An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors

Weidong Shi
shiw@cc.gatech.edu

Hsien-Hsin S. Lee
leehs@gatech.edu

Laura Falk[†]
laura@eecs.umich.edu

Mrinmoy Ghosh
mrinmoy@ece.gatech.edu

School of Electrical and Computer Engineering
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

[†]Department of Electrical Engineering
and Computer Science
University of Michigan
Ann Arbor, MI 48109

ABSTRACT

This paper presents a high-availability system architecture called INDRA — an INtegrated framework for Dependable and Revivable Architecture that enhances a multicore processor (or CMP) with novel security and fault recovery mechanisms. INDRA represents the first effort to create remote attack immune, self-healing network services using the emerging multicore processors. By exploring the property of a tightly-coupled multicore system, INDRA pioneers several concepts. It creates a hardware insulation, establishes fine-grained fault monitoring, exploits monitoring/backup concurrency, and facilitates fast recovery services with minimal performance impact. In addition, INDRA's fault/exploit monitoring is implemented in software rather than in hardware logic, thereby providing better flexibility and upgradability. To provide efficient service recovery and thus improve service availability, we propose a novel delta state backup and recovery on-demand mechanism in INDRA that substantially outperforms conventional checkpointing schemes. We demonstrate and evaluate INDRA's capability and performance using real network services and a cycle-level architecture simulator. As indicated by our performance results, INDRA is highly effective in establishing a more dependable system with high service availability using emerging multicore processors.

1. INTRODUCTION

Despite a considerable number of software [20, 12, 11] and hardware techniques [13, 31] proposed for detecting remote software exploit attacks, little attention was paid to the importance of software and service recovery. The continuation of disrupted services on compromised systems is typically hard, if not totally impossible. Conventional recovery practices remain not only cumbersome and error-prone but also costly in terms of administrative and financial resources. Sometimes, the disrupted services cannot be recovered for days or even weeks after the attacks.

Recently, new research has been conducted, such as buffer overrun aware compilers, automatic self-recovery, and automatic software patch generation [27, 28] for addressing the issue of service availability in the face of a remote exploit attack. In a broader sense, loss of network services could be caused by many reasons. Table 1 classifies the possible threats to network services and their previously proposed recovery schemes. The focus of this research is in the recovery of network services caused by malicious remote exploit attacks. During such attacks, the hacker sends a huge volume of network packets targeted at a specific network service or a network server. These malicious packets can corrupt the network service or cause the service to terminate thereafter. When compared with the effect of a random fault, a remote exploit attack is deterministic in nature. If a system is vulnerable, then it is possible that some or all of the packets sent during an attack are sure to compromise the server and its services. An unpatched and vulnerable machine will invite frequent and recurring attacks. Therefore, special handling is required when recovering from a service loss due to these remote attacks. Currently employed solutions such as replication of services or redundant execution, may not work for service recovery from remote exploits. When compared with other service failures, remote exploit attacks are a far more serious threat to service availability. In some cases, service providers are blackmailed by the attackers to either satisfy to their demands or suffer a denial-of-service (DoS) attack [22].

In this paper, we propose a system architecture called *INDRA* — an INtegrated framework for Dependable and Revivable Architectures.[1] INDRA provides necessary hardware support for constructing an efficient self-healing network service infrastructure in the event of a remote exploit attack. INDRA also introduces a new programming model to support better security, reliability, and availability for the emerging multicore processors or chip multiprocessors (CMP). It exploits the characteristics of a multicore processor in order to provide secure and non-disruptive network services. The main characteristics and advantages of our system are:

- **Consolidated security and revivability.** *INDRA* achieves better security and revivability by leveraging a multicore platform and designing the system around one or more protected processing components (called *resurrector* cores) that are insulated from remote attacks. This is done by imposing an asymmetric configuration to the different cores on a multicore processor. One or more cores are configured to run at higher privilege levels and assigned the role of monitoring the rest of the processor cores (or *resurrectee* cores). The resurrector core executes a runtime software module that monitors

[1]INDRA is a Hindu god who can revive dead warriors on the battlefield.

Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)
0-7695-2608-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Solutions		Causes of Network Service Loss					
		Accidental			Aging	Intentional	
		Transient	Heisenbugs	Permanent Damage		DoS	Buffer Overflow
Replication	Software-based [7]	√	√	√			
	Hardware-based [6, 15, 25]	√	√	√			
Rejuvenation [17]					√		
Checkpoint	App Level [10]	√	√	√			
	Hardware-based [29, 24]	√	√	√			
Remote exploit self-recovery [27, 28]						√	√

Table 1: Taxonomy of Network Service Loss

the services running on the resurrectee cores and detects traces of corruption. It ensures revivability in the face of remote exploit attacks and loss of service.

- **High efficiency monitoring, backup and recovery.** INDRA executes network services on real processors in native mode and uses novel architectural features for concurrent state monitoring and efficient state backup. When any resurrectee is either compromised or suffer a service failure as a result of a DoS attack, the resurrector will trigger the resurrectee to swiftly terminate the faulty service request, recover its corrupted state on-demand, and safely revoke the damage done by the remote exploits.

The rest of the paper is organized as follows. In Section 2, we briefly discuss service revivability issues caused by remote exploit attacks and network service loss due to these attacks. Section 3 presents the *INDRA* architecture. Section 4 shows security and performance evaluation, followed by related work in Section 5. Finally, Section 6 concludes the paper.

2. REMOTE ATTACK INSULATION AND SERVICE REVIVABILITY

The objective of this research is to create an autonomic system that supports revivable network services that are immune from remote exploit attacks. *Revivability* guarantees a non-disruptive service and swift recovery from erroneous states caused by remote exploit attacks. We achieve this goal by providing three key features in INDRA: 1) the ability to implement a component which is insulated from remote exploits. This component is the nucleus over which other security services such as faulty state monitoring and service recovery can be reliably deployed; 2) the ability to detect erroneous and corrupted states during software execution; 3) the ability to automatically recover compromised services with minimal performance impact.

2.1 Threat and Fault Model

Buffer overflow attacks remain the most popular exploited vulnerability [5]. Typical buffer overflows include stack smashing, which causes control transfer to maliciously injected code [5], overwriting of heaps or function pointer tables, and format string attacks [26]. The ability to exploit a buffer overflow allows an attacker to possibly inject arbitrary code into an execution path. If executed, the injected malicious code could give the attacker unauthorized access or allow malicious replication of code (e.g., worms). From a service availability perspective, a buffer overflow may corrupt the internal state of a service application and cause service failure. Moreover, even if the hacker did not gain root privileges through a buffer overflow, the attacks could still bring down the system due to the corruption of the application's memory space.

Furthermore, network services such as DNS and Email are also vulnerable and sensitive to DoS attacks. For example, in Windows NT, it is possible to bring down the entire system by sending remote out of band data to an established Windows connection [1]. Another example is the teardrop attack and its variants also targeted for NT [1]. These attacks can cause NT to freeze and eventually display the "blue screen of death" soon after the attack.

2.2 Intrusion Revivable and Instant Recoverable Multicore System

Remote exploit attack insulation and service revivability are two separate concepts. A remote exploit attack proof component within the system is a necessity for creating truly revivable network services. Such a component serves as the last line of defense for service recovery. Without it, the whole system could be compromised or suffers a service failure in which self-recovery would be impossible. However, the existence of such a component itself is not sufficient for creating revivable services unless there is some additional mechanism for state introspection, backup, and recovery.

