the crash.						
14	11037	2014/10/25 03:53:11.176	OK	2014/10/25 03:53:12.821	1645	Processes:277	latency:750	avg:45	WMA:50	Near-miss alert! Consider adding external memory to prevent the crash.						
15	11392	2014/10/25 04:00:19.893	OK	2014/10/25 04:00:20.077	184	Processes:277	latency:936	avg:45	WMA:48	Near-miss alert! Consider adding external memory to prevent the crash.						
16	11404	2014/10/25 04:00:33.232	OK	2014/10/25 04:00:35.124	1893	Processes:277	latency:37	avg:45	WMA:48	Near-miss alert! Consider adding external memory to prevent the crash.						
17	11418	2014/10/25 04:00:51.183	OK	2014/10/25 04:00:51.375	191	Processes:277	latency:194	avg:45	WMA:53	Near-miss alert! Consider adding external memory to prevent the crash.						

Figure 10.14: First near-miss alerts in the crash file

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
154	21027	2014/10/25 07:25:24.775	OK	2014/10/25 07:25:24.970	195	Processes:277	latency:505	avg:53	WMA:63	Near-miss alert! Consider adding external memory to prevent the crash.						
155	21127	2014/10/25 07:27:46.862	OK	2014/10/25 07:27:50.835	3973	Processes:277	latency:979	avg:53	WMA:73	Near-miss alert! Consider adding external memory to prevent the crash.						
156	21130	2014/10/25 07:27:56.637	OK	2014/10/25 07:27:56.858	221	Processes:277	latency:798	avg:53	WMA:76	Near-miss alert! Consider adding external memory to prevent the crash.						
157	21167	2014/10/25 07:28:52.847	OK	2014/10/25 07:28:53.037	190	Processes:277	latency:270	avg:53	WMA:80	Near-miss alert! Consider adding external memory to prevent the crash.						
158	21248	2014/10/25 07:30:33.329	OK	2014/10/25 07:30:33.419	190	Processes:277	latency:949	avg:53	WMA:68	Near-miss alert! Consider adding external memory to prevent the crash.						
159	21699	2014/10/25 07:41:02.841	OK	2014/10/25 07:41:07.988	5146	Processes:277	latency:947	avg:54	WMA:62	Near-miss alert! Consider adding external memory to prevent the crash.						
160	21757	2014/10/25 07:42:32.176	OK	2014/10/25 07:42:33.974	1799	Processes:277	latency:1113	avg:54	WMA:70	Near-miss alert! Consider adding external memory to prevent the crash.						
161	21782	2014/10/25 07:43:06.914	OK	2014/10/25 07:43:08.871	1957	Processes:277	latency:43	avg:54	WMA:77	Near-miss alert! Consider adding external memory to prevent the crash.						
162	21787	2014/10/25 07:43:12.847	OK	2014/10/25 07:43:14.886	2039	Processes:277	latency:63	avg:54	WMA:77	Near-miss alert! Consider adding external memory to prevent the crash.						
163	21823	2014/10/25 07:44:01.016	OK	2014/10/25 07:44:02.114	1098	Processes:277	latency:27	avg:54	WMA:73	Near-miss alert! Consider adding external memory to prevent the crash.						
164	21830	2014/10/25 07:44:08.986	Not OK	2014/10/25 07:44:09.971	986											
165																

Figure 10.15: Last near-miss alerts in the crash file

10.8 Evaluation of prototype implementation

10.8.1 Benefits

The prototype implementation was successful in the sense that it achieved the goals specified in its design in Chapter 8:

- Demonstrate the viability of the digital forensic process formulated in Chapter 4 to conduct a root-cause analysis of a software failure.
- Demonstrate the viability of detecting near misses at runtime.

In addition, it also showed the following:

- Near-miss detection indeed reduces the amount of relevant digital evidence that needs to be collected for root-cause analysis. This was one of the main statements of this thesis. Out of the 13 initial attributes in the crash file and the 9 attributes in the `iotop` output file, only four (*Latency*, processes, disk-reading bandwidth and disk-writing bandwidth) proved relevant for near-miss detection. This makes a significant data reduction possible. Since the four attributes are the most relevant for the root-cause analysis of the failure at hand, the collection effort could be limited to these attributes after a near-miss alert.
- In order to detect near misses, indicators of an upcoming failure need to be identified. A forensic analysis of the failure logs is a promising approach. The forensic analysis can provide both the root cause of the failure and its near-miss indicators.

Furthermore, the prototype implementation also provided an objective and effective method for conducting a forensic analysis of the logs of a software failure. Although the prototype was designed to demonstrate near-miss detection for failures due to resource exhaustion, the forensic analysis approach used is applicable to any type of software failure as long as logs providing relevant information about the system's operations are available. Although the techniques used were applied in a controlled lab environment, they are applicable to a real-world scenario with no prior knowledge of the root cause of the failure. The validity of the resulting near-miss formula demonstrates the reliability of the techniques, in other words the SOM analysis followed by a statistical analysis of the observed pattern. Since the SOM algorithm is optimised for large data sets, it is expected that this process can scale to a real-life failure with a higher number of logs than was used in the prototype. The detailed process is shown in Figure 10.16.

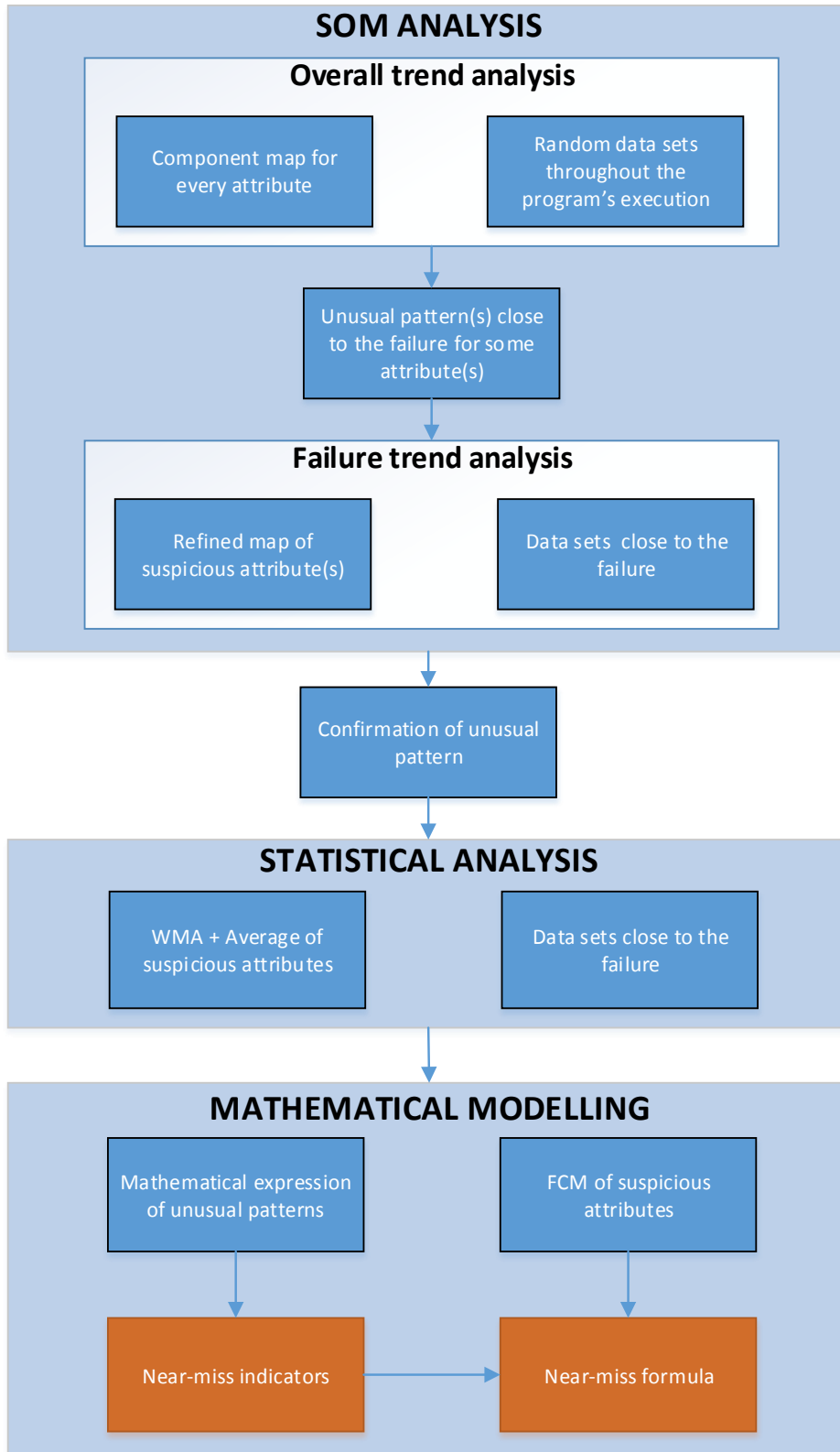


Figure 10.16: Proposed method for forensic analysis and identification of near-miss indicators

10.8.2 Limitations

Despite its benefits, the prototype implementation also had a number of limitations:

- Some limitation was encountered in the implementation of the complete formula as not all near-miss indicators could be detected simultaneously, due to their diverse sources. This resulted in a high number of false alarms. However, this problem was mitigated by a manual correlation between the two log files used to define the interdependencies between the near-miss indicators.
- Even after a manual correlation between all the near-miss indicators, a number of false alarms still remained. This could potentially be addressed by a prioritisation mechanism as proposed in Chapter 6.
- Although the created near-miss formula proved effective, it is not a direct application of the general formula presented in Chapter 6. As discussed in Section 10.2, the general formula was based on the concept of an SLA for the monitored system and such an SLA was not available for this prototype. However, a parallel between the two formulas can be drawn as follows.
 - Generally, failures cause downtime; hence downtime was used to detect near misses in the general formula. In the case of the current prototype, the failure does not result in downtime but in a performance slowdown. Indicators for this slowdown were therefore used to detect near misses.
 - Just like in the general formula, a threshold could have been set differently to specify the desired closeness of a near miss to the failure. In the prototype implementation, the threshold was implicitly set as the last 200 records before the failure in the calculation of the WMA for *Latency*. This threshold could have been higher (e.g. last 500 records) or lower (e.g. last 50 records), which would have resulted in an earlier or later generation of near-miss alerts.

10.9 Conclusion

This chapter was the last of a three-part series describing the prototype implementation of the NMS architecture. It followed on the previous two chapters that described the design phase and the data set (log files of a provoked software failure) used for prototyping the NMS architecture. The prototype was implemented to demonstrate the

viability of the NMS architecture, more specifically the detection of near misses at runtime.

Chapter 10 described the experiments that were conducted to analyse the data set obtained in the previous chapter and detect near misses based on this analysis. These three experiments included the following: (1) the forensic analysis of the data set to identify near-miss indicators; (2) the creation of a near-miss formula based on the indicators; and (3) the detection of near misses based on the formula.

The prototype implementation demonstrated the viability of the proposed NMS architecture through the successful detection of near misses from event logs before the occurrence of a failure. Although a number of false alarms were generated, their low number did not affect the validity of the results. However, certain limitations were encountered in the simultaneous detection of all the near-miss indicators, due to their diverse sources.

The prototype also demonstrated the suitability of the forensic process for conducting a root-cause analysis of software failure and the effectiveness of this process in identifying near-miss indicators. More importantly, the prototype showed how effectively near-miss detection could be used to help select the most complete and relevant digital evidence of a software failure for purposes of accurate root-cause analysis. Thus, the main claim of the current research was fully validated.

CHAPTER 11

CONCLUSION

11.1 Introduction

This study discussed the limitations of current approaches to failure analysis and proposed an original architecture of a near-miss management system (NMS) to address these shortcomings. The key aspect of the proposed architecture is a mathematical model developed to define, detect and prioritise near misses from a software system perspective. In the previous chapter, a prototype of the architecture, which focused on the detection of near misses, was implemented to test its viability. This last chapter now revisits the research question, and evaluates whether and how the goal of the research has been achieved. Finally, the main contributions of the research and recommendations for future work are discussed.

