VMM-based Process Shepherding

Haibo Chen, Pengcheng Liu, Rong Chen, Binyu Zang
{hbchen,pcliu,chenrong,byzang}@fudan.edu.cn
Parallel Processing Institute, Fudan University

Parallel Processing Institute Technical Report
Number: FDUPPITR-2007-08002
August 2007

Parallel Processing Institute, Fudan University
Software Building, 825 Zhangheng RD.
PHN: (86-21) 51355363-18
FAX: (86-21) 51355358
URL: http://ppi.fudan.edu.cn

VMM-based Process Shepherding

Haibo Chen, Pengcheng Liu, Rong Chen, Binyu Zang
{hbchen,pcliu,chenrong,byzang}@fudan.edu.cn
Parallel Processing Institute, Fudan University

Fudan University, Parallel Processing Insitute, PPITR-2007-08002
August 2007

Abstract

Processes in commodity operating systems are "*wild*" [1] in nature: They are usually granted with excessive privileges, yet can be easily compromised and abused. Unfortunately, since commodity operating systems are big, complex, thus inherently untrusted, monitoring process behaviors within them is inherently insecure and could be circumvented or tampered. In this paper, we present an approach, named VMM-based process shepherding, to prevent, detect and isolate harmful behaviors (e.g. intrusions) of wild processes. The key idea of our approach is using a virtual machine monitor (VMM) to shepherd all privileged operations made by a wild process, in terms of system calls. As a VMM can easily be adjusted to intercept all system calls from a process running within operating systems thereon, such interception is mandatory and can not be bypassed. Further, as our approach is completely implemented in VMMs, it can result in good operating system transparency and portability. We provide three techniques as building blocks to process shepherding. First, the VMM prohibits any unauthorized accesses to privileged resources (e.g. system configuration files) using policy-based system call auditing. Second, the VMM uses system call sequences to detect possible malicious behaviors. Unexpected system call sequences should be considered as evidences for misbehaving. Third, the VMM isolates all suspect operations and discard them once a wild process is identified as malicious. Based on the three techniques, our approach tends to be safe and non-intrusive, that is, it can isolate damages made by a wild process. We present Shepherd, a prototype system based on Xen VMM, and evaluate it against real-life applications. According to our evaluation, Shepherd is resistant against several recent real-life attacks. Performance measurements show that our implementation incurs only a small amount of performance overhead.

1 Introduction

Security and reliability have become two of the biggest problems facing contemporary computing systems. These systems are fragile and always susceptible to various attacks, intrusions and abuses. One of the major reasons for such problems is that contemporary operating systems are designed and optimized for openness, functionality and efficiency, but not for security and reliability. Processes in a contemporary operating system are usually granted with excessive privileges (Efstathopoulos et al., 2005), and thus capable of unrestricted accesses to many unauthorized resources. Hence, to increase system security and reliability, it is desirable to monitor and restrict the privileged accesses of wild processes.

Currently, most monitoring approaches choose operating systems as security building blocks (Wagner, 1999; Acharya and Paje, 2000; Sun et al., 2005). Unfortunately, most contemporary operating systems are unreliable in two aspects. First, they are big, complex and developed using unsafe languages, thus essentially insecure and untrustworthy. They can be tampered or penetrated due to design flaws, security vulnerabilities and implementation bugs (CVE, 2006; Securityfocus, 2007, e.g.). Second, there is poor isolation among processes in contemporary operating systems. These processes are usually granted with unrestricted privileges, yet can be easily tampered using software defects (CERT Coordination Center, 2006). A tampered process with root privilege can easily compromise a monitoring system.

[1]Here, a wild process refers to a process that is untrusted and may cause damages.

