Proceedings of the Tenth International Symposium on
Human Aspects of Information Security & Assurance (HAISA 2016)

Near-Miss Analysis and the Availability of Software Systems

J.H.P. Eloff and M.A. Bihina Bella,

Cybersecurity & Data Science Research Groups, Department of Computer Science,
University of Pretoria, Pretoria, South Africa
e-mail: eloff@cs.up.ac.za, mbihina@yahoo.fr

Abstract

Software failures often result in unavailability of systems causing disasters ranging from financial loss to loss of lives. Preventing their recurrence is therefore absolutely necessary. To this end, a post-mortem investigation of a software failure is usually conducted to identify its root cause. However, these investigations most often lack efficiency and accuracy, as they are dependent on human expertise and level of knowledge of the system, and are therefore subjective in nature. Furthermore, investigating a software failure can be challenging due to the usually high volume of failure data - such as log entries - to be scrutinised. To address this problem, near-miss analysis is proposed. Near-miss analysis is an incident investigation technique that detects indicators of a likely failure before the failure unfolds. As these indicators – known as near misses – that are very close to the point of failure, they are most likely to point to its root cause. Near-miss analysis therefore offers an objective method to root-cause analysis based on the data collected from the near misses. The near-miss analysis method proposed in this paper is based on the pattern analysis of a software system's behaviour close to a failure in order to identify near misses. The viability of the proposed method is demonstrated through an experiment.

Keywords

Software failure, near miss, pattern analysis

1. Introduction

Software failures disrupt the availability of a system, which results in data loss and data corruption and compromises the integrity of the related information. This often causes disasters ranging from financial loss to loss of lives. Preventing the recurrence of such major software failures is therefore crucial. Availability of software systems and information assets in general, together with confidentiality and integrity, forms the primary building blocks for safeguarding information and the related systems (Pfleeger & Pfleeger, 2007). Availability of software systems and in particular software failure analysis is of particular interest for the research at hand.

Consider for example the unavailability of a banking system 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 transact (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 the batch-scheduling software at the bank. As a result, deposits were not reflected in bank accounts, payrolls were delayed, credit ratings downgraded and utility bills not paid. In November 2014, RBS was fined 56 million pounds by British regulators for the 2012 outage (BBC News, 2014).

In cases like the above-mentioned a post-mortem investigation is usually required to identify their root cause. However, history shows that such an investigation is often conducted inefficiently and inaccurately, as it is dependent on human expertise and skills. Most often, investigators initially "diagnose" the software failures based on their own experience with the system and then do a number of troubleshooting attempts. Furthermore, there is no common procedure that an investigator can follow for investigating software failures in order to identify the root cause. This leads to a root cause investigation process that is based on human subjectivity. Other methods followed for failure analysis, although valuable, focus on performance improvement and not on preventing the recurrence of software failures; hence some manual guessing about the root cause of the failure is required (Neebula.com, 2012).

Due to the prevalence of catastrophes such as the Royal Bank of Scotland example quoted above, various studies (Stephenson, 2003; Hatton, 2004; Corby, 2011; Meyer, 2011) have focused on improving the usual ad-hoc approach to root-cause analysis. Most often adding structure and formal modelling to the investigation process does this. The research at hand rather focuses on the enablement of investigators to make the root cause investigation process more objective as well as to generate scientific evidence. This is accomplished through introducing near-miss analysis as a technique for investigating software failures (Bihina Bella et al. 2011).

Near-miss analysis is a technique used in the domain of risk analysis and safety for the prevention and investigation of accidents. Near-miss analysis refers to the detection and causal analysis of near misses. By definition, a near miss is a high-risk event that could have led to an accident, but did not due to some timely intervention or by chance (Jones et al. 1999). Almost all major accidents are preceded by a number of near misses (Phimister et al. 2004). Near misses are therefore warning signs or indicators of an upcoming failure. However, contrary to other indicators preceding the failure, a near miss is the closest to the point of failure; in other words, it is the closest to the time window during which the failure occurs. This concept can be better explained with an example.