11.2 Revisiting the problem statement

The main focus of this research was to address the lack of accuracy of current failure analysis practices in identifying the root cause of a major software failure. The solution that was proposed required that sound digital evidence of the failure be used as the basis for the root-cause analysis. To this end, digital forensics was suggested as a reliable and scientific method to accurately analyse such digital evidence. Additionally, near-miss analysis was presented as a novel approach to enable the collection of relevant and complete digital evidence and hence limit the collection of irrelevant or incomplete data.

Since the reactive approach of digital forensics only allows for data collection after a failure, there is always the potential risk that the data will be lost or damaged due to the crash and will therefore be incomplete. In addition, the common practice to log all system activity, irrespective of its relevance for failure analysis, leads to a situation where lots of irrelevant data is collected.

The main claim of this research was therefore formulated as follows:

Near-miss analysis can help identify and collect more relevant and complete digital evidence of a software failure. This has the potential to improve the accuracy of the ensuing forensic analysis of the failure.

11.2.1 Answering the main and secondary research questions

As near-miss analysis is usually conducted through NMS's, the goal of the research was to design an NMS by answering the following main research question:

What should the architecture for a near-miss management system look like such that it can improve the completeness and relevance of digital evidence of a software failure, thereby improving the accuracy of its forensic analysis?

The above question was answered through two sub-questions. The following is a discussion on how each sub-question was addressed in the study.

- *How can the methodology of digital forensics be applied to the investigation of software failures?*

The main differences between digital forensics and less formal investigation methods involve the use of scientific methods and techniques and the adherence to legal principles.

The scientific methodology of digital forensics was reviewed in Chapter 3 and addressed primarily the scientific method and mathematical analysis. Detailed examples of how these can be applied to the investigation of software failures were provided with real-life cases of software failures (as discussed in Chapter 2).

The scientific method was applied to a real-life software failure case to illustrate how it can be applied to the investigation of software failures. The scientific method was contrasted to the troubleshooting approach, as the literature review on recent cases of software failures points to troubleshooting as the most common immediate response to software failures. One significant difference between the scientific method and troubleshooting was found to be the process of falsification used in the scientific method. Falsification ensures that, besides the initial hypothesis regarding the root cause of the failure, alternative hypotheses are explored before concluding the investigation. This strategy promotes the comprehensiveness of the investigation, which is in sharp contrast

to troubleshooting, as the latter stops the investigation as soon as a plausible hypothesis (usually the most obvious or trivial one) has been confirmed, without considering any other possibility. As demonstrated with the reviewed cases of software failures, this often results in erroneous root-cause identification. The process of falsification was therefore identified as a valuable addition to failure analysis.

Mathematical analysis was also found to be an important element of a digital forensic investigation as it is used to authenticate the evidence and verify its integrity. It is primarily used through hash values of the collected files that are used as digital evidence of the event being investigated. Mathematical analysis can also be applied to the evidence of a software failure and is particularly valuable if the failure leads to legal proceedings.

The prototype implementation furthermore demonstrated the use of other scientific techniques for the analysis of the evidence of a failure, namely self-organising maps (SOM) and statistical analysis.

The legal principles adhered to by digital forensics are applied through best practice in handling the digital evidence, such as maintaining the chain of custody and ensuring the transparency of the investigation through an audit trail of the process followed. This differs significantly from the current practice in failure analysis where the investigation process is rarely made public – potentially giving rise to the assumption that it was not carefully documented. Best practice in digital forensics can also be applied to the investigation of software failures and allows for the investigation to be reproduced and verified independently.

Besides adhering to scientific methodology and legal principles, digital forensics also follows a distinctive standard investigation process. Since this process needs to be adjusted to the specific requirements of failure analysis, an adapted forensic investigation process was proposed for software failures. This process added a system restoration phase to the standard digital forensic process, which occurred after the data collection and before the data analysis in order to minimise system downtime. As downtime is very costly in the case of a major software failure, it is imperative to ensure that the forensic investigation does not unnecessarily prolong the downtime duration.

- *How can near-miss analysis be applied to the software industry effectively?*

The two main challenges to near-miss analysis are the detection and prioritisation of near misses. In the context of software systems, detecting near misses is particularly challenging as they do not cause any visible physical event. Regarding the prioritisation of near misses, a review of previous literature on the topic was conducted to identify solutions that are suitable for the software industry. However, a review of the existing prioritisation techniques indicated that they are generally specific to the industry concerned and often require prior knowledge about near misses from historical data. The latter is not available in the software industry, where the concept of a near miss, as used in other industries, is still largely unexplored.

In order to detect near misses, a suitable definition of a near miss for the software industry was required in order to recognise them. To this end, an objective measure to define a near miss was proposed in the form of a Service Level Agreement (SLA) that predefines the performance level of a system. The SLA is used to measure the severity of an event based on its downtime duration, in comparison with the downtime allowance specified in the SLA. A near miss is defined as an event that causes a downtime close to exceeding the downtime allowance specified in the SLA. In the current study an adjustable threshold was used to define the desired closeness to the downtime allowance for an event to be considered a near miss. This general definition of a near miss was formally expressed in a mathematical formula.

It is worth noting that downtime was used as the most common metric of an SLA, but depending on the type of system and failure at hand, other metrics (e.g. throughput, response time) could be used and may be more relevant. It is also worth noting that in the absence of an SLA, a near miss can still be defined by near-miss indicators identified from the root-cause analysis of a failure, as conducted in the prototype implementation.

The high volume of near misses was addressed with a prioritisation mechanism to ensure that only the events most likely to result in a failure were passed on for analysis. A mathematical near-miss detection and prioritisation model was subsequently developed to calculate the failure probability of a near miss and prioritise near misses based on this

probability. The calculation of such failure probability was based on a corresponding formula from the reliability theory of IT systems, which was extended to accommodate cases of near misses. The complete mathematical model, comprising the formulae for near-miss definition and failure probability, was presented in Chapter 6.

11.2.2 Achieving the goal of the research

The architecture of the proposed NMS includes all the partial solutions provided as answers to the above questions, and involves a thorough multi-stage filtering process that analyses and prioritises system logs for indicators of an upcoming failure. An alert is raised for potential near misses with the highest risk level (failure probability). The forensic investigation process is used throughout the architecture to collect and analyse data about the detected near misses. The NMS architecture helps improve the completeness and relevance of the digital evidence of the failure, as well as the accuracy of its forensic analysis as discussed below.

As soon as a potential high-risk near miss has been detected, a request is automatically made to collect all the data about the event and to store in a table for forensic analysis at a later stage. This automatic data collection ensures that the digital evidence is not affected by the ensuing failure, thereby ensuring that no evidence is lost or tampered with. As data is only collected for potential near misses with the highest risk level, i.e. events closest to a potential failure, the collected data is a fairly complete representation of the likely failure.

The above process (automatic data collection of potential near miss with highest risk level) is likely to result in digital evidence that is more complete than the evidence collected after a software failure. It also ensures that the collected evidence is relevant for the root-cause analysis as it only points to the system's behaviour close to the point of failure. Evidence about the normal system behaviour, which might be irrelevant to identify the root cause of the failure, is therefore not used for the forensic analysis.

Using limited but relevant and complete digital evidence is likely to facilitate the accurate identification of the root cause of the failure. Accurate root-cause analysis is further facilitated by the use of the forensic approach throughout the NMS architecture, ensuring that scientific methods and techniques are used to analyse the evidence. The forensic

process also ensures that all the requirements established for an accurate failure investigation are satisfied, namely objectivity, comprehensiveness, reproducibility and admissibility in court.

Results of the prototype implementation of the NMS architecture demonstrated the above as follows. The data collected following the detection of a near miss at runtime provided all the details pertaining to the potential failure and was therefore complete and suitable for the forensic analysis. The near-miss analysis also caused a significant reduction of irrelevant data. Indeed, the monitored software application had 22 attributes that were logged for an eventual root-cause analysis. However, through the process of identifying near-miss indicators and defining near misses for that application, only four of these attributes proved relevant to identify the root cause of the failure. Furthermore, the analysis of these four attributes accurately pointed to the root cause of the ensuing failure. The goal of the research was therefore achieved.

11.3 Main contributions

11.3.1 Advancing the state of the art

The first contribution of the research is the fact that it demonstrated that digital forensics can serve as an effective alternative for investigating major software failures. This is a new application of digital forensics, which is currently limited to the investigation of computer crimes, security incidents and civil matters. Digital forensics can help provide sound evidence of the root cause of a software failure. Applying digital forensics to failure analysis is not a new idea. It was referred to as Operational Forensics by Michael Corby in 2000 (Corby, 2000a). However, it has remained at a conceptual level with no end-to-end process. This research is the first attempt at proposing a complete “operational forensic” process and demonstrating its effectiveness through the implementation of a prototype.

The second contribution made by the current research is the fact that it introduced the concept of near miss in digital forensics. Near-miss analysis is an established field in many engineering disciplines but it is new to digital forensics. The research demonstrated

how near-miss analysis can benefit digital forensics for the purpose of investigating software failures.

The third and most important contribution of the research is the architecture of an NMS. Although some features of the architecture are based on the existing digital forensic process (e.g. collection of digital evidence of the failure, root-cause analysis of the failure based on the collected evidence), the overall architecture design is completely original as it includes a near-miss analysis process used to improve the validity of the evidence. The main novelty of the architecture is the mathematical model developed to define, detect and prioritise near misses. The prioritisation of near misses ensures that only near misses closest to the potential point of failure are passed on for root-cause analysis, thereby reducing the number of false alarms (i.e. raising an alert for an unsafe event that is not likely to result in a failure and which does not contain data pertaining to the potential failure).

The final contribution of the research is its proposal of an original process followed in the prototype implementation for identifying near-miss indicators from the forensic analysis of logs of a software failure. The forensic analysis used SOM analysis and statistical analysis to analyse the trends in the execution of a failing program and identify significant changes close to the point of failure. The SOM analysis provided a simple but accurate visual representation of the trends in the logged data, thereby facilitating the quick and accurate identification of the root cause of the failure. A diagram of the above process was provided in Figure 10.16 in the previous chapter. The process that was followed proved reliable due to the validity of the resulting near-miss formula and the low number of false alarms raised.

11.3.2 Publications produced

Throughout this study, results of the research have been published in the following conference and journal papers.

- Bihina Bella, M.A., Olivier, M.S. and Eloff, J.H.P. *Proposing a Digital Operational Forensic Investigation Process*. In Proceedings of the 6th International Workshop on Digital Forensics and Incident Analysis (WDFIA 2011), 7-8 July 2011, London, UK.

This paper presented the adapted digital forensic process proposed for the investigation of software failures. To demonstrate its viability, the process was applied to the case study of a real-life fatal software failure, namely the Therac-25 disaster, which is described in detail in Chapter 3. The case study demonstrated the advantages of using digital forensics rather than the troubleshooting approach that had actually been used in the case of the Therac-25 disaster.

- Bihina Bella, M.A., Eloff, J.H.P and Olivier, M.S. Improving System Availability with Near Miss Analysis. *Network Security*, October 2012, pp. 18-20.

This positioning paper introduced the concept of a near miss for software systems with examples of real-life near misses that preceded some severe software failures. The paper motivated the use of near-miss analysis as a novel approach to improve the availability of IT systems by preventing disruptive software failures.

- Bihina Bella, M.A., Olivier, M.S. and Eloff, J.H.P. *Near Miss Detection for Software Failure Prevention*. In Proceedings of the 2012 Southern Africa Telecommunications Network and Applications Conference (SATNAC 2012), 2-5 September 2012, George, South Africa.

This paper provided an overview of near-miss analysis and its potential application to the prevention of impending software failures. It presented the proposed definition of a near miss in the context of software systems and the initial high-level process for detecting and prioritising near misses. The application of the near-miss definition was illustrated with the example of a real-life mobile procurement system.

