Self-Defence of Information Systems in Cyber-Space

A Critical Overview

Mario Couture[*], Robert Charpentier[*],
Michel Dagenais[**], Abdelwahab Hamou-Lhadj[***], Abdelouahed Gherbi[*]

[*] DRDC Valcartier, 2459, Pie-XI Blvd North, Quebec, QC, Canada, G3J 1X5
[**] École Polytechnique de Montréal, 2900 Edouard-Monpetit Blvd, Montreal, QC, Canada, H3T 1J4
[***] Concordia University, 1455 Maisonneuve West, Montreal, QC, Canada, H3G 1M8

[*] [Robert.Charpentier] [Mario.Couture] [Abdelouahed.Gherbi] @ drdc-rddc.gc.ca

[**] Michel.Dagenais@polymtl.ca

[***] Abdelw@ece.concordia.ca

ABSTRACT

Nowadays information systems (ISs) that are utilized to support military operations are much more complex than a decade ago. These new levels of complexity have brought important technical problems that need to be addressed. For example, timing-related software bugs in multi-core and multi-processor contexts have become much harder to solve in the laboratory with the current development environments. This new technological complexity has also augmented the number of undetected software vulnerabilities in systems. These become widely exposed to malicious exploitation when they are connected to unsafe networks during operations. As the traditional security framework will probably continue to present limitations, it is expected that cyber-security gaps will continue to get larger, making information systems more vulnerable to increasingly sophisticated cyber-attacks.
New mechanisms and tools are needed to improve both deep software analyses in the laboratory and refined surveillance of systems during operations. This paper presents an overview of important results that were identified in six state of the art studies, and proposes new mechanisms and concepts that would complement the ones that are currently used in our development and security environments.

1.0 INTRODUCTION

Nowadays information systems (ISs) that are utilized to support military operations are much more complex than a decade ago. They now involve computers employing central processing units that have evolved from simple processors to symmetric or asymmetric multi-processor (SMP/ASMP), non-uniform memory access (NUMA) and more recently multi-core systems (SMP/ASMP on a single chip) [1]. ISs are also geographically distributed; making heavy use of networks to allow collaboration. This new technological complexity makes them more adapted and responsive to real operational needs, but at the same time it represents an important source of technical problems that must be addressed. For example, it brings on a number of new complex synchronization bugs as well as unknown vulnerabilities that are increasingly harder to detect and solve [2]. As ISs evolve on a continual basis, it is believed that these bugs and vulnerabilities will always be present in fielded systems.

The act of connecting ISs to network environments (that may contain hostile nodes) widely exposes unknown software vulnerabilities of these ISs. Combined with the limitations of the traditional security framework in detecting and eradicating cyber-threats, the new complexity levels will contribute to enlarging cyber-security gaps in the future. The expected consequence is that military ISs will continue to be the target of cyber-attacks of increasing frequency and sophistication. Current development and security frameworks thus need to be improved with new mechanisms that will insure deep and refined surveillance of ISs for debugging, vulnerability eradication and protection purposes.

DRDC Valcartier, like many other research communities around the world, is exploring new complementary strategies to significantly improve the debugging and self-defence of ISs. A three-year R&D project (herein called DRDC Project) [3] was initiated early in 2009. It aims to develop a new set of advanced mechanisms that will be utilized both during military operations (online) for protection and in laboratories (offline) for deep analysis of software (including cyber-forensics). The main strategy consists in putting emphasis on the *designed-for-tolerance-and-recovery* paradigm, but at the same time integrating and harmonizing with the traditional *software hardening* paradigm (and protecting against known threats) [4, 5, 6].

This paper summarizes state of the art studies that were conducted during the first year of the DRDC Project. The following emerging fields of self-defence systems design are covered: 1- mechanisms for adapted surveillance and analysis of ISs (Section 2); 2- innovative utilization of new architectural patterns (such as redundancy with diversity) in IS architectures (Section 3); 3- mechanisms to improve software resilience, self-adaptation and self-healing (Section 4). The level of technical details was purposely kept moderate throughout the paper.

2.0 SURVEILLANCE AND ANALYSIS OF INFORMATION SYSTEMS

The refined detection of unwanted software behaviours in an information system will involve the utilization of a monitoring system that is able to quickly collect information at specific locations (deep in the system), with appropriate resolution and very low performance impacts. The collected information is then analyzed with the goal of deriving system health and detecting anomalies in the system. This section presents the work that is being done to develop mechanisms for efficient and effective monitoring.

Tracing allows the capture of sequences of events that occurred during software execution, both in the user and kernel spaces of a system. Combined with other techniques (such as sampling), it represents a good option for software monitoring because it has access to almost any information on the system while it is running. Specific tasks or functions (in the form of code) are added into the source or binary code (source-level or binary-level *instrumentation*), precisely where events need to be captured. When these special instructions are encountered, they generate information (*trace events* and associated data) that are collected in *execution traces*, which can later be examined on demand for random audits or for the study of specific problems in systems.

The following use case examples show that tracing can be utilized in many contexts. 1- *Clusters of servers*: detailed execution traces can be generated from computers in a cluster, which are then globally analyzed online or offline; 2- *online system monitoring*: a distributed collaborative command and control IS in a battlefield is instrumented to continuously generate execution traces for remote analysis and detection of abnormal software behaviour; 3- *embedded systems*: the processes of embedded systems such as high-end cellular phone handsets can be traced for their duration (e.g. a phone call); 4- *offline dynamic deep analysis*: for vulnerability discovery and eradication, cyber-forensics, malware analysis, etc. Online software surveillance may provide indications regarding health states, performance and the presence of undesired software behaviours or states while the system is running.

Six state of the art studies (SOTA) were conducted in the following domains: 1- Adaptive fault probing, 2- Multi-level, multi-core distributed traces synchronization, 3- Trace abstraction, analysis and correlation, 4- Automated fault identification, 5- System health monitoring and corrective measure activation, and 6- Trace directed modelling. This section presents an overview of the first four; the last two SOTAs will be described in a later publication.

2.1 Adaptive fault probing[1]

A number of advanced tracing technologies are available on the market. Valgrind (Valgrind, 2008), DTrace [7], SystemTap [8] and GDB [9] are good examples that allow binary-level instrumentation. DTrace was built for the Solaris and Open Solaris operating systems. Instrumentation of these systems is achieved through loadable kernel modules that define a set of kernel probes which can be activated on demand. Impacts on traced information systems are negligible when probes are not activated, but the data extraction mechanism limits performance. Source-level tracing and user space tracing are also possible with DTrace. The SystemTAP tracer provides features that are similar to DTrace, but for the Linux operating system. It works by processing script files in which users specify probe points and associate handlers to them. While probing can be enabled asynchronously at regular time intervals, its architecture disallows instrumentation of source-code that is reached from NMI contexts. DTrace, SystemTap and GDB are typically used to add a few trace points or breakpoints with a small performance overhead (through a trap mechanism). Valgrind [10] is more commonly used for pervasive monitoring of the program (e.g. tracing every access to memory) with a considerable slowdown.