Consider for instance a potential car collision at a busy intersection. This potential accident could have been preceded by the following sequence of events: (1) a driver crossing a red traffic light, (2) the driver over speeding, and (3) the driver struggling to slow down when noticing an incoming car. In the above scenario, the last high-risk event, Event (3), is the near-miss event as it is the closest to the potential crash. The fact that the collision was avoided, maybe due to the carefulness of the driver of the incoming car, makes this sequence of events a near miss.

As near misses point to the possibly last indicator of an impending failure, they provide a fairly complete set of data about that failure. In the case of a software application, this data is likely to contain behavioural patterns close to the point of failure that can hint at its root cause. The evidence collection effort can therefore be limited to data about these patterns, eliminating the collection of less relevant data.

Common causes of software failures include resource exhaustion and logic errors. Resource exhaustion such as running out of available memory is taken as a use case for the research at hand. Consider for example a software system developed in C++ that does not give adequate warning should external memory become unavailable. This type of software failure can only be located through investigating output provided by the system executing. This output is most often in the form of failures logs and can be generated by either the operating system or the application itself. It is for this reason that the paper at hand demonstrates the application of near-miss analysis for investigating and preventing software failures by means of failure log analysis. This near-miss analysis is based on the pattern analysis of software failures.

The remainder of this paper is organised as follows. Section 2 reviews previous work on near-miss analysis to assess its suitability for software failure analysis. Section 3 presents our proposed near-miss analysis method. Finally, Section 4 presents an experiment to demonstrate the viability of the proposed method.

2. Previous work on near-miss detection

The detection of near misses usually involves assessing the risk level of an observed unsafe event or calculating the likelihood that this event can lead to a failure. Examples of such events include the degradation of plant conditions and the failures of safety equipment (Belles et al. 2000). A common technique proposed for this purpose is Bayesian statistics (Belles et al. 2000) to calculate the conditional probability of an accident given the occurrence of the risky event. Probabilistic risk analysis (PRA) is also a recurring suggestion. PRA 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). Some research has also been conducted to find generic metrics or signs of an upcoming accident, such as equipment failure rates (Leveson, 2015).

Some qualitative approaches to near-miss detection have also been proposed. For instance, in some organizations, the detection of near misses is simply based on a listing of potential hazards such as a toxic chemical leak or an improperly closed switchbox (Ritwik, 2002). These examples are often obtained from incident reporting systems. Another technique also used is the Delphi method. The Delphi method is a group decision-making tool that can be used to obtain information on the probability of an accident from a panel of experts (Pimister et al. 2004).

In all the above techniques, near misses are usually identified as those events that exceed a predefined level of severity. This limits their application to software failures. Indeed, 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. Besides, generic metrics of near misses might not be applicable to all types of systems and all types of failures. Another major limitation of the above techniques is the fact that they rely on the observation of physical events or conditions. However, in the case of software failures, some near misses might not be visible at all, as no failure actually occurred. A more flexible method to detect near misses is required in the case of software failures. The method proposed in this paper is described in the next section.

3. Proposed near-miss method for software failure analysis

The purpose of the near miss analysis method discussed in this section is twofold: Firstly, it demonstrates how to collect relevant information about an unfolding software failure. Secondly, it shows how to use the collected information for constructing near miss indicators that can play a role for the prevention of similar software failures in the future. This is illustrated in Figure 1.

