An Intrusion-Tolerant and Self-Recoverable Network Service System Using A Security Enhanced Chip Multiprocessor

Weidong Shi Hsien-Hsin S. Lee Guofei Gu Laura Falk†
Trevor N. Mudge† Mrinmoy Ghosh

School of Electrical and Computer Engineering
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{shi, guofei}@cc.gatech.edu
{leehs, mrinmoy}@ece.gatech.edu

†Department of Electrical Engineering
and Computer Science
University of Michigan
Ann Arbor, MI 48109
{laura, tnm}@eecs.umich.edu

ABSTRACT

This paper proposes a novel system design using a chip multiprocessor (CMP) to provide intrusion tolerance and self-recovery for server applications. Our platform provides three major advantages over previously proposed approaches, 1) security insulation from remote exploits and attacks; 2) close coupling between processor cores in a CMP to ensure immediate logging, fine-grained inspection and fast recovery; 3) concurrent and fine-grained inspection, logging and recovery techniques that are off of the critical path. We have designed a multi-point defense and recovery system to defeat remote exploits. We used a checkpoint based approach to recover server applications under attack. It takes a snapshot of the application's context and memory state before it handles the next request. If the request turns out to be malicious, the system can discard the malicious request and rollback the application's state to a known good one through checkpointing. We have also designed an rapid recovery system for kernel space rootkit attacks. Our intrusion survivable and self-recovery design provides reliable production services that System Administrators are seeking.

Keywords

Intrusion-tolerant computing, survivable service, chip multi processor, self-healing, rootkits, buffer overflow.

1. INTRODUCTION

In the face of remote attacks, quick recovery and service reliability are among the most important features that are sought after by System Administrators and Internet Service Providers. Although research has been devoted to the area of intrusion detection and prevention, little emphasis was made in addressing how to make a system tolerant of intrusions. Such a system would provide a highly efficient, non-intervening, autonomic service recovery system. Conventional service protection and recovery systems have the following characteristics. 1) Reliability of service, as a whole, is favored at the cost of individual services. When a new security vulnerability is detected for a particular application, the service is often terminated to avoid a more disastrous security breach until a patch is available; 2) The process is laborious and prone to human errors. After a system is compromised, hackers often conceal their activities by modifying the system logs and installing rootkits or back-doors. This is often accomplished through modifying some system binaries or the system call table [14, 15]. At this point, recovering a system after this level of compromise requires human intervention, usually to the degree of reloading the system.

More recently, new work [27, 21] has been initiated with a goal of providing self-healing and automatic recovery capability based on a single-processor, single-system model. Such an approach contains the following drawbacks. First, the degree of security is questionable. A single system based protection/recovery scheme places the security of the protection/recovery mechanism at the same level as the protected system. Once the system is compromised, the recovery mechanism could be bypassed, circumvented, or subverted. Second, self-healing and increased service availability is often achieved at the cost of performance. In order to recover from a compromised system state, timely logs or checkpoints must be maintained, which degrades performance. Third, most of the proposed systems provide only limited recovery capability. For example, [27] provides security for only selected functions.

A new class of self-healing techniques based on virtual machines (VM) have also been studied. When comparing single-system solutions with virtual machine based solutions, the virtual machines will be more secure because they provide isolation between the host system and the emulated system. However, Virtual Machines are vulnerable to network attacks when the VM along with the emulated system is viewed as one single network application. A VM is susceptible to network exploits such as buffer overflow just the same as any other regular network application.[1] A VM-

[1]Note that disabling network stack on the host side does not solve VM security problem completely because the whole

based self-healing system usually requires switching between native mode and VM mode. Most of the security countermeasures are implemented in the VM mode. In order to be fully capable of detecting and surviving any attacks, every state change and instruction needs to be executed and monitored by the virtual machine, leading to a substantial performance impact. In other words, a practical self-healing system using a VM needs to be coarse-grained or limited if a reasonable service throughput is to be maintained.[2]

In this paper, we propose a new intrusion survivable computing and self-healing scheme based on emerging chip multiprocessor (CMP) or multi-core platforms [16, 4]. The CMP is a natural evolution for future processor design, which utilizes the available transistors on chip and alleviate the complexity of design and verification. Almost all major processor vendors have already offered or released roadmaps for their own CMP designs [25, 28, 13]. Most of the current CMP designs adopt a *symmetric* CMP model, where each processor core is functionally identical. As we will show later, however, by imposing a carefully crafted security structure to the CMP cores, a new CMP programming model can facilitate the construction of a new wave of self-healing and intrusion tolerant systems. Comparing these with previous self-healing systems, our CMP-based autonomic system has the following characteristics and advantages:

- **Processor core differentiation.** In contrast to conventional CMP models, our security enhanced CMP model assigns different roles to different cores on a CMP. Through the BIOS settings, some processor cores (or monitor cores) can be configured to run at a higher security or privileged level than the remaining cores (or protected cores). These higher privileged cores can monitor, reset, or recover the protected cores.[3]
- **Attack/exploit/compromise insulation.** The protected cores and the monitor cores have different access privileges for hardware resources (e.g. main memory, storage, etc.) The monitor cores also run an entirely different operating system from the rest. Such isolation at both the physical layer (processor and other hardware resources) and the system layer (different operating systems) provides protection and assurance as to the level of security and recovery services that are available on the monitor cores. It will be extremely hard for any compromise on the protected cores to affect the monitor cores.
- **Rapid logging and recovery support on closely coupled CMP cores.** Our proposed CMP model allows different processor cores to share states. The monitor cores can log or checkpoint states of the protected cores for future recovery. When any protected core is corrupted or compromised, instead of rebooting the system or terminating the affected application, the monitor cores can perform a quick recovery based on the logs.
- **High performance intrusion tolerance and self-healing ability**. Since security tasks on the monitor cores, and regular applications on the protected cores, are executed concurrently, the performance overhead of the implemented security features are minimized.

