Tamper-resistant Monitoring for Securing Multi-core Environments

Ruchika Mehresh[1], **Jairaj J. Rao**[1], **Shambhu Upadhyaya**[1], **Sulaksh Natarajan**[1], **and Kevin Kwiat**[2]
[1]Department of Computer Science and Engineering, State University of New York at Buffalo, NY, USA
[2]Air Force Research Laboratory, Rome, NY, USA

Abstract - *Complex software is not only difficult to secure but is also prone to exploitable software bugs. Hence, an intrusion detection system if deployed in user space is susceptible to security compromises. Thus, this 'watcher' of other software processes needs to be 'watched.' In this paper, we investigate a tamper-resistant solution to the classic problem of 'Who watches the watcher?'*

In our previous work, we investigated this problem in a uni-core environment. In this paper, we design a real-time, light-weight, watchdog framework to monitor an intrusion detection system in a multi-core environment. It leverages the principles of graph theory to implement a cyclic monitoring topology. Since our framework monitors intrusion detection systems, the attack surface it has to deal with is considerably reduced. The proposed framework is implemented and evaluated using AMD SimNow simulator. We show that the framework incurs a negligible memory overhead of only 0.8% while sustaining strong, tamper-resistance properties.

Keywords: Attacks, Graph models, Intrusion detection, Multi-core, Processor monitoring, Recovery, User space components

1 Introduction

Growing connectivity of computer networks has made network services like wu-ftp, httpd, BIND, etc. a popular target for cyber attacks. Exploitation of these services makes the entire host vulnerable to further exploits. Since more and more functionalities are added to such software programs each day, their code base becomes larger and more complex. This increases the probability of existing software bugs, resulting in security vulnerabilities. There are many studies conducted over the years that document the increasing trend of software unreliability and growing intelligence of hacker community [1], [2], [3].

Over the years, industry and research communities have produced several prevention, detection and recovery methodologies. However, preventing all kinds of attacks in today's open networking environment is practically impossible. Therefore, the major burden of securing a system effectively lies with the detection techniques employed. Detection of attacks and suspicious behavior is generally achieved with the help of automated tools such as intrusion detection systems (IDS) [4]. Many IDS tools alert the authorized user when some malicious activity is suspected, or initiate a recovery without attempting to prevent the attack at all. Sometimes they can detect and prevent an attack before it causes any major damage – in which case, they are also called intrusion prevention systems (IPS). In this paper, we discuss intrusion detection systems but the findings equally apply to intrusion prevention systems.

Intrusion detection systems collect and analyze data at two broad levels: the network level and the host level. Host based IDSes have the advantage of proximity and hence can monitor a system closely and effectively. Traditionally, IDSes have been designed to operate in user space which makes them vulnerable to compromises. Advanced malware has been discovered that can disable them upon its installation [5], [6]. Recently, many compromises of IDSes were reported [7], [8], [9]. Moving IDSes completely or partially to the kernel space increases the trusted computing base (TCB), which in turn introduces further (and more serious) problems [10]. Therefore, a simpler and more effective way of ensuring the security of these systems is to design and deploy them in user space, and ensure their correct operation by tamper-resistant monitoring against subversion. If a malware is successful in switching off the IDS, the monitoring system should be able to report this change to an uncompromised authority. This addresses the problem of *'who watches the watcher'* that often arises in an end-to-end security system.

We have earlier studied this problem in a uni-core environment [11]. However, with the advent of multi-core technology, this problem needs to be revisited for two major reasons. First, a multi-core environment presents new security and design challenges [12], [13]. Therefore, existing security solutions need to be reevaluated for adoption in this new environment. Second, it offers the concurrency that can increase the efficiency and efficacy of our previously proposed framework [11], [14].