This paper describes the design and implementation of Shepherd, a process shepherding system that employs system virtualization to monitor and restrict the behaviors of wild processes in contemporary operating systems. The key idea is using a virtual machine monitor (VMM) to monitor and track all privileged operations made by a wild process. In contrast to other systems, Shepherd is superior in that it is tamper-resistant, non-intrusive and OS-transparency. First, since a VMM is relatively small and thus comparatively trustworthy, implementing security policies in it can result in superior security. Second, as a VMM is capable of intercepting all privileged operations, it has the same level of knowledge on the privileged behaviors of a process, compared to a user-mode monitoring system. Moreover, such interception is imperative that even a compromised process can not bypass it. Third, Shepherd implements most security-critical policies in the VMM, resulting in good OS-transparency and portability, yet incurs only a little impact on the code size of the trusted computing base (TCB). Finally, Shepherd is safe and non-intrusive in that it shields changes to critical resources by means of copy-on-write and discards these changes once they are identified to be malicious.

To effectively prevent, detect and isolate malicious behaviors of a wild process, Shepherd integrates three techniques to provide a safe execution environment to host a wild process. First, Shepherd prevents any unauthorized access to privileged resources using policy-based access control. Untrusted processes are prohibited to access security critical resources, including system configuration files and raw devices. Accesses to unauthorized resources are either silently suppressed with a successful return value or forbidden with a termination of the offending process.

Second, the VMM uses system call sequences to detect possible malicious behaviors. System call sequences (Forrest et al., 1996; Somayaji and Forrest, 2000) have been extensively studied and established as good candidates to detect malicious actions from an untrusted process. However, previous approaches are implemented either in kernel level or in user level, thus are susceptive to attacks and tampering. Shepherd utilizes system call sequences to detect misbehaving processes, but beneath the operating system and in the VMM, which results in superior security. Unexpected system call sequences are considered as evidence for misbehaving and the malicious actions will be discarded.

Third, Shepherd prevents a wild process from tainting the system using a shadow file system. All modifications to file systems from a wild process are handled by Shepherd in a copy-on-write manner. A write operation to a file triggers the creation of a shadow and private copy of the file, and all subsequent read and write operations are redirected to the shadow file. This is similar to isolation file system in (Sun et al., 2005). Shepherd incorporates the results from system call sequences detection and automatically discards the operations performed by a misbehaving process. Only if no anomaly is detected during the execution, can the modifications by a wild process be integrated to the main file system.

We have implemented a prototype system using Xen VMM (Barham et al., 2003), to shepherd suspect processes in Linux running atop. To demonstrate the applicability of our approach, we have evaluated Shepherd against several recent real-life security vulnerabilities. Our evaluation indicates that Shepherd can successfully prevent, detect and isolate these attacks and can seamlessly restore a tampered system to a clean state. Performance measurements show that Shepherd only incurs a small amount of performance overhead.

The rest of the paper is organized as follows: In section 2, we compare Shepherd with existing approaches. Then, we provide a review of VMMs and the general mechanisms of Shepherd. Section 4 presents the overall architecture of the process shepherding system. Section 5 comes up with the implementation issues of Shepherd. Next, we present both the functionality and performance evaluation in section 6. Finally, this paper ends up with a concluding remark in section 7.

2 Related Work

Sandbox-based approaches (Wagner, 1999; Acharya and Paje, 2000; Yu et al., 2006; Keahey et al., 2004; Sun et al., 2005) provide isolated environments for executing untrusted code. They can be classified into two categories: interposition based sandboxes (Wagner, 1999; Acharya and Paje, 2000; Sun et al., 2005), through accesses (e.g. system calls) interposition to limit the resources that an untrusted process can access; isolation-based sandboxes (Yu et al., 2006; Keahey et al., 2004), by creating dedicated, isolated environments (e.g. virtual machines) to host untrusted applications. The former approach ensures the monitored processes share the same knowledge on system resources as in normal executions, but results in inferior security because the sandbox may be tampered or circumvented (Garfinkel, 2003). The latter approach tends to be more secure and tamper resistant, but shares only restricted knowledge and resources because the sandbox is completely isolated from other resources, which may restrict the functionalities and behaviors of a monitored process. In contrast, Shepherd shares a good visibility with the former approach, yet embraces the security benefits from the latter one.