The major contributions of this paper are:

- This paper proposes a new security enhanced CMP model. There are two classes of processor cores: monitors and protected cores. The new computing model, equipped with intrusion-tolerance and self-healing techniques, makes it easy to install the operating system and software with greater assurance of service reliability.
- This paper describes a system architecture for our model. The proposed system uses a unique checkpoint approach to quickly repair services disrupted by remote exploits. It also uses a unique approach to protect the system from kernel level rootkits and prevent hackers from taking over the system.
- This paper evaluates the potential of such systems using real documented vulnerabilities and exploits.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the concept of intrusion tolerant computing and autonomic recovery and present an asymmetric CMP model to achieve this goal. Section 3 details the system architecture and demonstrates the capability of intrusion tolerant computing and fast recovery. Section 4 evaluates security and performance, followed by related work in Section 5. Finally, Section 6 presents concluding remarks.

2. AVAILABILITY AND SELF-RECOVERY ORIENTED COMPUTING

In this section, we introduce the concept of self-healing and intrusion-tolerant computing. A system with the capability of self-healing and intrusion tolerance can effectively improve its service reliability. In addition, we discuss a multi-dimensional scheme for designing and managing a self-healing and intrusion tolerant system. We also address why our security enhanced asymmetric CMP model is the appropriate platform for high assurance intrusion tolerant and self-healing systems. It is worth noting that this paper mainly focuses on self-healing and intrusion tolerance issues for end service systems. Improving service reliability using a network of computers and servers such as service replication is a separate problem and out of the scope of this paper. The approach presented in this paper provides rapid micro-level recovery.

2.1 Threat Model

Our threat model is based on network exploits and remote network attacks. We assume that service application software and parts of the OS contain vulnerabilities, including but not limited to, various forms of buffer overflows [3, 19, 22] that can be exploited, to bring down the application, or worse, suspend the entire system, or allow unlawful access to the system.

2.1.1 Remote attacks

Buffer overflows have been one of the most common remote intrusion techniques [3, 19]. A buffer overflow occurs when a program tries to store data into a buffer that is too small. Exploiting a buffer overflow allows a hacker to possibly inject arbitrary code into the execution path and gain unauthorized access to the system resources [3]. After a sys-

system (VM+emulated system) is one network application.

[2]By *limited* we mean that only certain types of state corruption and recovery can be supported.

[3]Throughout this paper, we call the processor core used for running server applications the "protected core" or "server core". The processor core that runs the security monitoring and recovery software is called the "monitor core".

tem is compromised, the adversary can hide their tracks by altering or erasing audit or system log files. The adversary can set up a concealed back-door by installing various *rootkits*, which he could use to return and wreak more havoc on the system. The rootkits install modified system binaries leaving, not only back-doors, but a means of concealing the hacker's activities by giving the appearance of a healthy system [14, 15]. The hacker can then set up malicious attacks such as distributed denial-of-service.

2.1.2 Availability and recovery

One severe consequence of network attacks is the loss of service, reducing service reliability. The current recovery scheme is error-prone and time-consuming, needing extensive human intervention. As shown in a real case study [15], in order to recover a compromised system and its services, it often involves the following steps. The administrator needs to spots the suspicious behavior in a compromised system. The system is then examined for known exploits and rootkits. The detection and removal of a rootkit is a laborious process [15] and usually requires a clean restore of the system. By analyzing network traces or service log files (given that these files are not deleted or altered by the hackers), the administrator may be able to pinpoint the suspicious activity and patch the application in question.

2.2 Self-healing and Intrusion Tolerant Computing

As shown above, conventional recovery processes requiring human intervention are time-consuming and expensive, thus reducing service reliability. To increase service reliability, a new recovery approach is required. Intrusion tolerant computing is such an alternative. Intrusion tolerant computing prolongs service reliability by running vulnerable applications or compromised systems and correcting them by a method of self-healing and autonomic recovery. In the face of remote exploits, instead of terminating the vulnerable or corrupted application or system, an intrusion-tolerant system will try to disconnect the infected service from the network, repair the damage, and restore the system/application in real-time back to normal. Typical damage caused by remote exploits are memory corruption, destruction of important data structures, system data corruption, or maliciously patched kernel text or data. The repair is based on timely logging of essential machine states. Intrusion tolerant computing allows for continued execution of services and systems before the nature of the exploit is fully understood. It serves as a temporary solution before a complete solution such as a new patch or rootkit removal utility is available. Recurring attacks can continue to "wound" the system. However, the system is capable of quickly recovering from the damage and can continue to serve legitimate and well-behaved requests. To achieve intrusion-tolerance and self-healing efficiently, the system must be supported with the following requirements:

- **Insulation.** An exploit insulation or immunity component as the host for services of self-healing and recovery. This component provides unconditional protection and assurance to the system that prevents the recovery and healing services from being bypassed, circumvented or subverted.
- **Introspection.** The ability to quickly detect attempted intrusions, state corruption, or system compromises.

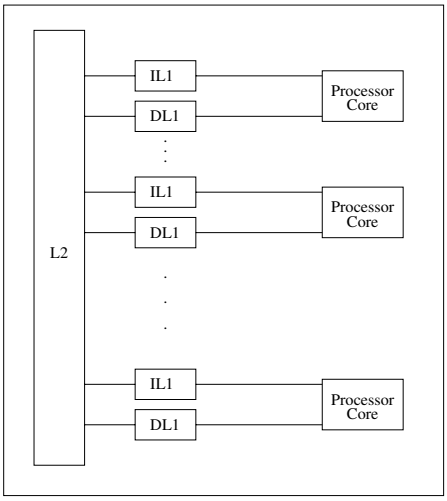

Figure 1: Chip Multiprocessor

- **Prevention.** The ability to prevent several intrusions and reduce the workload of the recovery process.
- **Recovery.** The ability to recover from a damaged system state to a known good state, and restore services to normal in a timely manner.
- **Performance.** The whole detection and recovery cycle needs to be efficient with minimal performance overhead to the service software for practical adoption.

2.3 Security Enhanced CMP

In order to meet these requirements, we propose to use a security enhanced asymmetric CMP platform for implementing an intrusion-tolerant computing system with high assurance. Figure 1 illustrates a conventional CMP model where all the processor cores have equal security privileges. Since the transistor count of a single chip doubles every 18 months, CMPs are a natural evolution of deep submicron processor design that fully utilizes the potential of technology advances while minimizing the complexity of design and verification. Today's CMP designs fall into the category of symmetric CMPs in which all the cores have equal security privileges. We propose a new CMP paradigm called the *asymmetric CMP model* that enables self-healing and intrusion tolerance by imposing different security regimes on different CMP cores. We next discuss the design principles of an asymmetric CMP.