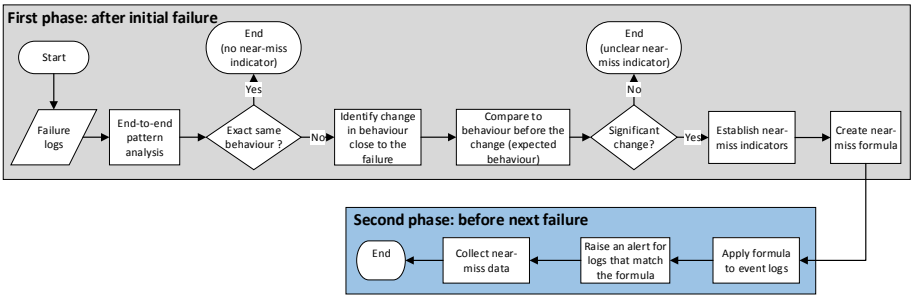


Figure 1: High-level flowchart of proposed near-miss method

The first phase is conducted after the occurrence of a new failure, from the analysis of the failure logs. The logs are analysed to compare the expected system's behaviour to the behaviour close to the failure in order to identify some "warning signs" of the failure. The expected behaviour is derived from the end-to-end pattern analysis of the system's transactions. The goal of this phase is the creation of a near-miss formula that defines near misses for the failure at hand. The near-miss formula combines all the identified warning signs of the failure, which we refer to as "near-miss indicators". The formula is a mathematical expression of the interdependencies between the near-miss indicators. These indicators signal a significant deviation from the operational expectation of a monitored system. The near-miss formula will obviously be specific to the failure and system at hand, but the process to create the formula can be applied to any system or failure type.

The second phase is the detection of near misses at runtime before the reoccurrence of a similar failure. The near-miss formula is applied to the logs of the monitored system as they are generated so that potential near misses can be detected prior to a failure. When some log entry matches the formula, an alert is raised and data relating to the near-miss indicators is collected for the suspicious log. This data is then used

as evidence for the ensuing root-cause analysis if the failure unfolds. This near-miss detection method identifies the information relevant to the root-cause analysis and it indicates when to collect this information.

4. Application of the near-miss detection method

This section demonstrates the application of the near-miss detection method through an experiment. The experiment shows the analysis of a software failure with a view to identifying near-miss indicators, so it is limited to the first phase of the method. The demonstration follows a scientific method: formulating a hypothesis, predicting evidence for the hypothesis, and testing the hypothesis with an experiment (Bernstein, 2009). Furthermore, it employs SOM (Self-Organizing-Maps) (Engelbrecht, 2007) data analysis technique to analyse the logs of the failure.

The reason for using SOM is that it enables pattern identification in big data sets. Patterns in a system's behaviour can be used to signal an unfolding systems' crash. Furthermore SOM is an unsupervised learning technique, which is useful seeing that the causes for a system crash are unknown before the crash occurs. Unsupervised learning classifies input data, represented as vectors, based on similarity. Similar vectors are grouped in the same cluster. For the research at hand these clusters can be used to signal a change in the system's behaviour from expected to unexpected. Previous work of the authors of the paper at hand focussed on the suitability and efficiency of the SOM investigations for forensic investigations (Fei et al. 2005).

4.1. The failure logs

Two types of logs were deemed relevant for this experiment: logs created by a software application busy executing as well as logs generated by the operating system. The following process was followed for obtaining the logs.

Logs generated by the application

A software failure whereby a C++ program would exhaust the memory of a flash disk was used for this experiment. The C++ program was running on a Linux machine and used a loop-structure that repetitively copies a video clip to a flash disk. Since a large data set was required for the subsequent SOM analysis, the crash file was chosen to maximise the number of records. This was done by running the program with the largest flash disk (128 GB) and the smallest video file at hand (3.91 MB), which resulted in a maximum of 31 001 potential records in the crash file (128 GB/3.91 MB). The size of the program's loop was set to be higher than 31 0001. 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 written to a file, subsequently referred to as the crash file. A total of 13 statistics were recorded, including the duration of a file operation (i.e. copying of the video clip), the latency (i.e. time delay between two file operations) and the associated memory statistics such as the amount of RAM used (Mem Used) and the amount of RAM used for caching of data (Cached). The latency and the duration were expressed in milliseconds (ms). Figure 2 shows a screenshot of the first entries in the crash file.

File Nr	Creation time	Mem Used	Mem free	Buffers	Cached	Swap Used	Swap Free	USB free space	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

Figure 2: Screenshot of crash file

System logs

The C++ program was performing significant input (reading copy of video clip) and output (copying video clip to new file) operations, it was therefore deemed most appropriate to use the iotop monitoring utility to show input and output (I/O) usage on the Linux disk. The iotop command continuously displays I/O statistics (9 in total) such as disk-reading and disk-writing bandwidth (Linux.die.net. 2014). The I/O statistics were used to corroborate the information in the crash file. Figure 3 shows a screenshot of the iotop output file.

