# A near-miss management system architecture for the forensic investigation of software failures

M.A. Bihina Bella and J.H.P. Eloff
ICSA Research Lab, Computer Science Department, University of Pretoria, Pretoria, South Africa

**Abstract**— Digital forensics has been proposed as a methodology for doing root-cause analysis of major software failures for quite a while. Despite this, similar software failures still occur repeatedly. A reason for this is the difficulty of obtaining detailed evidence of software failures. Acquiring such evidence can be challenging, as the relevant data may be lost or corrupt following a software system's crash. This paper proposes the use of near-miss analysis to improve on the collection of evidence for software failures. Near-miss analysis is an incident investigation technique that detects and subsequently analyses indicators of failures. The results of a near-miss analysis investigation are then used to detect an upcoming failure before the failure unfolds. The detection of these indicators – known as near misses – therefore provides an opportunity to proactively collect relevant data that can be used as digital evidence, pertaining to software failures. A Near Miss Management System (NMS) architecture for the forensic investigation of software failures is proposed. The viability of the proposed architecture is demonstrated through a prototype.

Keywords— Software failure, near miss, near-miss management system (NMS), digital evidence, digital forensics

## 1. Introduction

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". Therefore, for the purposes of the research in hand, a software failure is defined as the unplanned cessation of a software system to function as specified. Software systems can fail for various reasons including system overload, logic errors, security breaches, human errors, and glitches in routine maintenance operations (e.g. failed software upgrade) (Pertet & Narasimhan, 2005). As software is embedded in a range of devices and plays a vital role in a number of industries, a failed software application can affect any area of a user's day-to-day life and may even be fatal.

Consider for example the various cases of software failures in medical devices such as radiation therapy machines, external infusion pumps and implantable pace makers. In radiaton therapy machines in particular, failures of the embedded software system causes serious problems such as overdosage of radiation and administration of incorrect treatment that result in severe burns or deaths of the affected patients. Such catastrophic cases of radiation therapy software glitches have been reported many times in the media (Bogdanich & Rebelo, 2010) as well as on the portals of the FDA (the U.S. Food and Drug Administration) (FDA, 2013) and the IAEA (International Atomic Energy Agency) (IAEA, 2013).

Disastrous events as the above mentioned often result in lawsuits where a thorough post-mortem investigation is conducted. Comprehensive forensic reports are available for these cases, but do not address the software aspect of the investigation. Such an investigation is absolutely necessary to prevent the recurrence of these catastrophes. To this end, a digital forensic investigation is required to understand the root causes involved in the software failures.

Digital forensics is the process of methodically examining computer media as well as network components, software and memory for digital evidence (Vacca and Rudolph, 2010). This evidence is usually in the form of system logs, but may include other relevant data such as digital images. The digital evidence is used to provide clarity on the cause and circumstances of a computer-based event in support of the criminal justice system. As such, digital forensics is primarily used for the investigation of computer crimes and security-related events (e.g. breach of company policy). Nevertheless, we argue that it can also be applied to non-criminal events such as catastrophic failures that require a court case as the examples provided earlier. In such cases, using digital forensics instead of existing informal failure analysis techniques has the benefits of providing results admissible in a court of law due to the scientific foundation and the sound digital evidence used for the root-cause analysis.

However, being a reactive process, digital forensics can only be applied after the occurrence of a failure. This limits its effectiveness as data that could serve as potential evidence may be destroyed during and after the failure. Acquiring such data is necessary for the validity of the results of the forensic investigation. The international ISO/IEC 27037 standard – Guidelines for the Identification, Collection, Acquisition and Preservation of Digital Evidence (ISO/IEC 27037, 2012) – indeed recommends that the evidence collection should be prioritised based on volatility.

In order to address this limitation of digital forensics, it is suggested that 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 high-risk conditions, so-called forerunners to failures, are known as 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). Contrary to other precursors to 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 the following example.

*Corresponding author. Tel.: +27 731497384

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 overspeeding; 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.

Near-miss analysis, which refers to the detection and subsequent analysis of near misses, is a technique used in the domain of risk analysis and safety. Like a forensic investigation, near-miss analysis attempts to identify the root cause of accidents and prevent their recurrence and has been used successfully in various industries for decades (Phimister et al., 2004). It is suggested in this paper that this technique should also be applied to the forensic investigation of software failures. Reason being that the output of a near-miss analysis investigation can be used as digital evidence. Furthermore, it broadens the scope of a forensic investigation so to also prevent the recurrence of similar software failures. Indeed, as near misses point to the possibly last indicator of an impending failure, they provide a fairly complete set of data about that failure. By alerting system users of an upcoming failure, an opportunity is provided to collect this data at runtime and potentially prevent the failure from unfolding.

Near-miss analysis 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 that task.

Preliminary partial solutions to these challenges were presented in Bihina Bella et al. (2011) and Bihina Bella et al. (2012) respectively for digital forensics and near-miss analysis. An initial near-miss management model based on these solutions was presented as work-in-progress in Bihina Bella et al. (2014). The current paper presents the revised model and original NMS architecture that resulted from this previous work.

## 2. Overview of NMS

This section provides some background information on NMSs. It first presents the types of NMSs currently available and then reviews their functionality.

### 2.1 Types of NMSs

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 NMSs indicated that most of the research on near-miss analysis focuses on single NMSs.

Initially limited to the nuclear (Phimister et al., 2004) and aviation industries (NASA, 2006), research on the design of effective NMSs has received much attention in a wide range of industries over the last couple of years (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 et al., 2001; Aspden et al., 2004; Fried, 2009). Most NMSs in use today are proprietary systems designed specifically for the organisation that uses them. Barach and Small (2000) provide a comprehensive list of proprietary NMSs in various industries.