2.3.1 Security insulation by privilege partitioning

Processor cores of a CMP can be configured with different security privileges. The high privileged cores, or *monitor cores*, are granted access to all the hardware resources including the entire memory space, I/O devices, and all the DMA engines. The cores with low privilege levels called *protected cores* are only allowed access to certain physical memory areas and peripherals. The monitor cores start to boot at system startup before their low privileged counterparts. They boot from a light-weight secure OS (a few MB) and grant limited hardware access rights to the protected cores. After the monitor cores are up and running, the protected ones start to boot from a normal operating system such as a full-fledged Windows server or Unix/Linux production server. In terms of access rights, the monitor cores

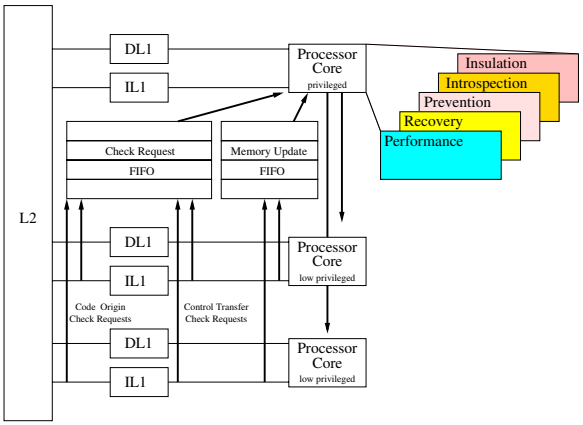

Figure 2: Asymmetric Chip Multiprocessor Model

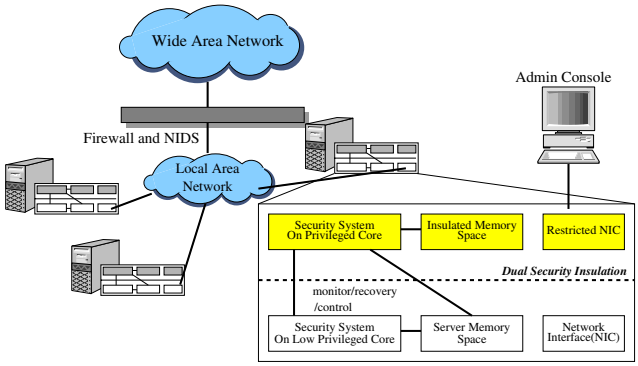

Figure 3: Deployment of Proposed System in a Network Environment

enjoy complete access to all the physical memory space, I/O memories, and peripheral devices whereas the protected cores have only limited access to those resources. In addition, the monitor cores are privy to all the system exceptions while the protected cores are only allowed to respond to certain interrupts and exceptions. The access rights are enforced through hardware protection. For example, a memory interface watchdog sitting between the processor cores and the front side bus (FSB) can inspect memory violations and I/O access rights (request to memory or peripheral bus is tagged with processor core id). An intelligent interrupt/exception routing controller also physically ensures that interrupts/exceptions are received by the entitled cores.

To restrict protected cores from accessing peripheral devices such as a network interface card on the PCI bus, the monitor cores signal the FSB controller to enforce access restrictions on the protected cores in I/O space. If an application's protected core violates the restriction and attempts to read/write in I/O space, an exception will be raised and handled by the monitor cores. In addition, the FSB controller also routes the request to the monitor cores. The monitor cores can check the request and carry out the request on behalf of the application's protected core. This solution uses the monitor cores as a relay for accessing I/O devices. The monitor cores can hide certain PCI devices by not reporting its presence to the application server system.

Such a CMP paradigm creates a hardware sandbox for production service system running on the low privileged cores while having a security system running on the high privileged cores. The two systems are not only physically insulated but also run different operating systems, a lean security runtime system and a full production server OS. This insulation provides higher assurance for service recoverability and reliability than a single-processor based self-healing system or a VM-based self-healing system.

2.3.2 *Fine-grained internal state logging*

Implementing multiple cores in the same chip allows efficient and high speed sharing of internal state information between processor cores. This enables the monitor cores to pull out internal information from the protected cores for inspection. The information pulled from the protected cores includes memory updates, fetched instructions from the unified L2 cache to the L1 instruction cache, [4] or other information necessary for detecting system corruption and compromise [12]. This capability can be easily implemented in a CMP because it only gathers information at the interface and requires no internal processor pipeline changes. The information collected can be sent to the monitor cores via a dedicated hardware FIFO. The monitor cores gather the information and inspect it for any suspicious behavior. If necessary, they use the log or checkpoint information for quick recovery.

2.3.3 *Close processor core coupling and control*

The close coupling of processor cores in a CMP allows a controlling mechanism to be easily installed so the monitor cores can stall, flush the pipeline, reset, and recover protected cores running application services. Since all the cores are on the same die, the monitor core can send signals to stall corrupted cores, flush its pipeline, restore its states and restart execution from a known good state. The close coupling among processor cores enables a mechanism of fast recovery.

3. SYSTEM ARCHITECTURE

In this section, we discuss how to create a self-healing and intrusion-tolerant system using the proposed asymmetric security enhanced CMP. The self-healing and recovery scheme uses multiple lines of defenses and recovery along the five aforementioned levels. The proposed scheme is implemented as security software on the high privileged cores. The software itself has been insulated from remote attacks targeting the protected system. This type of protection is described in Section 2.3.1. The security software on the high privileged cores can be a stand-alone system customized through the BIOS or configurable through a completely different management network. Figure 3 shows that a separate network can be setup to manage the system which connects the high privileged cores through a concealed network channel. The management network can be physically separated from the network used for hosting network services. Administrators can use it to configure or manage the monitoring and recovery systems. The existence of such a network is undetected by the server systems running on less privileged cores. Now we propose an architecture that enables two main forms of defense: (1) Application level exploit tolerance and recovery

[4]Hardware ensures code in the L1 instruction cannot be modified, which means that the L2 to the L1 interface is the natural point for monitoring injected code attacks.

using checkpointing; (2) System/kernel space recovery from rootkits.