Compilers, and source code parsing and transformation toolkits are also available to automatically add instrumentation code at specified points in the source code. For example, the GNU Compiler Collection [11] provides a number of instrumentation compilation options for debugging and instrumentation purposes. TXL and Javacc are other good examples. Other tracers, such as Evlog [12], LKST [13], and LTT [14], allow manual insertions of trace points at strategic locations in the source code. DTI API [15] is an architecture-independent tracer that was initially designed to enable developers of kernel drivers to log events in a circular buffer. DTI API is usable in both user and interrupts contexts and the data is transferred to user space with the Relay file system. Finally, tracing or logging libraries are also available for most programming languages (Java logging, .NET logging, and C++ logging).

Other tracers are available for Unix-like operating systems. KTAU [17] for example was developed for multiprocessor computers and allows both the profiling and tracing of the Linux kernel on a system-wide or per-process basis. Its capabilities are limited by the fact that collected information is provided in an aggregated form to limit resource utilization. FTrace is another recent tracer that provides system-wide tracing information for Linux kernel developers. It uses the trace point mechanism as its primary instrumentation mechanism. Its specialized trace analysis modules run in kernel-space and generate either a trace or analysis under the form of textual output to the user. As of its Linux 2.6.29 implementation, FTrace does not handle NMIs gracefully and the reliance on the scheduler clock for timekeeping may cause timing errors under certain conditions. K42 [16] is a research operating system developed by IBM. It contains a built-in tracer in the kernel.

None of these tracers meets all the objectives of the DRDC Project. The needed tracer had to provide: system-wide instrumentation coverage, basic support for all Linux architectures, minimal performance impact, near zero impact when not tracing, probe re-entrancy for all kernel execution context handlers (including NMIs and MCE), very high-frequency kernel events recording with small overhead under typical workloads, scaling to large multi-processor systems, predictable system real-time response changes, the ability to send trace events through the network or to hard disk, and the ability to dynamically activate and/or deactivate probes.

The Linux Trace Toolkit next generation (LTTng) [2, 18] was seen as the best solution because initial requirements (all fulfilled in 2009 [2]) were perfectly aligned with those of the DRDC Project. LTTng is made of a kernel patch, a tool chain, and a trace viewing and analysis program (called LTTV). It includes a set of kernel instrumentation points that are useful for debugging a wide range of bugs that are otherwise extremely challenging (such as performance problems on parallel and real-time systems). LTTng allows the tracing of the user space through markers and GDB trace points for x86-32 and x86-64 architectures,

[1] The text in this section is an overview of the SOTA that can be found in Dr Desnoyers' Ph. D. thesis [2].

and the kernel space through trace points and markers for the following architectures: x86-32, x86-64, SPARC, SPARC64, ppc, ppc64, sh, sh64, is64, s390, MIPS 32/64, and ARM. It is being actively developed at École Polytechnique de Montréal (since 2005) and supported by enterprises such as Ericsson, IBM, Google, Fujitsu, Sony, and Wind River. The source code instrumentation mechanism proposed by LTTng (trace point; called *Tracepoint*) was officially incorporated into the mainline Linux kernel 2008.

The ability of LTTng to directly send trace events through the network, combined with the possibility of dynamically activating and de-activating markers and trace points on demand represents an important asset for local/remote system surveillance and control. The choice of both the focus and resolution of events in execution traces can theoretically be based on results obtained from previous execution trace analyses. This *feedback-directed capability* of the protection system would contribute to optimize the production of more precise diagnostics (and control), with minimal performance impacts on systems.

2.2 Multi-level, multi-core distributed traces synchronization[2]

This section briefly introduces two mechanisms that can be utilized for the synchronization of trace events. It also describes how the selected synchronization mechanism was adapted for execution traces.

Probes installed in the different software layers[3] may be used to provide monitoring and tracing data. More precisely, each core within each processor generates a steady flow of events when tracing is activated. An event may thus be triggered by one core or another and, as each core has its own clock (and associated unique timing imperfections; e.g. drift [19]), event time-stamps do not all have the same reference time frame. A mechanism is thus needed to synchronize time-stamps generated by various clocks.

There are two ways of pursuing the synchronization of events that originate from distributed multi-core multi-processor systems: online synchronization or offline synchronization. The former consists of adjusting each clock during tracing. The most precise approaches to online synchronization involve the use of specialized hardware to physically distribute a clock signal to each traced node [20]. It is also possible to rely on software-only methods, such as the Network Time Protocol [21].

Offline synchronization is based on the analysis of events happening in multiple traces with a strict ordering relationship. The algorithms proposed by Duda et al. [22] form the seminal work in this area. They analyze events corresponding to network packet transmissions to map the time between the clocks of two systems. This approach has been extended to larger distributed systems [23]. Recent developments include the use of linear programming [24] and broadcast messages [25].

The choice of offline synchronization was driven by the need to limit impacts on ISs; Duda's convex hull algorithm was thus extended for tracing. This analysis guarantees that there will be no message inversions (e.g. packets that appear to travel backwards in time) in the synchronized traces. It also supplies accuracy bounds on the clock synchronization parameters. In its original form, however, the convex hull algorithm does not supply accuracy bounds on time conversions. The precision of the synchronization can only be estimated. To solve this problem, the linear programming approach of Sirdey was applied to the original convex hull algorithm. The objective function has been modified to identify accuracy bounds on time conversions at any point in the trace.

Through experimental study, we have identified some parameters that affect trace synchronization accuracy and precision. As an example, the use of a network with lower latency or a higher message rate improves trace synchronization; longer trace duration however reduces precision and accuracy. We've shown that this can be detected through metrics based on apparent message latency and broadcast

[2] The text in this section is an overview of a scientific paper that will be submitted for publication by Mr Benjamin Poirier, Dr Robert Roy and Dr Michel Dagenais (École Polytechnique de Montréal).

[3] Hypervisor, operating system, virtual machine, system libraries and applications.

reception times. During our experiments, we achieved a synchronization accuracy of ±15 µs and an estimated precision of 9 µs.

2.3 Trace abstraction, analysis and correlation[4]

Once trace events are collected and synchronized, their volume needs to be reduced to make it manageable. This process is called *trace abstraction*. It consists in extracting high-level abstractions from low-level trace events in order to facilitate the understanding, exploration, and analysis of trace contents.

Analysis may consist for example in comparing abstracted execution traces that originate from different but redundant nodes, which are simultaneously executing instances of the same application. Comparison in this case could reveal the presence of undesired software behaviours. It would be very hard to directly compare a huge amount of trace events without impacting system performance. Execution traces need to be reduced through trace abstraction before analysis.

Our review of the literature on trace abstraction and correlation techniques reveals that they can be grouped into three main categories: pattern detection, noise filtering, and visualization techniques. Although most of these techniques have been applied in the area of program comprehension and software maintenance, we believe that they can be easily adapted to security. This section presents results of the state of the art study and potential adaptation to cyber-security.