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

### 2.2 Functionality of NMSs

An ideal NMS is required to perform all activities pertaining to near-miss analysis. These activities are summarised in the following diagram by Phimister et al. (2000). The diagram uses the following notation:

Dissem: shortcut for dissemination of information
R.C.A: Root-cause analysis
Sol. I.D.: Solution identification

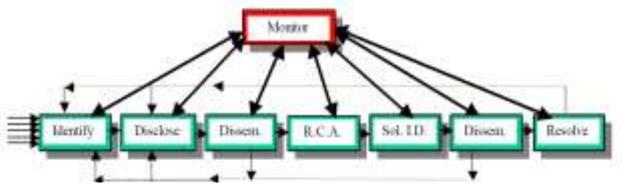


Figure 1: Near-miss management process (Phimister et al., 2000)

However, 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

#### *2.2.1 Techniques for the identification of near misses*

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.

For instance, 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).

Observed near misses such as the above are most often reported manually into the NMS. As such, NMSs are often called near-miss reporting systems. However, 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.

#### *2.2.2 Techniques for the prioritisation 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). 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. Only near misses closest to the impending accident are retained as they offer the most complete data about the particular accident. Various quantitative and qualitative approaches are used to prioritise near misses across industries.

The two main approaches used to prioritise near misses are risk-based classification and statistical analysis.

Risk-based classification ranks near misses according to the severity level of their potential consequences or their frequency. Determining the severity of the likely impact of a near miss can be done with a risk decision matrix that assigns a weight to the near-miss "relevancy" to help identify the potential worst-case scenario (Ritwik, 2002). Kleindorfer et al (2012) also propose the risk level of a near miss to be proportional to the amount of time that the event caused the system to exceed predefined safety and quality limits. 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.

In terms of statistical analysis, Bayesian statistics is often proposed to estimate the frequency of severe accidents based on the frequency of observed near misses (Bier & Mosleh, 1990; Johnson & Rasmuson, 1996). Other factors used to classify near misses include the existence of initiating events and the probability of successful recovery (Cooke & Goossens, 1990). 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).

#### *2.2.3 Techniques for the root-cause analysis of near misses*

As near misses and accidents have common causes, identifying the cause of a near miss is a valid method to identify the cause of the ensuing or potential accident (Andriulo & Gnoni, 2014). Root-cause analysis of near misses can be performed with investigation techniques taken from engineering disciplines, such as fishbone diagrams, event and causal factor diagrams, and failure mode and effects analysis (Phimister et al., 2003; Jucan, 2005; Hecht, 2007).

Applying the above techniques to near-miss analysis in the software industry presents a number of challenges that are discussed next.

## 3 Challenges to applying existing NMS techniques to the software industry

### 3.1 Challenges to the identification 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.

### 3.2 Challenges to the prioritisation of near misses

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 (Phimisteret al, 2004). Besides, generic metrics of near misses might not be applicable to all types of systems and all types of failures.

### 3.3 Challenges to the 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 as are discussed below.

The first challenge is the lack of forensic tools and techniques appropriate for software failure analysis. 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 through a hash algorithm) 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).

Another challenge to the application of the digital forensic process to software failure analysis is the need to minimise downtime following a software failure. System downtime can be very costly and minimising its duration is critical. This usually 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 the system is restored (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.

As indicated by the above, some work is still required to detect, prioritise and analyse near misses from a software system perspective. Requirements to guide this work for the design of an effective NMS are established in the next section.

### 3.4 Requirements for the NMS for the software industry

In the light of the above overview of NMSs and the challenges to near-miss analysis discussed earlier, the authors of this paper have identified requirements for an effective NMS for the software industry. Amongst others, the following requirements are discussed briefly:

- **A generic model** that is relevant to the software industry at large. The near-miss analysis techniques must be applicable to various types of systems and should be able to handle different types of software failures and near misses.
- **An automated system**. In order to limit human error and subjectivity, the identification and prioritisation of near misses as well as the evidence collection process should be automated.

## 4 Proposed solutions for the NMS design

### 4.1 Detection of near misses

Since near misses are defined in relation to the associated accidents, the definition of a software failure is used as a starting point to define and detect near misses in the software industry.

In the Introduction of this paper, a software failure was defined as an unplanned cessation of a software system to function as specified. Indeed, software systems are designed according to specifications in line with the requirements of the customer and intended users. The system specifications can be functional (the functionality and features of the system) as well as non-fonctional (such as quality of service, response time and uptime). Since no system is immune to malfunction, no system vendor can guarantee that the system will work continuously and perfectly as specified at all time. In other words, some periods of malfunction and downtime are expected. For this 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 a service level agreement (SLA) (Sevcik, 2008).

For instance, the performance requirements 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. Hence it violates the SLA and is considered to have failed.

For the purposes of the research in hand, an event is therefore 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. We consequently propose the following 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 specified downtime allowance.*

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. As a matter of fact, not all software systems have formal SLA's in place, although they have documented specifications. Examples include small or trivial applications developed in-house for internal use that are not made available to some external customer.