Recently, there are a considerable number of interests in using system virtualization to improve system security and reliability (Dunlap et al., 2002; Garfinkel and Rosenblum, 2003; Asrigo et al., 2006). For example, ReVirt (Dunlap et al., 2002) uses a virtual machine to analyze possible intrusions through VMM-based log and replay. VMM-based sensor (Asrigo et al., 2006) utilizes VMMs to implement sensors to monitor operating systems thereon, through dynamic instrumentation of the operating system code. Livewire (Garfinkel and Rosenblum, 2003) is a VM-based intrusion detection system (IDS). The IDS lies in a VM running together with a monitored VM on the same VMM. In contrast to Shepherd, these systems differ in their design goals and approaches. For example, they are designed to detect possible intrusions of the whole operating systems, not in providing monitored and isolated environments for individual processes. Further, they lack mechanisms to response to malicious behaviors in time and recover the intrusions.

Dynamic instruction flow tracking using dynamic translator (Kiriansky et al., 2002; Hu et al., 2006) is a fine-grained monitoring technique to prevent security attacks. By running programs in a dynamic translator, the translator monitors the control flow and data flow of the programs to enforce security policies. Compared to Shepherd, this technique operates in instruction level rather than in system-call level, and only enforces statically assigned policies. Yet, this technique can complement Shepherd to improve system security and reliability.

Detecting malicious actions using system call sequences have been extensively studied ever since it was first advocated by Forrest (Forrest et al., 1996) in 1996. Gao (Gao et al., 2004) gives a systematic study on the effectiveness of various anomaly detection system based on system call sequences. However, most systems are implemented in operating system level, which may be tampered or circumvented since operating systems may be vulnerable and penetrated. Shepherd integrates system call sequences to a VMM to detect malicious actions, resulting in superior security for the detection system.

Intrusion recovery (Goel et al., 2005; Hsu et al., 2006) is a promising technique to remove the damages caused by attacks or intrusions. Taser intrusion recovery system (Goel et al., 2005) enables revocation of malicious operations through logging all process, file and network operations and restoring them after an intrusion is detected. Back to the future (Hsu et al., 2006) preserves system integrity against malware in a similar approach. One-way isolation (Sun et al., 2005) employs an isolation file system to support changes in a copy-on-write fashion. When untrusted applications attempt to modify critical files, they operate on a temporary shadow copy of the files. Shepherd integrates such techniques into a VMM and incorporates them with an anomaly detector to automatically response to malicious actions, resulting in good OS-transparency and non-intrusiveness.

In the past, mandatory access control (MAC) systems have been widely used to enhance system security. One typical system is SELinux (Loscocco and Smalley, 2001), which integrates fine-grained access control policies to control access to many system resources in Linux. However, the policies are usually rather big and complex, which are hard to derive and to verify the completeness. Asbestos (Efstathopoulos et al., 2005) and Histar (Zeldovich et al., 2006) restrict the privilege of processes using capability-like mechanisms to explicitly track and restrict the process privilege. By contrast with Shepherd, Asbestos and Histar are not designed to retain backwards compatibility. Instead, they build new operating systems from scratch and require existing applications to be ported to run in them.

3 Background and Mechanisms

Shepherd relies on a VMM to monitor and restrict the behaviors of wild processes. The key point is how a VMM is informed when a process does some privileged operations. In this section, we first provide a review on the architecture of VMMs. Then, we present how Shepherd utilizes the features of VMMs to shepherd processes.