Revivable computing prolongs service availability by allowing vulnerable software or compromised systems to continue execution but doing so in an insulated environment where techniques of self-healing and fully autonomic recovery are applied. In the face of remote attacks, instead of terminating the vulnerable, corrupted application or system, an intrusion revivable system tries to repair damages instantly and restores the system/application in real-time to a normal safe flow. The damages include memory corruption, destruction of critical data structures resulting from buffer overflows, and system data corruption. The repair is based on timely backup of important machine states. It serves as a temporary solution before a complete solution such as a new patch is available. Recurring exploits may continue "infecting" or "wounding" the system. However, the system, ideally, can quickly recover from the "wounds" and continues to serve legitimate and well-behaved clients.

It is important to note that alternative solutions, such as restarting the compromised service or using replicated servers do not guarantee service availability under remote exploit attacks. The reason behind this is that the adversaries can repeatedly launch attacks (e.g. DoS), causing services to fail. After such a service failure, the data contained in the temporary storage areas such as file buffers with requests from well-behaved clients would be lost. After a reboot, legitimate clients may still suffer from denial of service because the servers might not be able to handle their traffic due to repeated attack attempts. INDRA solves this problem through a swift micro-level recovery. It has two distinctive features. First, it tries to repair damages caused by malicious requests in real time. Second, it tries to process every received service request. In this paper, we propose

IEEE COMPUTER SOCIETY

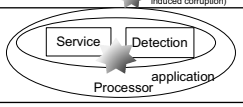
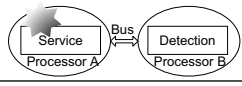
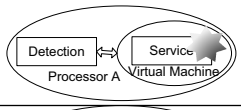
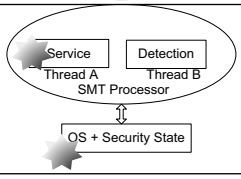
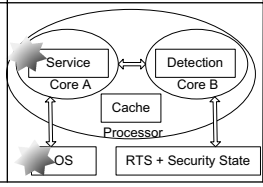
	Explanation (stand for remote exploit induced corruption)	Exploit Isolation /Immunity	Code origin/Control Flow Inspect	Performance Overhead
App (Instrument)		Almost no protection. App along security module can be crashed together.	Only for instrumented parts	Depends. App has to maintain memory log and conduct inspection
SMP		Can be supported if 1) resources are partitioned; 2) A and B run different OS.	Have to send internal cache and branch status across shared bus between A and B.	Constrained by bus throughput between A and B
Emulation		Supported. Exploit cannot cross virtual machine barriar.	Require frequent stall of virtual machine execution and mode switching for detailed inspection.	Slow
SMT		Difficult. Thread A and B share the same OS. Exploit can corrupt OS and security state.	Possible. Thread B has to access execution status of thread A.	Have performance overhead if A and B share pipeline resources,
Multicore		Good exploit isolation. Exploit of A has to cross OS and memory barrier,	Can be supported if core B has access to core A's execution status.	1)Concurrent inspection; 2) Minimal resource conflict; 3)On-chip bus throughput higher than off-chip bus.

Figure 1: Comparison of Different Design Paradigms

building such a highly dependable and revivable system using the emerging multicore processors.

2.3 Why Multicore Processors?

All the current multicore (or CMP) designs are configured symmetrically from the perspective of security. We are proposing a new multicore paradigm called asymmetric configuration. Our asymmetric multicore configuration would support self-healing and intrusion tolerant computing by imposing a security structure onto the multiple cores. Figure 1 compares several different design options for constructing self-recoverable services to counter remote exploit attacks. The figure compares asymmetric multicore, SMT, virtual machine, SMP, and application instrumentation in three key metrics. They are: 1) remote exploit attack immunity; 2) detectability; and 3) potential performance overhead. Among all the alternatives, the multicore, virtual machine, and SMP provide a means for the monitor's state to remain isolated from the application's state, thus may achieve better remote exploit immunity than the SMT or the application instrumentation based design. In terms of detectability, multicore is more flexible because the monitoring and the monitored cores are on the same die. It is possible to inspect internal execution status in finer granularity when using a multicore compared to an SMP. In terms of performance overhead, multicore enables concurrent state introspection and service execution, therefore causes less resource conflicts than designs that demand more processing resource sharing, such as virtual machine or SMT. SMP design is constrained by throughput of the external bus between processors while multicore can deliver higher inter-core throughput.

2.3.1 Multi-level Insulation

The processor cores on a multicore die can be configured with different security privileges. A security enhanced multicore BIOS may have a configuration setting that allows a local user to configure security privileges for different cores. The high privileged ones are granted access to all the hardware resources including the entire memory space and I/O devices and all the DMA engines. The low privileged cores can only access memory assigned to them with constrained physical memory space and limited access to the peripherals.

This asymmetric multicore configuration creates a *hardware sandbox* for a system running generic services on the low privileged cores and a system running security services on the high privileged cores. The two systems are not only physically insulated but also running different OSes. Such strong insulation provides high assurance for service recoverability and availability.

2.3.2 Fine-grained Internal State Logging

High speed sharing of internal state information between processors are made possible via a multicore that enables the privileged cores to examine internal information from the low privileged cores for security inspection. The information obtained from monitoring the low privileged cores include fetched instructions from the unified L2 cache to L1 exclusive I-cache,[2] control flow status, and other information necessary for detecting system corruption [20]. This capability can be easily implemented in a multicore because it only gathers information at the interface and requires no internal processor pipeline change. The gathered information is sent to the high privileged cores via a hardware FIFO. The high privileged core pulls out the information and inspects for suspicious behavior. A virtual machine monitor [14] is another technique for internal state inspection. When compared with a virtual machine monitor, a multicore has the following advantages. First, virtual machines comprise native execution and emulation. State inspection is only conducted when execution switches from the native mode to the emulation mode. The execution of native code is not monitored. This means that it has less inspection coverage. Second, emulation in a virtual machine is slow. INDRA executes all software in the native mode and achieves better performance than the virtual machine emulation.

2.3.3 Tight Processor Core Coupling and Control

The tightly coupled processor cores in a multicore system facilitates an easy implementation of controlling mechanisms under which the privileged cores can send signals to stall the corrupted low privileged cores, cause their pipelines to flush, trigger the state recovery of the compromised services in a timely manner, and resume their execution from a known good point.

2.3.4 Reconfigurability

An asymmetric multicore model can be configured to run in symmetric mode where each processor core has equal privilege. All the cores can be booted from the same operating system when security protection is not needed. Mode configuration can be done in the BIOS, thereby making such a system highly flexible under different given security requirements.

[2]Hardware ensures that codes in the L1 instruction (IL1) cache cannot be modified. This means that the L2 to L1 interface is the natural point for monitoring injected code attacks.

IEEE COMPUTER SOCIETY

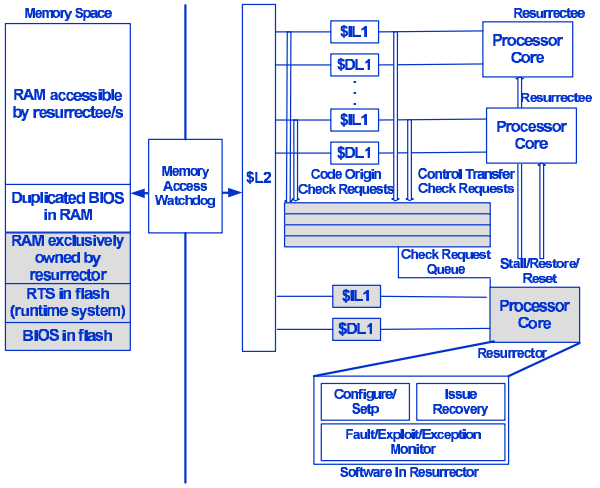


Figure 2: INDRA architecture

3. INDRA ARCHITECTURE

In this section, we detail INDRA and show how it is used to provide high-availability services. Figure 2 depicts a block diagram of the INDRA architecture. As illustrated, the architecture contains one core called *resurrector*[3] configured as our high privileged core and several cores called *resurrectees* with lower privileges. Service applications are deployed on the resurrectees. The resurrector runs special software that is responsible for monitoring the system for state corruption. The memory subsystem for INDRA is also partitioned based on privilege levels for the resurrector and resurrectee cores. The resurrectees may only access the un-shaded part of the system in the figure. They cannot change or read the physical memory space of the resurrector. This configuration is used to maintain proper insulation. The resurrector, however, can read and write the entire address space.