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, a 95% threshold (95% of the downtime allowed) would correspond to a total monthly downtime of 41 min and 2 s. In this case, a near miss is any monthly downtime of between 41 min and 2 s (the threshold) and 43 min and 12 s (the allowed downtime). This threshold-based definition of a near miss can be mathematically expressed as follows (Equation (1)):

$D_{experienced}$ is the experienced downtime
$D_{allowed}$ is the SLA downtime allowance
$\alpha$ is the near-miss threshold in percentage; $\alpha < 1$
$\alpha \times D_{allowed}$ is the near-miss threshold in time value
If $\alpha \times D_{allowed} \leq D_{experienced} \leq D_{allowed}$ then $D_{experienced}$ is a near miss (1)

Thus, using the above example of a 95% threshold, the following applies:

$\alpha = 0.95$
$D_{allowed} = 43$ min and 12 s

$\alpha \times D_{allowed} = 41$ min and 2 s
If 41 min and 2 s $\leq D_{experienced} \leq$ 43 min and 12 s
then $D_{experienced}$ is a near miss.

Figure 2 shows the downtime-based classification of events explained above.

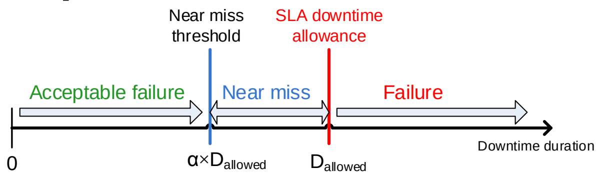


Figure 2: Classification of unsafe events based on their downtime duration

The above definition of a near miss is generic enough to be applicable to any type of software system and failure. It is also flexible enough to be adapted to the risk tolerance of any organisation.

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. An example of this scenario is provided in the prototype implementation in Section 6.

### 4.2 Prioritisation of Near Misses

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 MTTR can be obtained from the system vendor specifications or through historical observations. We are currently exploring formulas from the reliability theory of IT systems (Holenstein et al., 2003) to develop a suitable failure probability formula. 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 (2)):

$D_{expected}$ is the expected downtime due to a failure
$D_{allowed}$ is the SLA downtime allowance
P is the probability of failure, given the current unsafe situation
MTTR is the expected recovery time following an outage
$D_{expected} = P \times MTTR$
If $D_{expected} > D_{allowed} \rightarrow$ critical zone (2)

Only events in the critical zone are passed on for analysis. 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. 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 the failure, the root cause analysis is facilitated by the fact that the event data collection occurred before the system failed.

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 previously, 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 (2) is adjusted as follows:

$\alpha$ is the near-miss threshold; $\alpha \leq 1$
If $D_{expected} \geq \alpha$ x $D_{allowed} \rightarrow$ critical zone (3)

### 4.3 Root-Cause Analysis of Near Misses

The digital forensic process is followed for the root-cause analysis of the near misses and the failures. In Bihina Bella et al. (2011), a new forensic investigation process was proposed to cater for the challenges to failure analysis of the existing digital forensic process. This new process is an adapation of the digital forensic process and has four basic stages. The first two occur immediately after the failure has been detected: firstly, collect digital 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.

Phase 2 was introduced to limit the downtime duration before completing the root-cause analysis, which is critical (see discussion in Section 3.3). Phase 2 starts once all the evidence has been acquired. The failure is then fixed and the system is restored to its operational state as quickly as possible. 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). Such 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.

The root-cause analysis phase corresponds with the analysis phase of a digital forensic investigation. The digital evidence collected in the first phase is examined in a digital forensic laboratory 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, which in general involves three main steps: formulating a hypothesis; predicting evidence for the hypothesis; and testing the hypothesis with an experiment (Bernstein, 2009).

## 5 Proposed NMS architecture

### 5.1 The overall near-miss management 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) are flagged as potential near misses and 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 resulting architecture of an NMS is described next.

### 5.2 The NMS architecture

The proposed NMS architecture is shown in Figure 3. The architecture consists of five main components:

- 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. Each of the main components processes the event logs from the units of the system that is being monitored. The five main components of the system are used to perform a multi-staged filtering process that progressively discards "irrelevant" events and only retains near misses with the highest risk factor. A detailed description of the main components of the architecture follows.

#### *5.2.1 The Near-Miss Monitor*

The Near-Miss Monitor monitors the units of the system to identify potential near misses based on the near-miss definition formula in Equation (1). 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. 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.

#### *5.2.2 The Near-Miss Classifier*

The Near-Miss Classifier is expected to calculate the risk level of the potential near misses based on their failure probability and expected downtime. The complete formula for the failure probability is still a work-in-progress. The Near-Miss Classifier prioritises events according to their failure probability as explained in Equation (3). 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.

#### *5.2.3 The Near-Miss Data Collector*

This component 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. The data is then stored in the Event Data table and transferred to the Failure Prevention module.

#### *5.2.4 The Failure Prevention*

With this component, 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 component.

#### *5.2.5 The Event Investigation*

Based on the outcome of the recovery process in the previous component, the Event Investigation 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 forensic 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 architecture meets the objectives of an effective NMS for the software industry specified earlier. 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. An additional benefit of the architecture is that it enables the prevention of an impending failure if appropriate corrective actions are executed timely.

The prototype implementation of the NMS architecture is documented in the next section.