- Bihina Bella, M.A., Eloff, J.H.P and Olivier, M.S. *A Near-miss Management System to Facilitate the Forensic Investigation of Software Failures*. In Proceedings of the 13th European Conference on Cyber Warfare and Security (ECCWS 2014), 3-4 July 2014, Piraeus, Greece.

This paper presented the architecture of the NMS proposed in this research. The paper first reviewed challenges to near-miss analysis specific to the software industry and presented proposed solutions to these challenges. The paper also

presented the refined definition of near misses for software systems, as the basis for the design of the NMS architecture.

- Bihina Bella, M.A. and Eloff, J.H.P. Forensic investigation of software failures. *Computer Fraud & Security*, December 2014, submitted for publication.

At the time of press, this article was under review. The article motivated the combined use of digital forensics and near-miss analysis to improve the root-cause analysis of software failures. It discussed the advantages of this approach using the case of the RBS software failure discussed in Chapter 1.

11.4 Future research

This research is the first attempt at applying near-miss analysis to digital forensics with a view to improving the accuracy of the investigation of software failures. The proposed NMS architecture achieved the goal of the research to the extent described above, as demonstrated through the prototype implementation. However, the prototype still suffers some limitations that should be addressed by future research:

- The prototype only implemented a subset of the NMS architecture, namely the near-miss detection process and the forensic analysis of a software failure. The implementation of a complete prototype to test the viability of the entire NMS architecture is recommended.
- The failure investigated for the prototype implementation was caused by a simple program and had little impact. Simulating a major failure with significant impact would have been costly and risky, hence the choice of a simplistic example. However, it is advisable that the practicality of the NMS architecture be tested in future with the case of a major complex software failure.

An additional avenue for future research is the generalisation of the proposed near-miss detection process to accommodate various types of failures, since the current research was limited to demonstrating potential solutions by focusing on failures due to resource exhaustion. A general methodology to pinpoint near-miss indicators emerged from the forensic analysis in the prototype implementation and provides a starting point.

Furthermore, future research may consider extending the application of near-miss analysis to the field of software reliability. The detection of near misses at runtime can be followed by an automatic response to apply appropriate countermeasures and prevent the impending failure from unfolding. Although the proposed NMS architecture has a Failure Prevention component that can be used for this purpose, it was not implemented in this research as it fell outside the scope of this study.

As a final note, despite its limitations, this research formally introduced the technique of near-miss analysis to the software industry. Near-miss analysis has proved beneficial in a number of safety-critical industries and it is continuously applied to a growing number of disciplines. Although the software industry is not particularly safety conscious, this research has shown that major software failures can and do affect safety. Near-miss analysis is therefore a technique worth exploring more formally and in more-depth in software applications as it can be beneficial to various areas of software improvement including failure analysis and system reliability.

APPENDIX 1

TECHNICAL DETAILS ON THE SOM ANALYSIS

SOM map creation process

The pre-processing steps that were taken for the SOM map creation involved the following:

- The parameter File Status was used as a nominal attribute to distinguish between the records marked as “OK” (video copied successfully) and those marked as “not OK” (video not copied).
- Viscovery SOMine automatically converted the time values to numerical values.

The creation of maps in Viscovery SOMine follows a simple manual step-by-step process from importing the input file, selecting attributes to be processed, defining nominal attributes, and specifying the parameters to train the map. Training parameters include map size (number of nodes) and training schedule (processing speed from fast to normal). The default training parameters were kept. The resulting map is automatically created and displayed after this process and information about each cluster is provided.

How to read Viscovery SOMine output maps: Example of first 1000 records

Figure 12.1 shows the maps of the first 1000 records. Two types of maps are displayed: in the bottom frame, the overall map from all attributes, and in the top frame, the component maps for each attribute.

The overall map has four clusters, each one displayed in a different colour. Details about each cluster are provided in the panel on the right side of the map and include the average values for each of the attributes in the cluster. The frequency or distribution of each cluster

in the main map is also shown. So, the biggest cluster, which is represented in light blue on the map, has a frequency of 46.7%, which means that 46.7% of the records in the data set (records 1 to 1000) belong to this cluster. In other words, the values of their attributes are close to the values of the attributes in this cluster.

The component maps show the distribution of the values in the data set over time for each attribute. The scale of the values in the data set is displayed on a bar below each map. Values range from lowest on the left to highest on the right of the bar. Values on the map are differentiated by their colour on the scale. This means that lowest values are in blue and highest values are in red, with various shades of blue, green and yellow in between.

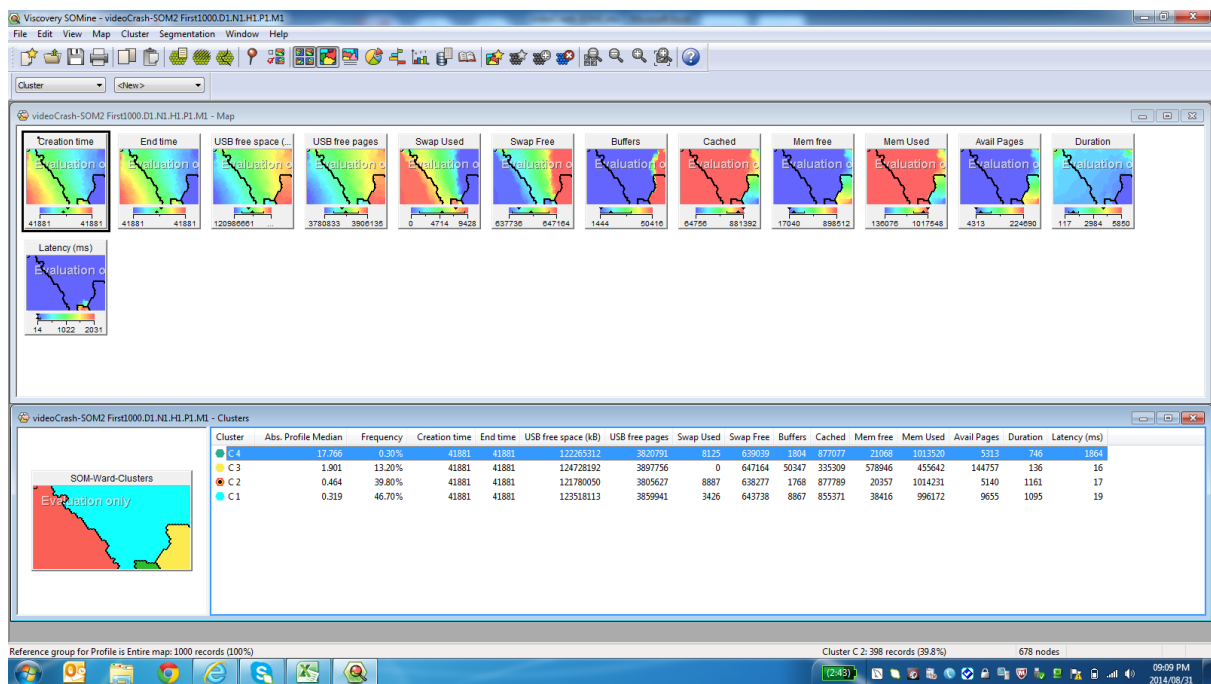


Figure 0.1: SOM output maps for first 1000 records

All component maps have the same topology, so any node (record) on one map has the exact same position on another map. For example, the first record, which has the highest value for *Duration* (5850 ms, refer to Figure 12.2) appears as an outlier in red in the top right corner of the *Duration* map below (it is circled in black). This record also has the lowest value for memory used (dark blue in the top right corner of the *Mem Used* map) and the highest value for free memory (red in the top right corner of the *Mem free* map). Similar observations can be made on the *Cached* map and the *Avail Pages* map. This is

understandable and to be expected, since at that point (record number 1), the program has just started running and little memory has been used.

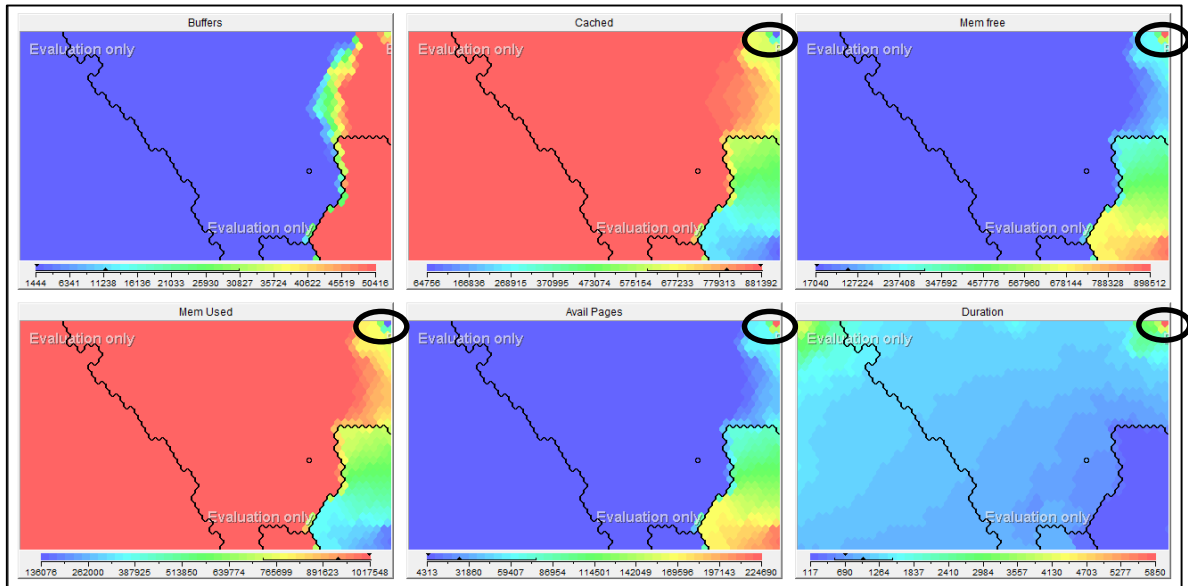


Figure 0.2: Some component maps of first 1000 records – first record appears as an outlier

Adding the number of running processes to the C++ program

The Linux command `ps -eLf | wc -l` was used to obtain the total number of processes and threads running on the system and then to display them on the screen. The `ps` command displays information about active processes. The `eLf` option was used to display all active threads and processes, followed by the `wc` command to get a count of the listed processes, counting each line (`-l` option) of the output. The researcher decided to get a count of all processes and threads as the `ps` command did not provide the option to display only those processes and threads doing I/O activity, which is what the `iostat` command provided.

APPENDIX 2

GLOSSARY OF TERMS

Accident: An undesirable event resulting in injury or damage (Jones et al., 1999).

Analysis: process of evaluating potential digital evidence in order to assess its relevance to the investigation (ISO/IEC 27042, 2015).

Accident sequence: Sequence of events that result in an accident. The accident sequence starts with an initiating event such as a human error, and ends when the accident unfolds, also known as the accident end-state (Saleh et al., 2013).

Accident sequence precursor (or accident precursor): conditions, events and sequences that precede and lead up to accidents” (Phimister et al., 2004). They are also defined as “events that must occur for an accident to happen in a given scenario” (Carroll, 2004).

Cause: A condition or an event that results in or participates in the occurrence of an effect. Causes can be classified as:

- **Direct Cause:** A cause that resulted in the occurrence.
- **Contributing Cause:** A cause that contributed to an occurrence but would not have caused it by itself.
- **Root Cause:** The cause that, if corrected, would prevent recurrence of this and similar occurrences. The root cause usually has generic implications to a broad group of possible occurrences, and it is the most fundamental aspect of the cause that can logically be identified and corrected.

Causal analysis: the analysis of the cause of an event. In this research, this term is used interchangeably with root-cause analysis.

Condition: Any system state, whether precursor or resulting from an event, that may have adverse implications for the normal system's functionality (Jucan, 2005).

Data reduction: the process of identifying and discarding data that is irrelevant for the forensic analysis (Walker, 2011). Similarly, it can also be defined as the process of identifying and extracting data relevant for the forensic analysis. Data reduction is conducted to reduce the amount of data that an investigator needs to analyse in order to reconstruct an event and find its root cause.