3.1 Asymmetric Multicore and Insulation

INDRA enhances a multicore with security features to enable revivable services. The key idea is to create a security hierarchy among different processing cores by assigning different privilege levels and tasks to different cores. For clarity, we limit our discussion to a multicore configured with one single resurrector core and multiple resurrectee cores. The design principle, however, can be extended easily to a multicore with more than one resurrector. Now we present how INDRA enhances insulation for security services in order to prevent remote exploits.

3.1.1 Remote exploit insulation

Specific design features are proposed in INDRA in order to enable remote attack insulation of the resurrector. These features isolate the resurrector and the resurrectees in such a manner so that the resurrector is invisible and transparent from the resurrectees. Corrupted memory space or a bad state in the resurrectees are self-contained which makes it impossible for them to affect the state of the resurrector. This is guaranteed by the following specific design features.

- *Dual or multiple-systems*: The resurrector and the resurrectees run different operating systems. Corruption or compromise of the system executed by the resurrectees have no effect on the system of the resurrector. The two systems are isolated from each other. In addition, the two systems may be even two different operating systems. For example one is Windows and the other is Linux.
- *Memory space isolation*: The resurrector has access to all the physical memory space while the resurrectees can only access the physical memory space allocated to them by the resurrector. The limited access privilege to the physical memory space is enforced by the INDRA hardware memory watchdog.
- *Network isolation*: The resurrector does not host network services. The resurrector is basically quarantined from the remote exploits.

[3]Only one resurrector core is shown in the figure for clarity. Having more resurrector cores is possible.

Table 2: Remote Exploit Inspection

Inspection	Stack Smash	Injected Codes	Function Pointer / Virtual Function
Function Call / Return	√		
Code Origin		√	
Control Transfer Inspection			√

3.1.2 Boot sequence

The booting procedure of INDRA is also unique. When an INDRA multicore system is configured to run in the asymmetric mode with one or more processing elements as the resurrectors, one resurrector will be selected as the bootstrap processor and boot first from the regular BIOS and run the light-weight RTS stored in the flash memory. After the resurrector core is successfully booted, it sets security parameters and assigns access rights to the physical memory space to the resurrectee cores. The resurrector will hide the original BIOS and the RTS from the resurrectees and exclusively allocate space for itself. Each memory access issued by either the resurrector or a resurrectee core is tagged with the core's ID. A simple hardware check is implemented to guarantee that the resurrectees are prohibited from accessing space belonging to the resurrector core. The resurrector duplicates a version of the BIOS in the space accessible by the resurrectees. Then it informs the resurrectees so that they can start to boot from the full operating system.

3.2 Monitoring and Introspection

One of the primary tasks of the resurrector is monitoring. It not only detects corruption caused by remote exploit attacks, but also checks for the well-being of the resurrectees. All the monitoring services are implemented in software, hence it is straightforward to configure the monitoring service based on the security requirements and policies. When any error, misbehavior, or corruption is detected, the resurrector will stall the affected resurrectee core and signal the recovery process.

To monitor the resurrectees, necessary information about them must be provided to the resurrector. This is achieved by the resurrectee core hardware (alternatively, traces or necessary information can also be provided by instrumented software on the resurrectee cores). Different exploit attacks can be detected using different types of trace or log information.[4] Table 2 summaries the exploits and how they can be detected. The *resurrector* receives trace information from

[4]How to use trace information for detecting remote attacks were well studied and is outside the scope of this paper.

0-7695-2608-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

either a designated port or from specially mapped I/O registers. For example, The *resurrectee* sends committed instruction information traces to the *resurrector* through a FIFO. The *resurrector* fetches data from the FIFO through special input registers. The following subsections discuss state inspection in details.

3.2.1 Function Call/Return

In one approach, the resurrectee can spit a trace of function calls (target address, return address, and stack pointer) and function returns to the resurrector. The resurrectee directs the function call/return trace into the shared FIFO. The resurrector retrieves the information and verify that for every function call, execution always returns to the correct next instruction after the function call is returned. This is sufficient for detection of remote exploit attacks that overwrite the function return address, which accounts for the majority of buffer overflow based attacks. The resurrector's monitor ensures that each function always returns to the next instruction following the call. Some special cases include setjmp or longjmp. Setjmp or longjmp targets need to be verified against the list of valid control transfer targets, thus it is also considered as part of the introspection mechanism. The env by setjmp will restore the register state, thus the call/return monitoring and introspection will be resumed after the longjmp completes and program resumes from the instruction after setjmp.

3.2.2 Code Origin Inspection

Inspecting the origin of the executing code is a simple, effective way to prevent the majority of code injection attacks. Using a patched kernel based on IA32, non-executable restriction can be applied to both stack pages [2] and data pages [3]. In [20], a dynamic software rewriting kernel called RIO is used to inspect the code origin during code transformation. Implementing code origin inspection in INDRA is straightforward. Either the application or the OS process manager can inform the resurrector regarding the execution privileges associated with the application's memory pages once the application's binary is loaded from disk. During runtime, the resurrector must ensure that only the instructions loaded from the original memory pages, along with its assigned execution privileges, can be moved to the instruction L1 cache. Inspection is performed only once before the instructions are moved to the IL1 and they cannot be altered in any way through software exploits. In INDRA, if the code does not come from the original memory page with the designated execution privilege, loading it to the IL1 will trigger a fault.

INDRA also supports dynamically modifiable code and self-modified code in a manner similar to [20]. The code must be explicitly declared and given a reserved memory space. The *resurrectee* provides the address space of any self-modifying code to the *resurrector*. Execution of dynamic code is restricted to its own memory space and enforced by the *resurrector*. Since the information is kept in the resurrector and hidden from the resurrectees, it is impossible for remote attacks to forge security attributes of an arbitrary memory page. As shown in [20], inspecting code origin can prevent most of the buffer overflow attacks.

The workload of code origin verification will be proportional to the number of IL1 misses. To further reduce the workload of code origin monitoring, INDRA uses a simple filtering mechanism which maintains a small set of recently encountered fetched code page addresses in a content addressable memory (CAM). When a line is fetched, the resurrectee core will look up the block's page address in the CAM, if there is no match, the resurrectee will send the page address to the monitor for inspection. Our research shows that a CAM of only 32 page addresses can effectively filter out more than 90% code origin checks.

Note that the proposed code origin verification approach provides better protection than the simple hardware based solution using execution flag for memory pages because the execution flag does not prevent tampering of the execution flag, thus a non-executable page can become executable [20].