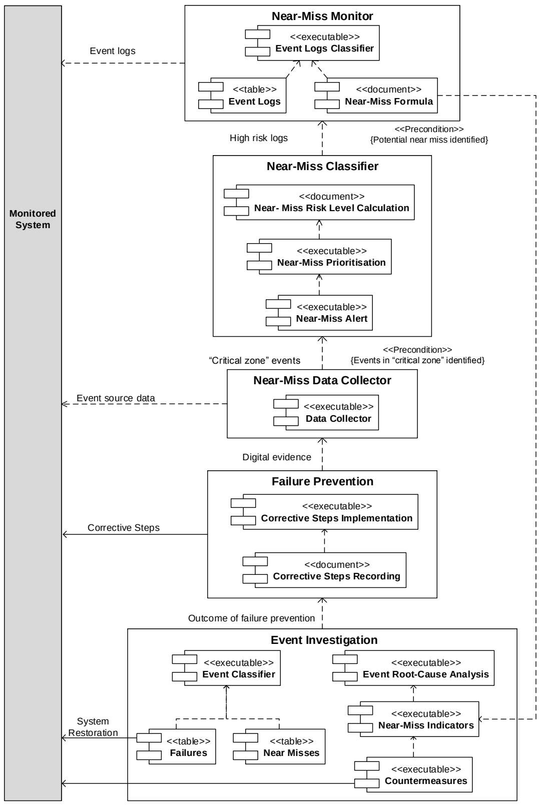

Monitored System
Event logs
Near-Miss Monitor
<<executable>> Event Logs Classifier
<<table>> Event Logs
<<document>> Near-Miss Formula
High risk logs
<<Precondition>> {Potential near miss identified}
Near-Miss Classifier
<<document>> Near- Miss Risk Level Calculation
<<executable>> Near-Miss Prioritisation
<<executable>> Near-Miss Alert
"Critical zone" events
<<Precondition>> {Events in "critical zone" identified}
Near-Miss Data Collector
Event source data
<<executable>> Data Collector
Digital evidence
Failure Prevention
Corrective Steps
<<executable>> Corrective Steps Implementation
<<document>> Corrective Steps Recording
Outcome of failure prevention
Event Investigation
<<executable>> Event Classifier
<<executable>> Event Root-Cause Analysis
<<executable>> Near-Miss Indicators
System Restoration
<<table>> Failures
<<table>> Near Misses
<<executable>> Countermeasures

Figure 3: Component diagram of NMS architecture

## 6 Prototyping the NMS Architecture

In line with the goal of the paper to use the results of a near-miss analysis investigation as digital evidence and so to prevent the recurrence of major software failures, the prototype implementation only addresses the following two aspects of the NMS architecture: the definition and detection of near misses and the root-cause analysis of software failures. The other elements of the model were not discussed in detail in the paper and are therefore not part of the prototype. The goal of the prototype implementation is therefore twofold:

- Demonstrate the viability of the digital forensic process formulated in Section 4.3 to conduct a root-cause analysis of a software 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.
- The identification of near-miss indicators that constitute a near miss.
- The development of a near-miss formula.
- The identification of potential near misses using a set of event logs.
- The detection of near misses at runtime.

The prototype therefore focuses on the following two components of the architecture: Near-Miss Monitor and Event Investigation. As the prioritisation formula is a work-in-progress, it is not included in the prototype implementation.

Conducting a forensic analysis of a software failure was the basis for the prototype's goal. 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, all of which are described in the discussion that follows.

### 6.1 The Test Data Set

In order to demonstrate the viability of the proposed NMS arachitecture, real-life failure logs with no prior knowledge of the cause of the failure were required. However, for confidentiality reasons, such real-life data could not be obtained. We therefore opted to simulate a software failure and generate logs of the event. Two types of logs were deemed relevant for this demonstration: logs created by a simple application being executed as well as logs generated by the computer system. The process descrcibed below was followed for obtaining the logs.

**Logs generated by the application**

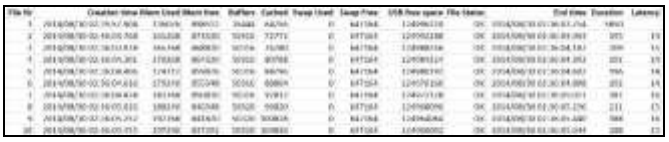
A software failure was simulated by designing a C++ program that would exhaust the memory of a flash disk. The C++ program was running on a Linux machine and was designed to run as a loop structure that repetitively copies a video clip to a flash disk. In this simulation a failure was the inability of the C++ program to copy the video clip to the flash disk. A near miss would occur when the program was close to exhausting the flash disk space or the program would show behavioural patterns similar to the ones observed close to a failure. These behaviourial patterns were used to define near-miss indicators.

The failure simulation program was purposefully designed to be simple so as to focus on the near-miss indicators instead of on the complexity of the software system. For this purpose, the program was intentionally designed to be flawed with no code to monitor disk space in order to enable a failure. In the absence of real-life failure data, a more complex and smarter program could make the illustration and understanding of the near-miss analysis concept more difficult.

Since a large data set was required for the subsequent root-cause analysis, the crash file was designed to maximise the number of records. This was achieved 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). In order to force a failure, the size of the program's loop was deliberately set to be higher than 31 001. It was set to 31 150.

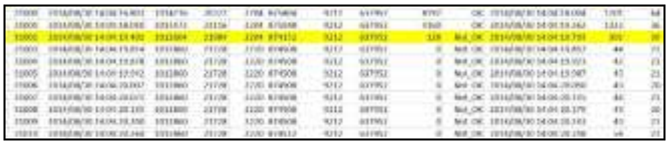
Every time a new copy of the video clip was made, various statistics about the C++ program, the Linux machine and the flash disk were written to a file, subsequently referred to as a crash file. A total of 13 statistics were recorded, including the duration of a file operation (i.e. copying of the video clip), the latency (i.e. time delay between two file operations) and the associated memory statistics such as Mem Used (Amount of RAM used) and Cached (Amount of RAM used for caching of data). The latency and the duration were expressed in milliseconds (ms).