The overall scheme works as follows. We use a cyclic, tamper-resistant monitoring framework that uses light-weight processes (referred to as process monitors in the sequel) to monitor the IDS. Though we focus on monitoring the host-IDS, this framework can generally monitor any crucial process. Thus, this framework can also assist in reducing the size of a system's TCB. Process monitors in this framework

are responsible for performing simple conditional checks. A primary conditional check is to continuously monitor if a process is up and running. Rest of the conditional checks can be implemented and enabled according to security and efficiency requirements of the system. If any of the conditional checks fail, an alert notification is sent to the uncompromised authority (mostly the root). These process monitors monitor each other in a cyclic fashion without leaving any *loose* ends. If one of them is killed, the next in the order raises an alert. Since the monitoring is cyclic, no process monitor is left unmonitored. One of these process monitors has an additional responsibility of monitoring the host-IDS. If an attacker intends to subvert the IDS, he first needs to subvert the process monitor monitoring it. Since this process monitor is being monitored by another process monitor, and so on, the subversion becomes almost impossible (we discuss the possible attack scenarios in Section V). Loop architectures and concepts from graph theory have long been used to make designs reliable and robust [15]. In this paper, we identify the benefits of a cyclic monitoring framework, the issues in maintaining it and the kind of performance overhead it incurs.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 states the assumptions. Section 4 gives the system architecture. Section 5 discusses the threat model, while Section 6 presents the various framework topologies. Section 7 describes the experiments and results, followed by conclusion and future work in Section 8.

2 Related Work

As discussed previously, intrusion detection systems are traditionally located in the user space. Tools like DWatch [16] are implemented completely in user space and monitor other daemon processes. Implementing IDS in user space may have its performance advantages [19] but access to these detection systems is not very well protected. Hence, they are equally susceptible to a security compromise as any other process. There are many intrusion detection systems that are implemented completely inside the kernel [17]. Kernel space implementation of an intrusion detection system is a very tempting choice because it provides strong security (process privileges required). However, such an implementation has high associated overhead. Each time a kernel-implemented IDS event is invoked, there is expensive context switching. Besides, IDS being complex software has a high probability of residual software bugs. These bugs can either cause severe negative performance impacts at kernel level, or make the entire kernel vulnerable to a security. Also, IDS tools generally need to apply frequent updates because new attacks are discovered each day. This not only results in high performance overheads but carries a risk of infecting the TCB with bad code. Implementations like the Linux security modules (LSM) [21] provide a diffuse mechanism to perform checks inside the kernel at crucial points, but it is an expensive solution. There are some hybrid detection techniques [18] that are partially in kernel space and rest in the user space. Such techniques inherit some strengths as well as weaknesses from both the domains. For instance, an entire IDS implementation in kernel space, with just the notification mechanism in user space can be subverted by curbing the notifications.

Some monitoring architectures secure intrusion detection systems by proposing the use of isolated virtual machines [5]. Such systems solve the problem of securing the watcher, but require installation of separate virtual machines and process hooks. These features increase the cost of such security solutions.

Our framework is an extension of the user space framework proposed by Chinchani et al. [11]. They proposed a tamper-resistant framework in a uni-core environment to secure user space services. In this paper, the framework is extended to protect user space components in a multi-core environment.

3 Assumptions

i) All software components are assumed to be susceptible to security attacks.

ii) We assume a zero-trust model. Therefore, an attacker cannot predict the order of process monitors in the framework topology by observing system behavior.

iii) All process monitors are identical and light-weight.

4 System Architecture

The proposed framework runs on a K-core host. We assume symmetrical multiprocessing (SMP), where all cores are managed by the same operating system instance. This system runs a user space, host-based IDS that monitors other host processes for any suspicious activity. We mentioned earlier that some malware can attack the IDSes upon their installation. Since the IDS here is in the user space, it is susceptible to security compromise as any other process. Therefore, we design a monitoring framework to ensure that the IDS is running in a tamper-resistant mode at all times.

This framework primarily consists of light-weight process monitors. These process monitors are simple programs that monitor other processes for specific conditions. The primary condition is to check continuously if the monitored process is running. If it is killed or any other condition fails, the process monitor detects it and sends an alert to the root user.

In the simplest topology of this framework, process monitors are arranged in a cyclic fashion (as shown in Figure 1). Since there are no loose ends in a cycle, every process monitor is monitored by another. To ensure parallel monitoring, IDS and all the process monitors run on separate

cores. For instance, in a K-core system, if the IDS is running on Core 1, rest of the K-1 cores run the process monitors, one on each. Process monitor on Core 2 monitors the IDS on Core 1, as well as the process monitor on Core K.