2.3.1 Pattern detection

A *trace pattern* is defined as a sequence of events that is repeated non-contiguously in an execution trace. The more patterns are present in the trace, the less time is required to understand its content, since an analyst does not need to look at the same sequence twice. Patterns, once detected, are often replaced with high level descriptions that are understandable by humans.

The detection of patterns requires the use of some sort of similarity metrics to determine when two sequences of events can be deemed similar. For this purpose, several matching criteria have been proposed [26, 27, 28, 29]. These criteria vary significantly depending on the type of trace that is used. One example is to measure the distance between two given sequences of events using the edit metric. A threshold needs to be defined above which two sequences can be considered similar. Many other matching criteria have been the subject of several studies, with a focus on the analysis of routine call traces. Examples include comparing two call trees by treating their calls as a set, ignoring the order of calls, limiting the stack depth.

Although pattern detection techniques have been shown to be useful in many applications, they suffer from some limitations such as the difficulty of understanding how the matching criteria can be combined and their impact on the resulting abstractions.

We followed a knowledge-based approach to detect patterns from system call traces generated from the Linux kernel. We used these patterns to abstract out the content of kernel-space traces and turned them into a more compact and readable form while preserving the key information. To achieve this, we have built a pattern library that contains key Linux operations for file, socket, and process management. The patterns are described as state machines composed of lists of events and states. The states conform to the modes of execution in an LTTng trace (e.g. USER_MODE, SYSCALL, etc), whereas the events represent the system calls that appear in the trace.

To build the pattern library, we have studied the Linux kernel to understand the system call mechanism. We have also executed a number of applications with different operations and generated traces using the LTTng tracer. We studied the generated traces in order to uncover the common patterns. The pattern

[4] The text in this section is an overview of a scientific paper that will be submitted for publication by Dr W. Hamou-Lhadj and Mr W, Fadel (Corcordia University).

library has been validated by Linux kernel experts. We are currently in the process of applying it to abstract out the content of large system call execution traces.

2.3.2 Noise filtering

Execution traces tend to contain a considerable amount of noise that clutters the trace without adding much value to its content. Hamou-Lhadj and Lethbridge [30] studied the concept of utility components, which they defined as low-level implementation details (i.e. noise), and proposed a metric that measures the extent to which a routine can be considered as a utility. They have developed a trace abstraction method based on the removal of utilities. They were successful in extracting high level abstractions from low-level trace events. Many other researchers agree that removing noise from traces can significantly improve the quality of the extracted abstractions [26, 31, 32, 33, 28, 27, 34]. They propose tools that enable users to remove information from the trace before the abstraction process takes place.

When applied to Linux kernel system call traces, we have investigated what constitutes noise by studying the Linux kernel and working with the users of the LTTng tracer. We have found that most memory management operations, page faults, and hardware interrupts tend to appear anywhere in the trace in a non-predictable way, and they do not add valuable information to the system behaviour. As a result, we categorized these operations as noise and proposed that removing them would result in better abstractions.

2.3.3 Visualization techniques

Several proposed techniques in the area of trace abstraction rely on some sort of visualization technique to allow the users to manipulate the trace according to the needs of the task at hand. Using these techniques, an analyst can, for example, browse, animate, slice, group events, hide specific events, or search the traces. Bennett et al. [35] divided the features implemented in trace analysis tools into two groups: *presentation features* and *interaction features*. They defined the former as the set of features affecting the layout in which the trace is displayed, such as showing multiple views, hiding information and using animation. Presentation features can be further divided according to the following attributes:

- *Layout*. This represents the way a trace is displayed. For example, a system call trace can be represented in a linear view. A routine call trace is usually represented as a tree structure or a UML sequence diagram. Many other layouts have been proposed including 3D layouts.

- *Multiple Linked Views*. These views display information about a trace at different levels of abstraction. The views are linked in a way that makes it easy for the user to move from one view to another.

- *Highlighting*. Highlighting a part of the sequence corresponding to user selection.

- *Hiding*. The ability to hide information such as noise, or specific processes not needed for the task at hand.

- *Visual Attributes*. Using colours and shapes that help users to recognize certain information. For example, colour-coding can be used to distinguish trace patterns so as to enable the user to quickly spot the most important ones.

- *Labels*. The ability to add descriptions to patterns, label particular places in a trace, etc.

- *Animation*. The ability to animate the content of a trace by playing dynamically the flow of execution.

Interaction features, on the other hand, are those implemented to enable users to interact with the tool by navigating, querying, and manipulating the trace content [35]. They can be further divided as follows:

- *Selection*. This feature enables users to select elements to manipulate, filter, or slice.
- *Focusing*. This feature allows the user to focus on a particular aspect of a trace (e.g. by collapsing parts of a trace).
- *Zooming and scrolling*. This feature permits enlarging or reducing parts of the trace view as well as moving up, down, left or right within the diagram.
- *Querying and slicing*. Querying refers to identifying and filtering information, while slicing refers to selecting specific parts related to the selected component.
- *Grouping*. This feature groups events into patterns. This could be performed automatically (using the pattern detection techniques discussed earlier) or manually.
- *Annotating*. This feature is used to describe grouped events such as patterns, to store user notes while exploring the trace, and to provide messages to users sharing the trace.
- *Saving and restoring views*. Users can save the state of the trace after several trace abstraction techniques have been applied, for later re-use.

Mechanisms allowing abstraction of execution traces in the context of cyber-protection are under development.

2.4 Automated fault identification[5]

Continual analyses will be made on abstracted execution traces in order to detect system health degradation and the presence of undesired software behaviours. Excessive swapping, lock contention, undue latency, inefficient task scheduling, attempts to erase system logs, and modification of system files are some examples of these. Some can be related to software design defects, others to inefficiencies or malicious activities. As for the abstraction process, mechanisms for automated fault identification must be very fast, with low impact on traced systems. Similar approaches to the ones used for Intrusion Detection Systems (IDS) are chosen to provide a flexible automated mechanism for the identification of unwanted software behaviours in execution traces. The main goal is to allow systems to trigger alarms during operations when specified problematic conditions, scenarios or patterns are detected. This section summarizes the state of the art study that was conducted in this domain.

There are two main types of IDS: *network-based IDS* (NIDS) monitor the network, and *host-based IDS* (HIDS) monitor host systems. IDS can further be divided based on the techniques they use for detection: *signature-based*, *anomaly-based* and *policy-based*. Signature-based IDS define known attacks through scenarios and their associated signatures. Comparisons of observation with scenarios allow the detection of problems. Anomaly-based IDS work differently. They involve two complementary phases: a learning phase and a detection phase. The learning phase consists in capturing normal behaviours of the system, while the detection phase compares observations with the pre-learned behaviour and detects deviations between the two. Policy-based IDS, finally, define policies that allow (or prevent) accesses to system resources. Violations of these policies are considered *intrusions*. Six main types of languages (and corresponding technologies that make use of these languages) were examined. They were chosen based on their potential adaptation for security and trace analysis contexts. Note that this list is not exhaustive.

[5] The text in this section summarizes the work done by Dr B. Ktari and Mr H. Mohamed-Waly (Laval University).