Screenshots of the resulting crash file are provided in Figure 4 (beginning of file) and Figure 5 (point of failure). The highlighted row in Figure 5 (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 | File Status | End time | Duration | Latency |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | |
| 2 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| 3 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| 4 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| 5 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| 6 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| 7 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| 8 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| 9 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| 10 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |

Figure 4: Crash file at beginning of program



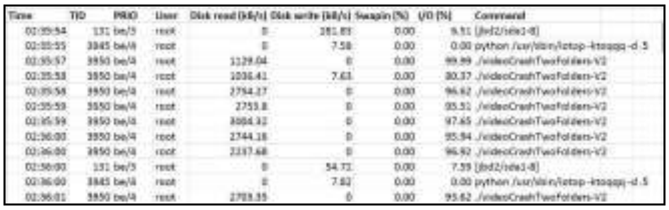
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | Not_OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | Not_OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | Not_OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | Not_OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | Not_OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | Not_OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | Not_OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | Not_OK | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0 | Not_OK | [illegible] | [illegible] | [illegible] |

Figure 5: Crash file – point of failure

**System logs**

Since the C++ 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 `iotop` monitoring utility to show input and output (I/O) usage on the Linux disk. The `iotop` command continuously displays I/O statistics such as disk-reading and disk-writing bandwidth for the various processes running on the system (Linux.die.net. 2014). The I/O statistics were used to corroborate the information in the crash file.

Figure 7 shows a screenshot of the first entries in the `iotop` output file, which had nine fields. The relevant fields for the prototype were *Time, Disk read* (disk-reading bandwidth), *Disk write* (the disk-writing bandwidth) and *Command*, which refers to the name of the process. *Disk read* and *Disk write* were expressed in kilobytes per second (kB/s).



| 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 | 9.51 | [jbd2/sda1-8] |
| 02:35:55 | 3845 | 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 | 30.37 | ./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 | 2755.8 | 0 | 0.00 | 95.51 | ./videoCrashTwoFolders-V2 |
| 02:35:59 | 3950 | be/4 | root | 3084.32 | 0 | 0.00 | 97.65 | ./videoCrashTwoFolders-V2 |
| 02:36:00 | 3950 | be/4 | root | 2744.18 | 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 | 54.71 | 0.00 | 7.59 | [jbd2/sda1-8] |
| 02:36:00 | 3845 | 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 | 95.62 | ./videoCrashTwoFolders-V2 |

Figure 7: `iotop` output file

### 6.2 THE FORENSIC INVESTIGATION TOOL

Ideally, one should use a digital forensic tool to conduct a forensic (root-cause) analysis of the log files. However, these tools are not equipped to handle software failures. Indeed, current digital forensic tools are designed to handle criminal and security-related events, not software failures of an accidental nature. Therefore they are designed to help identify evidence of a known crime, not evidence of a failure whose cause is *a priori* unknown. Since one of the goals of 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, although it is not a digital forensic tool *per se*. The SOM is a powerful pattern classifier algorithm optimised for large data sets (Engelbrecht, 2003), which enables the easy visualisation of clusters in a data set. The suitability and efficiency of the SOM for forensic investigations was already demonstrated by earlier researchers (Fei et al., 2005).

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 experiment. Little pre-processing of the crash file was required to create the SOM maps 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 steps and map creation process are provided below.

- 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.

Since the SOM only provides a visual representation of the patterns in a data set, we also planned on using a statistical analysis to express the observed patterns mathematically. Statistics such as the average and the weighted moving average (WMA) of attributes in the logs were considered relevant for this purpose. A WMA gives more weight to the most recent data in a time series and attaches less importance to older data. It was therefore used for trend analysis and forecasting, which was particularly relevant for the study in hand (Holt, 2004). We planned to compare the normal average to the WMA to determine deviation from normal behaviour.

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:

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

### 6.3 THE TEST PLAN

Since the forensic analysis was conducted with a view to identifying near-miss indicators, the whole demonstration was oriented towards that purpose.

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 maintain a stable operating mode under normal conditions (when enough memory was available on the flash disk that was used as an external memory device) and that this normal behaviour would be disrupted when memory became insufficient. Therefore the forensic analysis was expected to reveal some unusual changes in the monitored attributes close to the point of failure, in other words close to the exhaustion of the flash disk's free space. The correlation between the near-miss indicators would be used to create a near-miss formula to define near misses and detect them as they occur.

### 6.4 Forensic analysis of the crash file

The forensic analysis of the crash file followed the scientific method described 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 demonstration, the source of the failure was already known (chosen) to be memory exhaustion, which 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*

Indicators of performance degradation in the execution of the C++ program were expected from the crash file. In addition, as memory was being depleted, 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 calcualted 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. Profiling the system's behaviour was performed in three steps. Firstly, trends 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 made.

#### *6.4.1 Behaviour of the system before the failure*

In order to observe trends 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. An explanation of how to read the maps is provided next.

The component maps below 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 with various shades of blue, green and yellow in between. All component maps have the same topology, so any node (record) on one map has the exact same position on another map for the same set of records. For example, the first record, which has the highest value for *Duration* (5850 ms, refer to Figure 8) 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 highest value for free memory (red in the top right corner of the *Mem free* map). Component maps of some of the other attributes are provided in Figure 9.

Clusters in the data set are delimited by black lines on the maps. Each cluster groups together the records with close values for the various attributes. As we used the trial version of the Viscovery SOMine tool for the SOM analysis, an "Evaluation only" watermark appears on the maps.