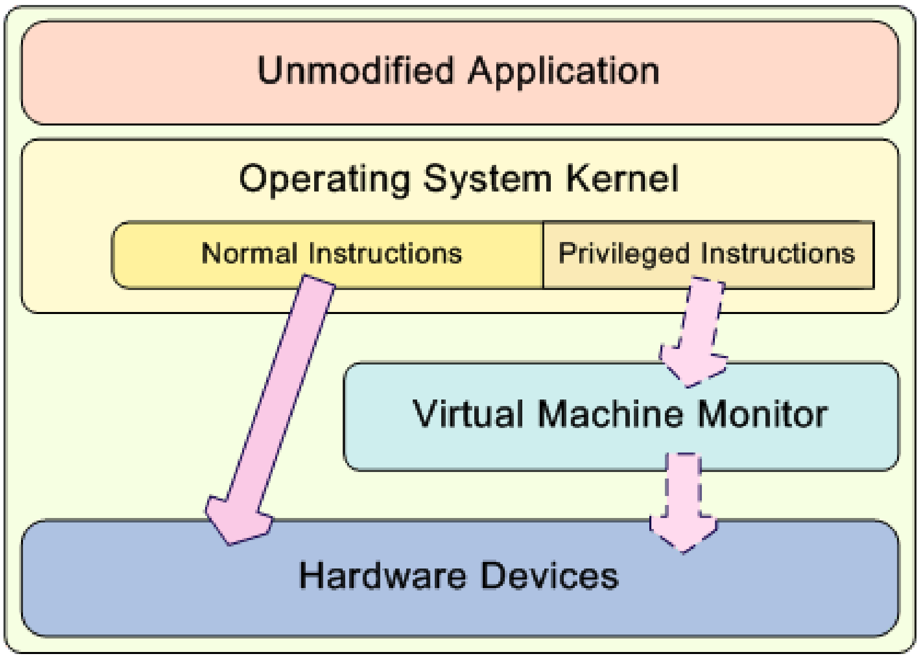

Figure 1: General structure of virtualization.

3.1 A Review of VMMs

As depicted in Fig. 3.1, a virtual machine monitor (VMM) is a software layer lying between operating systems and hardware [2]. It manages the underlying hardware resources and exposes them to the above operating systems in the form of a virtual machine (VM) (Goldberg, 1974). For the sake of performance, a VMM only intercepts privileged instructions that change or observe the hardware state, leaving other instructions running without intervention. To provide strong isolation among various VMs, the interception of privileged instructions is mandatory and can not be bypassed by an operating system atop. Otherwise, it can result in an inconsistent state and the system may collapse. In modern computers, the interrupt lines and some process control state (e.g. page table root) are definitely privileged state, thus accesses to them should be intervened by the VMM.

Contemporary operating systems are abstracted in a single-kernel, multiple-user-process manner: User processes time-share the kernel. User processes can not directly access the hardware resources but instead use the interfaces provided by the kernel, in the form of system calls, to invoke the services provided by the kernel. In a virtualized environment, a VMM is the key resource manager and provider. Therefore, accesses from user processes to kernel, in the form of interrupts, are intercepted by the VMM. Besides, switches of processes, i.e., context switches, are also captured and controlled by the VMM.

3.2 Mechanisms of Shepherd

To securely and safely isolate an untrusted process from tainting the system, Shepherd utilizes the features that a VMM can intercept all system calls made by a process, and a VMM can track the creation, context switches and termination of processes. Essentially, Shepherd uses the VMM as a middle-man between user processes and the operating system kernel, and implements policies in the VMM to prevent, detect and isolate malicious accesses from user processes to the kernel.

Normally, a system call is essentially an interrupt [3] (0x80 in Linux) intercepted and forwarded by the

[2]Here, we only focus on VMMs which directly execute on bare hardware. Shepherded can also be similarly implemented on VMMs running on a host operating system (e.g. VMWare workstation).

[3]Other forms such as *sysenter* (Intel) and *syscall* (AMD) can be similarly handled.

VMM. Accesses to the interrupt line are captured by the VMM that manages the hardware resources. Fig. 2 depicts the execution flow of system calls in Shepherd. Upon capturing a system call, Shepherd audits it according to pre-install policies and reacts using predefined actions. If the system call passes the auditing, it will be forwarded to the operating system kernel to serve it.