Declarative languages model what is to be computed, rather than how it is computed; the logic of computation is considered rather than the control flow of the program [36]. These languages are often considered as domain specific because their syntax is tightly related to the domain in which they are applied. Examples of technologies that make use of these languages are: the open source NIDS Snort 2009 [37, 38, 39] and the commercial SECnology 2009 [40]. *Imperative languages* model the steps or algorithms that represent attacks. Examples include: RUSSEL [41], DTrace [7], and SystemTap [8], with their scripting languages. *Automata-based languages* use finite state machines (FSM) to describe attacks. The system triggers a transition from one state to another when specific events occur. An attack is a suite of states and transitions between them. Some technologies based on FSM are: STATL [42], SMC [43], and Ragel [44, 45]. *Temporal logic* makes use of first-order logic (or its derivatives) to describe attacks. Statements about trace events are built from atomic propositions, which express the content of a trace. Temporal operators allow specifying how atomic propositions should be arranged in order to constitute a security violation. Chronicle [46] is an example that permits the recognition of anomalies in a flow of events. *Policy-based languages* describe security policies rather than attacks. Policies control the type of operations that can be done on system objects (including files, pipes, FIFOs, sockets, shared memory buffers, etc.). As mentioned, two intrusion detection techniques are available: signature-based and anomaly-based. Policy violations are considered attacks. Blare [47] is a policy-based intrusion detection system.

Finally, *Expert systems* make use of inference rules to reproduce the reasoning made by an expert, and aid in complex decision-making. The two main techniques used to infer new facts are forward chaining, which starts with basic facts to deduce new ones, and backward chaining, which starts from a proposed hypothesis and proceeds to collect supportive evidence. Examples of expert systems are ADeLe [48], P-BEST [49], and LAMBDA [50]. Important properties that were identified in this work and that will be studied more in depth are listed and briefly described in the following lines.

1. *Scenario based on multiple events*. The language should allow the study of attacks based on the occurrence of a sequence of events.

2. *Non-Occurrence of events*. The language should allow testing for the non-occurrence of events.

3. *Real-time constraints*. The language should allow modelling of the timing between events.

4. *Counting*. The language should allow modelling of repetitions of specific events.

5. *Conditional transitions*. The language should allow modelling transitions from events (or states) that depend on one or many specific conditions.

6. *Variables*. The language should be able to save variables and it should be possible to retrieve their values upon request.

7. *Grouping*. The language should allow the grouping of specific variables into a structure and checking whether a certain value appears in that group.

8. *Synthetic events*. The language should allow the saving of scenarios in a knowledge base, which could be used to describe more complex scenarios in the future.

9. *Knowledge acquisition*. The language should allow the dynamic capture and saving of information.

10. *Suitable for kernel tracing*. The language should allow the modelling of malicious activities in

execution traces and their abstractions (both in user and kernel spaces).

11. *Online*. The language should allow online detection.

The definition of a dedicated language and mechanism to be used in the context of trace analysis for cyber-protection are under development.

3.0 REDUNDANCY AND DIVERSITY IN ARCHITECTURES

The ability to take reactive or proactive actions when undesired states or behaviours are detected in information systems is another important element of a protection system. These actions may range from the small scale (closing specific Internet ports) to the larger scale (modifying the whole architecture of the system while in use). This section introduces an approach that consists in *adding diversity to redundancy* (under the form of new *architectural patterns*) in software and hardware architectures. For example, a set of two nodes simultaneously running the same OS and the same application is not a secure solution in the cyber-threat context because the same method or mechanism can be used to exploit the same vulnerability that is present in both systems. Running two different but similar operating systems (such as BSD and Linux) will significantly lower chances that the whole system will be successfully put down by the same cyber-attack; inherent vulnerabilities in both operating systems are relatively different[6]. It is expected that these new architectural patterns will contribute to improving the protection against cyber-attacks.

Even though the relevance of diversity to attack tolerance has caught the interest of researchers [51, 52, 53], it is only recently that the utilization of diversity in redundant architectures to build secure software systems were recognized [54]. Some of the major research programs that were launched in the last decade with the goal of investigating intrusion tolerance, resilience and survivability include MAFTIA [55], OASIS [56], SRS [57] and ReSIST [58]. Essentially, two categories of architectures can be drawn from these projects. The first one is represented by DIT [59(39)], SITAR [60], HACQIT [61], and DPASA [62]. They are instances of a general architectural pattern in which servers are shielded from end-users by proxies. Monitoring and voting mechanisms are used to check the health of the system, validate the results and detect abnormal behaviours. As this approach does not involve the modification of applications on the server side, it appears to be well suited when COTS or legacy or closed-source applications have to be integrated.

The second category of architectures (represented by ITUA [63, 64], MAFTIA [55, 65], and ITDOS [66]) use a middleware to provide intrusion tolerance functionalities. It eliminates the need to build custom solutions for each software application. These are built aware of the intrusion tolerance services provided by the middleware services.

The evaluation of the extent to which security is achieved through these architectures is usually approached using *model-based analysis techniques*. The *probabilistic approach* represents a basis for many of these evaluation techniques [67]. *Qualitative approaches* such as Scenarios Analysis using fault-trees [68] are often used. Quantitative approaches are based on stochastic models (Semi Markov Processes [69] and Stochastic Reward Nets [70]). As an example, [71] presents a state transition model that describes the behaviour of a generic intrusion-tolerant system. This model has been used to describe the behaviour of SITAR [70].

This state of the art study has identified a number of ideas that could contribute to improving the protection of information systems against cyber-threats. The following points group them and provide a brief description.

[6] It is worth mentioning that redundant architectures (with or without diversity) allow major software upgrades and other maintenance operations without downtime.

- *Redundancy with diversity*. Our state of the art study revealed that different intrusion-tolerant and survivable architectures such as DIT, SITAR, ITUA, DPASA, and the Willow architecture [72] implemented the principle of diversity. However, open questions regarding the manageability and control of diversity, and consequently the different levels of security that can be achieved, remain to be addressed. Notwithstanding the existence of limited mechanisms for adaptive redundancy (such as in DIT), diversity models that are similar to the redundancy models (as defined by the “Availability community”) do not exist at this moment. Architectural patterns involving the utilization of redundancy and diversity at different levels or dimensions[7] must be developed for the specific context of cyber-security, and tested in many operational situations. While diversity will contribute to reducing risks of correlated vulnerability, it will induce an increase in complexity levels that must be taken into account as well.

- *Indirection*. The principle of indirection consists in isolating or shielding from the end-users the servers implementing system functionalities and services. Essentially, it aims to hide systems’ implementations (and systems’ vulnerabilities among other things) from “outside”, making more complex the preparation of cyber-attacks. It is implemented in a variety of architectures through the use of proxies (such as in DIT, SITAR, DPASA). Its flexibility to dynamically relocate services (reactively or proactively) lowers the chance of success of multi-staged cyber-attacks. Proxies are thin interfaces having deterministic behaviours that are much less complex than servers’ functionalities. They are therefore easier to protect.