3.1 Tolerance and Recovery from Application Level Exploits

The application servers running on the low privileged cores are vulnerable and subject to remote attacks. These attacks include various forms of buffer overflows [3, 19, 22] and other malicious exploits for gaining root level privileges. Table 1 summaries intrusion tolerance and service recovery at the level of service applications. It shows what is done at each of the five security levels.

3.1.1 Introspection

As indicated in [12], the majority of the remote exploits can be detected using a simple inspection mechanism on function call/return addresses, code origins and program control flow transfer. Code origin inspection can detect and prevent execution of injected code. However, only codes originally loaded from the disk can be executed. Call/return address checks prevent the function return address from being modified. Restricted program control flow is an effective way to defeat modified or injected program jumps or function calls. To facilitate inspection, cores running service applications output a trace of fetched instructions (from the unified L2 cache to the instruction L1 cache) and control transfer traces including call/return instructions to a FIFO which is running the inspection software executing on the high privileged cores. The protected cores (low privileged) also send a trace of their memory updates to another FIFO. Once the high privileged cores receive the information, they maintain a log in their isolated memory space. Figure 2 illustrates such a scenario using our proposed CMP model.

When a server application starts for the first time, the server core sends necessary security information to the monitor core by writing to a shared buffer. This information includes execution attributes associated with the application's memory pages, symbol table, and import/export function list. The monitoring software stores the received information in its private and protected memory space and organizes it on a per-application basis. It uses the information to search for signs of intrusion when the server application is running. Introspection is performed concurrently with application execution. When the server application issues major I/O operations or raises a system call or exception, it stalls until the FIFO containing the information trace is drained by the monitor core. This synchronization between the application and monitoring software reduces the scope of the damage done by remote exploits before they are discovered.

Once an intrusion has been detected, certain damage has already been inflicted on the system. There are two specific types of damage. First, the system might be compromised by hackers who have installed a root shell to gain access to the system. Second, the system has been damaged by the intruders, e.g. corrupted memory data, run-away program flow, and so on. Synchronization between the server and the monitoring software prevents server compromise in the event that an unauthorized shell is executed. This is caught by the monitor because execution of a shell requires a system call. This is the first line of defense. Our system also uses two other techniques to repair a damaged or compromised server. The first one uses a checkpoint based recovery scheme to repair damages done by intrusions. The second one is a kernel space protection/recovery mechanism to prevent adversaries from installing rootkits.

To support introspection, applications have to communicate certain information such as function export/import table, addresses of original executable code pages, etc. to the monitor core. This can be achieved through shared memory. For each server application, OS on the protected core can write required information for security inspection in a restricted memory location accessible by the monitor core and the privileged OS before the server application is started. The design allows only high privileged OS code to send information to the monitor cores. Also certain information such as important static OS kernel data, once recorded by the monitor core, cannot be changed without a system reboot. Alternatively, certain inspection information can be downloaded to the monitor core through the administration network separate from the application services.

3.1.2 Checkpoint-based recovery

Our checkpoint-based recovery system has been designed and configured with network based service applications in mind. Upon receipt of a new network request by a server application, the server takes a snapshot of its process context and incremental memory state. The logging process captures the register states, program counter, open file descriptors, sockets, pipes, etc. The request is then handled by the application. If later, a buffer overflow or other type of intrusion is detected by the monitoring software, the monitor cores will interrupt the application's protected core(s). The application's memory state is rolled back to a state before the malicious request was handled. Its backed up process context is then compared with the current context. It releases system resources allocated after the checkpoint and prepares to restore the application to a known good state based on the backed up (or logged) process context. This approach attains intrusion tolerant computing and self-healing by nullifying the damage of network requests containing malicious contents. If the next request is from a legitimate and well behaved client, the application would continue as expected based on the communication protocol. It should be noted that the server does not undo all the socket reads or file descriptors when rolling back the malicious request. After rollback, the server application is ready to handle the next request.

It is important to note that if the server terminates the application and restarts it, the previous context, including file buffers with requests, from well-behaved clients will be lost. There is also no guarantee on the recovery time. Under repeated attacks, without the ability to quickly recover, legitimate users will suffer from service disruption.

In order to log memory states more efficiently, our system uses *copy-on-write* techniques. When the application issues a request for a checkpoint through a system call, the server will mark the memory pages of the entire application's address space including data, stack, and heap. When the application has resumed, it starts to modify data in its memory space, the attempt will trigger a memory fault, and will be handled by the server. At that time, the server will allocate a backup page and copy the data to it. It then changes the page attributes to writable and resumes application execution. Each checkpoint results in one set of logged information. To avoid running out of resources, the

Table 1: Rapid Recovery From Exploits on Service Applications

Security Function	Implementation
Insulation	physical memory insulation, memory state of security software on privileged cores are invisible to the protected system that is exposed to potential exploits and compromises
Introspection	function call/return address, code origin, call control flow matched with (import/export) function table [12]
Prevention	stall system calls and file operations until application codes executed are verified to be benign.
Recovery	maintain a log of memory updates or take checkpoints after service requests, iteratively roll back to to a known good state or checkpoint
Performance	concurrent verification and state inspection

server only supports a limited window of certain number of checkpoints, for instance five. When more are requested, information older than the original five will be overwritten.

With respect to process context and resources, our system maintains register context, the program counter, file descriptors, the process environment, and other allocated system resources. During recovery, resources allocated after the checkpoint are freed. Files opened after the checkpoint are closed. Files opened before the checkpoint will remain open after the rollback. All child processes (one might be malicious) spawned by the application after checkpointing will be killed and any new memory pages they allocated are reclaimed.

However, our system does not backup or restore all the possible memory states. States associated with inter-process communication, messages, and signals are not recovered. The system does not undo any changes to the files, or messages and signals already sent. If the application logs the malicious request to a file, the information will remain in the file. It is important to point out that our checkpoint based recovery approach is designed for network oriented server applications and remote exploits. It is not meant to be a general recovery solution for any type of application. Our study shows that the proposed scheme is sufficient to heal a majority of applications. In case the application cannot be properly recovered, the system will use the traditional approach, i.e. terminate the application and start over again.