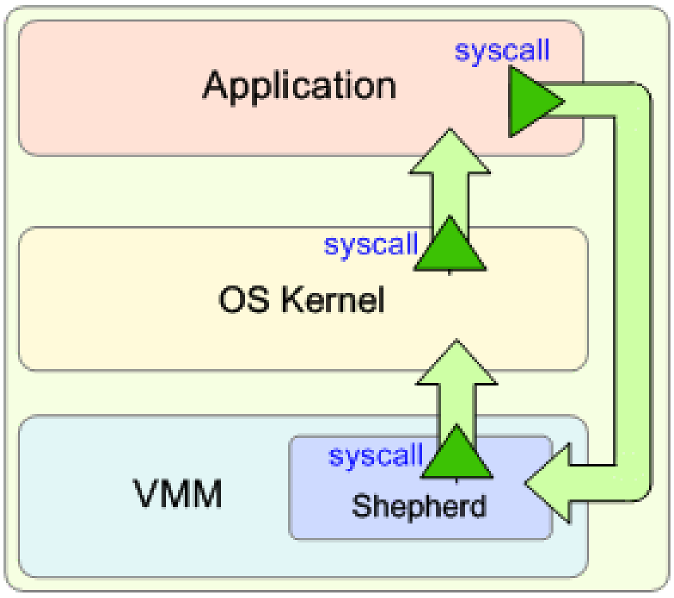

Figure 2: Flows of System Calls in Shepherd.

To efficiently manage monitored processes, Shepherd needs to track the creation, termination of each process. It also needs to track the context switches of processes and the relationships among various processes. To track process creation and termination, Shepherd only needs to monitor the corresponding system calls to create and terminate a process. To maintain process relationships (i.e. parenthood), Shepherd tracks the system calls spawning (i.e. fork) new processes. Tracing the context switches of processes can be done by monitoring specific hardware events (e.g. changes of CR3 in x86).

One concern is the semantic gap (Chen and Noble, 2001) between the VMM and the process being shepherd. In the case of process shepherding, we argue that a VMM shares a comparable knowledge to user level monitors. The most critical events to process shepherding are process, file, and network accesses. All these accesses are based on system calls, thus are visible to the VMM. Meanwhile, the VMM itself can issue system calls to inform users about the occurrences of some dedicated events (i.e. an intrusion is detected and handled), either through writing log files or displaying them in the screen. For example, to print a warning message to screen, the VMM only needs to issue a system call to the kernel that writes the standard output file descriptor.

4 Architecture of Shepherd

Shepherd is designed to provide a safe, tamper-resistant and non-intrusive environment to shepherd the execution of untrusted applications. To achieve such goal, Shepherd is built within a VMM and relies on three components. In this section, we first illustrate the general model of process shepherding. Then we describe details of the three components of Shepherd: the permission auditor, the anomaly detector and the isolation manager.

4.1 Process Shepherding Model

Fig. 3 illustrates the general work flow of Shepherd. Shepherd utilizes three components that loosely cooperate to prevent, detect and isolate possible attacks and intrusions. The permission auditor does some general checks to prohibit possible damages to critical system resources. The anomaly detector and the isolation manager collaborate with each other to prevent possible damages to the system. The isolation manager ensures that the malicious actions made by an adversary can be restored when an anomaly is detected by anomaly detector.

All system calls made by a wild process first flow into Shepherd lying in the VMM. A system call is first checked by the permission auditor, which examines the system call number and its corresponding arguments against predefined rules. Upon a violation, the auditor informs Shepherd and redirects the control flow to the anomaly handler, which will either suppress the error by silently returns a success code or informs the operating systems to kill the offending process.

The system call then flows into the anomaly detector in Shepherd. The detector first logs the system calls and its arguments for further use. Then, it checks the validity of the system call by matching it together with previous system calls against a predefined profile. The profile records all normal system call sequences and is loaded into Shepherd during a process creation. If no profile is provided, Shepherd automatically creates a profile based on the elapsed execution. Once the profile is stable, the generated profile is used for anomaly detection.

Shepherd classifies system calls based on whether they involve file system operations. For those do not operates on file systems, Shepherd simply audits, logs and examines the system call sequence. If they are valid, they are forwarded directly into the kernel to serve them. For file system related system calls, it is more complicated to handle them. To prevent modifications made by a wild process from tainting other normal processes, Shepherd isolates them in a copy-on-write fashion. Shepherd takes an approach rather similar to the isolation file system in one-way isolation (Sun et al., 2005). Wild processes can only operate on a shadow file system.