3.2.3 Control Transfer Inspection

INDRA allows arbitrary security policies on program control transfer. As discussed in [20], many of the deployed exploits on program control transfer can be stopped by enforcing a restricted control transfer policy. For example, function export and import lists created by the compiler can be used to verify the validity of each cross segment function call. Indirect function calls to shared libraries can be checked to ensure that they always invoke library functions through defined entry points. A piece of software on the resurrector can inspect executed control transfer instructions and verifies the target and source address against application's symbol table and the shared library's export/import list. Such control transfer information can be provided by the resurrectee to the resurrector when a service program is started. Under the assumption that code pages are protected with read-only restriction, only computed control transfer and indirect calls require vigorous target address inspection.

3.2.4 False Positive vs. False Negative

It is important to point out that INDRA's security inspection is behavior based. Unlike some intrusion detection systems that use circumstantial evidence or packet signatures, INDRA rarely has false positives. When one of the three aforementioned inspection conditions is satisfied, something is certainly wrong. For instance, if a function does not return to its caller, there is certainly an error. According to the code origin inspection, if information stored in a stack or data page is executed, one can conclude that functionality is incorrect. If, during control flow inspection, execution jumps to an instruction that was not originally defined as a jump target by the compiler, one can conclude that the functionality is incorrect.

However, the above three inspection schemes are not sufficient to cover all possible state corruptions and consequences of remote attacks. Therefore, INDRA could potentially generate false negatives, meaning some corruption triggered by remote exploit attacks may not be detected. This issue may be fixed at the software level as the inspection module itself is based on software. How to detect new exploits is a topic for security computing and requires future research.

3.2.5 Synchronization

The relative speed between regular software execution in the resurrectees and the speed of monitoring in the resurrector is a critical determinant in the overall performance of INDRA. In many cases, tens or even hundreds of instructions on the resurrector side need to be executed for every instruction from the resurrectees that requires verification. However, the real gap between these two is far less than the disparity shown by counting the number of instructions because the resurrectee cores may stall and wait for cache misses or I/O inputs to continue, which provides needed slack for the resurrector. Furthermore, the resurrector does not check every retired or executed instruction. For example, to verify code origin, only code missing IL1 re-

IEEE COMPUTER SOCIETY

quires checking. In certain situations, the resurrectees and the resurrector have to synchronize with each other. First, when the resurrectees are writing to the I/O memory or DMA-writing to peripheral devices such as the hard-drive or network interface, the resurrectees will stall their I/O operations until all the previous instructions are verified by the resurrector. Second, when a resurrectee issues a system call, software interrupt, or context switch, it will stall until all the previous instructions are properly verified and checked. Third, when the shared FIFO queue is full, the resurrectees who want to add more check requests to the queue will stall until some space in the queue is released. According to our profile analysis, a buffer of a few KB is sufficient to eliminate the majority of stalls caused by buffer synchronization.

3.3 State Backup and Recovery

High efficiency memory state backup is crucial for instant service recovery. INDRA's state backup and recovery are based on the observation that the majority of network service applications are driven by network requests. The idea itself is simple. Upon the receipt of a new network service request by a server application, the server will issue a request to the hardware. The hardware will take a virtual state snapshot to allow rollback to this state later if an exploit or memory corruption is detected. INDRA's backup and recovery mechanism handles three types of states, application's execution state (register context and program counter), memory state, and system resource allocation state.

3.3.1 Memory State Backup and Recovery

There are many existing software [27, 28] and hardware techniques [30] for memory state backup. INDRA uses a novel and efficient *delta page based approach* for high speed memory state backup and instant rollback. The cost of state backup is amortized over regular software execution. The technique assigns a physical backup page in memory to each virtual page requiring backup. Only the cache lines that are modified are stored in the backup page.

Table 3 compares several macro-level memory backup schemes. Note that INDRA cannot take advantage of some of the micro level checkpointing schemes that keep checkpointing state or incremental memory updates in on-chip cache or buffer [18, 24, 29]. Most of these techniques were designed to support state recovery for branch misprediction, precise interrupt, or transient faults that often have a short time window between checkpointing and recovery. This does not apply to corruption induced by remote exploits such as buffer overflow. It often takes hundreds of thousands to even million of instructions for a server application to process a network service request. In this large time window, there could be tens of thousands to hundreds of thousands of memory updates or even context switches. It is impractical to hold all backup changes or memory logs on-chip.

Note that using hardware to incrementally store memory update into a log stack [18, 33] is fast in terms of backup speed but slow at recovery because the hardware has to undo the changes sequentially for each record in the log. Another technique of replacing an active page with an entire backup page in TLB facilitates fast recovery but with very slow backup because it has to back up the entire active page regardlessly. Our profiling study shows that for each network request, a server application may modify many memory pages but for each modified page, only a small number of lines are modified. The traditional virtual checkpointing technique of copying an entire dirty memory page incurs too much overhead. INDRA's memory backup is designed to address this problem by storing only dirty lines on-demand.

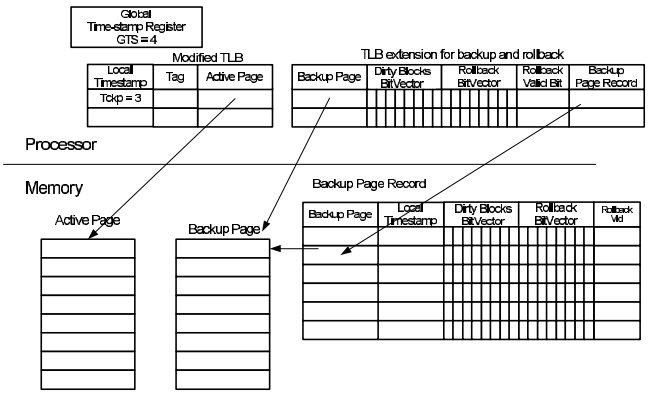


Figure 3: Backup Page Record

Global and Local Checkpointing Timestamp. Virtual memory checkpointing employs a global memory checkpoint counter, called Global TimeStamp (GTS).[5] Each time a server application receives a new network request, the server application issues a system call that increments the GTS register. For each virtual memory page, there is a Local memory checkpoint TimeStamp (LTS). When the *resurrectee* core writes a memory page, if the GTS is greater than the page's LTS, it implies that the dirty line needs backup. If a backup page is not assigned yet, the hardware will allocate a new backup page. A backup page stores the original values of all the first time modified lines of a virtual page since a global checkpoint. The virtual page with all the current values is called an active page. For example, a backup page for virtual page x with LTS=5 means that it stores the original memory lines of page x that are modified since the GTS becomes 5. INDRA also assigns a *dirty block bitvector* to indicate which lines were backed up prior to modification.

Backup Page Record. For each backup page, there is a backup page record stored in the external memory as shown in Figure 3. The backup page record maintains four fields: the physical address of a backup page, the LTS, the dirty bitvector, and the rollback bitvector. The rollback bitvector is a bitmap of all the memory lines whose values in an active page should be replaced by the values of its backup page during a recovery. The use of these two bitvectors enables INDRA to perform incremental backup and state rollback concurrently without the overhead of an "explicit" memory rollback. In other words, whenever a malicious exploit is detected, INDRA will immediately process the next request without inducing an explicit rollback, i.e. no memory copying. During the processing of the subsequent request, INDRA will automatically store new changes and rollback the previous changes made by the previous troublesome request. This way, both the overheads of memory checkpointing and recovery are amortized with the application's execution.

To expedite the access, INDRA extends the TLB to include the corresponding backup page record for each page stored in TLB as illustrated in Figure 3. For clarity, the example assumes that there are only eight cache lines for each memory page. When a new virtual page is brought into the TLB, its corresponding backup page record is also brought

[5]This GTS is maintained in a hardware register for each service application as part of the process context. It needs to backup during context switches.