Digital evidence: after-the-fact digital information derived from digital sources for the purpose of facilitating or furthering the reconstruction of the events (Willasen & Mjølshnes, 2005). Information or data, stored or transmitted in binary form, that may be relied on as evidence.

Digital forensic process: structured procedure followed during a digital forensic investigation to ensure forensic soundness of the evidence, in other words to ensure that the data is complete and has not been tampered with throughout the investigation.

Digital forensics: the use of scientifically derived and proven methods towards the preservation, collection, validation, identification, analysis, interpretation and presentation of digital evidence derived from digital sources for the purposes of facilitating or furthering the reconstruction of events found to be criminal, or for anticipating the unauthorised actions shown to be disruptive to planned operations (Palmer, 2001).

Downtime: the period of time during which a system or component is not operational or has been taken out of service" (IEEE, 1990). It is also referred to as 'outage'.

Event: A real-time factual occurrence that could seriously impact the system operation (Jucan, 2005).

Failure: the inability of a system or component to perform its required functions within specified performance requirements (IEEE, 1999).

Forensic investigation: an investigation where the scientific procedures and techniques used will allow the results (digital evidence) to be admissible in a court of law. It is also known as *forensic examination*, *digital forensic investigation*, or *digital investigation* (Köhn, 2012). ISO 27943 (2015) provides a comprehensive definition for a **digital investigation** as the use of scientifically derived and proven methods towards the identification, collection, transportation, storage, analysis, interpretation, presentation, distribution, return, and/or destruction of digital evidence derived from digital sources, while obtaining proper authorizations for all activities, properly documenting all activities, interacting with the physical investigation, preserving digital evidence, and maintaining the chain of custody, for the purpose of facilitating or furthering the reconstruction of events found to be incidents requiring a digital investigation, whether of criminal nature or not.

Incident: Any undesirable event, including accidents and near misses (Jones et al., 1999).

Investigation: application of examinations, analysis, and interpretation to aid understanding of an incident (ISO/IEC 27042, 2015).

Forensic soundness: the preservation of the integrity and completeness of the data throughout the investigation (McKemmish, 2008). Digital evidence is deemed **forensically sound** when it has not been tampered with and has remained complete throughout the investigation.

Near miss: In the general sense, a near miss is a hazardous situation, event or unsafe act where the sequence of events could have caused an accident if it had not been interrupted (Jones et al., 1999). In the context of software failures and for the purposes of this study, the author defines a near miss as an unplanned high-risk event or system condition that could have caused a major software failure if it had not been interrupted either by chance or timely intervention.

Near-miss analysis: the process of identifying near misses and determining their root cause with a view to preventing and predicting accidents (Phimister et al., 2004).

Near-miss management system (NMS): software tool used to report, analyse and track near misses (Oktem, 2002). Also known as a near-miss system or a near-miss reporting system.

Operational failure: software failure that occurs when a system is in production, after the design, development and testing phases.

Outage: Period of time when a system is down. It is used as a synonym for downtime.

Post-mortem: analysis of an event held soon after it has occurred, to determine why it was a failure (Oxford Dictionary of English, 2010). In the context of this research, a post-mortem investigation is conducted soon after a software failure to determine its cause.

Potential digital evidence: information or data, stored or transmitted in binary form, which has not yet been determined, through the process of examination and analysis, to be relevant to the investigation (ISO/IEC 27042, 2015).

Root-cause analysis: logical sequence of steps that leads the investigator through the process of isolating the facts surrounding an event or failure. Once the problem has been fully defined, the analysis systematically determines the best course of action that will resolve the event and assure that it is not repeated (Moblely, 1999). Root cause analysis uncovers the fundamental issues (root causes) that generate a problem, as opposed to troubleshooting that seeks immediate solutions to resolve the user visible symptoms (Jucan, 2005).

Runtime: the period during which a computer program is executing.

Scientific method: process used by scientists to conduct an objective investigation of an event. Its aim is to minimise bias or prejudice from the experimenter and ensure the accuracy of the results (Bernstein, 2009).

Self-organising map (SOM): a model of unsupervised neural networks used for the analysis and visualisation of multi-dimensional data (Engelbrecht, 2003). It is a data analysis technique that identifies and displays clusters of similar records in the data set.

Service Level Agreement (SLA): The SLA is the entire contract that specifies what service the customer can expect from the provider, and the responsibilities of both parties (Sevcik, 2008). It defines the expected performance level of a system.

Software failure analysis: logical sequence of steps that leads the investigator through the process of isolating the facts surrounding a software failure

Software failure: an unplanned cessation of a software system or component to function as specified

System restoration: Putting a system back into its normal mode of operation after a failure

Troubleshooting: discovering why something does not work effectively and making suggestions about how to improve it (Cambridge English dictionary). It is the process of diagnosing the source of a system failure by recreating the problem to identify its cause. It can be subjective as it is not a scientific process but it is the most common first response to a failure.

Volatile data: It commonly refers to data that is likely to be lost when a machine is rebooted or overwritten during the course of the machine's normal use (Amari, 2009). This data is especially prone to change and can be easily modified. Change can be switching off the power or passing through a magnetic field. Volatile data also includes data that changes as the system state changes (SO/IEC 27037, 2012). This data is stored in volatile memory, such as RAM (random access memory), which is computer storage that retains its data only when the machine is switched on, by opposition to persistent storage on the hard disk. Examples of volatile data include information about running

processes, open files, network connections, passwords and cryptographic keys, hidden data, and malicious code (Amari, 2009).

BIBLIOGRAPHY

AccessData.com. Forensic Toolkit 5. [Online] Available from: <http://www.accessdata.com/products/digital-forensics/ftk#.Ub2w65w3jcw> [Accessed: 17 June 2013].

Amari, 2009. Techniques and Tools for Recovering and Analyzing Data from Volatile Memory, SANS Institute InfoSec Reading Room, Available from: <http://www.sans.org/reading-room/whitepapers/forensics/techniques-tools-recovering-analyzing-data-volatile-memory-33049> [Accessed: 21 June 2013].

Ammer, C. (2013). *The American Heritage® Dictionary of Idioms*. 2nd edition. Houghton Mifflin Company. [Online] Available from: http://dictionary.reference.com/browse/near_miss [Accessed: 26 November 2014].

Andriulo, S. & Gnoni, M. (2014). Measuring the effectiveness of a near-miss management system: An application in an automotive firm supplier. *Reliability Engineering and System Safety*, 132, 154-162.

ANSI/IEEE. (1991). *Standard Glossary of Software Engineering Terminology*, STD-729-1991, ANSI/IEEE.

Arnold, D.N. (2000). *The Explosion of the Ariane 5*. (23 August). [Online] Available from: <http://www.ima.umn.edu/~arnold/disasters/ariane.html> [Accessed: 7 February 2013].

Aspden, P., Corrigan, J.M., Wolcott, J. & Erickson, S.M. (2004). *Patient safety: Achieving a new standard for care*, The National Academy Press, Washington, DC. [Online] Available from: <http://www.nap.edu/catalog/10863.html>. [Accessed: 8 December 2014].

Barach, P. & Small, S.D. (2000). Reporting and preventing medical mishaps: Lessons from non-medical near miss reporting systems. *British Medical Journal*, 320(7237), 759-763. March.

Basel Committee on Banking Supervision. (2003). *Risk Management Principles for Electronic Banking*. Bank for International Settlements. [Online] Available from: <http://www.bis.org/publ/bcbs98.pdf> [Accessed: 21 February 2011].

BBC News. (2014). RBS fined £56m over 'unacceptable' computer failure. (20 November) [Online] Available from: <http://www.bbc.com/news/business-30125728>. [Accessed: 18 December 2014].

Belles, R-J., Cletcher, J.W., Copinger, D.A., Dolan, B.W., Minarick, J.W., Muhlheim, M.D, O'Reilly, P.D., Weerakkody, S. & Hamzehee, H. (2000). *Precursors to Potential Severe Core Damage Accidents: 1998 – A Status Report*. NUREG/CR-4674 ORNL/NOAC-232 Vol. 27. Oak Ridge National Laboratory, US. Nuclear Regulatory Commission Office of Nuclear Regulatory Research Washington, DC 20555-0001.

Bernstein, M. (2009). *Scientific Method Applied to Forensic Science*. [Online] Available from: <http://marybernstein.wordpress.com/2009/05/27/scientific-method-applied-to-forensic-science/> [Accessed: 09 June 2013].

Bier, V.M. (1993). Statistical methods for the use of accident precursor data in estimating the frequency of rare events. *Reliability Engineering and System Safety*, 41, 267-280.

Bier, V.M. (1998). *Accident Sequence Precursors and Probabilistic Risk Assessment*. Madison, Wis.: University of Wisconsin Press.

Bier, V.M. & Mosleh, A. (1990). The analysis of accident precursors and near misses: implications for risk assessment and risk management. *Reliability Engineering and System Safety*, 27, 91-101.

Bihina Bella, M.A., Eloff J.H.P. & Olivier, M.S (2009). A fraud management system architecture for next-generation networks, *Forensic Science International*, vol. 185, pp.51-58, March.

Bihina Bella, M.A., Eloff, J.H.P. & Olivier, M.S. (2012). Improving system availability with near miss analysis. *Network Security*, October, 18-20.

Bird (Jr), F.E. & Germain, G.L. (1996). *Practical Loss Control Leadership*. Loganville, Georgia: Det Norske Veritas Inc.

Bogdanich, W. (2010a). Radiation offers new cures, and ways to do harm. *The New York Times*. [Online] Available from: http://www.nytimes.com/2010/01/24/health/24radiation.html?hp=&pagewanted=all&_r=1& [Accessed: 1 April 2013].

Bogdanich, W. (2010b). As technology surges, radiation safeguards lag. *The New York Times*. 26 January. [Online] Available from: http://www.nytimes.com/2010/01/27/us/27radiation.html?ref=radiation_boom [Accessed: 1 April 2013].

Bogdanich, W. (2011). Radiation boom. *The New York Times*. [Online] Available from: http://topics.nytimes.com/top/news/us/series/radiation_boom/index.html [Accessed: 1 April 2013].

Bogdanich, W. & Rebelo, K. (2010). A pinpoint beam strays invisibly, harming instead of healing. *The New York Times*. 28 December. [Online] Available from: http://www.nytimes.com/2010/12/29/health/29radiation.html?pagewanted=all&_r=0 [Accessed: 1 April 2013].

Borg, B. (2002). Predictive Safety from Near Miss and Hazard Reporting. [Online] Available from: <http://www.signalsafety.ca/files/Predictive-Safety-Near-Miss-Hazard-Reporting.pdf> [Accessed: 27 January 2014].

Borrás, C. (2006). Overexposure of radiation therapy patients in Panama: Problem recognition and follow-up measures. *Rev Panam Salud Publica*, 20(2-3), 173-187. ISSN 1020-4989.

Breitinger, F., Stivaktakis, G. & Baier, H. (2013). FRASH: A framework to test algorithms of similarity hashing, *Digital Investigation*, vol. 10, Supplement, pp. S50-S58, December.

Breeuwsma, M. (2006). Forensic imaging of embedded systems using jtag (boundary-scan), *Digital Investigation*, 3(1), pp. 32–42.

Brosgol, B. (2011). Safety in medical device software: Questions and answers. *Electronic Design*. 6 June. [Online] Available from: <http://electronicdesign.com/embedded/safety-medical-device-software-questions-and-answers> [Accessed: 25 March 2013].

Brown, J.F., Obenski, K.S. & Osborn, T.R. (2003). *Forensic Engineering Reconstruction of Accidents*. Second Edition. p. 4. Charles C. Thomas: Springfield, Illinois, USA.

Bytemark.co.uk. My Machine is too slow. [Online] Available from: https://www.bytemark.co.uk/support/document_library/vm_too_slow/ [Accessed: 07 November 2014].

Callum, J.L., Kaplan, H.S., Merkley, L.L., Pinkerton, P.H., Rabin-Fastman, B., Romans, R.A., Coovadia, A.S. & Reis, M.D. (2001). Reporting of near-miss events for transfusion medicine: improving transfusion safety. *Transfusion*, 41, 1204-1211. October. [Online] Available from: http://www.iakh.de/tl_files/oldcontent/literatur/nearmiss.pdf. [Accessed: 22 November 2014].