If an anomaly is detected by the anomaly detector, the isolation manager either discards the shadow file system and informs the kernel to kill the process, or suppresses it silently. Only if no anomaly is detected by shepherd, can the changes made by the wild processes take effect. The shadow file system is integrated into the normal file system according to a specific criterion.

4.2 Permission Auditor

The permission auditor is responsible to check the permission of system calls made by a wild process. Many attacks involve writing to some specific privileged resources. For example, a common way to attack a server is to tamper the configuration files (e.g. /etc/xinetd.conf in Linux) to give attackers the root permission. To prevent this, the system call number and the arguments are examined and compared to predefined rules. The rules specify the prohibited operations on some privileged resources for wild processes.

Shepherd does not aim to provide an accurate and complete specification to prevent all malicious accesses, because it is impossible for many applications. A complex specification is difficult to derive and likely introduces false positives, while a simple one surely produces a large set of false negatives. Here, Shepherd only provides a general and global specification to prevent possible false positives. It instead relies on the anomaly detector to detect further malicious behaviors.

4.3 Anomaly Detector

It is generally hard for system administrators to decide whether an execution is normal and harmless. Thus, it is unreasonable to leave such arduous work to users. Instead, Shepherd determines the anomaly of system calls by comparing the current execution with a predefined profile, similar to the approach in (Forrest et al., 1996; Somayaji and Forrest, 2000).

To get a profile, Shepherd is designed to work in two modes: training mode and monitoring mode. The training mode is to used collect normal execution sequences by running the program several times in a relative safe environment, e.g., in a clean system and within a self-contained networked environment. After getting a steady profile, the profile is used in the monitoring mode to detect anomalies. The profile is loaded into Shepherd when a new process is created with a new executable binary image. Processes with the same image share a single profile. A profile is composed of a set of normal system call sequences. Upon intercepting a new system call, the anomaly detector compares the combination of the system call and its predecessors

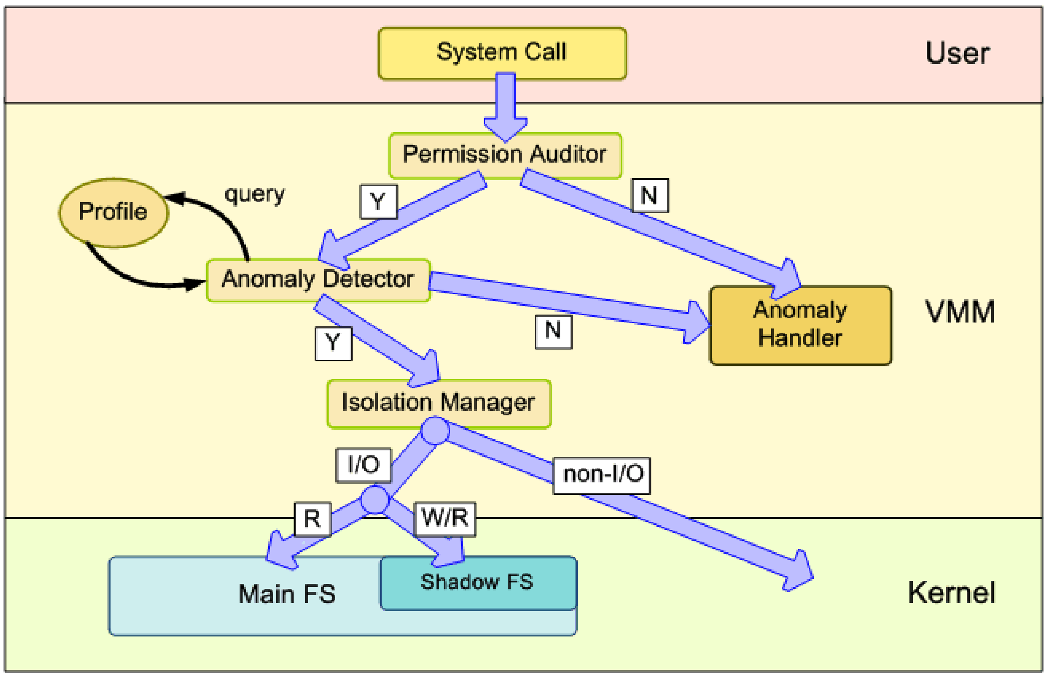

Figure 3: The **logical** model of Process Shepherding. The VMM and the kernel are placed in reversed order because a system call is first intercepted by the VMM. Only if it passes the audit and processing by Shepherd in VMM, can it be forwarded to the operating system kernel.