0-7695-2608-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Table 3: Comparison of Macro Memory Backup Approaches Supporting Recovery

Approach	Explanation	Backup	Recovery
software checkpointing [23]	backup entire page when modified	copy all dirty pages, slow	fast, modify page translation
memory update log [28]	transactional memory update, memory update log	fast	undo update according to the log, slow
hardware supported virtual checkpointing [8]	backup dirty pages on demand	copy dirty page on demand, slow	fast, modify TLB entry
INDRA	delta backup pages, backup only dirty cache lines	fast, no page copy	fast, no page copy

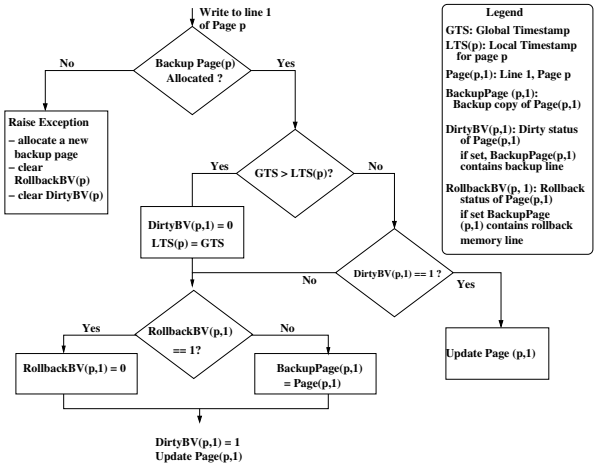


Figure 4: Processing of Memory Write

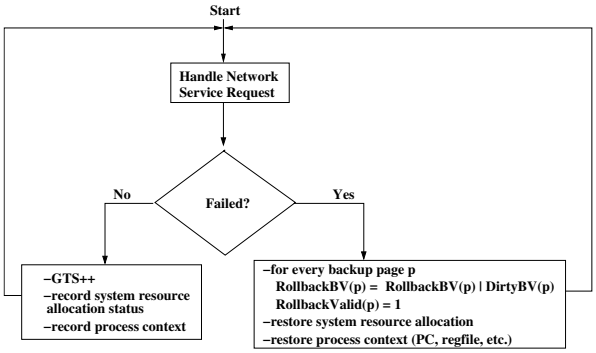


Figure 6: Processing of Service Request

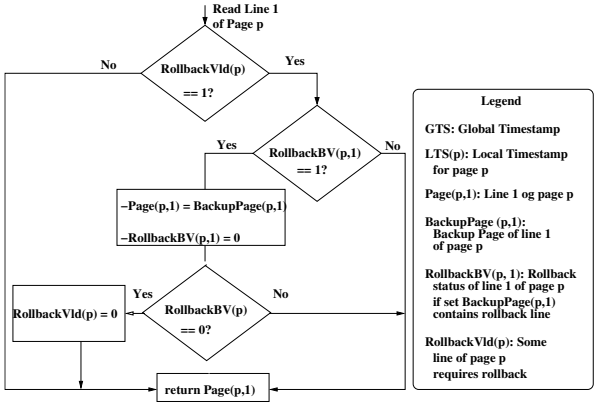


Figure 5: Processing of Memory Read

into the backup page table by the recovery system.

Flowcharts in Figure 4 and Figure 5 show how INDRA processes server application's memory read and write accesses. Figure 6 illustrates how a server application uses INDRA's backup and recovery mechanism. To elucidate these flowcharts, the following example is provided.

An Example of INDRA Recovery. This example illustrates how INDRA incrementally stores memory backup and performs rollback recovery when a malicious network request triggers a fault. We use one application memory page, called page p as example. Figure 7 shows the history of memory accesses to page p for several network requests and the status related to page p's backup information.

Action 1 (the first row) shows the initial state of page p's backup page. Assume that page p's LTS is 3 while the GTS was increased to 5. Action 2 is a memory write to

Action	Memory Access History(page p)	Operation	LTS	Dirty Blocks Bitvector	Rollback Bitvector	Rollback Valid
1	Page p Initial State Last Request OK GTS =5		3			0
2	GTS =5 Wr Memory Line 7	DirtyBV(p) = 0 BackupPage(p,7) = Page(p,7) DirtyBV(p,7) = 1 LTS(p) = GTS	5			0
3	GTS =5 Wr Memory Line 2	BackupPage(p,2) = Page(p,2) DirtyBV(p,2) = 1	5			0
4	GTS =5 Wr Memory Line 2		5			0
5	Failure Next Request GTS =5	RollbackBV(p) = RollbackBV(p) | DirtyBV(p) DirtyBV(p) = 0 RollbackVld(p) = 1	5			1
6	GTS =5 Rd Memory Line 7	Page(p,7) = BackupPage(p,7) RollbackBV(p,7) = 0	5			1
7	GTS =5 Wr Memory Line 1	BackupPage(p,1) = Page(p,1) DirtyBV(p,1) = 1	5			1
8	Failure Next Request GTS =5	RollbackBV(p) = RollbackBV(p) | DirtyBV(p) DirtyBV(p) = 0 RollbackVld(p) = 1	5			1
9	GTS =5 Rd Memory Line 1	Page(p,1) = BackupPage(p,1) RollbackBV(p,1) = 0	5			1
10	GTS =5 Wr Memory Line 2	DirtyBV(p,2) = 1 RollbackBV(p,2) = 0 RollbackVld(p) = 0	5			0
11	OK Next Request GTS =6		5			0
12	GTS =6 Wr Memory Line 6	DirtyBV(p) = 0 BackupPage(p,6) = Page(p,6) DirtyBV(p,6) = 1 LTS(p) = GTS	6			0

Figure 7: Example — History of Backup States

0-7695-2608-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

line 7 of page p. Since the current GTS is greater than the page's LTS, INDRA clears the old dirty bitvector, copies line 7 into a backup page before overwriting it, sets the corresponding bit of the dirty bitvector, and updates the page's LTS with the GTS. Action 3 is another memory write to page p that sets the dirty bitvector and backs up line 2 without updating p's LTS as p's LTS was synchronized to the GTS already. The next Action is yet another write to the same line 2. INDRA checks the dirty bitvector which indicates that a backup copy prior to the current request already exists. Therefore, INDRA overwrites line 2 directly.

If a buffer overrun or other type of intrusion or state corruption is detected by the resurrector core, the resurrector will interrupt and stall the resurrectee core. For each backup page, the resurrector will set the rollback valid bit, update the rollback bitvector by bitwise OR-ing the backup page's dirty bitvector and its current rollback bitvector, and clears the page's dirty bitvector. The 5th Action in Figure 7 shows the resulting status. Then the resurectee's interrupt handler will rollback the application's context (e.g. PC, regsiter file) back to the state when the GTS is incremented to 5, which corresponds to the state before the faulty request was processed. Then the resurrectee resumes the server application and continues processing the next service request.

Action 6 shows that the server application issues a memory read to line 7 of p while its corresponding rollback bit is set. Instead of reading the line from the active page, INDRA will copy the same line from the backup page, and clear the corresponding bit of the rollback bitvector to indicate that the line has been recovered. The 7th Action shows a memory write to line 1 of p. The same operations in Action 3 are applied.

Assume that the next network request is also malicious, causing the server application to fail again. In this case, INDRA has to rollback damages caused by both the current network request and the one before. INDRA needs only to follow the same procedure as shown in Action 5. After that, the server application is resumed to process the next request.

If nothing goes wrong during the process of this request, the server application will increment the GTS for the next request. The last Action in Figure 7 shows a memory write to page p after the GTS is increased.