3.2 Recovery from Kernel Level Rootkits

A kernel level rootkit modifies the OS kernel code and data in its memory space. This is often accomplished through malicious loadable kernel modules (LKM) residing in the kernel space or directly patched kernel text code through /dev/kmem accessible to user-space processes [23]. Malicious LKMs, if loaded, can modify the kernel's critical data such as the system call table and the interrupt descriptor table (IDT). A typical malicious act imposed by a kernel level rootkit is to redirect a system call to custom kernel code that hide illegitimate files and processes. It also creates a concealed communication channel (back-door). Most rootkits are capable of hiding themselves from system report utilities and removing currently loaded security modules.

Recently, there has been a great deal of research done in addressing the issue of kernel level rootkits [15, 21]. The authors in [21] categorize rootkits according to the way they infect a system's kernel space. A rootkit can 1) modify the interrupt handler for system calls (0x80) to use a different system call table (called table redirection), or 2) alter syscall entries to redirect individual calls to hacker supplied code (called entry redirection), or 3) patch or replace kernel code used for handling system calls with the hacker's malicious code. Note that enforcing write-protection on ktext and syscall table does not necessarily prevent rootkit installation because it can be subverted or bypassed.

Many tools and utilities have been developed to detect or prevent kernel rootkits including Chkrootkits [5], KernCheck [11], check-IDT [10], etc. Chkrootkits can detect rootkits using rootkit signatures. KernCheck compares infected machine's system call table with information defined in the system map to uncover entry redirection. CheckIDT is a tool to detect table redirection. In [21], the authors provide details on how administrators can detect and recover systems compromised by kernel level rootkits. They mention that virtual machines might be useful for automatic recovery from these rootkits. Later we will demonstrate that an asymmetric CMP is another platform for carrying out rapid recovery from kernel level rootkits.

3.2.1 Introspection and Recovery

To detect the presence of kernel level rootkits and recover from malicious changes to the kernel space, the monitor cores must maintain a clean copy of each protected core's kernel image. This includes a clean version of the system call table (system call target addresses), a clean map of the kernel symbols, and a clean copy of kernel text code. The monitor cores keep this information in their protected memory and storage space, which is insulated from the rest of the system. Monitoring software inspects memory updates on the server and tries to spot attempts to modify the system call table or ktext. It also checks the kernel text and syscall table fetched from the external memory to prevent the system from loading modified system binaries from outside. For example, hackers may modify the kernel image stored in a networked file system. Next time, when the system is rebooted, the malicious kernel will be loaded. This suggests that a simple solution of enforcing a non-writable kernel text/syscall table by the monitor core is insufficient for protection.

Once these kernel level rootkits are installed, the information provided by the server can no longer be trusted. Our scheme is capable of recovering from kernel level rootkits effectively because the monitor cores have direct access to each protected core's kernel space. Moreover, the monitor cores maintains a clean copy of the protected core's kernel data/text code for an immediate recovery of the modified syscall table and ktext. The monitor cores can be trusted to provide introspection and recovery services even when the server has been compromised.

Since the monitor cores maintain a clean copy of the kernel text code and syscall table, they can quickly repair the compromised protected server core's kernel when the kernel level rootkits are detected.

3.3 Limitation

The proposed hardware model and system architecture are capable of providing intrusion tolerance and service reliability better than previous approaches. However, our approach is not a cure-all for all network based exploits. The proposed scheme does not guarantee that all conceivable attacks are caught or that recovery is possible from every corrupted machine state. However, it does create a hardware and system architecture that allows for future or more advanced intrusion tolerant and autonomic techniques to be deployed. Note that the proposed protection and recovery scheme is end-system based. It does not address vulnerabilities or exploits that target the network itself. For example, it does not solve denial-of-service attacks that saturate the network. Also note that the security software in the monitor cores is insulated from remote exploits, it is subject to potential local physical tampering such as replacing or reflashing the memory that holds the monitoring and recovery software. Furthermore, the proposed scheme is not designed to replace the conventional process of fixing software bugs and vulnerabilities. It is an intrusion tolerant and recovery technique that prolongs service availability under remote attacks. Before a new patch is released, service providers can use our approach to recover from corrupted state and continue serving clients and customers.

Although rootkit use has been significantly impacted under our scheme, the rapid kernel recovery procedure does not completely remove all the rootkit files from the system. Since they can no longer hide from system auditing and inspection facilities, administrators can easily spot them and remove them from the system.

It is possible to design an independent and separate intrusion detection agent and recovery system for the persistent storage using our asymmetric CMP model. The system would provide similar services as described in [18].

4. EVALUATION AND ANALYSIS

We carried out our study with a whole system emulator. As such, we can evaluate our proposed hardware architecture, its supporting system components, and the service applications altogether. We first describe the implementation of our system and software followed by the security evaluation using real cyber exploits. We also discuss the performance impact for four server applications.

4.1 Implementation

4.1.1 Emulation environment

We used an in-order x86 whole system simulator derived from a whole system emulator called Bochs [9]. Bochs is an open source IA32 processor emulator capable of performing full system emulation. It provides modeling of the entire PC platform including hard drive, VGA monitor, network device, and other devices to support the execution of a complete operating system and its applications.

Our simulator emulates two processor cores concurrently with shared memory for synchronization and data communication. One core is configured to run a full-blown Linux OS (RedHat 6.0 distribution, kernel version 2.2) and network applications. The second core is designated as the monitor processor. It runs a simple runtime system based on a stripped-down tiny Linux stored in a flash memory. The monitor core that boots from the runtime system and the entire system including the security software is less than 10MB. We implemented our proposed hardware support to enable internal state inspection. This included automatic L2 cache to instruction L1 cache tracing, memory update inspection, processor state control. An example of the later includes the ability to stall and reset a protected processor core from its monitor counterpart.

The underlying machine that Bochs is running on, is a dual 2.4GHz Intel Xeon processor machine with 2GB RAM. The emulated platform and applications can respond to network requests in real time.