with normal sequences. If the current sequence is not within the set of normal sequences, then Shepherd concludes that an abnormal system call is detected.

Yet, for unknown applications, e.g., those unknown programs downloaded from the Internet, they do not have profiles before execution. For such applications, Shepherd simply provides an empty profile at the beginning of execution. The detector automatically collects the sequences of system calls and stores it into a sequence table. Once the table is stable enough (i.e. remains unchanged for a relative long time) and no violation is detected by the permission auditor, the profile will be used as a basis to detect anomaly system calls.

4.4 Isolation Manager

The isolation manager prevents changes made by a wild process from interfering with other normal processes before the process is identified as harmless. By tracking file system related system calls, the isolation manager uses a copy-on-write manner to shield the changes to file systems by a wild process.

Shepherd adopts the approach of isolation file systems proposed in (Sun et al., 2005). For a system call that modifies file systems, the isolation manager inspects the type and arguments of the call, and creates a shadow copy of the modified files. Then, all related read and write operations from the process is redirected to the shadow one instead of the main file system. Note that all processes derived from the process should access the shadow file system as well.

If no anomaly is detected throughout the whole execution, the shadow file systems should be integrated into the main file system. If the shepherded processes do not overlap with normal processes in file modifications, the files in the shadow file system are simply copied back to the main file system. The directories are adjusted accordingly to reflect the changes of files. The difficulty is that a normal process and a shepherded process may have modified the same file. To resolve such a conflict, Shepherd adopts a similar commit criterion with that in (Sun et al., 2005): if all read operations from wild processes happen *after* all write operations from normal processes on the files. To determine such *happen after* relationship, Shepherd maintains the last time of write operations in the main file system and compares the timestamps associated with the shadow file. If the condition is held, Shepherd merges the two versions of the file. Otherwise, Shepherd simply keeps the two versions of the file to prevent possible data loss. Yet, Shepherd renames the files modified by shepherded processes and informs administrators to make decision on handling the two versions of files.

5 Implementation

We have implemented a prototype system for Linux running on Xen-3.0.2. All components are implemented in Xen VMM, without modifications to the Linux kernel. The hardware platform is x86 architecture. We chose Xen as a base platform because of its open-source nature and robustness. Although our current implementation was specific to Linux on Xen for x86, we believe the architecture and the design of Shepherd could be similarly implemented on other operating systems, VMMs and processor architectures.

Shepherd is implemented in a highly portable and customizable way. The three main components of Shepherd are abstracted as processing phases that can be easily adjusted and replaced. Further, operators can selectively specify which component should be in use for each individual application.

The following subsections discuss the specific implementation of Shepherd. First, we present in detail the implementation of system call interposition in Xen VMM. Next, we discuss how Shepherd tracks and manages processes. Then, we provide some specific issues in implementing the shadow file system of Shepherd in Xen VMM. Finally, we present the implementation efforts and discuss the impact on TCB size.

5.1 Implementing System Call Interposition

5.1.1 Intercepting System Calls

In Linux, system calls are made through triggering the 0x80 interrupt line, i.e., *int 0x80*. In the original Xen VMM, a system call is implemented as *a direct trap* to improve performance. That is, the system call handler in Linux is installed directly into the hardware trap table (*idt_tables*). Thus, all system calls in Linux are directly caught by Linux kernel, with no interception from Xen.