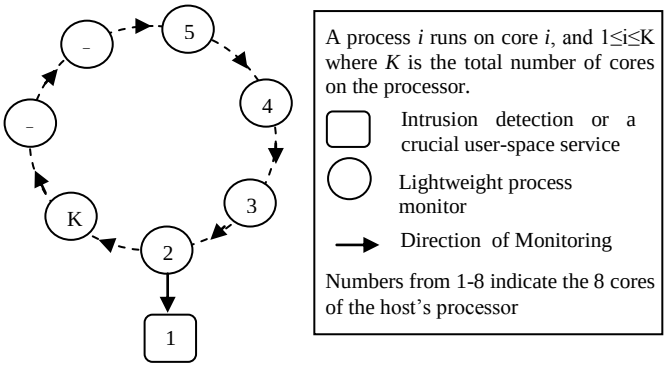

Figure1: A simple-ring topology on a K-core system.

5 Threat model

i) *Denial of Service attacks in multi-core systems:* Memory hogging [12] is a denial of service attack where one core consumes shared memory unfairly. This results in performance degradation at other cores due to resource scarcity. This kind of attack can be handled by our framework via conditional checks. A process monitor can raise an alert if it observes exceptionally high scheduling delays affecting the monitored process.

ii) *Window of vulnerability:* There are windows of vulnerability introduced because of multi-core scheduling. We assume that process monitors hosted on separate cores can continuously monitor each other. This, however, is not practically feasible. If the core hosting the process monitor has other processes scheduled on it, the process monitor will have to go back into the scheduling queue periodically. During this window, if monitored process is attacked, the process monitor monitoring it cannot raise an alert. It can only do so when it is rescheduled to run again. If such windows are identified and exploited in order, it is possible to subvert the entire framework. This vulnerability can be patched by employing multiple degree of incidence, meaning that one process monitor is monitored by (and monitors) multiple other process monitors. This way, even when some of them go back into scheduling queues, we can still dynamically maintain at least one monitoring cycle with a high probability. However, this arrangement leads to a performance-security trade-off. Higher degree of incidence provides stronger security but at a higher monitoring cost.

iii) *Exploiting system vulnerabilities (crash attacks, buffer overflow, etc.):* An attacker can try to crash (kill) any of these process monitors by exploiting system vulnerabilities. These vulnerabilities could be introduced by other software running on the system. We will see in Section 5 how such attacks are handled.

6 Framework Topologies

A monitoring framework, cyclic or not, can have numerous topologies. For a K-core system, we can choose from a simple ring topology (as shown in Figure 1) to a topology with multiple degree of incidence (as shown in Figure 2). We will present a few basic topologies here that provide strong tamper-resistance properties.

In order to compare the various topologies, we need to understand the basis on which they can be evaluated. There are two questions that can be asked:

i) How secure a topology is?
ii) How efficient a topology is?

Any topology that can be compromised with a high probability is insecure. In [11], Chinchani et al. discuss the subversion probabilities of simple replication and layered hierarchy (onion peel) topologies. The paper claims that a circulant digraph configuration provides the strongest tamper resistance properties.

Topology 1: Simple Ring

Simple ring topology represents an ordered cycle of process monitors, as shown in Figure 1. It offers a much lower probability of subversion compared to the onion peel model [11]. Since we assumed a zero-trust model, an attacker needs to try (n!-1) permutations (in the worst case), before he figures out the right order. This topology works considerably well for scenarios where all participating cores are minimally loaded, and the process monitors run for most of the times. However, when the workload increases, these process monitors have to wait in scheduling queues for some finite amount of time. If during this time, the processes that they are monitoring are compromised, an alert cannot be raised. Therefore, heavy system load can create windows of vulnerability that if exploited in a certain order, can lead to successful subversion of the framework.

Topology 2: Circulant Digraph

Earlier research proposed circulant digraph as the primary approach to increase efficacy of this monitoring framework (reduce false negatives) [11]. However, in a multi-core environment it has an added benefit of reducing the creation of windows of vulnerability. Higher the degree of incidence, lower is the probability that a process monitor remains unmonitored itself.