4.1.2 Monitor and recovery software

On the security programming side, we implemented a simple software security monitor. The program runs in the monitor core and receives logged instructions and memory updates from the other processor cores through designated I/O ports.

Upon receipt of a new instruction, the monitor first verifies the code's origin against the recorded code page attributes. When an application is created and started by the protected core, the system specifies page attributes, and the import/export call table, and posts them to the monitor core via a FIFO. Applications are distinguished according to information in CR3 control register. In the IA32 architecture [8], CR3 is used to store the physical address of a process's page table, which is unique for each process. Each instruction received by the monitor core is paired with its corresponding CR3 value so that the monitor core can decide which set of information should be used for security checks. The monitoring software then decides whether the instruction is a function call or the type of control transfer that requires further security checks based on the instruction's opcode. For control transfer, the program uses the recorded application's symbol table, function export/import lists to verify the legitimacy of the control transfer. The program also records a stack of expected function return addresses and verifies each function's return. The monitor core also manages a small buffer of memory logs (a windowed history of memory updates — several million most recent updates). The memory log is required when the monitor core detects errors, corruption, or compromise in the server's application and decides to rollback the application's memory state.

The Linux kernel is modified to support the rollback of applications. We implement a new system call for applications to capture checkpoints of their internal state. We used four open source network server applications to evaluate both security and performance. They are Wu-ftpd's file transfer protocol server software (*FTPD daemon*), Apache web server software (*HTTPD daemon*), email server software (*Sendmail daemon*), and Bind's domain name server software (*Named daemon*). We modified these applications so they could make use of the recovery mechanism supported by the hardware and system. Before the next client request is handled by these applications, it invokes the new system call to take a snapshot of its state. It then reads the next request and starts processing. If an intrusion is detected by the monitor cores, it will first verify the integrity of the sys-

tem's recovery routine (rotate detection and recovery) and interrupts the protected core. The core's OS invokes the recovery routine and rolls back the state of the application, re-schedules the application, and then resumes execution.

4.2 Security Evaluation

We validate our recovery approach using a set of real exploits against selected popular server applications.

4.2.1 Attack on server applications

Real exploits were applied to attack a few popular server applications: Apache, Bind, and FTP.[5] Attack codes or scripts are collected from various hacker websites and security agent websites.[6]

For Apache service, we used an exploit reported at (CAN-2003-0651) [1] that overflowed the buffer in the `mylo_log` logging function (`my_sql_real_escape_string`) for `mod_mylo 0.2.1` and earlier. This exploit allows remote attackers to execute arbitrary code with user privileges via a long HTTP GET request. This exploit actually only affects the logging process of the web server. This attack can be detected because it overwrites the return address and executes injected code. Applications can recover from this attack by rolling back the application's state to that of one before the exploit was executed.

BIND8 (domain name server) contains a buffer overflow attack associated with the processing of the request transaction signature (VU#196945) [2]. BIND uses the same stack buffer for storing the request and generating the response. It composes a response by appending a transaction signature to an existing request. However, during this process, BIND does not properly update the size of the packet buffer. This potential attack can also be detected by checking code origin and return address overwrites. Recovery from this attack can be done by going back to the previous checkpoint and processing the next request.

Wu-ftpd FTP server (wu-ftpd 2.5.0 through 2.6.2) contains a remotely exploitable off-by-one bug in the `fb_realpath()` function which allows attackers to execute arbitrary code (CAN-2003-0466) [1]. The overflow occurs when the length of a constructed path is equal to the MAXPATHLEN+1, one character larger than the size of the buffer. The bug stems from the misuse of the rootd variable in the calculation of the concatenated string length. Hackers can exploit this vulnerability by sending certain FTP commands to trigger buffer overflow. This attack can be detected when hackers try to gain root access. Recovery is possible by restoring the application's state back to the state prior to the FTP commands.

4.2.2 RootKits

We used real rootkits available from the Internet to examine the effectiveness of our design for detecting and recovering from rootkit. We infected the emulated Linux server with kernel rootkits and tested our detection and recovery scheme. We used five rootkits, *SucKIT*, *heroin*, *adore*, *itf*, and *knark*.

SucKIT [23] is loaded through /dev/kmem. It supplies hackers with a password protected remote back-door, and can hide processes, files and connections. Except *SuckIT*, all the rest rootkits use syscall table modification to subvert a system. Rootkit *heroin* hides files and processes. *Adore* can also hide files and processes. It provides a local back-door for hackers to execute a process with root privileges. Rootkit *itf* [20] installs a back-door and hides files and processes, and redirect commands. *Knark* is another rootkit that can hide itself and give root privilege to an intruder. All rootkits use kernel modification to hide themselves and cannot be removed by using system utilities such as rmmod (remove kernel module).

In our implementation, the monitoring system keeps a clean image of the syscall table, IDT entry, and ktext. The server sends a trace of memory updates to the monitor core for inspection. When any attempt to modify syscall, IDT entry or ktext is detected, the monitor core will stall the execution of its protected server core, drain the update trace FIFO and undo all the changes to the syscall table or ktext, then signal the protected core to resume execution. In our experiments, none of the rootkits succeeded in altering the application kernel.

In summary, our design prevents rootkits from modifying a known good system kernel and quickly patches the compromised kernel text and system table by restoring them back to a known good image.

4.3 Performance Analysis

We incorporated an accurate cache and SDRAM memory emulation into our performance emulator. Each processor core has its own L1 instruction and data cache. By default, each one is a direct-mapped 8KB cache with a 32B line. Each core has a four-way unified 512KB L2 cache. Latency of the L1 cache is one cycle and 8 cycles for the L2 cache. The emulated memory system runs at 200MHz and the emulated processor runs at 1GHz. The FIFO between the cores for storing trace information has 64 entries. The monitoring software on the monitor core fetches data from the FIFO through dedicated ports. When the FIFO is full and the monitored core has data to be pushed into the FIFO, it stalls and waits until a FIFO slot becomes available.