Overhead of Backup Space. It is important to point out that INDRA allocates delta backup pages on demand. Thus it does not have significant physical memory overhead. Our profiling study shows that on average about 50 pages are touched during the inter-packet processing interval. Note that the actual number of dirty lines inside the 50 pages are much smaller. The overall overhead is small comparing with the size of today's system memory.

Protection of Backup Space. The backup page records and all backup pages are allocated and managed by the INDRA's operating system and invisible to the service applications. Therefore, they cannot be corrupted by the exploits into the application's virtual memory space. Note that the rollback, when a recovery is needed, is completely handled by the INDRA recovery mechanism, not the service application itself.

3.3.2 Hybrid Recovery Scheme

Each time INDRA increments the GTS register, it assumes that either the application is in a healthy state or the application has recovered from a previous failure. This assumption is generally true if a malicious request immediately triggers corruption or service failure after it is processed. This covers a majority of the DoS exploit attacks. In other words, INDRA's micro recovery mechanism always rolls back by one service request when something is wrong. However, there may exist some well-contrived exploit attacks we call "dormant" attacks. These "dormant" attacks may cause damage to server applications but the server is able to continue serving many new requests before it fails due to damage inflicted by past requests. It is recommended that INDRA combine slow paced macro checkpointing technique with its swift request based recovery. After responding to a number of requests, the server OS will issue an application checkpoint [23]. The result is a hybrid dual recovery mechanism as shown in Figure 8. INDRA combines its micro-level per request based recovery with the slow and infrequent application level recovery. Since the software checkpoint is performed infrequently. e.g. once every 10,000 processed requests, the performance impact will be small. If something bad happens, INDRA first tries the micro recovery process by assuming that the damages are caused by the previous handled service request. If INDRA cannot recover a corrupted server application using this swift recovery approach, it will use the traditional application checkpointing instead.

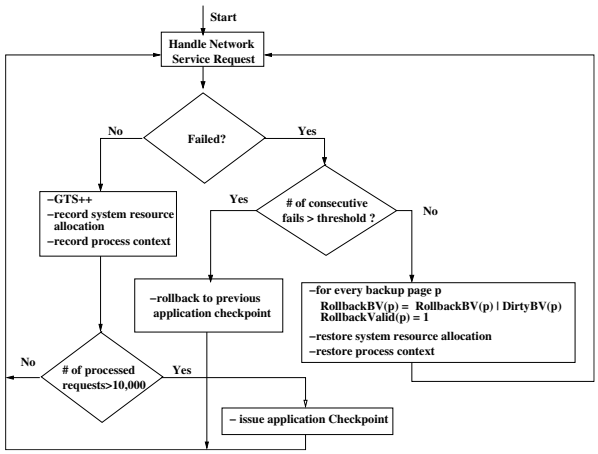


Figure 8: Hybrid Recovery Scheme

3.3.3 System Resource Recovery

With respect to process context and resources, INDRA maintains the file descriptors, the environment of the process and other allocated system resources. During recovery, resources allocated after the backup request will be freed. Files opened after the checkpoint will be closed. Opened files before the checkpoint will remain open after rollback. All child processes (might be a malicious child process) spawned by the application after backup are killed and newly allocated memory pages are reclaimed. However, INDRA does not restore all the possible system states. States associated with inter-process communication, messages, and signals are not recovered. The system does not rollback any changes to the files, or messages and signals already sent. If the application logs the malicious request to a log file, then the information will remain in the file for audit and inspection. Note that INDRA's recovery approach is designed for the scenario of network oriented server applications and remote exploit attacks. It is not meant to be a general recovery approach for any type of programs. In case the application cannot be properly recovered, the system will use traditional

IEEE COMPUTER SOCIETY

recovery approach.

3.3.4 Connection State Recovery

It is important to point out that INDRA recovers at the application level, not the network or transport protocol level. INDRA does not cause connection upset at transport layer for both stateless protocol such as UDP or state protocol such as TCP. In addition, different from system network, transport protocols designed for Internet such as TCP has built-in support for tolerating lost packets and re-synchronizing states between two end points. Furthermore, different from the conventional transaction based recovery, INDRA does not intend to recover the failed request and respond to it again. INDRA does not care or bother to recover the connection between the server application and a malicious client. As a response of recovery, a server application may after recovery terminate the faulty or malicious connection. For most server applications such as HTTP and DNS, the application level connection is stateless. This means that every new network request will initiate a new connection and the connection is terminated after the request is served. It is important to keep in mind that INDRA only cares for connections from well-behaved clients who have not sent any requests that corrupt the server.

3.4 Limitation

Although INDRA provides better capability of intrusion tolerance and service revivability than prior approaches, there are limitations. INDRA does not promise to handle all conceivable attacks and recover from all possible corrupted machine states. It focuses mainly on service loss caused by malicious network input. However, INDRA does create a system architecture that allows for future advanced detection and recovery techniques to be studied and deployed. In addition, INDRA's architectural design does not attempt any file system recovery assuming that all disk writes are issued by verified program execution (synchronization described in Section 3.2.5) and properly checked. In addition, the resurrector, even though insulated from remote exploit attacks, is subject to physical tampering, e.g. re-flashing the flash memory that activates the resurrector. INDRA is also not a replacement for the conventional means of patching software vulnerabilities. Last, INDRA does not handle attacks that jam a network channel, e.g. router flooding. Such problems are outside the server and should be countered by network-oriented security solutions.

4. EVALUATION

To evaluate the proposed INDRA and its monitoring and recovery techniques, we used Bochs [19] and TAXI [32] framework. Bochs, a full-system x86 emulator, models the entire platform including network device, hard drive, VGA, and other devices to support the execution of a complete OS and its applications. TAXI is a Simplescalar simulator with x86 front-end for our performance analysis. Architectural support for INDRA such as automatic L2 cache to instruction L1 cache trace, memory state backup/rollback and the required processor state rollback were implemented.

Our hardware framework emulates a dual-core processor with shared memory for synchronization and data communication. One core is configured to run a full-blown Redhat Linux 6.0 and network applications and the second core is designated as the resurrector running a simple runtime system based on a stripped down tiny Linux stored in a flash memory. The resurrector boots from the runtime system; the entire system including the security software is less than 10MB.

Parameters	Values
Fetch/Decode width	8
Issue/Commit width	8
L1 I-Cache	DM, 16KB, 32B line
L1 D-Cache	DM, 16KB, 32B line
L2 Cache	4way, Unified, 64B line, WB cache 512KB for each core
L1/L2 Latency	1 cycle / 8 cycles (512KB)
I-TLB	4-way, 128 entries
D-TLB	4-way, 256 entries
Memory Bus	200MHz, 8B wide
Memory Latency	X-5-5-5 (core clocks) X depends on page status
CAS latency	20 mem bus clocks
Pre-charge latency (RP)	7 mem bus clocks
RAS-to-CAS (RCD) latency	7 mem bus clocks

Table 4: Processor model parameters

Within our security software framework, we implemented a simple software security monitor based on our prior description. The program receives snooped monitor traces from the other processor through designated I/O ports. Upon the receipt of a new instruction, the monitor first verifies the code origin against recorded code page attributes. Information of the application code space is determined by the resurrectee and posted to the resurrector through the shared FIFO queue. Applications are distinguished according to information in the CR3 control register which is used to store the physical address of a process's page table. It is unique for each process. An instruction sent to the resurrector is paired with the CR3 value so that the resurrector can decide which set of informations should be used for security check. Alternatively, the trace information can be tagged with its process ID. As a result, the resurrector's monitoring program decides whether the instruction is a function call or a control transfer instruction based on the opcode. For control transfer, the program uses recorded application symbol table, function export/import lists to verify the legitimacy of the control transfer. Mechanism of memory state checkpointing and rollback is implemented in both the Bochs and the performance simulator. We also integrated an accurate DRAM model [16] to improve the system memory modeling, in which bank conflicts, page miss, row miss are all modeled based on the PC SDRAM specification. The processor parameters are listed in Table 4.