Carper, K.L. (2000). *Forensic Engineering*. Second Edition, pp. 2-4, CRC Press: Boca Raton.

Carrier, B. (2003). *Open Source Digital Forensic Tools – The Legal Argument*. September. [Online] Available from: http://www.digital-evidence.org/papers/opensrc_legal.pdf [Accessed: 9 June 2013].

Carrier, B.D. & Spafford, E.H. (2004). Defining event reconstruction of digital crime scenes. *Journal of Forensic Sciences*, 49(6).

Carroll, J.S. (2004). Knowledge management in high-hazard industries: Accident precursors as practice. In: Phimister, J.R., Bier, V.M. & Kunreuther, H.C. (Eds.), *Accident precursor analysis and management: reducing technological risk through diligence*. Washington, DC: The National Academies Press; pp. 127-36.

Casey, E. (2004). *Digital Evidence and Computer Crime*, Second Edition. Elsevier.

Casey, E. (2010). *Handbook of digital forensics and investigations*. Elsevier Academic Press, Chapter 2, pp. 23-24.

Cashmore, S. (2012). Adding muscle to mobile apps. *Brainstorm IT Web*, 11(08), 38-39. April.

Caudill, M. (1989). *Neural Network Primer: Part I*. *AI Expert*, 2(12), pp.46-52, December.

Charette, R. (2010). Software problem blamed for woman's death in Minnesota. *IEEE Spectrum*. [Online] Available from: <http://spectrum.ieee.org/riskfactor/computing/it/software-problem-blamed-for-womans-death-in-minnesota> [Accessed: 28 March 2013].

Coetzee, M. & Eloff, J. (2006). A framework for web services trust. In: *Proceedings of the IFIP TC-11 21st International Information Security Conference (SEC 2006)*, 22-24 May, Karlstad, Sweden.

Cooke, R. & Goossens, L. (1990). The accident sequence precursor methodology for the European Post-Seveso era. *Reliability Engineering and System Safety*, 27, 117-130.

Cooke, R. M., Ross, H. L., & Stern, A. (2011). *Precursor Analysis for Offshore Oil and Gas Drilling: From Prescriptive to Risk-Informed Regulation*, Resources for the future, RFF DP 10-61, January, [Online]. Available from: www.rff.org [Accessed: 10 May 2012].

Corby, M.J. (2000a). Operational Computer Forensics – The New Frontier. *Proceedings of the 23rd National Information Systems Security Conference*, Baltimore, USA, 16-19 October. [Online]. Available from: <http://csrc.nist.gov/nissc/2000/proceedings/papers/317slide.pdf> [Accessed: 10 May 2010].

Corby, M.J. (2000b). Operational Forensics. *Information Security Management Handbook*. Fourth Edition. Vol. 2, chapter 28. Auerbach Publications: Boca Raton.

Corby, M. (2007). Operational Forensics. In: Tipton, H.F. & Krause, M. (Eds.), *Information Security Management Handbook*, Sixth Edition, chapter 211, pp. 2773-2779. Auerbach Publications: Boca Raton.

Corby, M.J. (2011). Forensics: Operational. *Encyclopedia of Information Assurance*. Taylor & Francis.

Corcoran, W.R. (2004). Defining and analyzing precursors. In: Phimister, J.R., Bier, V. & Kunreuther, H. (Eds.), *Accident Precursor Analysis and Management: Reducing Technological Risk through Diligence*, Washington DC, USA: The National Academies Press, pp. 79-84.

Craiger, P. (2006). Computer forensics methods and procedures. In: Bigdoli, H. (Ed.), *Handbook of Information Security*, New York, John Wiley and Sons, 2, pp. 736-755.

Daily Mail Reporter. (2010). *Wrong organs removed from donors after computer glitch lay undiscovered for 10 years*. 20 October. [Online]. Available from:

<http://www.dailymail.co.uk/news/article-1322081/Wrong-organs-removed-donors-glitch.html> [Accessed: 03 March 2013].

Das, A. (2012). Maximizing Profit Using SLA-Aware Provisioning, In: *Proceedings of Network Operations and Management Symposium (NOMS)*, 16-20 April 2012, Maui, Hawaii, pp.393-400.

Dependability Research Group. University of Virginia. Software Forensics (aka Forensic Software Engineering). [Online] Available from: http://dependability.cs.virginia.edu/info/Software_Forensics [Accessed 5 May 2012].

Devore, J.L. & Berk, K.N. (2012). *Modern mathematical statistics with applications*. Second Edition. New York, London: Springer. pp. 79-81.

DevTopics.Com. (2008). 20 Famous Software Disasters. 12 February. [Online] Available from: <http://www.devtopics.com/20-famous-software-disasters/> [Accessed: 8 April 2013].

Dershowitz, N. (2013). Software Horror Stories. [Online]. Available from: <http://www.cs.tau.ac.il/~nachumd/horror.html> [Accessed: 15 March 2013].

Diaz, J.M. (2004). Cuatro años de prisión para físicos del ION. *PanamaAmerica*. [Online] Available from: <http://www.panamaamerica.com.pa/notas/479486-cuatro-anos-de-prision-para-fisicos-del-ion>. [Accessed: 21 March 2013].

Dimaio, V.J. & DiMaio, D. (2001). *Forensic Pathology. Second Edition*. Florida: CRC Press LLC. pp. 3-6. ISBN 0-8493-0072-X.

Dolinak, D., Matshes, E.W. & Lew, E.O. (2005). *Forensic pathology: Principles and practice*. p. 68, Elsevier, Oxford.

Durand-Parenti, C. (2009). *Une erreur informatique à 300 millions d'euros*, in *Le Point*. 12 May. [Online] Available from: <http://www.lepoint.fr/actualites-societe/2009-05->

[12/une-erreur-informatique-a-300-millions-d-euros/920/0/342633](#) [Accessed: 4 March 2013].

Evolgen.com. (2012). *Incident Investigation*. [Online] Available from: <http://www.evolgen.com/labels/incident-investigation.html> [Accessed: 28 January 2013].

Evolgen.com. (2013). Gartner Says IT Operations Analytics to Supplement APM. (2 January). [Online] Available from: <http://www.evolgen.com/blog/what-is-it-operations-analytics.html> [Accessed: 28 January 2013].

Expert Glossary. (2012). *Outage*. [Online] Available from: <http://www.expertglossary.com/telecom/definition/outage> [Accessed: 18 October 2012].

FDA. (2004a). NEURON'VISIONPROGRAMMER. *MAUDE Adverse Event Report: #2182207-2004-00681*. 30 April. [Online] Available from: http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfMAUDE/Detail.cfm?MDRFOI_ID=527622 [Accessed: 25 March 2013].

FDA. (2004b). *Medtronic announces nationwide, voluntary recall of model 8870 software application card*. 22 September. [Online] Available from: <http://www.fda.gov/MedicalDevices/Safety/ListofRecalls/ucm133126.htm>. [Accessed: 25 March 2013].

FDA. (2007). Baxter healthcare pte. ltd. colleague 3 cxe volumetric infusion pump 80frn. MAUDE Adverse Event Report #6000001-2007-09468. July. [Online] Available from: http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfMAUDE/Detail.cfm?MDRFOI_ID=914443 [Accessed:26 March 2013]

FDA. (2013). *MAUDE - Manufacturer and User Facility Device Experience*. [Online] Available from: <http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfMAUDE/TextSearch.cfm> [Accessed: 26 March 2013].

Fei, B., Eloff, J., Venter, H. & Olivier, M. (2005). Exploring Forensic Data with Self-Organizing Maps, *Advances in Digital Forensics*, 194, 113-123. Springer.

Fei, B., Eloff, J., Venter, H. & Olivier, M. (2006). The use of self-organising maps for anomalous behaviour detection in a digital investigation. *Forensic Science International*, 162(1-3), 33-37. November.

Feldman, J. (2011). RIM outage explanation leaves big questions. *Information Week*, 13 October. [Online] Available from: www.informationweek.com/news/global-cio/interviews/231900785 [Accessed: 22 July 2012].

Fernandez, M. (2009). Computer Error Caused Rent Troubles for Public Housing Tenants. *The New York Times*. 6 August. [Online] Available from: http://www.nytimes.com/2009/08/06/nyregion/06rent.html?_r=0 [Accessed: 4 March 2013].

Finnegan, M. (2013). RBS apologises as customers hit by another IT outage. *Computerworld UK*, [Online] Available from: <http://www.computerworlduk.com/news/it-business/3491865/rbs-apologises-as-customers-hit-by-another-it-outage/> [Accessed: 5 February 2013]

Fox News. (2012). *United Airlines fixes problem with computer system after thousands of travellers delayed*. 15 November. [Online] Available from: <http://www.foxnews.com/travel/2012/11/15/united-airlines-fixes-problem-with-computer-system-after-thousands-travelers/> [Accessed: 20 March 2013].

Free Online Law Dictionary. (2013). Legal definition of Forensic Science. [Online] Available from: <http://legal-dictionary.thefreedictionary.com/Forensic+Science> [Accessed: 18 May 2013].

Fried, E. (2009). Near Miss Project Update. *Near Miss Project Newsletter*, vol. I, issue 3. [Online] Available from: http://www.nyacp.org/files/public/Near%20Miss%20Newsletter_Issue%203_Email%20Version.pdf [Accessed: 10 December 2014].

Garfinkel, S. (2005). History's worst software bugs. *Wired.com*. 11 August. [Online] Available from: <http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=1> [Accessed: 14 March 2013].

Gerhards, R. (2009). *The Syslog Protocol*. RFC 5424. [Online] Available from: <http://tools.ietf.org/html/rfc5424> [Accessed: 29 July 2011].

Girault, C. & Valk, R. (2003). *Petri Nets for Systems Engineering: A Guide to Modeling, Verification and Applications*. Springer.

Gnoni, M.G., Andriulo, S., Nardone, P. & Maggio, G. (2013). Lean occupational safety: an application for a near-miss management system design. *Safety Science*, 53, 96-104. March.

Gogolin, G. (2013). *Digital Forensics Explained*. CRC Press: Boca Raton. Chapter 2.

Goode, N., Salmon, P., Lenne, M. & Finch, C. (2014). *UPLOADS: An incident reporting and learning system for the outdoor activity sector*. PowerPoint slides. [Online] Available from: <http://uploadsproject.files.wordpress.com/2014/05/goode-n-salmon-p-2013-uploads-5th-asia-oceania-camping-congress.pdf> [Accessed: 28 November 2014].

Grady, R.B. (1996). Software failure analysis for high-return process improvement decisions. *Hewlett-Packard Journal*. August. [Online] Available from: <http://www.hpl.hp.com/hpjournal/96aug/aug96a2.pdf> [Accessed: 10 May 2012].

Greene, T. (2011). Financial firm fined \$25M for hiding software glitch that cost investors \$217M. 4 February. [Online] Available from: <http://www.networkworld.com/news/2011/020411-axa-rosenberg-group-glitch.html> [Accessed: 28 February 2013].

Greenwell, W.S., Knight, J.C. & Strunk, E.A. (2003). Risk-based classification of incidents. In: *Workshop on the Investigation and Reporting of Incidents and Accidents*. Department of Computer Science, University of Virginia. [Online] Available from: <http://shemesh.larc.nasa.gov/iria03/p03-greenwell.pdf> [Accessed: 7 May 2012].

Grobler, M. & von Solms, B. (2009). A best practice approach to live forensic acquisition. In: *Proceedings of the Information Security South Africa 2009 (ISSA 2009)*, 6-8 July 2009, Johannesburg, South Africa.

Grottke, M., Li, L., Vaidyanathan, K. & Trivedi, K.S. (2006). Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 55(3), 411-420.

Guarino, A. (2013). Digital forensics as a big data challenge. *ISSE 2013 Securing Electronic Business Processes*, 197-203.