Time	TID	PRIO	User	Disk read (kB/s)	Disk write (kB/s)	Swapin (%)	I/O (%)	Command
02:35:54	131	be/3	root	0	281.83	0.00	6.51	[jbd2/sda1-8]
02:35:55	3945	be/4	root	0	7.58	0.00	0.00	python /usr/sbin/iotop -ktoqqq -d .5
02:35:57	3950	be/4	root	1129.04	0	0.00	99.99	./videoCrashTwoFolders-V2
02:35:58	3950	be/4	root	1036.41	7.63	0.00	80.37	./videoCrashTwoFolders-V2
02:35:58	3950	be/4	root	2754.27	0	0.00	96.62	./videoCrashTwoFolders-V2
02:35:59	3950	be/4	root	2753.8	0	0.00	95.51	./videoCrashTwoFolders-V2
02:35:59	3950	be/4	root	3004.32	0	0.00	97.65	./videoCrashTwoFolders-V2
02:36:00	3950	be/4	root	2744.16	0	0.00	95.94	./videoCrashTwoFolders-V2
02:36:00	3950	be/4	root	2237.68	0	0.00	96.92	./videoCrashTwoFolders-V2
02:36:00	131	be/3	root	0	54.72	0.00	7.59	[jbd2/sda1-8]
02:36:00	3945	be/4	root	0	7.82	0.00	0.00	python /usr/sbin/iotop -ktoqqq -d .5
02:36:01	3950	be/4	root	2703.35	0	0.00	93.62	./videoCrashTwoFolders-V2

Figure 3: Screenshot of `iotop` output file

4.2. The SOM implementation tool

A commercial SOM tool, Viscovery SOMine (Viscovery.net, 2014), was used.

4.3. The test plan - analysis of the software failure

The root-cause analysis was conducted with a view to identifying near-miss indicators. Identifying near-miss indicators was based on the assumption that it was possible to see the failure emerging by monitoring the relevant attributes - such as memory usage statistics - provided in both the crash file and the iotop output file. Indeed, it was expected that the C++ program would have a stable operating mode under normal conditions (when enough memory was available on the flash disk) and that this normal behaviour would be disrupted when memory became insufficient. Therefore the analysis was expected to reveal some unusual changes in the monitored attributes close to the exhaustion of the flash disk free space.

4.3.1. Analysis of the crash file

The analysis of the crash file followed the scientific method as follows.

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 this

experiment, the source of the failure was already known (chosen) to be memory exhaustion that results in performance degradation. Nonetheless, it is more important to understand that the purpose of this demonstration is not only to identify a root cause but rather to identify indicators that contribute to the root cause of a failure.

Predict evidence for the hypothesis - Indicators of performance degradation 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.

Test hypothesis with experiment - It was assumed that the above pattern in the memory statistics would be visible from a pattern analysis of the behaviour of the system (Linux machine) as the program was running. Profiling the system's behaviour was performed in three steps. Firstly, patterns in the overall end-to-end behaviour were outlined. Then the focus shifted to the system's behaviour close to the point of failure, and finally a comparison between these two profiles was performed.

Behaviour of the system before the failure

In order to observe patterns in the system's behaviour, we created SOM maps for several random sets of 1000 records throughout the crash file. Four sets of records were selected: first 1000, 10 000 to 11 000, 20 000 to 21 000 and the last 1000 before the failure. In line with the expected evidence for memory exhaustion mentioned earlier, the focus of the SOM analysis was on the following attributes in the crash file: Creation Time, Buffers, Cached, Swap Used, Duration and Latency. Therefore the SOM component maps were only generated for the attributes mentioned above. 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. 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 dark blue and highest values are in red. Clusters in the data set are delimited by black lines on the maps. Each cluster groups records with close values for the various attributes together. The maps for Latency are shown in Table 1 as an illustration. As we used the trial version of the Viscovery SOMine tool for the SOM analysis, an "Evaluation only" watermark appears on the maps.