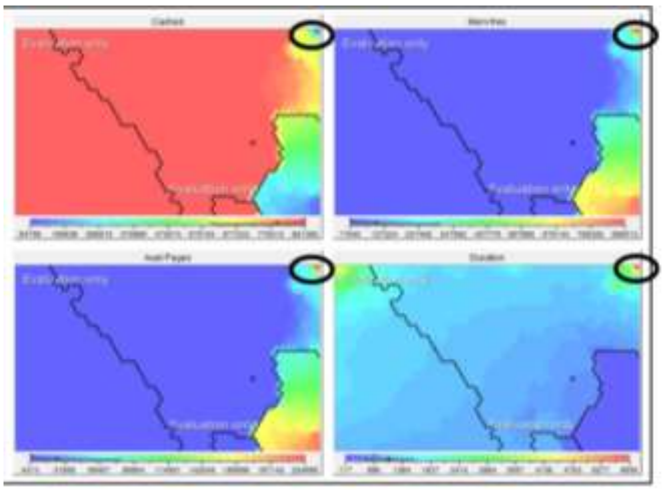


Figure 8: Some component maps of first 1000 records – first record appears as an outlier

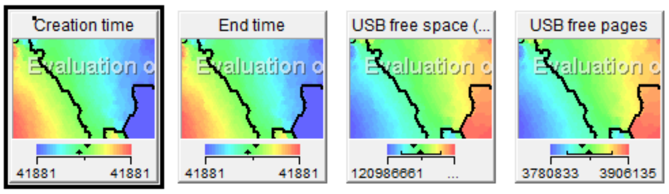


Figure 9: Component maps of first 1000 records for some attributes.

From the component maps that were generated for all attributes in the crash file, we note a high correlation between the following attributes:

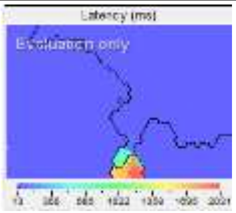
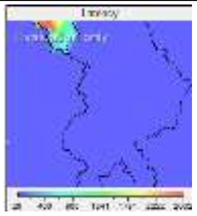
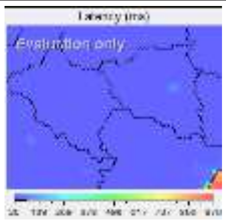
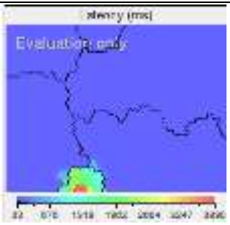
- USB free space and USB free pages
- Mem Free and Avail pages
- Cached and Mem used

As these pairs of attributes point to the same memory source, their maps have the exact same pattern of value distribution. Therefore, for each pair, one of the maps is discarded in future analysis. As we were looking for attributes whose distribution would change over time, we also discarded maps of attributes whose pattern was the same throughout the program. This includes both USB memory attributes, and Avail pages, whose values decrease linearly. We also noticed that *Cached* is the direct inverse of *Mem free*, as their values follow inverse movements (as *Cached* increases, *Mem free* decreases). We therefore also discarded *Mem free* from the forensic analysis. We also discarded the map of *End time* as it had the exact same distribution as the map of *Creation time*. Therefore, we conducted the forensic analysis with the following remaining attributes*: Creation time*, *Buffers*, *Cached*, *Swap Used*, *Duration* and *Latency*.

**Results**

A study of the component maps shows that the values for the remaining six 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).

Table 1: SOM maps for *Latency*

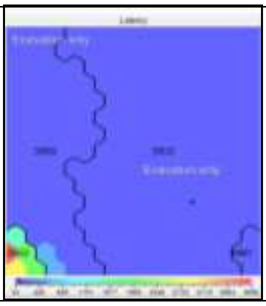
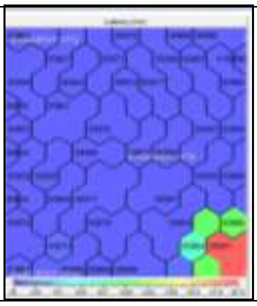
| Records 1-1000 | Records 10 000 to 11 000 |
|---|---|
|  |  |
| Records 20 000 to 21 000 | Last 1000 records before crash |
|  |  |

In order to find more detailed and 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.

*6.4.2 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. These maps are displayed in Table 2.

Table 2: SOM maps of *Latency* close to the failure

| Last 100 records before failure | Last 50 records before failure |
|---|---|
|  |  |

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 forensic analysis of the 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*, which 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.

### 6.5 Forensic analysis of `iotop` output file

The forensic analysis of the `iotop` output file followed the same process as with the crash file. SOM maps were generated for the following attributes in the file: *Time*, *Disk read*, *Disk write*, *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 as 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.

### 6.6 Creating a near-miss formula from the near-miss indicators

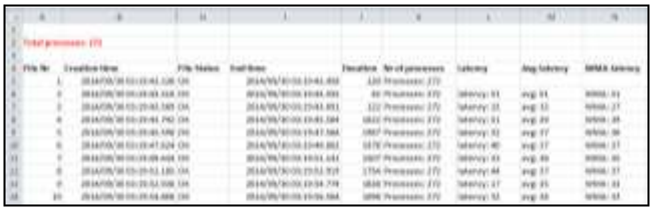
In order to provide reliable results, the near-miss formula had to be not only accurate, but also relevant when executing the program with different variables. These variables were the size of the video clip to be copied, the amount of free space on the flash disk, and the number of processes running in parallel to the C++ program. To this end, the program was executed a number of times with various values for these variables, each time changing the size of the video clip or the amount of free space on the flash disk, as well as running fewer or more programs concurrently. The validity of the previously identified near-miss indicators was verified in every new execution of the program.

A number of flash disks with factory settings were used to run the tests several times. Once all the flash disks had been used at least once, their content was deleted to empty the flash disks and get back to their original free space and similar results were obtained (same number of maximum video files stored in flash disk). No specialised tool was used the delete the files stored in the flash drives.

In order to observe the trends in the values of *Latency*, its overall average and the WMA were calculated for every new record as follows.