- *Adaptive responses and reconfiguration*. Intrusion-tolerant architectures involve mechanisms that ensure the continuity of services even under attack. For example, mechanisms are reactively or proactively triggered to isolate components showing undesired software behaviours. Other mechanisms make systems adapt to changing circumstances during operations through reconfiguration (such as in ITUA). Also, the injection of uncertainty in system responses may contribute to making the systems unpredictable from an adversary’s point of view. *Temporal diversity* varies the structure and/or the behaviour of systems with respect to time, while *spatial diversity* consists in deploying different components implementing the same functionalities.

- *Monitoring*. Behaviour monitoring and analysis are important elements because they provide the necessary information upon which efficient automatic & manual decision-making processes are based. As LTTng and sibling software analysis mechanisms allow deep continual feedback-directed local and/or remote monitoring and analysis of software behaviours of systems, they appear to be the technology of choice. Their fine tuning will allow automatic and manual alarm management as well. In the context of redundant/diverse architectures, the work of Gao et al. [95, 96] are good alternatives that could be improved for measuring “distance” between software behaviours that originate from an application that is running on two different systems.

- *Communication infrastructure*. The use of redundancy and diversity in architectures will involve configuration management (such as the replication of functionalities, components, etc.). Current communication protocols such as group membership protocol [73], totally ordered reliable multicast protocol [74], and even the Byzantine fault-tolerant protocol [75] must be studied more in depth in order to identify potential solution options for the context of redundant and diverse architectures.

- *Analysis techniques*. A number of architectural patterns involving redundancy and diversity should be available during operations in order to provide information systems with the capability to face different problematic situations. Their ability to address specific security requirements

[7] Such as: hardware platforms and components, and software (e.g. operating systems, software applications, virtualization mechanisms, security mechanisms).

should be characterized in the laboratory, prior to their utilization in operations. Mechanisms allowing deep analysis of these architectural patterns are thus needed to generate full specifications and mechanisms for online transition from one pattern to another.

4.0 SOFTWARE RESILIENCE, SELF-ADAPTATION AND SELF-HEALING

Feedback-directed system protection (Section 2.1) will also contribute to system optimization through positive feedback. Ultimately it will make systems self-adapt and self-heal. This section presents an overview of the state of the art study that was conducted in this relatively "new" domain.

A *self-healing system* is a system that attempts to heal itself from a fault in order to regain its normal operational state prior to disruption [76]. Self-healing systems should also be *self-adaptive*, attentive to the changes triggered by the environment for improved performance or to simply adapt to various situations [77]. Some researchers view self-healing systems as just another type of fault-tolerant system, in which detection and resistance to faults is an important component. Although self-healing systems must also be equipped with some sort of fault detection mechanism, it is important to emphasize that the focus is on the healing and recovery process rather than on fault analysis and modelling as is the case for traditional fault-tolerant systems. In other words, a self-healing system is more recovery-oriented and should be able to restore itself to normalcy independently of the type, source, and severity of the fault. This makes such a design particularly suitable to security, especially in the military context, since understanding various types of faults caused by malicious attacks might be a tedious (and sometimes impractical) task, and since—even if it is done successfully—the system still needs to heal itself from other types of faults.

One of the key aspects of a self-healing system is its ability to decide whether it is functioning properly or not. To this end, it is important to study what constitutes a "*normal*" or "*healthy*" behaviour of the system. The common approach is to measure various characteristics of the system in a laboratory environment that can later be used as a baseline for comparison. A fault detection technique can then be developed by observing, using monitoring capabilities, any deviations of the deployed system from these measurements [78]. The obvious drawback of this approach is that it does not account for the changes that the system may undergo as its environment also changes. In addition, it is difficult to anticipate the various ways the system can be used before it is put in operation [79]. Shaw argues that what constitutes a healthy system varies in time and from one user to another and it is therefore not reasonable to expect that the line between a "healthy" and a "broken" state be clearly defined in advance [79]. She proposes instead that there is a gradual transition state that she calls the *degraded state*. Many other researchers (e.g. [78]) support this view, which has led to a self-healing process in which the key element is the need for a system to initiate rectification not only when it is in the broken state but also (and more importantly) when it starts exhibiting signs of decline, i.e., when the fault or an attack has begun to take effect. This requires of a self-healing system that it not only recovers from faults but also continuously maintains its health.

A self-healing process encompasses three main activities: maintaining the system health, detecting system failures, and recovering the system from the failures. In what follows, we describe each of these activities along with a brief discussion of the main techniques that are employed. Some of these techniques have also been discussed in [78]:

1. *Maintaining the system health*. This activity consists of continuously checking the health of the system in order to maintain its normal functionality. Several strategies have been proposed, among which the most popular ones are based on a redundancy and diversity architecture [51, 52, and 80]. Other techniques include performance log analysis, in which performance measures are collected and used for the diagnosis of system health. Software aging and rejuvenation techniques have also been employed [81, 82], which consist of diagnosing the system based on empirically studied signs of decline. Finally, some researchers have used architecture description languages [83] to periodically compare the architecture recovered from a running system with healthy

reference architecture.

2. *Detecting system failures*. Most existing fault detection techniques rely on some sort of run-time monitoring and system probing technique. The idea is to compare the execution of a healthy system to the execution of the system in operation. For this purpose, a reference model of what constitutes a healthy system is needed. The techniques used to build reference models vary significantly from one study to another, including the use of natural language processing techniques [84], probabilistic context free grammars [85], Markov chains [86, 87], metric correlation models [88], machine learning [89], symbolic execution [90], and architectural model-based techniques [83]. These methods also vary depending on whether the learning process is performed online (i.e. during operation) or offline (in a controlled environment), on the way they deal with false positives, on whether they are platform specific or not, etc. In addition, other approaches such as scheduled announcements and checkpoints and policy-based monitoring have also been proposed. Checkpoints have been used to periodically check the system for responsiveness [91, 92]. Policy-based monitoring techniques have also been used in which a set of policies (e.g. the amount of time it takes to execute a particular query) are established in advance and the system is checked against these policies [93].

3. *Recovering the system health*. System recovery techniques ensure that the system recovers from an attack and returns back to its normal state. Although existing techniques vary depending on the type of system architecture on which they are applied, they all share a common repair method which consists of replacing a faulty component with a healthy one, performing a clean-up, etc. [78] A recovery strategy, however, must be guided by a recovery and repair plan that is designed in advance.

Although the above techniques have been shown to be useful, our review of the literature shows that most of them are still at the experimental stage and have not been applied to large systems with stringent security requirements. This hinders the evaluation of their effectiveness in the context of our research.

In addition, there are many research issues that still remain unaddressed. First, although most researchers agree that knowledge of the system to be healed (performance data, system architecture, etc.) is needed for the healing process to be successful, little has been said on how this knowledge should be represented, validated, and managed. This is particularly important since self-healing systems are expected to make healing and recovery decisions based on this knowledge. They are in many ways very similar to decision support systems found in organizational and managerial studies, and in which information is the key enabler to decision-making.