Six popular open source network server applications were used: file transfer server (ftp daemon), web server (apache http daemon), email sever (imap daemon), domain name server (bind daemon), mail transmission server (sendmail), and network file system server (nfs daemon). All server applications are executed as standalone daemons.

4.1 Security Evaluation

The effectiveness of using code origin, return address, and control flow restriction to detect real world remote exploit attacks and server fault is extensively tested and verified in [20] using a security monitor implemented inside a dynamic binary code rewriter. Though INDRA's monitor implementation is different from [20], both schemes check the same types of information. The idea of recovering server applications on per request basis by repairing the damages has been validated by a number of recent studies [27, 28]. All the studies used real world exploits and showed that typical network applications can effectively recover from faults induced by remote exploit attacks on per network request basis.

We also validated our recovery approach using a set of real exploits against the selected popular server applications. Exploit codes or scripts are collected from various

IEEE COMPUTER SOCIETY

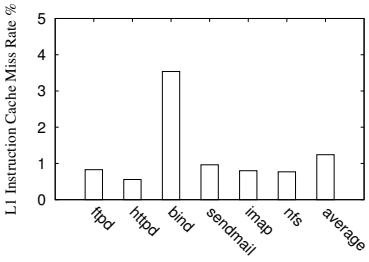


Figure 9: L1 instruction cache miss rate

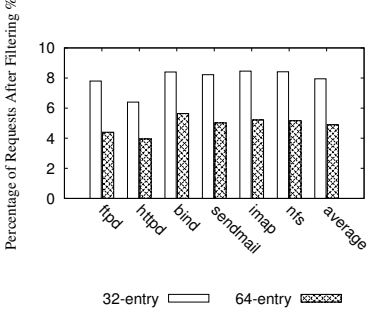


Figure 10: Effectiveness of Code Origin Check Filtering

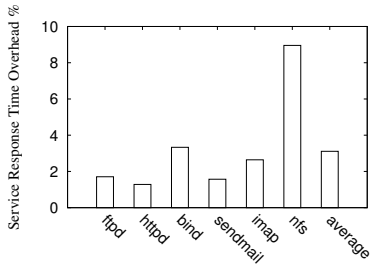


Figure 11: Monitoring Overhead

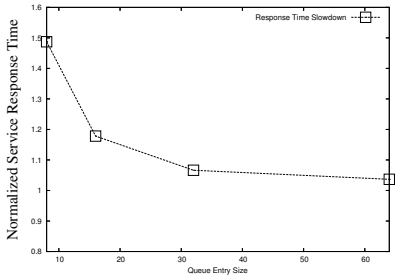


Figure 12: Impact of Shared Queue Size

hackers' websites or security agent websites.[6] Particularly, we launched remote exploit attacks targeted for the following server vulnerability, CVE documented attack CAN-2003-0651 [1] on HTTP server, exploit (VU#196945) [4] on bind daemon, documented exploits (CAN-2003-0466) [1] on FTP server, and documented exploit (CAN-2004-0640) [1] on SSLtelnet daemon. Experiment shows that INDRA can detect and recover from these exploit attacks.

4.2 Performance

One major difference between INDRA and other per-request based self-recovery approaches is INDRA's efficiency in monitoring and memory state recovery. We wrote a set of scripts to automatically send network requests to the simulated platform and handle the responses. For DNS service, the script sends a sequence of queries to the server. For FTP, the script automatically logs in into the simulated machine, downloads and uploads a few files. For imap, a python script is used to automatically authenticate and check new emails. For http, wget is used to download a set of web pages from the simulated server recursively. For sendmail, a shell script is used to automatically send a sequence of text mail files to the server continuously. For NFS, a shared file directory in the simulated machine is mounted by the simulation host. NFS is tested with a script that first copies a whole directory of Apache html manual files to the simulated server then copies them back to the client.

4.2.1 Monitor

We evaluated the time overhead required to complete the service requests for the six network applications. We implemented a network packet dump module inside our simulator so that it can produce a trace of network packets that help identify each packet's receiving and sending time by the simulated server. The results are based on five runs of the test scripts. As aforementioned, code origin check workload is proportional to the IL1 miss rate. Figure 9 shows the IL1 miss rates. As can be seen, the miss rates are relatively low. Figure 10 shows the percentage of code origin checks after using the simple page address filtering mechanism. The filter CAM removes the majority of redundant code origin checks. According to the figure, on average, 92% and 95% of the code origin checks can be waived with a 32 and 64-entry filter CAM, respectively.

Figure 11 shows the service response time overhead with monitoring against a system without monitoring support. Overheard due to backup and rollback are not accounted for and will be discussed in Section 4.2.2. As suggested by the results, INDRA monitoring incurs only a small percentage of performance degradation.

Another factor that affects the performance of INDRA is the size of the request FIFO between the *resurrectee* and the *resurrector*. Figure 12 shows average normalized request response time with 16, 32, and 64 entries of monitor request FIFO. As indicated by the results, a queue of 16 request entries is too small and it could cause extra stall of the resurrectee cores. When it is increased to 32 or more, the performance starts to saturate.

4.2.2 State Backup and Recovery

Figure 13 shows the average instruction count between

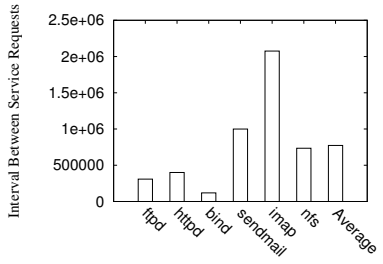


Figure 13: Number of instructions between requests

[6]They include www.securiteam.com, www.insecure.org, and www.k-otik.com.

0-7695-2608-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

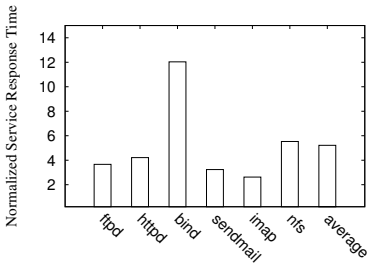


Figure 14: Slowdown using traditional memory virtual checkpointing

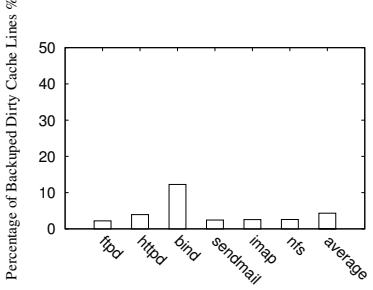


Figure 15: Percentage of backup dirty lines

two back-to-back requests. The numbers are in the range from hundreds of thousands to millions of instructions. Since the interval is relatively short, using traditional memory checkpointing schemes will incur intolerable overhead. Figure 14 shows the slowdown of response time if dirty memory pages are backed up using the conventional memory virtual checkpointing. Most of the overhead is due to the frequent page-to-page memory copying. Figure 15 suggests that the dynamic number of cache lines that require backup over all the modified lines is relatively small, showing that INDRA's delta page based backup can be orders of magnitude more efficient.