The WMA of the last 200 records was displayed continuously. This was based on the results of the previous analysis indicating that some behavioural change was observed in the last few hundred records before the failure. 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. The crash file resulting from the above calculations is shown in Figure 10.



Total processes: 277

| File Nr | Creation time | File Status | End time | Duration | Nr of processes | Latency | Avg latency | WMA latency |
|---|---|---|---|---|---|---|---|---|
| 1 | [illegible] | OK | [illegible] | 120 | Processes: 277 | | | |
| 2 | [illegible] | OK | [illegible] | 44 | Processes: 277 | latency: 51 | avg: 51 | WMA: 51 |
| 3 | [illegible] | OK | [illegible] | 122 | Processes: 277 | latency: [illegible] | avg: [illegible] | WMA: [illegible] |
| 4 | [illegible] | OK | [illegible] | 1822 | Processes: 277 | latency: [illegible] | avg: [illegible] | WMA: [illegible] |
| 5 | [illegible] | OK | [illegible] | 1907 | Processes: 277 | latency: [illegible] | avg: [illegible] | WMA: [illegible] |
| 6 | [illegible] | OK | [illegible] | 1079 | Processes: 277 | latency: [illegible] | avg: [illegible] | WMA: [illegible] |
| 7 | [illegible] | OK | [illegible] | 1907 | Processes: 277 | latency: [illegible] | avg: [illegible] | WMA: [illegible] |
| 8 | [illegible] | OK | [illegible] | 1754 | Processes: 277 | latency: 44 | avg: [illegible] | WMA: [illegible] |
| 9 | [illegible] | OK | [illegible] | 1848 | Processes: 277 | latency: [illegible] | avg: [illegible] | WMA: [illegible] |
| 10 | [illegible] | OK | [illegible] | 1896 | Processes: 277 | latency: [illegible] | avg: [illegible] | WMA: [illegible] |

Figure 10: Adapted crash file for near-miss detection

The outcome of this verification process was that results similar to the initial forensic analysis were obtained, except for the number of processes that would either be smaller than or equal to the initial number. No explanation could be provided for this observed pattern. Consequently, the final near-miss indicators used to define the near-miss formula were identified as follows:

- The WMA of *Latency* is greater than its average.
- 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.

Based on the above analysis, the formula to detect potential near misses was defined as follows:

```
If Nr-Processes <= Initial-Nr-Processes AND
WMA-latency > Avg-latency AND
DR > (Avg-DR x 2) AND
DW == 0
⇨   Near Miss
```

The near-miss formula uses the following notation:

- *Nr-Processes*: number of processes
- *Initial-Nr-Processes*: Initial number of processes
- *WMA-latency*: WMA of latency
- *Avg-latency*: average of latency
- *Avg-DR*: Average of Disk reading bandwidth

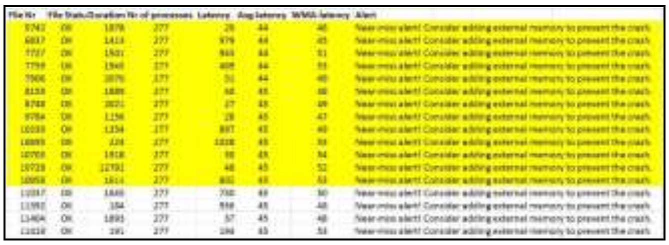
It is worth noting that the above formula was specific to the software failure at hand, the conditions of its occurrence (lab experiment) and its analysis (`iotop` used for correlation to program's logs). However, it can be a starting point for the identification of near misses for similar types of failures. The formula was used in the next and last step of the experiment to detect near misses during the program's execution, and the process involved is documented in the next section.

## 6.7 Detection of near misses at runtime

The goal of this step was to verify whether near misses could be detected during the execution of the program by using the formula developed in the previous step. The plan was to apply the near-miss formula to the logs as they were generated. Any log whose attributes matched the near-miss formula was then labelled as a potential near miss and an alert was sent. The alert was a notification message with some suggestion to prevent the failure. Preventing the failure was outside the scope of this study, which means that no attempt was made to implement the suggested countermeasures. The alert was generated to enable the collection of digital evidence of the potential near miss before the impending software failure occurred. However, the collection of evidence was not performed as it was beyond the scope of the experiment.

The near-miss formula was inserted in the program's loop after calculation of all the necessary attributes. The crash file was generated with 21 829 records before the failure, a loop size of 22 000, and the original video clip. The formula matched 162 records at various instances in the crash file, mostly close to the failure.

The first of the near-miss alerts started at record 5 742, indicating that the near-miss formula did not apply to the early records in the file, as was expected (see Figure 11). Only 9% of the near-miss alerts (13) appeared in the first half of the program's execution and are highlighted in Figure 8. These are clearly false alarms. The remaining 91% of the near-miss alerts were generated in the second half of the program's execution before the failure. This again confirms that the near-miss indicators mostly emerged close to the failure. The last alert was generated 6s 872 ms before the failure. This indicates that it was signalling a near miss as it was not immediately followed by the failure, although it was very close to its occurrence.