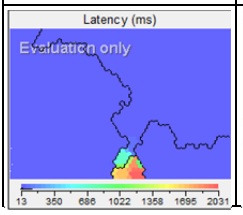
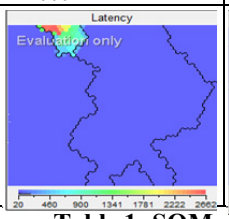
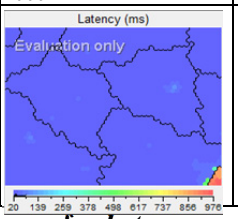
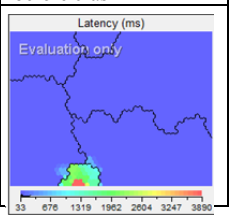
Records 1-1000	Records 10 000 to 11 000	Records 20 000 to 21 000	Last 1000 records before crash
			

Table 1: SOM maps for *Latency*

Results

A study of the component maps shows that the values for the various attributes listed above remain fairly constant throughout the execution of the C++ program. For instance, *Duration* remains around 1000 ms, with occasional big jumps throughout the various data sets. However, one attribute that shows a distinctive change throughout the program as well as close to the failure is *Latency*. Indeed, *Latency* increases over time. As shown in Table 1, the minimal value goes from 13 ms to 20 ms and finally to 33 ms and the maximum value increases from 2031 ms to 3890 ms. There are occasional big increases, but the biggest increase occurs in the last data set, closer to the failure (3890 ms). In order to find more usable information about the observed pattern in *Latency*, a more detailed SOM analysis for that attribute was conducted. The analysis was performed with data sets close to the failure and is described in the next section.

Behaviour of *Latency* close to the failure

A more detailed analysis of *Latency* was performed with the last 50 and the last 100 records before the failure. An examination of the resulting component maps confirmed the previous observations with more specific evidence. The SOM maps showed that in the last 100 records, *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. The SOM maps also indicated a lack of homogeneity in the records close to the point of failure. Indeed, the number of clusters in the data of the last 50 records was considerably higher than the number of clusters in the previous data sets. This indicates that these records are erratic in terms of the other attributes used to train the maps, confirming the lack of correlation between *Latency* and the other attributes.

Conclusion based on 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 increase in *Latency* was used as our first near-miss indicator. After establishing a near-miss indicator from the crash file, the same analysis was conducted with the `iotop` output file.

4.3.2 Analysis of `iotop` output file

The analysis of the `iotop` output file followed the same process as with the crash file. SOM maps for the same sets of 1000 records were generated for the following attributes in the file: *Time, Disk read* (disk reading band-width)*, Disk write* (disk writing bandwidth), *I/O* and *Command* (process name). Significant changes were observed in *Disk read, Disk write,* and *Command* and were used to identify near-miss indicators, which are specified below.

- The number of running processes declines towards the point of failure.
- The values of *Disk read* are more than double the overall average.

- In the last few hundred records before the failure, the value of *Disk write* drops to 0 at various instances.

4.4. Evaluation of experiment

Although the experiment was based on a failure with a simple software application and did not demonstrate the full implementation of the near-miss detection method, it was successful in the sense that it showed the benefit of this approach in terms of an objective investigation of software failures. Indeed, out of the 13 initial attributes in the crash file and the 9 attributes in the `iotop` output file, only 4 (*latency, processes, disk-reading bandwidth* and *disk-writing bandwidth*) proved relevant for near-miss detection and hence for the root-cause analysis. This is a significant reduction in the volume of data to be analysed. Since the SOM algorithm is optimised for large data sets, it is expected that the process followed to identify near-miss indicators can scale to a real-life failure with a higher number of logs than was used in the experiment.

It is worth noting that the identified near-miss indicators are system-level patterns that would not be visible to the end-user otherwise. It is also important to notice that these indicators are specific to the software failure at hand, the conditions of its occurrence (lab experiment) and its analysis (`iotop` used for correlation to program's logs). However, they can be a starting point for the identification of near misses for similar types of failures in the future once a near-miss formula has been created. A deeper analysis of the collected evidence can potentially explain the observed patterns (e.g. fluctuating number of processes) and find their root cause. Ultimately a repository of near-miss indicators and formulas for various types of failure could be obtained to facilitate their analysis.

The experiment discussed suffers some limitations that will be addressed in future work. The experiment only implemented the near-miss detection process and the analysis of a software failure, referred to as Phase 1 of the method in Fig 1. The failure investigated for the experiment 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 use case.