We wrote a set of scripts to automatically send network requests to the emulated platform and handle the responses. For DNS service, the script sends a sequence of queries to the server. For FTP, the script automatically logs into the emulated machine, downloads and uploads a few files. For Apache (HTTPD), wget is used to download a set of webpages from the emulated server recursively. For Sendmail, a shell script is used to continuously send a sequence of text mail files to the server. We implemented a packet log system similar to tcpdump inside the emulation software that can record receive and send time (in emulation time) for each request and response. This way, we can measure the time taken by each server application to respond to user's requests.

We evaluated the overhead introduced by our system for the selected applications, by measuring the time required to complete a set of service requests. The results are based on multiple runs of the test scripts (average of five runs). Figure 4 shows normalized execution time (normalized against un-instrumented server applications without security monitoring) when the server's OS does the copy-on-write to log the application's memory space. The figure indicates the slowdown for each application with our scheme. The result

[5] All selected server applications are executed as standalone daemons.

[6] We acquired these codes from www.k-otik.com/exploits/, www.insecure.org/sploits.html, and www.securiteam.com

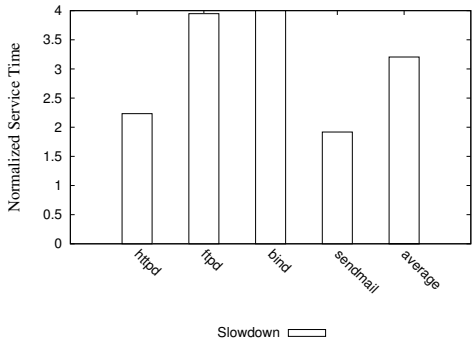

Figure 4: Normalized Service Time

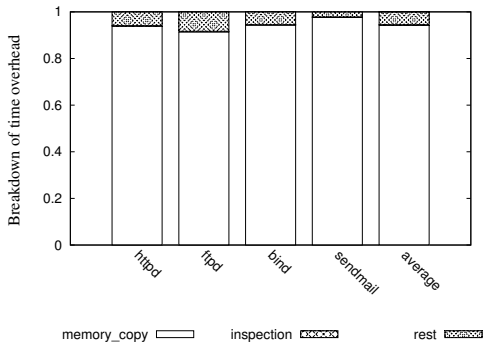

Figure 5: Breakdown of Latency Overhead

shows that Bind has the greatest overhead and that Sendmail has the least. HTTP and FTP are in the between. On average, the service times double and in some cases triple. To locate the main sources of slowdown, we broke the service time into pieces: 1) the time spent on copying an application's memory pages by the kernel (copy-on-write); 2) the delay introduced by inspecting traces; and 3) the time overhead (shown as "rest") including syscall for initiating backup and etc. Figure 5 shows the breakdown. It clearly points out that memory state logging of the application is the dominant performance overhead. Figure 6 displays the average interval between two consecutive checkpoint operations from the same server application.

Without using CMP, the memory state logging bottleneck is hard to remove. However, in our design, this overhead can be handled by using the monitor core to concurrently keep a memory update log for each server application between

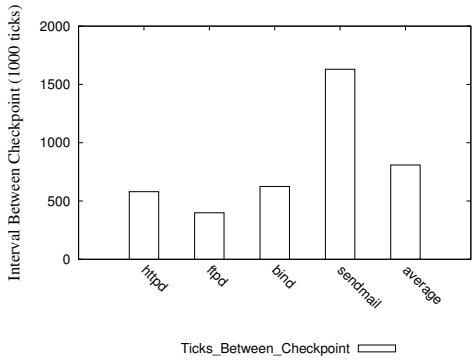

Figure 6: Average Interval Between Checkpoint

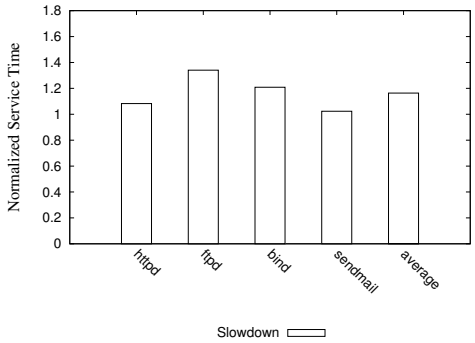

Figure 7: Normalized Service Time

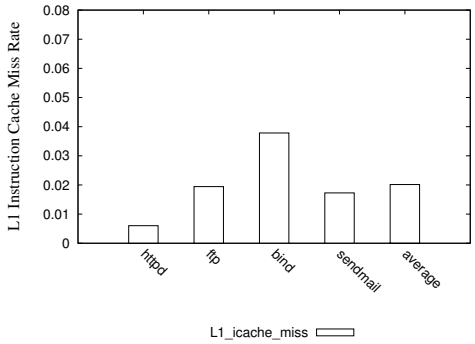

Figure 8: L1 Instruction Cache Miss Rate

checkpoint requests. The log only keeps the original values of modified memory locations instead of copying the entire page (copy-on-write duplicates the whole page even though only one byte is changed). Figure 7 shows the results of the new service times after the memory state log is captured by the monitoring software. According to the results, the overhead significantly decreases. The monitoring software uses an efficient way to store memory logs. It aggregates memory update information inside its processor core. The non-temporal memory updates are written to memory.

One indicator of the workload on the monitoring software is the L1 cache instruction cache miss rate, because instructions fetched from the unified L2 cache are subject to code origin inspection. Figure 8 shows the L1 miss rate.

5. RELATED WORK

There are prior studies on the subject of detecting remote attacks [17, 12], prevention, rootkits discovery and removal [21]. The purpose of our study is to demonstrate the possibility of including security related features in an asymmetric CMP which will provide intrusion tolerance and self-healing computing. Since we are not providing another new detection technique, our analysis of related work will mainly focus on previous research involving recovery and service reliability.

5.1 Self-healing and Intrusion Tolerant Systems

In this paper, we categorize intrusion tolerant end systems into three area, single processor and single system based, single processor and multi-system based, and CMP multi-system based. Virtual machine based systems can be viewed as a single processor, multi-system based where one proces-

sor is used to run both a host and at least one client system. For this kind of system, security measures such as inspection and repair can be implemented as services within the emulation software on the host system [7]. Single processor and single system based solutions do not use a full-fledged virtual machine. Instead, they incorporate inspection and recovery codes into the application or system itself [27]. Those approaches often require availability of the system or the application source codes so that the applications can be instrumented and re-compiled.