Another important question with regards to healing is whether the healing and recovery are achieved with or without human intervention, and to which extent human intervention is needed. It is also important to investigate techniques designed to help the analyst make sense of the information collected from the system, especially when run-time monitoring techniques are employed—since it is well recognized that the amount of run-time information to be processed tends to be overwhelmingly large. There is also a need for visualization techniques to visualize, in a usable manner, the state of the system, its health status, the sources of potential problems, etc., that can help an analyst use this information to analyze the feedback received from the system and make the necessary adjustments. This important requirement for an assisted self-healing system is almost absent in most existing studies.

Finally, we believe that not all faults should be treated the same, as is the case in most studies in the area of self-healing. In our view, there should be an acceptable security range that requires system adaption and healing. This is particularly important in order to reduce the overhead associated with switching from one node to another in the context of redundant architectures. We propose that a set of security modes be defined based on the severity of the attacks and system health attributes. For example, security attacks can be classified into severe, moderate, and minimal. In a minimal mode, an analyst may decide to terminate the infected process, whereas a severe attack can lead to a complete shut-down of the host node. This adds to the complexity of the determination of the proper threshold above which a system decline should be considered as a fault.

5.0 CONCLUDING REMARKS

New inherent complexity levels of information systems induce new complex bugs and vulnerabilities that are much harder to solve than a decade ago. Because the traditional security framework does not keep up with the cyber-security arms race, unsolved bugs and vulnerabilities have the opportunity of causing dramatic damages during operations [94]. R&D efforts are underway in Canada to try to identify new complementary mechanisms that would significantly improve the decision-making process as well as thoughtful analysis and response, while maximizing the time available for risk mitigation. The selected approach is based on *feedback-directed tracing, analysis and protection* of both user and kernel spaces of information systems. It allows the online selection of: 1- the focus and resolution of execution traces, 2- analysis mechanisms, and 3- corrective mechanisms based on the results of the immediately preceding analyses (which are conducted on a continual basis). The LTTng tracer will provide the necessary functionalities and flexibility to fully support the development of this approach.

State of the art studies presented in this paper have identified a number of solution options that would contribute to address a number of important problems. For example, an algorithm is currently being adapted to allow the synchronization of trace events originating from local or geographically distributed CPUs and cores in CPU. Its utilization on a continual basis will allow the production of synchronized execution traces that can then be sent for deep analysis. Abstraction approaches allowing execution trace volume reduction into higher-level behavioural objects (more significant behaviours) were also identified. Based on these approaches, new mechanisms producing abstracted execution traces on a continual basis are currently being prototyped and tested. Based on current approaches found in the scientific literature, new analysis mechanisms are also being prototyped and tested. The goals are to measure the heath of information systems as well as detect and appropriately report potential anomalies and problems, and propose reactive and proactive corrective measures.

New concepts and mechanisms improving software resilience, self-adaptation and self-healing are being studied as well. For example, system redundancy and diversity combined with carefully implemented decentralization of system control appear to be a good approach for self-adaptation and basic healing capabilities in ISs. The overviewed approaches seem technically feasible at this time since nowadays computing capabilities are greatly exceeding the computational requirements of modern decision systems.

This article provides an overview of the current state of the art for the key technical topics. Authors will be pleased to exchange results from current S&T investments with allies from the NATO community.

6.0 REFERENCES

[1] K.-P. Faxén (ed.), C. Bengtsson, M. Brorsson, H. Grahn, E. Hagersten, B. Jonsson, C. Kessler, B. Lisper, P. Stenström, B. Svensson. Multicore computing--the state of the art. Not published. (http://eprints.sics.se/3546/), [Accessed: February, 2010], 2008.

[2] M. Desnoyers. Low-impact operating system tracing. Ph.D. thesis, École Polytechnique de Montréal, 202 pages, (http://lttng.org/content/documentation#thesis), [Accessed: February, 2010], 2009.

[3] M. Couture, M. Dagenais, D. Toupin, R. Charpentier, G. Matni, M. Desnoyers, and P.-M. Fournier. Monitoring and tracing of critical software systems - State of the work and project definition.

DRDC/RDDC Valcartier, TM 2008 144, December 2008.

[4] P. Verissimo, M. Correia, N. F. Neves, and P. Sousa. Intrusion-Resilient Middleware Design and Validation. In Annals of Emerging Research in Information Assurance, Security and Privacy Services, H. Raghav Rao and Shambhu Upadhyaya (eds.), Elsevier, 2008.

[5] J. McDermott, A. Kim, and J. Froscher. Merging paradigms of survivability and security: stochastic faults and designed faults. In Proceedings of the 2003 Workshop on New Security Paradigms, Ascona, Switzerland, August 18–21, 2003. C. F. Hempelmann and V. Raskin, Eds. NSPW '03. ACM, New York, NY, pp. 19–25.

[6] H. Lala Jaynarayan. Intrusion Tolerant Systems. In Seventh Pacific Rim International Symposium on Dependable Computing (PRDC'00), Advanced Research Projects Agency, 2000.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In USENIX, (http://www.sagecertification.org/events/usenix04/tech/general/full_papers/cantrill/cantrill.pdf) [Accessed: February, 2010], 2004.

[8] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In Proceedings of the Ottawa Linux Symposium, (http://sourceware.org/systemtap/systemtap-ols.pdf), [Accessed: February, 2010], 2005.

[9] GDB. http://www.gnu.org/software/gdb/, [Accessed: February, 2010].

[10] Valgrind. http://valgrind.org/, [Accessed: February, 2010].

[11] GNU Compiler Collection. http://gcc.gnu.org/, [Accessed: February, 2010].

[12] Evlog. http://evlog.sourceforge.net/, [Accessed: February, 2010].

[13] LKST. http://lkst.sourceforge.net/, [Accessed: February, 2010].

[14] LTT. http://www.opersys.com/LTT/, [Accessed: February, 2010].

[15] DTI. http://sourceforge.net/projects/dti/, [Accessed: February, 2010].

[16] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, and al. K42: building a complete operating system. In EuroSys ’06: Proceedings of the 2006 EuroSys conference, pp. 133–145, 2006.

[17] A. Nataraj, A. Malony, S. Shende, and A. Morris. Kernel-level measurement for integrated parallel performance views: the KTAU project. In IEEE International Conference on Cluster Computing, 2006.

[18] LTTng. http://lttng.org/, [Accessed: February, 2010].

[19] H. Marouani, M. Dagenais. Internal clock drift estimation in computer clusters. Journal of Computer Systems, Networks, and Communications, article 9, volume 2008, 2008.

[20] P. Ashton. The Amoeba Interaction Network: Initial Results. Publication TR-COSC 09/95, Department of Computer Science, University of Canterbury, 1995.

[21] D. L. Mills. Precision synchronization of computer network clocks. ACM SIGCOMM Computer

Communication Review 24(2), volume 24, Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 28–43, 1994.

[22] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Estimating global time in distributed systems. Proc. 7th Int. Conf. on Distributed Computing Systems, Berlin, volume 18, 1987.

[23] J. M. Jezequel. Building a global time on parallel machines. Proceedings of the 3rd International Workshop on Distributed Algorithms, LNCS, volume 392, 136–147, 1989.