A circulant digraph $C_K(a_1, \ldots, a_n)$ with K vertices $v_0, \ldots v_{K-1}$ and jumps $a_1, \ldots, a_n$, $0 < a_i < \lfloor K/2 \rfloor$, is a directed graph such that there is a directed edge each from all the vertices $v_j \pm a_i \bmod K$, for $1 < i < n$ to the vertex v_j, $0 < j < K - 1$. It is

also homogeneous, i.e., every vertex has the same degree (number of incident edges), which is 2n, except when $a_i = K/2$ for some i, when the degree is 2n−1. Figure 2 shows a circulant digraph with 8 process monitors, degree of incidence 3 and jumps {1, 2}.

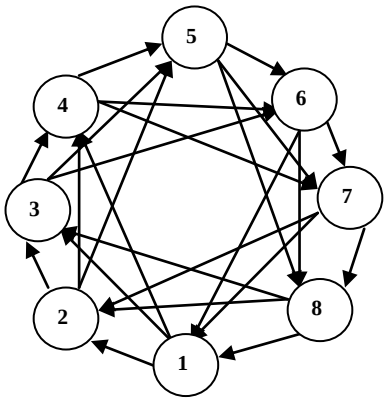

Figure 2: Circulant digraph with 8 process monitors running on 8 cores. One process monitor per core. This circulant digraph has a degree of incidence 3 and jumps {1,2}.

Simple ring topology is a special case of circulant digraph topology with degree of incidence 1. However, a circulant digraph topology (with degree of incidence > 1) is much more secure than a simple ring topology. This is because the number of attempts required to find the right permutation increases exponentially in the worst case (since the attacker does not know the degree of incidence, the jump and the order of process monitors).

Topology 3: Adaptive Cycle

Since raising a large number of alarms is counter-productive to a system's performance, a circulant topology though effective, is not optimal. Even if an attacker is not in a position to attack, he can tamper with the framework to make it raise a large number of useless alerts. To counter this threat and reduce the number of alerts produced by the circulant topology, we propose an adaptive topology. It predicts the system load and tries to maintain cyclic monitoring at all times. This requires that process monitors at each core track the load on other cores. As shown in Figure 3, the initial state of this topology is set to be a simple ring. If process monitor on core 2 realizes that core K has just been assigned a lot of new processes, it starts monitoring process monitors K and K-1, both. Similarly, if core 2 gets heavily loaded, process monitor 3 starts monitoring process monitors 2, K and K-1. So, the cores that are lightly loaded take up the responsibility of monitoring the process monitors on heavily loaded cores and the process monitors they were respectively responsible for.

Therefore, the final state of an adaptive cyclic topology can be formally defined: For a K-core processor, a process monitor on core *i* where $1 \leq i \leq K$, monitors process monitors on all cores *j*, where $i+1 \leq j \leq K$, if there is a directed edge from *i* to *j*, or if there exists a *z* such that there is a directed edge from *i* to *z* and from *z* to *j*.

The probability of subversion for adaptive cycle topology is equal to the probability of subversion for circulant digraph topology. However, the number of attempts required to find the right order of process monitors in an adaptive topology is much larger than in circulant digraph topology (in the worst case). This is because the degree of incidence and jumps are always changing dynamically. Therefore, an adaptive topology provides better performance and stronger security as compare to the circulant digraph topology.

7 Experiment Design

Companies like Intel, AMD, etc., have made significant progress in multi-core technology. Clearspeed's CSX600 processor with 96 cores [19] and Intel's Teraflops Research chip with 80 cores [20] are the latest in this line. However, such systems do not have a strong presence in the commercial market yet. This generally restricts researchers to use a small number of 2-6 cores. In order to bridge this gap between unavailability of present technology and researching the future needs of this technology, simulators have been developed [21], [22]. These simulators emulate the functioning of a multi-core platform on a system with lesser number of cores (even uni-core processor). Amongst the many open source multi-core simulators that are available today, AMD SimNow closely emulates the NUMA architecture. Therefore, we use it as a test-bed to experiment with simple ring and circulant digraph topologies.