| File Nr | File Status | Duration | Nr of processes | Latency | Avg latency | WMA latency | Alert |
|---|---|---|---|---|---|---|---|
| 5742 | OK | 1878 | 277 | 28 | 44 | 46 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 6837 | OK | 1414 | 277 | 979 | 44 | 45 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 7737 | OK | 1501 | 277 | 944 | 44 | 51 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 7799 | OK | 1940 | 277 | 409 | 44 | 55 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 7906 | OK | 1876 | 277 | 31 | 44 | 46 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 8153 | OK | 1889 | 277 | 48 | 45 | 48 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 8748 | OK | 2021 | 277 | 27 | 45 | 49 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 9784 | OK | 1156 | 277 | 28 | 45 | 47 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 10033 | OK | 1154 | 277 | 897 | 45 | 46 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 10895 | OK | 224 | 277 | 1018 | 45 | 53 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 10769 | OK | 1918 | 277 | 98 | 45 | 54 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 10729 | OK | 12781 | 277 | 48 | 45 | 52 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 10958 | OK | 1614 | 277 | 802 | 45 | 53 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 12017 | OK | 1849 | 277 | 700 | 45 | 50 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 12592 | OK | 184 | 277 | 558 | 45 | 48 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 13404 | OK | 1893 | 277 | 57 | 45 | 48 | Near-miss alert! Consider adding external memory to prevent the crash. |
| 13618 | OK | 191 | 277 | 196 | 45 | 51 | Near-miss alert! Consider adding external memory to prevent the crash. |

Figure 11: First near-miss alerts in the crash file

## 6.8 Evaluation of prototype

The prototype was successful in the sense that it achieved the specified goals, which were to demonstrate the viability of using the digital forensic process for software failure analysis and demonstrate the viability of detecting near misses at runtime. In addition, it also showed the following:

- Near-miss detection can reduce the amount of relevant digital evidence that needs to be collected for root-cause analysis. Indeed, out of the 13 initial attributes in the crash file and the nine attributes in the `iotop` output file, only four (latency, processes, disk-reading bandwidth and disk-writing bandwidth) proved relevant for near-miss detection.

- In order to detect near misses, indicators of an up-coming 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 validity of the resulting near-miss formula demonstrates the reliability of the technique used for the root-cause analysis, in other words the SOM analysis. 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. Previous research using the SOM analysis to analyse real-life system logs support this claim (Fei et al, 2005).

On the downside – although the created near-miss formula proved effective, it was not a direct application of the general formula presented in Equation (1). Nonetheless, near misses were defined based on the characteristics of a failure, as is the case in Equation (1), where failures are characterised by their downtime. Additionally, some false alarms were observed. Handling these false alarms automatically could be addressed with a prioritisation mechanism based on time. E.g. how close the alarm was generated to the beginning of the program execution or to the exhaustion of the flash disk or using the number of alerts per time period. However, this is for future work as it was not one of the goals of the prototype.

## 7 Conclusion

This paper proposed the technique of near-miss analysis to assist in the digital forensic investigation of software failures. Such a forensic investigation may be required following catastrophes due to the failure and resulting in a court case. The concept of a near miss was used to help identify the root cause of a software failure and to prevent its recurrence. This was achieved through the pattern analysis of the system logs close to the point of failure. An original architecture of a near-miss management system (NMS) was proposed to combine near-miss analysis and digital forensics to automate the detection of near misses and identify relevant digital evidence required for the root-cause analysis and the potential failure prevention. The validity of these two features (i.e. near-miss detection and evidence identification) of the NMS architecture was demonstrated through a prototype. Future work that is suggested involves the completion of a prioritisation scheme to select the most relevant near misses that can be used for evidence collection and reduce false positives.

## References

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*, vol. 132, pp. 154-162.

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.

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.

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., Olivier, M.S. & Eloff, J.H.P. (2011). *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, London, UK.

Bihina Bella, M.A., Olivier, M.S. & Eloff, J.H.P. (2012). *Near Miss Detection for Software Failure Prevention*. In Proceedings of the 2012 Southern Africa Telecommunications Network and Applications Conference (SATNAC 2012), 2-5 September, George, South Africa.

Bihina Bella, M.A., Eloff, J.H.P. & Olivier, M.S. (2014). *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, Piraeus, Greece.

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].

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].

Cooke, R. & Goossens, L. (1990). The accident sequence precursor methodology for the European Post-Seveso era. *Reliability Engineering and System Safety*, 27, 117-130.

Corby, M.J. (2000). Operational Forensics. Information Security Management Handbook. Fourth Edition. Vol. 2, chapter 28. Auerbach Publications: Boca Raton.

Engelbrecht, A.P. (2003). *Computational Intelligence: An Introduction.* John Wiley & Sons.

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.

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].

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.

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].

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.

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.

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].

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 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].

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.

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*, vol. 12, pp. 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].

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.

Laprie, J.C. (Ed.). (1992). *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Wein, New York.

Leveson, N. (1995). *Safeware*. Boston: Addison-Wesley Publishers.

Leveson, N. (2015). A systems approach to risk management through leading safety indicators. *Reliability Engineering and System Safety,* vol. 136, pp. 17-34.

Linux.die.net. (2014). Iotop(1) – Linux man page. [Online] Available from: http://linux.die.net/man/1/iotop [Accessed: 14 November 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].

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 [Accessed: 1 December 2014].

Near-miss Management LLC. *Dynamic Risk Predictor Suite*. [Online] Available from: http://www.nearmissmgmt.com/products.html [Accessed: 03 December 2014].

Pertet, S. & Narasimhan, P. (2005). *Causes of failures in Web Applications*. Carnegie Mellon University: Parallel Data Lab Technical Report CMU-PDL-05-109. (December).

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].

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.

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].

Ritwik, U. (2002). Risk-based approach to near miss. *Hydrocarbon Pro-*

*cessing*, pp. 93-96. October.

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].

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.

Vacca, J.R. & Rudolph, K. (2010). *System Forensics, Investigation and Response*. Chapter 1, pp. 2-16. Sudbury, Mass.: Jones & Bartlett Learning.

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].

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].