[24] R. Sirdey, and F. Maurice. A linear programming approach to highly precise clock synchronization over a packet network. 4OR: A Quarterly Journal of Operations Research 6(4), volume 6, Springer, 393–401, 2008.

[25] B. Scheuermann, W. Kiess, M. Roos, F. Jarre, and M. Mauve. On the Time Synchronization of Distributed Log Files in Networks With Local Broadcast Media. Networking, IEEE/ACM Transactions on 17(2), volume 17, 431–444, April 2009.

[26] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Swanson, and J. Isaak. Visualizing Dynamic Software System Information through High-level Models. In Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, p.p. 271-283, 1998.

[27] A. Hamou-Lhadj, T. C. Lethbrridge, L. Fu. SEAT, A Usable Trace Analysis Tool. In Proc. of the 13th International Workshop on Program Comprehension, pp. 157-160, 2005.

[28] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, and J. Vlissides, J. Yang. Visualizing the Execution of Java programs. In Proc. of the International Seminar on Software Visualization, LNCS 2269, pp. 151-162, 2002.

[29] A. Hamou-Lhadj A. and T. Lethbridge. An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls. In Proc. of the 1st International Workshop on Dynamic Analysis (WODA), Co-located with ICSE, Portland, Oregon, USA, 2003.

[30] A. Hamou-Lhadj and T. C. Lethbridge. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. International Conference on Program Comprehension (ICPC), Athens, Greece. pp. 181–190, 2006.

[31] T. Systä, Understanding the Behaviour of Java Programs. In Proc. of the 7th Working Conference on Reverse Engineering, pp. 214-223, 2000.

[32] T. Richner and S. Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. In Proc. of the 18th International Conference on Software Maintenance, pp. 34-43, 2002.

[33] D. Jerding, J. Stasko, and T. Ball. Visualizing Interactions in Program Executions. In Proc. of the International Conference on Software Engineering, pp. 360-370, 1997.

[34] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. Journal of Systems & Software, 81(11), pp. 2252–2268, 2008.

[35] C. Bennett, D. Myers, M. A. Storey, D.M. German, D. Ouellet, M. Salois, and P. Charland. A Survey and Evaluation of Tool Features for Understanding Reverse Engineered Sequence Diagrams. Journal of

Software Maintenance and Evolution: Research and Practice, 20(4), pp. 291-315, 2008.

[36] J. W. Lloyd. Practical Advantages of Declarative Programming. In Joint Conference on

Declarative Programming, GULP-PRODE ’94, 1994.

[37] Snort. http://www.snort.org/docs, [Accessed: February, 2010].

[38] Deadlock. http://en.wikipedia.org/wiki/Deadlock, [Accessed: February, 2010].

[39] The Snort Project. Snort users manual version 2.8.5, October 22, 2009.

[40] Secnology. http://www.secnology.com, [Accessed: February, 2010].

[41] A. Mounji Naji Habra, B. Le Charlier and I. Mathieu. Asax: Software architecture and rule-based language for universal audit trail analysis. Computer Security ESORICS 92, 648/1992:435-450, April 2006.

[42] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. Journal of Computer Security, 10(1/2): 71-104, 2002.

[43] The state machine compiler. http://smc.sourceforge.net/, [Accessed: February, 2010].

[44] Ragel. http://www.complang.org/ragel/, [Accessed: February, 2010].

[45] A. Thurston. Ragel State Machine Compiler - User Guide, 2007. http://www.complang.org/ragel/, [Accessed: February, 2010].

[46] B. Morin and H. Debar. Correlation of intrusion symptoms: an application of chronicles. In Proceedings of the 6th International Conference on Recent Advances in Intrusion Detection, RAID '03, pages 94-112, 2003.

[47] J. Zimmermann, L. Mé, and C. Bidan. An improved reference of control model for policy-based intrusion detection. In ESORICS, pages 291-308, 2003.

[48] C. Michel, L. Mé, and L. M. Adele. An attack description language for knowledge-based intrusion detection. In In Proceedings of the 16th International Conference on Information Security, IFIP/SEC 2001, pages 353-368, 2001.

[49] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (p-best)*. IEEE Symposium on Security and Privacy - Oakland, California, May 1999.

[50] F. Cuppens and R. Ortalo. Lambda: A language to model a database for detection of attacks. In Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection, RAID '00, pages 197-216, London, UK, 2000.

[51] P. E. Ammann and J. C. Knight. Data Diversity: an Approach to Software Fault Tolerance. IEEE Transactions on Computers, vol. 37, no. 4, 1988.

[52] J. P. J. Kelly, T. I McVittie, and W. I. Yamamoto. Implementing Design Diversity to Achieve Fault Tolerance. IEEE Software, 1991.

[53] Y. Deswarte, L. Blain, and J.-C. Fabre. Intrusion Tolerance in Distributed Computing Systems. IEEE Symposium on Security and Privacy, pp. 110–121, 1991.

[54] S. Forrest, A. Somayaji, and D. H. Acley. Building Diverse Computer Systems. In Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, 1997.

[55] D. Powell and R. Stroud. Conceptual Model and Architecture of MAFTIA. MAFTIA deliverable D21, Technical Report CS-TR-787, University of Newcastle upon Tyne, 2003.

[56] J.H. Lala. OASIS - Foundations of Intrusion Tolerant System. In Foundations of Intrusion Tolerant System. 2003.[Organically Assured and Survivable Information Systems], pp. x–xix, 2003, IEEE Computer Society, 2003.

[57] Self-Regenerative Systems (SRS). http://www.darpa.mil/ipto/Programs/srs/srs.asp, [Accessed: February, 2010].

[58] ReSIST. http://www.resist-noe.org/overview/overview.html, [Accessed: February, 2010].

[59] A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saıdi, V. Stavridou, and T. Uribe. Dependable Intrusion Tolerance: Technology Demo. In 3rd DARPA Information Survivability Conference and Exposition (DISCEX-III 2003), pp. 128–130, IEEE Computer Society, 2003.

[60] F. Wang, F. Gong, R. Sargor, K. Goseva-Popstojanova, K. Trivedi, and F. Jou. SITAR: a scalable intrusion-tolerant architecture for distributed services. In Foundations of Intrusion Tolerant Systems, 2003.

[61] J. C. Reynolds, J. E. Just, E. Lawson, L. A. Clough, R. Maglich, and K. N. Levitt. The Design and Implementation of an Intrusion Tolerant System. In International Conference on Dependable Systems and Networks (DSN'2002), pp. 285?292, IEEE Computer Society, 2002.

[62] J. Chong, P. P. Pal, M. Atighetchi, P. Rubel, and F. W., Franklin. Survivability Architecture of a Mission Critical System: The DPASA. In 21st Annual Computer Security Applications Conference (ACSAC 2005), pp. 495–504, IEEE Computer Society, 2005.