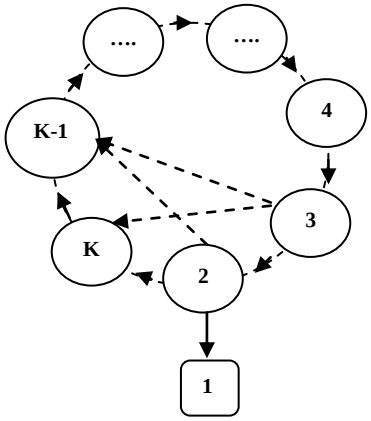

Figure 3: Adaptive topology when cores 2 and K are heavily loaded.

7.1 Configuration

Experiments are conducted on Intel Pentium Core2Duo 2.1 Ghz processor with 4GB RAM. AMD SimNow is installed on Ubuntu 10.04 which is the host operating system. Inside AMD SimNow, we run a guest operating system, i.e., FreeBSD 7.3. All experiments run on this guest operating system. This system is configured to use emulated hardware

of AMD Awesim 800Mhz 8-core processor with 1024 MB RAM.

We use kernel level filters to implement process monitoring. This is because inter-process communication support provided by UNIX-like systems (like pipes or sockets) does not suffice for our framework. Inter-process communication delivers messages only between two live processes. However, we require that a communication (alert) be initiated when a process is terminated. For this purpose, we use an event delivery/notification subsystem called Kqueue, which falls under the FreeBSD family of system calls. Under this setup, a process monitor interested in receiving alerts/notifications about another process creates a new kernel event queue (kqueue) and submits the process identifier of the monitored process. Specified events (kevent) when generated by the monitored process are added to the kqueue of the process monitor. Kevent in our implementation is the termination of the monitored process. Process monitors can then retrieve this kevent from their kqueues at any time. A process monitor can monitor multiple processes in parallel using POSIX threads.

Experimental setup consists of 8 simulated cores with process monitors running on each one of them. We report on the evaluation of only the circulant digraph topology as a representative result. We experiment with different circulant digraph topologies with varying number of process monitors and degrees of incidence. The primary performance metrics in a multi-core system are time and memory overheads. Each reading in this analysis represents an average of 100 runs.

7.2 Execution Performance

The initial setup time is defined as the time taken for the kqueue subsystem to get loaded before an attacker tries to subvert the process monitors. This is the only major time delay this system has been observed to incur. As shown in Figure 4, initial setup time increases linearly with increasing degree of incidence. With 8 process monitors in a circulant digraph topology, the worst case initial setup delay of 0.3ns is obtained with a maximum degree of incidence (i.e., 7).

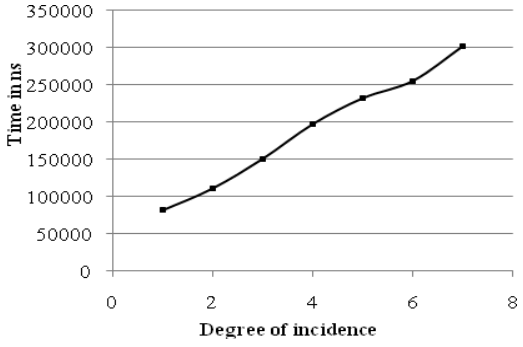

Figure 4: Initial Setup overhead for circulant digraph topology with 8 process monitors.

7.3 Memory Performance

We define the memory overhead to be the amount of memory consumed by a running instance of the framework as a percentage of the entire system memory capacity. Memory overhead is observed to increase linearly with the degree of incidence. A circulant topology with 8 process monitors and degree of incidence 7 incurs a 0.8% (0.1% per process) memory overhead.

Table 1: Categorization of circulant digraph topologies

Configuration	Number of processes	Degree of Incidence
Series1	2	1
Series2	3	1,2
Series3	4	1,2,3
Series4	5	1,2,3,4
Series5	6	1,2,3,4,5
Series6	7	1,2,3,4,5,6
Series7	8	1,2,3,4,5,6,7

7.4 Attack Tolerance

We experimented with different circulant digraph topologies with varying number of process monitors and degree of incidence, as shown in Table 1. For all topologies, jumps start from a minimum of 1, incremented by 1, until it satisfies the degree of incidence.