GuidanceSoftware.com. EnCase Forensic v7. [Online] Available from: <http://www.guidancesoftware.com/encase-forensic.htm> [Accessed: 17 June 2013].

Hallman, T. (2014). 20 get out of jail free after new Dallas County records system debuts. [Online] Available from: <http://www.dallasnews.com/news/metro/20140618-20-get-out-of-jail-free-after-new-dallas-county-records-system-debuts.ece> [Accessed: 22 June 2015].

Harris, C. (2011). IT Downtime Costs \$26.5 Billion in Lost Revenue. *Information Week*. 24 May. [Online] Available from: <http://www.informationweek.com/storage/disaster-recovery/it-downtime-costs-265-billion-in-lost-re/229625441> [Accessed: 8 October 2012].

Hatton, L. (2004). *Forensic software engineering: An overview*. CIS, University of Kingston, UK. 19 December. [Online] Available from: http://www.leshatton.org/wp-content/uploads/2012/01/fse_Dec2004.pdf [Accessed: 5 May 2012].

Hatton, L. (2007). Forensic software engineering: Taking the guesswork out of testing. [Online] Available from: http://www.leshatton.org/wp-content/uploads/2012/01/LH_EuroStar07.pdf [Accessed: 17 August 2013].

Hatton, L. (2012). Forensic Software Engineering and not before time. [Online] Available from: <http://www.leshatton.org/wp-content/uploads/2012/01/A7.pdf> [Accessed: 5 May 2012].

Hecht, M. (2007). Use of software failure data from large space systems. Presented at the *Workshop on Reliability Analysis of System Failure Data*. Cambridge, UK.

Heydebreck, P., Klofsten, M. & Krüger, L. (2011). F2C – An innovative approach to use fuzzy cognitive maps (FCM) for the valuation of high-technology ventures. *Communications of the IBIMA*, Vol 2011, Article ID 483882, 14 pages. [Online] Available from: <http://www.ibimapublishing.com/journals/CIBIMA/2011/483882/483882.html> [Accessed: 12 November 2014].

Highleyman, W.H. (2008). *Why are active/active systems so reliable?* [Online] Available from: www.availabilitydigest.com [Accessed: 1 March 2012].

Holenstein, B., Highleyman, B. & Holenstein, P.J. (2003). *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, 1, 27-28. December. Bloomington, USA: Authorhouse.

Holt, C.C. (2004). Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20, 5-10. January-March.

Hood, B. (2010a). Modelling for operational forensics. *Digital Forensics Magazine*, 3. 1 February.

Hood, B. (2010b). Psychosocial forensics – Exploring a number of novel approaches to operational forensics. *Digital Forensics Magazine*, 4. 1 August.

Horton, J. (2008). How BlackBerry outages work. *HowStuffWorks.com*. 15 May. [Online] Available from: <http://electronics.howstuffworks.com/blackberry-outage1.htm> [Accessed: 26 July 2012].

Hower, R. (2013). *Software QA and Testing Frequently-Asked-Questions, Part 1*. [Online] Available from: http://www.softwareqatest.com/qatfaq1.html#FAQ1_3 [Accessed: 6 February 2013].

Huckle, T. 2014 General web site on bugs and reliability. *Institut für Informatik, TU München*. [Online] Available from: <http://www5.in.tum.de/~huckle/bugse.html#WWW> [Accessed: 6 February 2013].

IAEA (International Atomic Energy Agency). (2001). Investigation of an accidental exposure of radiotherapy patients in Panama/ Report of a Team of Experts, 26 May–1 June 2001. [Online] Available from: http://www-pub.iaea.org/MTCD/publications/PDF/Pub1114_scr.pdf [Accessed: 19 March 2013].

IAEA. (2013a). Prevention of Accidental Exposure in Radiotherapy. *Training course*. Module 2.7. Error in TPS data entry - Panama (2,111 KB). [Online] Available from: https://rpop.iaea.org/RPOP/RPoP/Content/AdditionalResources/Training/1_TrainingMaterial/AccidentPreventionRadiotherapy.htm [Accessed: 19 March 2013].

IAEA. (2013b). Prevention of Accidental Exposure in Radiotherapy. *Training course*. Module 2.10. Accident update, some newer events - UK, USA & France. [Online] Available from: https://rpop.iaea.org/RPOP/RPoP/Content/AdditionalResources/Training/1_TrainingMaterial/AccidentPreventionRadiotherapy.htm [Accessed: 19 March 2013].

IEEE. (1990). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*.

IJoFCS. (2012). *The International Journal of Forensic Computer Science*. [Online] Available from: <http://www.ijofcs.org/> [Accessed: 18 June 2013].

ISMP-Canada (Institute for Safe Medication Practices Canada). (2014). *Definitions of Terms*. [Online] Available from: <http://www.ismp-canada.org/definitions.htm>. [Accessed: 24 November 2014].

ISO/IEC. (2007). *Information technology - Security techniques - Code of practice for information security management*. Switzerland, Geneva. International Standard ISO/IEC 27002.

ISO/IEC 27037. (2012). *Information technology — Security techniques — Guidelines for identification, collection, acquisition, and preservation of digital evidence*. [Online] Available from: http://www.iso.org/iso/catalogue_detail?csnumber=44381. [Accessed: 8 April 2015]

ISO/IEC 27042. (2015). *Information technology - Security techniques - Guidelines for the analysis and interpretation of digital evidence*. [Online] Available from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=44406. [Accessed: 21 June 2015]

ISO/IEC 27043. (2015). *Information Technology - Security Techniques - Incident investigation principles and processes*. [Online] Available from: http://www.iso.org/iso/catalogue_detail.htm?csnumber=44407. [Accessed: 21 June 2015]

ItsGov.com. (2011). *History of forensic pathology*. [Online] Available from: <http://www.itsgov.com/history-of-forensic-pathology.html> [Accessed: 05 December 2014].

Jain, M. & Gupta, R. (2011). Redundancy issues in software and hardware systems: an overview. *International Journal of Reliability, Quality and Safety Engineering*, 18(1), 61-98.

Jeyaraman, S. & Atallah, M.J. (2006). An empirical study of automatic event reconstruction systems. *Digital Investigation*, 3, 108-115.

Johnson, J.W. & Rasmuson, D.M. (1996). The US NRC's Accident Sequence Precursor Program: an overview and development of a bayesian approach to estimate core damage frequency using precursor information. *Reliability Engineering and System Safety*, 53, 205-216.

Johnson, C. (2002). Forensic software engineering: are software failures symptomatic of systemic problems? *Safety Science*, 40(9), 835-847. December.

Jones, A. (2012). 10 Seriously epic computer software bugs. 24 December. [Online] Available from: <http://listverse.com/2012/12/24/10-seriously-epic-computer-software-bugs/> [Accessed: 8 April 2013].

Jones, S., Kirchsteiger, C. & Bjerke, W. (1999). The importance of near miss reporting to further improve safety performance. *Journal of Loss Prevention in the Process Industries*, 12, 59-67.

Jucan, G. (2005). *Root Cause Analysis for IT Incidents Investigation*. [Online] Available from: <http://hosteddocs.ittoolbox.com/GJ102105.pdf> [Accessed: 10 October 2010].

Karp, G. (2012). United Airlines experiences yet another major computer glitch. *Chicago Tribune*. 15 November. [Online] Available from: http://articles.chicagotribune.com/2012-11-15/business/ct-biz-1116-united-outage-20121116_1_jeff-smisek-charlie-hobart-reservation-system [Accessed: 1 March 2013].

Khan, M.N.A., Chatwin, C.R. & Young R.C.D. (2007). A framework for post-event timeline reconstruction using neural networks. *Digital Investigation*, vol.4, pp. 146-157.

Kent, K., Grance, T., Chevalier, S. & Dang, H. (2006). Guide to Integrating Forensic Techniques into Incident Response. *NIST Special Publication 800-86*, National Institute of Standards and Technology, Gaithersburg, USA. [Online] Available from:

<http://csrc.nist.gov/publications/nistpubs/800-86/SP800-86.pdf> [Accessed: 9 February 2011].

Kessler, G.C. (2009). Computer forensics hardware and software. Processing hard drives and cell phones. *Tutorial*. 6 October. [Online] Available from: <http://www.garykessler.net/presentations/HICSS-2009.zip> [Accessed: 9 February 2011].

Kirwan, B., Gibson, W.H. & Hickling, B. (2007). Human error data collection as a precursor to the development of a human reliability assessment capability in air traffic management. *Reliability Engineering and System Safety*, 93(2), 217-33.

Kleindorfer, P., Oktem, U.G., Pariyani, A. & Seider, W.D. (2012). Assessment of catastrophe risk and potential losses in industry. *Computers and Chemical Engineering*, 47, 85-96.

Köhn, M.D. (2012). Integrated Digital Forensic Process Model. Master's dissertation, University of Pretoria, Department of Computer Science, Pretoria, South Africa. November.

Kohonen, T. (1990). The Self-Organizing Map. *Proceedings of the IEEE*, 78 (9). Sept.

Kohonen, T. & Honkela, T. (2007) *Kohonen Network*. Scholarpedia, p. 7421.

Krigsman, M. (2012). *RBS Bank joins the IT failures 'Hall of Shame'*. (25 June). [Online] Available from: <http://www.zdnet.com/blog/projectfailures/rbs-bank-joins-the-it-failures-hall-of-shame/15685> [Accessed: 7 February 2013].

Lai, R. (2013). *Operations Forensics: Business Performance Analysis Using Operations Measures and Tools*, MIT Press, March 2013.

Laprie, J.C. (Ed.). (1992). *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Wein, New York.

Laprie, J.C., Arlat, J., Béounes, C., Kanoun, K. & Hourtolle, C. (1987). Hardware and software fault tolerance: Definition and analysis of architectural solutions. In: *Digest of 17th FTCS*, pp. 116-121. Pittsburgh, PA.

Linstone, H.A. & Turoff, M. (2002). *The Delphi Method: Techniques and Applications*. (electronic version). [On-line] Available from: <http://is.njit.edu/pubs/delphibook/delphibook.pdf> [Accessed: 17 May 2012].

Linux.die.net. (2014). Iotop(1) – Linux man page. [Online] Available from: <http://linux.die.net/man/1/iotop> [Accessed: 14 November 2014].

Lyu, M.R. (2007). Software Reliability Engineering: A roadmap. In: *Proceedings of Future of Software Engineering. FOSE '07*. Minneapolis (23-25 May). pp. 153-170.

Mabuto, E.K. & Venter, H.S. (2011). State-of-the-art digital forensic techniques. *Proceedings of 2011 ISSA conference*. Johannesburg, South Africa. [Online] Available from: http://icsa.cs.up.ac.za/issa/2011/Proceedings/Research/Mabuto_Venter.pdf [Accessed: 9 June 2013].

Macrae, C. (2007). *Analyzing Near-Miss Events: Risk Management in Incident Reporting and Investigation Systems*. December [Online] Available from: <http://www.lse.ac.uk/researchandexpertise/units/carr/pdf/dps/disspaper47.pdf> [Accessed: 02 December 2014].

Mappic, S. (2013). *How much does downtime cost?* 4 September. [Online] Available from: <http://www.appdynamics.com/blog/devops/how-much-does-downtime-cost/> [Accessed: 8 February 2014].

March, J.G., Sproull, L.S. & Tamuz, M. (1991). Learning from Samples of One or Fewer. *Organization Science*, 2(1), 1-13.

Marcus, E. & Stern, H. (2003). *Blueprints for High Availability: Designing Resilient Distributed Systems*. Second Edition. Chapter 2 and 3. 19 September. John Wiley & Sons (US).

Martin, S.K., Etchegaray, J.M., Simmons, D., Belt, W.T. & Clark, K. (2005). Development and implementation of the University of Texas Close Call Reporting System. In: Henriksen, K., Battles, J.B., Marks, E.S. et al. (Eds.), *Advances in Patient Safety: From Research to Implementation* (Volume 2: Concepts and Methodology). Rockville (MD): Agency for Healthcare Research and Quality (US). [Online] Available from: <http://www.ncbi.nlm.nih.gov/books/NBK20498/#A2153> [Accessed: 22 November 2014].