[63] T. Courtney, J. Lyons, H. V. Ramasamy, W. H. Sanders, M. Seri, M. Cukier, M. Atighetchi, P. Rubel, C. Jones, F. Webber, P. Pal, R. Watro, and J. Gossett. Providing Intrusion Tolerance with ITUA. International Conference on Dependable Systems and Networks (DSN), 2002.

[64] P. P. Pal, P. Rubel, M. Atighetchi, F. Webber, W. H. Sanders, M. Seri, H. Ramasamy, J. Lyons, T. Courtney, A. Agbaria, M. Cukier, J. M. Gossett, and I. Keidar. An architecture for adaptive intrusion-tolerant applications: Experience with Auto-adaptive and Reconfigurable Systems. Software – Practice & Experience, 36(11-12), 1331–1354, 2006.

[65] P. Verissimo, N. F. Neves, C. Cachin, J. A. Poritz, D. Powell, Y. Deswarte, R. J. Stroud, and I. Welch. Intrusion-tolerant middleware: the road to automatic security. IEEE Security & Privacy, 4(4), 54–62, 2006.

[66] D. Sames, B. Matt, B. Niebuhr, G. Tally, B. Whitmore, and D. E. Bakken. Developing a Heterogeneous Intrusion Tolerant CORBA System. In the Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN ’02), pp. 239–248, Washington, DC, IEEE Computer Society, 2002.

[67] B. Littlewood and L. Strigini. Redundancy and Diversity in Security. 9th European Symposium on

Research in Computer Security (ESORICS ’04), Lecture Notes in Computer Science (LNCS), vol. 3193, pp. 423–438, 2004.

[68] R. J. Stroud, I. S. Welch, J. P. Warne, and P. Y. A. Ryan. A Qualitative Analysis of the Intrusion-Tolerance Capabilities of the MAFTIA Architecture. In International Conference on Dependable Systems and Networks (DSN 2004), pp. 453–, 2004 IEEE Computer Society, 2004.

[69] B. B. Madan, K. Goseva-Popstojanova, K. Vaidyanathan, and K. S. Trivedi. A method for modeling and quantifying the security attributes of intrusion tolerant systems. Perform. Eval., 56(1-4), 167–186, 2004.

[70] D. Wang, B. B. Madan, and K. S. Trivedi. Security analysis of SITAR intrusion tolerance system. In the proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems (SSRS ’03), pp. 23–32, ACM , 2003.

[71] K. Goseva-Popstojanova, F. Wang, R. Wang, F. Gong, K. Vaidyanathan, K. Trivedi, and B. Muthusamy. Characterizing intrusion tolerant systems using a state transition model. In the proceedings of DARPA Information Survivability Conference & Exposition II, DISCEX ’01, Vol. 2, pp. 211–221, 2001.

[72] J. Knight, D. Heimbigner, A. L. Wolf, A. Carzaniga, J. Hill, P. Devanbu, and M. Gertz. The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. Technical Report CU-CS-926-01, University of Colorado, Department of Computer Science, 2001.

[73] O. Rodeh, K. P. Birman, D. Dolev. The architecture and performance of security protocols in the ensemble group communication system: Using diamonds to guard the castle. ACM Transactions on Information and System Security 4(3): 289-319 (2001) [74] (to be completed)

[74] C. Cachin, K. Kursawe, F. Petzold, V. Shoup. Secure and Efficient Asynchronous Broadcast Protocols. Advances in Cryptology, LNCS 2139, pp. 524?541, J. Kilian, ed., Springer-Verlag, 2001.

[75] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS), Volume 20, Issue 4, November 2002.

[76] P. Koopman. Elements of the self-healing system problem space. In: Workshop on Software Architectures for Dependable Systems (WADS2003), International Conference on Software Engineering (ICSE’03), Portland, Oregon, May 3–11, 2003.

[77] R. Laddaga. Active software. 1st International Workshop on Self-Adaptive Software, 2000.

[78] D. Ghosh, R. Sharman, H. R. Rao, and S. Upadhyaya. Self-healing systems - survey and synthesis. In the Journal of Decision Support Systems, vol. 42, no. 4, Elsevier, 2007.

[79] M. Shaw. Self-healing: softening precision to avoid brittleness. WOSS'02: Workshop on Self-Healing Systems, 2002.

[80] R. Sharman, H. R. Rao, S. Upadhyaya, P. Khot, S. Manocha, and S. Ganguly. Functionality defense by heterogeneity: a new paradigm for securing systems. In Proceedings of the 37th Hawaii International Conference on System Sciences, 2004.

[81] Y. Hong, D. Chen, L. Li, and K. Trivedi. Closed loop design for software rejuvenation. In Proceedings of the Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.

[82] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In Proceedings of the 25th International Symposium on Fault-Tolerant Computing, 1995.

[83] R. De Lemos and J. L. Fiadeiro. An architectural support for self-adaptive software for treating faults. In Proceedings of the First Workshop on Self-Healing Systems, 2002.

[84] H. Chen, G. Jiang, K. Yoshihira. Failure Detection in Large-Scale Internet Services by Principal Subspace Mapping. IEEE Transactions on Knowledge and Data Engineering, 2007.

[85] E. Kiciman, A. Fox. Detecting and localizing application-level failures in Internet services. IEEE Transactions on Neural Networks, 2005.

[86] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff. A Sense of Self for Unix Processes. In Proc. of the 1996 IEEE Symposium on Security and Privacy, 1996.

[87] S. Forrest, S. A. Hofmeyr and A. Somayaji. The Evolution of Systemcall Monitoring. In Proc. of the Annual Computer Security Applications Conference, pp. 418-430, 2008.

[88] M. Jiang, M. A. Munawar, T. Reidemeister, P. Ward. System monitoring with metric-correlation models: problems and solutions. In Proc. of the 6th international conference on Autonomic computing, pp. 13-22, 2009.

[89] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In Proceedings of the 21st ACM Symposium on Operating Systems Principles, 2009.

[90] M. Costa, M. Castro, L. Zhou, L. Zhang, M. Peinado. Bouncer: securing software by blocking bad input. In Proc. of the ACM Symposium on Operating Systems Principles, pp. 117-130, 2007.

[91] C. Dabrowski and K. L. Mills. Understanding self-healing in service discovery systems. In Proceedings of the First Workshop on Self-Healing Systems, 2002.

[92] Service Availability Forum. www.saforum.org, [Accessed: February, 2010].

[93] J. Aldrich, V. Sazawal, C. Chambers, and D. Nokin. Architecture-centric programming for adaptive systems. In Proceedings of the First Workshop on Self-Healing Systems, 2002.

[94] S. Landau, M. R. Stytz, C. E. Landwehr, and F. B. Schneider. Overview of Cyber Security: A Crisis of Prioritization. In IEEE Security and Privacy, vol. 3, no. 3, pp. 9–11, May/June, 2005.

[95] D. Gao, M. K. Reiter, and D. Song. Behavioral distance for intrusion detection. In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, 2005.

[96] D. Gao, M. K. Reiter, and D. Song. Behavioral distance measurement using Hidden Markov Models. In Proc. of the 9th International Symposium on Recent Advances in Intrusion Detection, 2006.