The following attack scenarios were executed in order to test the security strength of the framework.

Experiment 1: Killing process monitors without delay (under light system load)

We experiment with the worst case scenario where the attacker already knows the correct order of the nodes in this topology. We assume that he also identifies the windows of vulnerability and uses them to his advantage (again, the worst case). In Figure 5, the number of alerts generated shows the sensitivity of this framework toward a crash attack executed using SIGKILL.

Experiment 2: Killing process monitors without delay (under heavy system load)

Experiment 1 was repeated under heavy load conditions to determine the impact of increasing system load on framework's sensitivity (number of alerts) to an attack. A heavy load condition is simulated by running Openssl benchmark in the background. In this emulated multi-core environment, a maximum of 6,164 processes can run on FreeBSD operating system. We ran 6,000 processes to achieve nearly 100% CPU consumption for all cores. As seen in Figure 6, the framework generates lesser number of alerts. This is because the process monitors have to wait in the scheduling queue longer than in Experiment 1.

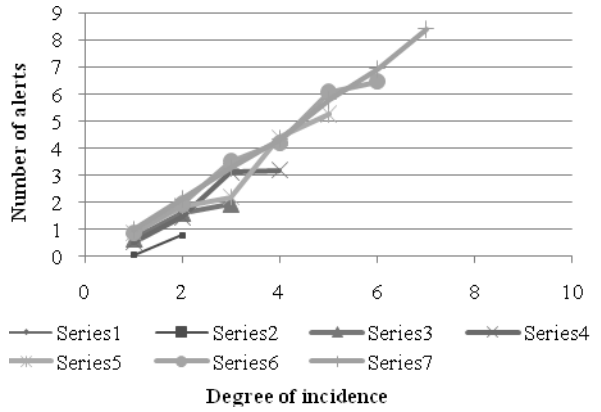

Figure 5: Alerts generated for killing process monitors in sequential order without delay, under light system load.

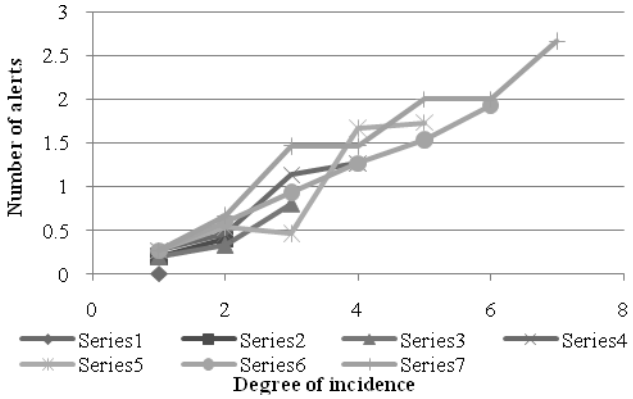

Figure 6: Alerts generated for killing process monitors in sequential order without delay, under heavy system load.

Experiment 3: Group Kill attack

This framework is created by forking a process into child processes. All child processes forked from the same parent belong to the same group by default (identified by the same group ID). An external process can easily identify the group ID (GID) from the kernel proc structure using commands such as 'ps' from the user space. Any crash attack on this process monitor group can be represented by a SIGKILL signal sent to the GID of the process monitors. This attack successfully subverts the framework and no alerts are raised by any of the process monitors. Thus, this constitutes a successful attack on the framework, where the property of all the process monitors belonging to a common default group becomes a vulnerability.

In order to increase framework's resistance to these kinds of attacks, we organized alternate process monitors under two different groups. Process monitors with even PIDs (process IDs) retain their default GID, which is the PID of the parent process. The GID of process monitors with odd PIDs is changed to their respective PIDs. Now, a SIGKILL signal sent to the default GID of the group will successfully kill the processes with even PIDs, but the odd ones will raise alerts.