5. Conclusion

This paper proposed the use of near-miss analysis for the enablement of investigating software failures. A method was proposed to detect near misses in software applications through the identification of unusual patterns in the system behaviour close to a likely failure. The viability of the method was demonstrated through an experiment that applied the scientific method combined with the near-miss detection method to identify relevant evidence of a software failure. Results of the experiment are promising but need to be further validated through the creation of a near-miss formula based on this method to determine whether near misses can be accurately detected at runtime. A prioritisation mechanism to only collect data for the near

misses with the highest risk level might also be required to handle possible false alarms.

6. References

BBC News. (2014), "RBS fined £56m over 'unacceptable' computer failure", 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. and Hamzehee, H. (2000), "Precursors to Potential Severe Core Damage Accidents: 1998 – A Status Report", http://pbadupws.nrc.gov/docs/ML0037/ML003733843.pdf, (Accessed 02 April 2013)

Bernstein, M. (2009), "Scientific Method Applied to Forensic Science", http://marybernstein.wordpress.com/2009/05/27/scientific-method-applied-to-forensic-science, (Accessed 18 December 2014)

Bihina Bella, M.A., Olivier, M.S. and Eloff, J.H.P. (2011), "Proposing a Digital Operational Forensic Investigation Process", Proceedings of the 6th International Workshop on Digital Forensics and Incident Analysis, London, UK, 2011

Corby, M.J. (2011), "Forensics: Operational", in McGhie, L. (Ed). Encyclopedia of Information Assurance. Taylor & Francis, ISBN: 1-4200-6620-X.

Engelbrecht, A.P. (2007), Computational Intelligence: An Introduction, 2nd edition. John Wiley & Sons, Ltd, ISBN: 978-0-470-03561-0.

Fei, B., Eloff, J., Venter, H., and Olivier, M. (2005), Exploring Forensic Data with Self-Organizing Maps. Advances in Digital Forensics, Vol. 194, pp113-123. Springer.

Finnegan, M. (2013), "RBS apologises as customers hit by another IT outage", Computerworld UK, http://www.computerworlduk.com/news/ it-business/3491865/rbs-apologises-as-customers-hit-by-another-it-outage, (Accessed 5 February 2013)

Hatton, L. (2004), "Forensic software engineering: An overview", http://www.leshatton.org/wp-content/uploads/2012/01/fse_Dec2004.pdf, (Accessed 5 May 2012).

ISO/IEC 27037. (2012), "Information technology — Security techniques — Guidelines for identification, collection, acquisition, and preservation of digital evidence", http://www.iso.org/iso/catalogue_detail?csnumber=44381, (Accessed 8 April 2015)

Jones, S., Kirchsteiger, C. and Bjerke, W. (1999), "The importance of near miss reporting to further improve safety performance", Journal of Loss Prevention in the Process Industries, Vol.12, pp59-67

Leveson, Nancy. (2015), "A systems approach to risk management through leading safety indicators" Reliability Engineering and System Safety, Vol.136, pp17–34

Linux.die.net. (2014), "Iotop(1) – Linux man page", http://linux.die.net/man/1/iotop (Accessed 14 November 2014)

Meyer, B. (2011), "Again: The one sure way to advance software engineering", ACM communications blog, http://cacm.acm.org/blogs/blog-cacm/101891-again-the-one-sure-way-to-advance-software-engineering/fulltext, (Accessed 17 February 2012)

Neebula.com. (2012). Success Factors for Root-Cause Analysis. [Online] Available from: http://www.neebula.com [Accessed: 26 March 2013].

Pfleeger, C.P. and Pfleeger, S.L. (2007), Security in Computing, 4th edition, Pearson Education, Inc, United States.

Phimister, J., Vicki, R., Bier, M. and Kunreuther, H.C. (2004), Accident Precursor Analysis and Management: Reducing Technological Risk through Diligence, National Academies Press, http://www.nap.edu/catalog/11061.html. (Accessed 15 May 2012)

Ritwik, U. (2002), "Risk-based approach to near miss", Hydrocarbon Processing, pp93-96

Stephenson, P. (2003), "Formal Modeling of post-incident root cause analysis" International Journal of Digital Evidence, Vol. 2, Issue 2

Viscovery.net (2014), "Viscovery SOMine 6 - Explorative data mining based on SOMs and statistics", http://www.viscovery.net/somine, (Accessed 5 November 2014)