Figure 16 shows the results of service response time slowdown for two configurations. The left bar shows the slowdown caused by monitoring and backup, while the right bar shows the slowdown when a rollback is needed for every other network request. The results indicate that both INDRA's memory backup and rollback are more efficient than the page copy based checkpointing scheme shown in Figure 14. The only outlier is bind (DNS) having a more than 2x slowdown. The reasons are twofold. First, the interval (150,000 instructions) is much shorter than the others as shown in Figure 13. Second, the number of dirty lines is also much higher in Figure 15. As shown by our results, INDRA

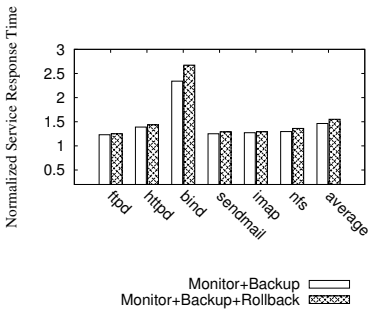


Figure 16: Slowdown by backup and rollback

is suitable for recovering server applications from frequent maliciously induced faults with reasonable overhead.

5. RELATED WORK

We discuss related work on the area of automatic service recovery from faults caused by remote exploit attacks. Studies of recovery from transient fault, random circuitry fault or hardware design fault [6, 15, 21, 25] are different issues because faults caused by remote exploit attacks often have different characteristics and require different recovery approach as described earlier. INDRA is also different in both solution and purpose from the studies of software rejuvenation that are aimed to solve software reliability problems caused by software aging [17].

5.1 Exploit Detection

Compiler techniques such as StackGuard [12] have been developed to detect buffer overflow attacks. Another technique called program shepherding [20] uses binary interpreter to prevent an attack from executing injected codes. In [13, 31], hardware support for tracking information flow is proposed for preventing external untrusted information from being executed as code or used as function pointers. Most of those techniques focus on intrusion detection only and does not provide means for swift recovery like what INDRA does.

5.2 Recovery

5.2.1 Traditional Recovery

Traditional error recovery terminates faulty applications or reboots the entire system. Many techniques have been proposed to speed up the reboot process including recent study on microreboot [9]. INDRA is different from these solutions because it attempts to maintain the service availability by recovering it from crash or corrupted states without reboot or disruption to the legitimate users.

5.2.2 Reactive Immune System and DIRA

[27] uses instrumented applications and program emulation to discover and fix faulty or vulnerable server software. Another approach called DIRA [28] was proposed for service recovery by repairing damaged memory states based on memory log obtained by instrumenting application source codes. The approaches in [27, 28] implement intrusion monitor and service recovery at the application level which relies on the availability of the application itself. This kind of design severely limits the effectiveness of service recovery because the applications themselves are vulnerable to direct remote exploits.

5.2.3 Reliability and Security Engine

[21] presents a study, called RSE, using dedicated hardware modules and logic for detecting various faults including both buffer overflow based and transient faults. RSE can recover processor pipeline from many transient faults. However, the nature of faults induced through remote exploits requires some special treatment which is absent in RSE's checkpointing and recovery approach.

5.2.4 Memory State Recovery

Memory state backup and recovery have been studied exhaustively in the past using designs such as virtual checkpointing, on-chip history file, on-chip checkpoint buffer, memory update stack, and etc [30, 8, 29]. However, as we have discussed before, due to some unique properties of

IEEE COMPUTER SOCIETY

remote attack induced memory error or corruption, many of the efficient memory state recovery techniques designed for recovering from miss-branch prediction, transient fault, or miss-speculation are not directly applicable to the cases of swift recovery of damaged network services. First, the inter-request execution window for network server applications is too large for micro-level checkpointing, in the range of millions of instructions or even more. Second, recovery of network applications involves possible system resource reallocation and recovery. Third, to fight against DoS attacks, *INDRA* has to provide both high speed state back up and high speed frequent rollback. The proposed INDRA recovery scheme is sufficient for such purpose.

6. CONCLUSION

In this paper, we present INDRA — a framework that integrates novel security mechanisms into the emerging multicore platform to provide highly available, revivable, and continuous services in the face of remote network attacks and network attacks induced memory errors. The framework also introduces a new configurable programming model to address dependability concerns in the enterprising computing domain using multicore systems. In contrast to the previous research conducted on software-based recovery, INDRA creates a remote attack immune hardware sandbox based on asymmetric configuration among different cores to create a solid insulation against malicious exploits. Cores configured as the resurrectors perform monitoring and introspection for service applications running on the resurrectee cores. Furthermore, INDRA proposes a novel delta backup scheme for resurrectees to enable high speed recovery when an attack or a fault is detected by their resurrector. As shown in our experiments, INDRA not only provides better dependability and availability for high performance production servers hosting high volume networked services but also facilitates a fast backup and recovery mechanism that shows a substantial improvement against the conventional checkpointing schemes.

7. ACKNOWLEDGMENT

This work is supported in part by NSF grants CCF-0326396, CNS-0325536, Intel's Multicore curriculum development fund, and a Department of Energy Early CAREER Award. The authors also thank Professor Trevor Mudge for his support and valuable comments.

8. REFERENCES

[1] CVE: Common Vulnerabilities and Exposures. *http://www.cve.mitre.org/*.

[2] PaX Team, Non Executable Data Pages. *http://pageexec.virtualave.net/pageexec.txt*.

[3] Solar Designer, Non-executable User Stack. *http://www.openwall.com/linux/*.

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

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

[6] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, 1999.

[7] A. Avizienis. The Methodology of N-Version Programming, 1995.

[8] N. S. Bowen and D. K. Pradhan. Virtual Checkpoints: Architecture and Performance. *IEEE Transactions on Computers*, 41(5):516–525, 1992.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *Proc. 6th Symposium on Operating Systems Design and Implementation*, 2004.

[10] P.-Y. Chung and C. Kintala. Checkpointing and Its Applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 22–31, 1995.

[11] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[12] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[13] J. R. Crandall and F. T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[14] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *In Proc. Net. and Distributed Sys. Sec. Sym*, February 2003.

[15] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of the 30th annual International Symposium on Computer Architecture*, pages 98–109, 2003.

[16] M. Gries and A. Romer. Performance Evaluation of Recent DRAM Architectures for Embedded Systems. In *TIK Report Nr. 82, Computing Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich*, November 1999.

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

[18] W.-M. W. Hwu and Y. N. Patt. Checkpoint Repair for Out-of-order Execution Machines. In *Proceedings of the International Symposium on Computer Archtecture*, 1987.

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

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

[21] N. Nakka, J. Xu, Z. Kalbarczyk, and R. K. Iyer. An Architectural Framework for Providing Reliability and Security Support. In *Proceedings of International Conference on Dependable Systems and Networks*, June. 2004.

[22] D. Neal. Online Blackmail Grows. In *IT Week*, 08, Mar 2005.

[23] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. Technical report, 1994.

[24] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective Architectural Support for Rollback Recovery in Shared-memory Multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[25] J. Ray, J. C. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-fault Detection and Recovery. In *Proceedings of the 34th International Symposium on Microarchitecture*, 2001.

[26] Scut. Exploiting Format String Vulnerabilities. *http://www.team-teso.net/articles/ formatstring/*, 2001.

[27] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

[28] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attack. In *The 12th Annual Network and Distributed System Security Symposium*, 2005.

[29] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors With Global Checkpoint/recovery. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[30] M. E. Staknis. Sheaved Memory: Architectural Support for State Saving and Restoration in Pages Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating System*, 1989.

[31] G. E. Suh, J. Lee, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[32] S. Vlaovic and E. S. Davidson. TAXI: Trace Analysis for X86 Interpretation. In *Proceedings of the 2002 IEEE International Conference on Computer Design*, 2002.

[33] K. L. Wu, W. K. Fuchs, and J. H. Patel. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), 1990.

IEEE COMPUTER SOCIETY