8 Conclusion and Future work

This paper proposed a tamper-resistance framework to monitor the intrusion detection systems (IDS) in a multi-core environment. We identified the benefits of our framework and the related issues. We also analyzed two framework topologies, viz. simple ring and circulant digraph. They are found to incur low time and memory overhead, while still retaining strong tamper-resistance properties.

As a future work, we plan to investigate the adaptive ring and other topologies. We plan to add more attack scenarios to this analysis. For instance, a smart attacker can replace a process monitor with a dummy process to subvert the framework.

9 Acknowledgement

This research is supported in part by a grant from the Air Force Office of Scientific Research (AFOSR). The work is approved for Public Release; Distribution Unlimited: 88ABW-2011-1929 dated 31 March 2011.

10 References

[1] A. Nhlabatsi, B. Nuseibeh, and Y. Yu, "Security requirements engineering for evolving software systems: A survey," *Journal of Secure Software Engineering*, vol. 1, pp. 54–73, 2009.

[2] N. Dulay, V. L.Thing, and M. Sloman, "A Survey of Bots Used for Distributed Denial of Service Attacks," *International Federation for Information Processing Digital Library*, vol. 232, 2010.

[3] T. Heyman, K. Yskout, R. Scandariato, H. Schmidt, and Y. Yu, "The security twin peaks," *Third international conference on Engineering secure software and systems*, 2011.

[4] F. Sabahi, and A. Movaghar, "Intrusion Detection: A Survey," *Third International Conference on Systems and Networks Communications (ICSNC)*, pp. 23-26, 2008.

[5] B. D. Payne, M. Carbone, M. Sharif, and L. Wenke, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *IEEE Symposium on Security and Privacy (SP)*, pp. 233-247, 2008.

[6] T. Onabuta, T. Inoue, and M. Asaka, "A Protection Mechanism for an Intrusion Detection System Based on Mandatory Access Control," *Society of Japan*, vol. 42, 2001.

[7] Greg Hoglund, "Malware commonly hunts down and kills anti-virus programs," *Computer Security Articles* 2009.

[8] Hermes Li, "Fake Input Method Editor(IME) Trojan," *Websense Security Labs*, 2010.

[9] Christopher Null, “New malware attack laughs at your antivirus software,” *Yahoo! News*, 2010.

[10] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, pp. 77-110, 2005.

[11] R. Chinchani, S. Upadhyaya, and K. Kwiat, "A tamper-resistant framework for unambiguous detection of attacks in user space using process monitors," *First IEEE International Workshop on Information Assurance (IWIAS)*, pp. 25-34, 2003.

[12] T. Moscibroda, and O. Mutlu, "Memory performance attacks: denial of memory service in multi-core systems," *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (SS)*, 2007.

[13] C. E. Leiserson, and I. B. Mirman, "How to Survive the Multicore Software Revolution (or at Least Survive the Hype)," *Cilk Arts Inc.*, 2008.

[14] S.P. Levitan and D. M. Chiarulli, "Massively parallel processing: It's Déjà Vu all over again," *46th ACM/IEEE Design Automation Conference (DAC)*, pp. 534-538, 2009.

[15] S.L. Hakimi, and A. T. Amin, "On the design of reliable networks," *Networks*, vol. 3, pp. 241-260, 1973.

[16] U. Eriksson, "Dwatch - A Daemon Watcher," *http://siag.nu/dwatch/*, 2001.

[17] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *Foundations of Intrusion Tolerant Systems (OASIS)*, pp. 213, 2003.

[18] N. Provos, "Improving host security with system call policies," *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[19] Y. Nishikawa, M. Koibuchi, M. Yoshimi, A. Shitara, K. Miura, and H. Amano, "Performance Analysis of ClearSpeed's CSX600 Interconnects," *IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 203-210, 2009.

[20] Intel, "Teraflops Research Chip " *http://techresearch.intel.com/ProjectDetails.aspx?Id=151*.

[21] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," *Sixth IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1-12, 2010.

[22] A. Vasudeva, A. K. Sharma, and A. Kumar, "Saksham: Customizable x86 Based Multi-Core Microprocessor Simulator," *First International Conference on Computational Intelligence, Communication Systems and Networks*, pp. 220-225, 2009.