Martinez, H. (2009). How Much Does Downtime Really Cost? *Information Management.com*. 6 August. [Online] Available from: http://www.information-management.com/infodirect/2009_133/downtime_cost-10015855-1.html. [Accessed: 15 October 2012].

McDanel, S.J. (2006). Space shuttle Columbia post-accident analysis and investigation. *Journal of Performance of Constructed Facilities*, 42(3), 159-163.

McKemmish, R. (2008). When is digital evidence forensically sound? In: Ray, I. & Sheno, S. (Eds.), *Advances in Digital Forensics IV*, Springer, Chapter 1, pp. 3-16.

Merriam Webster Dictionary. Forensic. [Online] Available from: <http://www.merriam-webster.com/dictionary/forensic> [Accessed: 2 December 2014].

Meyer, B. (2011). Again: The one sure way to advance software engineering. *ACM communications blog*. (13 January). [Online] Available from: <http://cacm.acm.org/blogs/blog-cacm/101891-again-the-one-sure-way-to-advance-software-engineering/fulltext> [Accessed: 17 February 2012].

Minarick, J.W. & Kukielka, C.A. (1982). *Precursors to Potential Severe Core Damage Accidents: 1969-1979, A Status Report*, USNRC Report NUREG/CR-2497

(ORNL/NSIC-1 82N/1 and V2). Union Carbide Corp., Nuclear Div., Oak Ridge National Laboratory and Science Applications.

MIC (Mary Immaculate College). (2014). *Accident, incident and near miss reporting*. [Online] Available from: <http://www.mic.ul.ie/adminservices/healthsafety/Pages/AccidentIncidentandNearMissReporting.aspx> [Accessed: 26 November 2014].

Mobley, R.K. (1999). *Root Cause Failure Analysis*. Butterworth Heinemann, ISBN: 978-0-7506-7158-3.

MTL Instruments. (2010). *Availability, Reliability, SIL: What's the difference?* [Online] Available from: http://www.mtl-inst.com/images/uploads/datasheets/App_Notes/AN9030.pdf [Accessed: 28 February 2012].

Mukasey, M.B, Sedgwick, J.L. & Hagy, D.W. (2008). *Electronic Crime Scene Investigation: A Guide for First Responders*. Second Edition. NIJ Special Report. April. [Online] Available from: <http://www.nij.gov/pubs-sum/219941.htm> [Accessed: 11 June 2013].

Murff, H.J., Byrne, D.W., Harris, P.A., France, D.J., Hedstrom, C. & Dittus, R.S. (2005). "Near-miss" reporting system development and implications for human subjects' protection. In: Henriksen, K., Battles, J.B., Marks, E.S. et al. (Eds.). *Advances in Patient Safety: From Research to Implementation* (Volume 3: Implementation Issues). Rockville (MD): Agency for Healthcare Research and Quality (US). [Online] Available from: <http://www.ncbi.nlm.nih.gov/books/NBK20529/> [Accessed: 10 December 2014].

Mürmann, A. & Oktem, U. (2002). The near-miss management of operational risk. *The Journal of Risk Finance*, 4(1), 25-36. 23 July.

Napochi. (2013). *AlmostMe. Near Miss medical error reporting solution*. [Online] Available from: <http://www.almostme.com/> [Accessed: 1 December 2014].

NASA. (2006). Safety Depends on "Lessons Learned". The ASRS celebrates its 30th anniversary. *Callback*, Number 317, March/April 2006. [Online] Available from: http://asrs.arc.nasa.gov/publications/callback/cb_317.htm [Accessed: 29 October 2013].

NASA. (2011). *NASA accident precursor analysis handbook* (NASA/SP-2011-3423). December. [Online] Available from: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120003292_2012003430.pdf [Accessed: 28 October 2013].

Nashef, S.A. (2003). What is a near miss? *The Lancet*, 361(9352), 180-181 (January). [Online] Available from: [http://www.thelancet.com/journals/lancet/article/PIIS0140-6736\(03\)12218-0/fulltext](http://www.thelancet.com/journals/lancet/article/PIIS0140-6736(03)12218-0/fulltext) [Accessed: 1 December 2014].

Nassif, L.F. & Hruschka, E.R. (2011). Document clustering for forensic computing: An approach for improving computer inspection. In: *Proceedings of the Tenth International Conference on Machine Learning and Applications (ICMLA)*, 1, 265-268. IEEE Press.

Near-miss Management LLC. *Dynamic Risk Predictor Suite*. [Online] Available from: <http://www.nearmissmgmt.com/products.html> [Accessed: 03 December 2014].

Neebula.com. (2012). Success Factors for Root-Cause Analysis. [Online] Available from: <http://www.neebula.com> [Accessed: 26 March 2013].

Neumann, P. (2013). *Illustrative Risks to the Public in the Use of Computer Systems and Related Technology*. 17 December. [Online] Available from: <http://www.csl.sri.com/users/neumann/illustrativerisks.html> [Accessed: 8 February 2014].

Nguyen, A. (2012). RBS says UK – not Indian – IT staff caused outage. 2 July. *Computerworld UK*. [Online] Available from: <http://www.computerworlduk.com/news/it-business/3367358/rbs-says-uk-not-indian-it-staff-caused-outage/> [Accessed: 8 March 2013]

Nichol, K. (2012). *The safety triangle explained*. 18 July. [Online] Available from: <http://crsp-safety101.blogspot.com/2012/07/the-safety-triangle-explained.html>

[Accessed: 31 October 2013].

NIST. (2013). *NSRL project website*. [Online] Available from: <http://www.nsrll.nist.gov/>

[Accessed: 17 June 2013].

Noblett, M.G., Pollitt, M.M. & Presley, L.A. (2000). Recovering and examining computer forensic evidence. *Forensic Science Communications*, 2(4). October. [Online] Available from:

<http://www.fbi.gov/about-us/lab/forensic-science-communications/fsc/oct2000/computer.htm> [Accessed: 9 June 2013].

Noon, R.K. (2001). *Forensic Engineering Investigation*. First edition, p. 1, CRC Press: Boca Raton.

Oktem, U.G. (2002). *Near-Miss: A Tool for Integrated Safety, Health, Environmental and Security Management*. 37th Annual AIChE Loss Prevention Symposium – Integration of Safety and Environmental Concepts. New Orleans, LA.

Oktem, U. & Meel, A. (2008). Near-Miss Management: A participative approach to improving system reliability. In: Melnick, E. & Everitt, B. (Eds.). *Encyclopedia of Quantitative Risk Assessment and Analysis*. Chichester, UK: John Wiley & Sons, pp. 1154-1163.

Oktem, U.G., Seider, W.D., Soroush, M. & Pariyani, A. (2013). Improve process safety with near-miss analysis, *CEP Magazine*, May. [Online] Available from: <http://www.aiche.org/sites/default/files/cep/20130520.pdf> [Accessed: 10 December 2014].

Oktem, U.G., Wong, R. & Oktem, C. (2010). Near-miss management: Managing the bottom of the risk pyramid. *Risk & Regulation*, pp. 12-13. July. ESRC Centre for Analysis of Risk and Regulation, Special Issue on Close Calls, Near Misses and Early Warnings.

OMG Unified Modeling Language (UML), *Superstructure*, 2.1.2, 143. [Online] Available from: <http://doc.omg.org/formal/2007-11-02.pdf> [Accessed: 31 October 2014].

Onyiaorah, I.V. (2013). Role of forensic pathology in clinical practice and public health: Need for a re-birth. *Afrimedical Journal*, 4(1).

ORH (Office of Radiological Health). (2005). *ORH Information Notice 2005-01*. 25 March. New York City Department of Health and Mental Hygiene. [Online] Available from: http://www.health.ny.gov/environmental/radiological/radon/radioactive_material_licensing/docs/berp2005_1.pdf [Accessed: 21 March 2013].

Palmer, G. (2001). A road map for digital forensics research. *Technical report, Digital Forensics Research Workshop Group*. August.

Palomo, E.J., North, J., Elizondo, D., Luque, R.M. & Watson, T. (2012). Application of growing hierarchical SOM for visualisation of network forensics traffic data. *Neural Networks*, 32, 275-284.

Pariyani, A., Seider, W.D., Oktem, U.G. & Soroush, M. (2012). Dynamic risk analysis using alarm databases to improve safety and quality: Part II – Bayesian Analysis. *American Institute of Chemical Engineers (AIChE) Journal*, 58(3), 826-841.

Perlin, M. (2012). Downtime, outages and failures – Understanding their true costs. 18 Sept. [Online] Available from: <http://www.evolver.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html> [Accessed: 16 March 2013].

Pertet, S. & Narasimhan, P. (2005). *Causes of failures in Web Applications*. Carnegie Mellon University: Parallel Data Lab Technical Report CMU-PDL-05-109. (December).

Pettinger, C. (2013). Are near misses leading or lagging indicators? *Safety and Health*. (August). [Online] Available from: <http://www.safetyandhealthmagazine.com/articles/9153-near-misses> [Accessed: 30 November 2014].

Phimister, J.R., Oktem, U., Kleindorfer, P.R. & Kunreuther, H. (2000). *Near-Miss System Analysis: Phase I*. Wharton School, Center for Risk Management and Decision Processes. [Online] Available from: <http://opim.wharton.upenn.edu/risk/downloads/wp/nearmiss.pdf> [Accessed: 19 July 2011].

Phimister, J.R., Oktem, U., Kleindorfer, P.R. & Kunreuther, H. (2003). Near-miss incident management in the chemical process industry. *Risk Analysis*, 23(3), 445-459.

Phimister, J., Vicki, R., Bier, M. & Kunreuther, H.C. (2004). *Accident Precursor Analysis and Management: Reducing Technological Risk Through Diligence*, National Academies Press. [Online] Available from: <http://www.nap.edu/catalog/11061.html>. [Accessed: 15 May 2012].

Pingdom. (2009). *10 historical software bugs with extreme consequences*. 10 March. [Online] Available from: <http://royal.pingdom.com/2009/03/19/10-historical-software-bugs-with-extreme-consequences/> [Accessed: 25 March 2013].

Ponemon Institute. (2011). Calculating the cost of data center outages. *Benchmark Study of 41 US Data Centers*. February. [Online] Available from: http://emersonnetworkpower.com/en-US/Brands/Liebert/Documents/White%20Papers/data-center-costs_24659-R02-11.pdf [Accessed: 8 February 2014].

Presuhn, R. (2002). Management Information Base (MIB) for the Simple Network Management Protocol (SNMP). *RFC 3418*. December. [Online] Available from: <http://tools.ietf.org/html/rfc3418>. [Accessed: 29 July 2011].

Pullum, L.L. (2001). *Software fault tolerance techniques and implementation*. Artech House.

Quick, D. & Choo, K.R. (2014). Impacts of increasing volume of digital forensic data: A survey and future research challenges, *Digital Investigation*, vol. 11, pp. 273-294.

RealityCharting.com. (2013). *Various RCA Methods and Tools in Use Today*. [Online] Available from: <http://www.realitycharting.com/methodology/conventional-wisdom/rca-methods-compared> [Accessed: 11 June 2012].

Mashable, L.U. (2011) BlackBerry's outage caused by huge e-mail backup. CNN. [Online] Available from: <http://edition.cnn.com/2011/10/12/tech/mobile/blackberry-outage-email-backup/index.html> [Accessed: 22 June 2015].

Renault, M. (2012). Orange s'explique sur la grande panne de juillet. *Le Figaro*. 20 September. [Online] Available from: <http://www.lefigaro.fr/hightech/2012/09/19/01007-20120919ARTFIG00606-orange-s-explique-sur-la-grande-panne-de-juillet.php> [Accessed: 4 March 2013].

Ritwik, U. (2002). Risk-based approach to near miss. *Hydrocarbon Processing*, pp. 93-96. October.

Roberts, M. (2010). Organ donation errors "avoidable". *BBC News*. 19 October. [Online] Available from: <http://www.bbc.co.uk/news/health-11572898> [Accessed: 20 March 2013].