5.2 Virtual Machine Based Security Model

Virtual machine emulation of a full system including hardware architecture has been used for many security techniques including *honeypots* [24], remote attack behavior analysis, and attack replay [6]. In [6], a virtual machine monitor called ReVirt is proposed to log detailed execution information and later used as a forensic tool to replay and identify steps involved in an intrusion. In [7], a virtual machine monitor based intrusion detection system (IDS) is proposed to isolate IDS from the monitored host. Researchers in [21] used virtual machine to recover systems infected with rootkits. What we propose in this paper differs from virtual machine in many ways. First, our CMP based system runs production services in native mode without using any emulation. Therefore, it has less overhead than VM-based approaches. Second, our system uses secure insulation to isolate security software on the monitor cores and the application systems. It is more secure and more reliable than virtual machines at detecting and recovering from remote and local attacks. It is important to note that virtual machine emulators and the client system together can be viewed as one networked application. The application as a whole, just like other service applications, is subject to remote exploits. Third, our system does not require frequent switching between native execution mode and virtual emulation mode. Fourth, our approach is capable of doing fine-grained inspection and micro-level recovery which a VM-based approach cannot accomplish unless every instruction and memory update is emulated, which is very costly.

5.3 Single Processor, Single System Based

In [26, 27], several techniques are proposed to automatically evolve source codes and patch faulty software applications in the face of attacks. In [27], an on-line recovery technique derived from [26] using source code instrumentation and selective emulation is described. Our approach has several advantages over the technique in [27]. First, [27] uses instruction emulation which could undermine server performance. Second, [27] uses selective emulation. As a consequence, the protection and recovery are only applied to the selected functions. This reduces both its security and ability to recover cleanly. Third, single processor, single system based approaches do not provide an exploit insulated component for reliable security services. This limits the reliability of the security service provided by this kind of system. This, in turn, limits its capability of self-healing and being intrusion tolerant. Fourth, [27] mainly focuses on applications while our system provides a multi-line and multi-point defenses against remote exploits.

6. CONCLUSION

This paper presents an intrusion tolerant and self-recovery architecture that uses a unique security enhanced asymmetric CMP model in order to provide highly reliable network services and systems. In contrast to previous single-processor, single-system and single-processor, multiple-system (VM) based approaches, our approach uses an asymmetric security enhanced CMP based multi-system to achieve self-healing and intrusion-tolerant computing. The new model provides three main advantages which ensures self-healing and intrusion tolerance: (1) the monitor cores are insulated from the rest of the system to ensure resistance from remote exploits and attacks; (2) the close coupling between the processor cores of a CMP system enables rapid logging, fine-grained inspection and fast recovery of a corrupted or compromised system; and (3) the concurrent and fine-grained inspection, logging and recovery techniques that are off of the critical path. Our system provides a high level of service reliability and recoverability by deploying inspection and recovery agents to several vulnerable points in a system. It uses a checkpoint based approach to recover server applications and quickly repairs the kernel in an event of a rootkit installation.

7. REFERENCES

[1] Common vulnerabilities and exposures. *http://cve.mitre.org/*.

[2] US-CERT Vulnerability Notes. http://www.kb.cert.org/vuls.

[3] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a Scalable Architecture based on Single-chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 282–293. ACM Press, 2000.

[5] Chkrootkit. http://www.chkrootkit.org/.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of the 2002 Symposium on Operating Systems Design and Implementation*, Dec 2002.

[7] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection based Architecture for Intrusion Detection. In *Proc. of the Internet Society's 2003 Symposium on Network and Distributed System Security*, February 2003.

[8] Intel. *Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2002.

[9] K. Lawton. Welcome to the Bochs x86 PC Emulation Software Home Page. http://www.bochs.com.

[10] Kad. Handling Interrupt Descriptor Table for Fun and Profit. *Phrack*, 11(59), 2002.

[11] Kerncheck. http://la-samhna.de/library/kern_check.c.

[12] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proc. of the 11th Usenix Security Symposium*, Aug 2002.

[13] K. Krewell. Sun's Niagara Pours on the Cores. *Microprocessor Report*, September 13 2004.

[14] S. Labs. Loadable Kernel Module Rootkits. *http://la-samha.de/library/lkm.html*, July, 2002.

[15] J. G. Levine, J. B. Grizzard, and H. L. Owen. A Methodology to Detect and Characterize Kernel Level Rootkit Exploits Involving Redirection of the System Call Table. In *Proc. of the 2nd IEEE International Information Assurance Workshop*, pages 107–128, 2004.

[16] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proc. of the 7th International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 2–11. ACM Press, 1996.

[17] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks, 31(23-24)*, pages pp. 2435–2463, 14 Dec. 1999.

[18] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. Soules, G. R. Goodson, and G. R. Ganger. Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior. In *Proc. of the 12th USENIX Security Symposium*, August 2003.

[19] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. In *IEEE Security and Privacy, 2(4)*, pages 20–27, 2004.

[20] Plaguez. Weakening the Linux Kernel. *Phrack*, 8(52), 1998.

[21] P. Samarati, D. Gollmann, and R. Molva, editors. *Computer Security - ESORICS 2004, 9th European Symposium on Research Computer Security, Sophia Antipolis, France, September 13-15, 2004, Proceedings*, volume 3193 of *Lecture Notes in Computer Science*. Springer, 2004.

[22] Scut. Exploiting format string vulnerabilities. 2001.

[23] Sd and Devik. Linux On-the-fly Kernel Patching without LKM. *Phrack*, 11(58), 2002.

[24] K. Seifried. Honeypotting with VMware: Basics. *http://www.seifried.org/security/ids/20020107-honeypot-wmware-basics/html*.

[25] A. Shilov. Amd targets quad-core microprocessors for 2007. http://www.xbitlabs.com/news/cpu/display/20040813040951.html.

[26] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the 12th International Workshop on Enabling Technologies*, page 220. IEEE Computer Society, 2003.

[27] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services . Report CUCS-038-04, Columbia University, Newyork, NY, 2004.

[28] J. M. Tendler, J. S. Dodson, J. S. Fields, H. L. Jr., and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.