Roberts, P. (2012). *FDA: Software failures responsible for 24% of all medical device recalls*. 20 June. [Online] Available from: http://threatpost.com/en_us/blogs/fda-software-failures-responsible-24-all-medical-device-recalls-062012 [Accessed: 25 March 2013].

Rodrigues, G. (2009). *Flushing out pdf flush*. 1 April. [Online] Available from: <http://lwn.net/Articles/326552/>. [Accessed: 8 November 2014].

Roughton, J. (2008). The Accident Pyramid. *Safety Culture Plus*. July. [Online] Available from: <http://emeetingplace.com/safetyblog/2008/07/22/the-accident-pyramid/> [Accessed: 30 August 2012].

Roussev, V. (2009). Hashing and data fingerprinting in digital forensics. *IEEE Security and Privacy*, pp 49-55. March/April.

Rusling, D. (1999). *The Linux tutorial – swapping out and discarding pages*. [Online] Available from: <http://www.linux-tutorial.info/modules.php?name=MContent&pageid=311>. [Accessed: 8 November 2014].

Saferstein, R. (2010). *Criminalistics: An Introduction to Forensic Science*. Tenth Edition, Prentice Hall.

Saleh, J. H., Saltmarsh, E. A., Favaro, F. M., & Brevault, L. (2013). “Accident precursors, near misses and warning signs: Critical review and formal definitions within the framework of Discrete Event Systems”. *Reliability Engineering and System Safety*, vol. 114, pp. 148-154, February.

Santosa, M. (2006). *When Linux runs out of memory*. 30 November. [Online] Available from: <http://www.linuxdevcenter.com/pub/a/linux/2006/11/30/linux-out-of-memory.html>. [Accessed: 8 November 2014].

Scarfone, K., Grance, T. & Masone, K. (2008). Computer security incident handling guide. *NIST Special Publication 800-61*, Revision 1 (March). National Institute of Standards and Technology, Gaithersburg, USA. [Online] Available from: <http://csrc.nist.gov/publications/nistpubs/800-61-rev1/SP800-61rev1.pdf> [Accessed: 22 February 2011].

Scherer, B. (2012). What quants can learn from the Axa case. 20 May. [Online] Available from: <http://www.ft.com/intl/cms/s/0/65a66800-9eb4-11e1-9cc8-00144feabdc0.html#axzz2MCol3BWr>. [Accessed: 28 February 2013].

Seveso II (1997). Council Directive 96/82/EC of 9 December 1996 on the control of major-accident hazards involving dangerous substances. *Official Journal of the European Communities*, Luxembourg.

Sevcik, P. (2008). Service Level Agreements for Business-Critical Applications. *NetForecast Report 5091*. January. [Online] Available from: <http://www.netforecast.com/wp-content/uploads/2012/06/NFR5091SLAsforBusiness-CriticalApplications.pdf> [Accessed 16 February 2013].

Skogdalen, J.E. & Vinnem, J.E. (2011). Combining precursor incidents investigations and QRA in oil and gas industry. *Reliability Engineering and System Safety*, 101, 48-58.

Sleuthkit.org. *The Sleuth Kit*. [Online] Available from: <http://www.sleuthkit.org/sleuthkit/> [Accessed: 17 June 2013].

Smith, C.L. & Borgonovo, E. (2007). Decision making during nuclear power plant incidents: A new approach to the evaluation of precursor events. *Risk Analysis*, 27(4):1027-42.

Smith, E. & Eloff, J.H.P. (2002). A prototype for assessing information-technology risks in health care, *Computers & Security*, 21(3), 266-284.

Sommer, J. (2010). The Tremors From a Coding Error. *New York Times*. 19 June. [Online] Available from: http://www.nytimes.com/2010/06/20/business/20stra.html?_r=0 [Accessed: 17 June 2013].

Sovani, K. (2013). *Linux: The Journaling Block Device*. 17 February. [Online] Available from: <http://pipul.org/2013/02/linux-the-journaling-block-device/>. [Accessed: 8 November 2014].

Splunk Wiki. (2012). *Community: Performance Troubleshooting*. 7 May. [Online] Available from: <http://wiki.splunk.com/Community:PerformanceTroubleshooting> [Accessed: 13 November 2014].

Springboard. (2014). *Career of the Week in Public Health: Forensic Pathologist*. 10 February. [Online] Available from: <http://www.springerpub.com/w/public-health/career-of-the-week-in-public-health-forensic-pathologist/> [Accessed: 12 April 2014].

SQS. (2012). *SQS identifies the highest profile software failures of 2012*. [Online] Available from: <http://www.sqs.com/portal/news/en/press-releases-173.php> [Accessed: 8 February 2014].

SRI International. (1981). *Computer Science Laboratory*. 15 March. [Online] Available from: <http://www.csl.sri.com/> [Accessed: 17 June 2013].

Stamou, K.; Kantere, V.; Morin, J.-H. (2013). SLA data management criteria. In: *Proceedings of IEEE International Conference on Big Data, Silicon Valley, California, U.S.*, pp. 34-42.

Stephenson, P. (2003). Modeling of post-incident root cause analysis. *International Journal of Digital Evidence*, 2(2).

Stephenson, P. (2004). The application of formal methods to root cause analysis Of digital incidents. *International Journal of Digital Evidence*, 3(1).

Sujan, M.A. (2012). A novel tool for organisational learning and its impact on safety culture in a hospital dispensary. *Reliability Engineering & System Safety*, 101, 21-34. May.

Swedien, J. (2010). Software failure linked to women's death. *Daily Globe*. 2 June. [Online] Available from: <http://www.dglobe.com/event/article/id/36982/> [Accessed: 28 March 2013].

Symantec. (2012). *State of the Data Center Survey – Global Results*. September. [Online] Available from: http://www.symantec.com/content/en/us/about/media/pdfs/b-state-of-data-center-survey-global-results-09_2012.en-us.pdf?om_ext_cid=biz_socmed_twitter_facebook_marketwire_linkedin_2012Sept_worldwide_StateofDataCenter. [Accessed: 18 October 2012].

TechMediaNetwork. (2013). *Disk Imaging Software Review*. [Online] Available from: <http://disk-imaging-software-review.toptenreviews.com/> [Accessed: 18 June 2013].

TheFreeDictionary.com. (2013a). *Chain of custody*. [Online] Available from: <http://legal-dictionary.thefreedictionary.com/Chain+of+custody>. [Accessed: 18 May 2013]

TheFreeDictionary.com. (2013b). *Corroborating evidence*. [Online] Available from: <http://legal-dictionary.thefreedictionary.com/corroborating+evidence> [Accessed: 7 August 2013].

TheFreeDictionary.com. (2013c). *Non-repudiation*. [Online] Available from: <http://encyclopedia.thefreedictionary.com/Nonrepudiation>. [Accessed: 7 August 2013].

The RISKS DIGEST. [Online] Available from: <http://catless.ncl.ac.uk/Risks/> [Accessed: 15 March 2013].

Treanor, J. (2012). RBS computer failure to cost bank £100m. *The Guardian*. 2 August. [Online] Available from: <http://www.guardian.co.uk/business/2012/aug/02/rbs-computer-failure-compensation>. [Accessed: 19 March 2013].

Trigg, J. & Doulis, J. (2008). Troubleshooting: What can go wrong and how to fix it. *Practical Guide to Clinical Computing- Systems: Design, Operations, and Infrastructure*. Chapter 7, pp. 105-128. Elsevier: London.

Turner, P. (2007). Applying a forensic approach to incident response, network investigation and system administration using Digital Evidence Bags. *Digital Investigation*, 4(1), 30-35. March.

US Department of Defense. (2005). *Dictionary of Military and Associated Terms*. [Online] Available from: [http://www.thefreedictionary.com/near+miss+\(aircraft\)](http://www.thefreedictionary.com/near+miss+(aircraft)) [Accessed: 30 November 2014].

US Department of Labour. (2010). Accident/Incident Investigation. [Online] Available from: https://www.osha.gov/SLTC/etools/safetyhealth/mod4_factsheets_accinvest.html [Accessed: 30 November 2014].

Usmani, A.S., Chung, Y.C. & Torero, J.L. (2003). How did the WTC towers collapse: A new theory. *Fire Safety Journal*, 38(6), 501-533.

Vacca, J.R. & Rudolph, K. (2010). *System Forensics, Investigation and Response*. Chapter 1, pp. 2-16. Sudbury, Mass.: Jones & Bartlett Learning.

Van der Schaaf, T.W., Lucas, D.A. & Hale, A.R.(1991). *Near-Miss Reporting as a Safety Tool*. London: Butterworth-Heinemann.

Vesely, W.E. (2011). Probabilistic Risk Assessment. In: S.B. Johnson, T.J. Gormley, S.S. Kessler, C.D., Mott, A., Patterson-Hine, K.M. Reichard & P.A. Scandura, *System Health Management: With Aerospace Applications*. Chichester, UK: John Wiley & Sons.

Vijayaraghavan, G.V. (2003). A taxonomy of e-commerce risks and failures. Master's thesis. Florida Institute of Technology.

Vinnem, J.E., Hestad, J.A., Kvaløy, J.T. & Skogdalen, J.E. (2010). Analysis of root causes of major hazard precursors (hydrocarbon leaks) in the Norwegian offshore petroleum industry. *Reliability Engineering and System Safety*, 95(11), 1142-53.

Virtualbox.org. (2014). Welcome to VirtualBox.org! [Online] Available from: <https://www.virtualbox.org/> [Accessed: 17 October 2014].

Viscovery.net. (2014). Viscovery SOMine 6 - Explorative data mining based on SOMs and statistics. 30 August. [Online] Available from: <http://www.viscovery.net/somine/>. [Accessed: 5 November 2014].

Vision Solutions. [2006]. *Understanding Downtime. A Vision Solutions White Paper*. May. [Online] Available from: http://www.mik3.gr/docs/VWP_Downtime.pdf [Accessed: 16 March 2012].

Whittaker, Z. (2011). BlackBerry's outage post-mortem: Where did it all go wrong? *ZDNet*. 10 October. [Online] Available from: <http://www.zdnet.com/blog/btl/blackberrys-outage-post-mortem-where-did-it-all-go-wrong/60801> [Accessed: 26 July 2012].

Willasen, S.Y. & Mjølunes, S.F. (2005). Digital Forensics Research. *Teletronikk Journal*, 1, 92-97. [Online] Available from: <http://www.telenor.com/teletronikk/> [Accessed: 20 May 2013].

Wong, W.E., Debroy, V., Surampudi, A., Kim, H. & Siok, M.F. (2010). *Recent Catastrophic Accidents: Investigating How Software Was Responsible*. Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement.

Worstall, T. (2012). *RBS/NatWest Computer Failure: Fully Explained*. (25 June). [Online] Available from: <http://www.forbes.com/sites/timworstall/2012/06/25/rbsnatwest-computer-failure-fully-explained> [Accessed: 7 February 2013].

Wu, W., Yang, H., Chew, D.A.S., Yang, S., Gibb, A.G.F. & Li, Q. (2010). Towards an autonomous real-time tracking system of near-miss accidents on construction sites. *Automation in Construction*, 19, 134-141. [Online] Available from: <http://202.114.89.42/resource/pdf/5720.pdf> [Accessed: 22 November 2014].

Wyld, H.C. (1961). *The universal dictionary of the English language*. Hazel Watson & Viney LTD, Aylesbury, Bucks, England.

Young, T. (2007). *Forensic Science and the Scientific Method*. [Online] Available from: <http://www.heartlandforensic.com/writing/forensic-science-and-the-scientific-method>. [Accessed: 08 June 2013].

ZDNet.com. (2005). Average large corporation experiences 87 hours of network downtime a year. January 20. [Online] Available from: <http://www.zdnet.com/blog/itfacts/average-large-corporation-experiences-87-hours-of-network-downtime-a-year/268> [Accessed: 26 March 2012].

Zhou, Q. & Yu, T.X. (2004). Use of high-efficiency energy-absorbing device to arrest progressive collapse of tall building. *Journal of Engineering Mechanics*, 130(10), 1177-1187.