To catch system calls in Xen, system calls should be implemented as the normal interrupt forwarding mechanisms. A system call should be first captured by the handler in Xen. Shepherd in Xen then does some auditing, monitoring and isolating work. Afterwards, the call is forwarded to Linux to serve it. To achieve this, we provide a system call handler in Xen and installed it into the idt_tables. The handler serves as the entry point for processing system calls in Shepherd.

5.1.2 Preventing Dangerous System Calls

To prevent unauthorized modifications to privileged resources, Shepherd uses some general rules to restrict the accesses to privileged files. In current implementation, Shepherd prohibits specific accesses to several categories of system resources for wild processes. Table 1 gives a rough description on the resources and its access types that wild processes are prevented from accessing.

Table 1: A summary of system call categories and their corresponding access permission.

Categories	Target Resources	Access Permission
Module Management	All Modules	Deny All
IPC	Normal Processes	Deny All
File Operations	System Files	Deny Write
File Operations	Devices Files	Deny Write
Process Management	PID, GID	Permit Non-root

As shown in Table 1, Shepherd prevents write accesses to system configuration files, devices files (except for some pseudo devices such as /dev/null and tty). Also, Shepherd disallows a monitored process from communicating with other processes outside the monitoring group. One exception is that IPCs between wild processes and X-server process should be allowed, or the monitored applications can not correctly function. Moreover, a shepherded process can not be granted with root permission, nor can it load or delete kernel modules. Note that the rules can easily be adjusted to use in different scenarios. Shepherd also allows installing multiple auditing policies and changing it dynamically for different processes. Shepherd audits a system call by examining its system call number and the corresponding arguments.

As pointed by Garfinkel (Garfinkel, 2003), indirect paths and symbolic links may fool a naively designed detecting system. For indirect paths, Shepherd handles this by maintaining the current directory of each running process, and always uses a full path name for auditing. For symbolic links, Shepherd tracks the symbolic related system calls and maintains the relationships between a file and its links.

5.1.3 Detecting Abnormal System Calls

As mentioned before, system sequences are used to detect possible violations. Normal system sequences are stored in a *profile* file (Somayaji and Forrest, 2000). A Profile is essentially a table of system call pairs that describe permitted system calls. The profile is loaded into memory during a process being executed with a new image.

A newly intercepted system call is compared with the profile, together with recent system calls. Shepherd uses a matching algorithm through comparison in *pair wise*. Shepherd mains a window of recent system calls for each process. For comparison in pair wise, Shepherd combines current system call and all previous ones in the window in pairs, and searches the pairs in the profile. The distance between system calls in the window is also accounted as a matching criterion. If no match is found for the search, Shepherd concludes that an unexpected system call is detected. Note that Linux kernel could also made system calls via *int 0x80* and will be captured by Shepherd. Such system calls disturb the sequence of normal system calls. Thus, Shepherd only considers system calls made from user mode.

Upon detecting an anomaly, Shepherd terminates the offending process or suppresses the violation and silently returns the processes with a successful return code. The latter case is important if operators need to dig deep into the runtime behaviors of an untrusted application. In both cases, Shepherd logs such violation and saves the execution context to help operators to diagnose the problem (e.g. intrusion analysis).

5.2 Running Wild Processes

Users run wild processes by simply appending an additional argument namely – *shepherd mode* to the command line arguments. The mode arguments specify the shepherding policies, which is a combination of the three components of Shepherd. Shepherd inspects the arguments passed to *sys_execve*, resolves the third argument and find if it contains *–shepherd*. To efficiently support multiple untrusted processes, Shepherd manages wild processes in groups. All processes spawned by a shepherded process should be shepherded as well and inherit the shepherding policies from its parent. All offspring processes are in the same group with the first shepherded process. Processes in the same group share the same view on system resources, e.g., share the same shadow file system.

5.2.1 Managing Shepherded Processes

To manage wild processes, Shepherd needs to efficiently track each process in Linux. Here, the page directory is used to uniquely identify a process, as that in Antfarm (Jones et al., 2006). Shepherd maintains a control structure for each process. The control structure contains a window of logged system call, a profile describing the normal system call sequence and opened files, some control information on the shadow file system, among others. All process creations, terminations and context switches are caught by Shepherd, which allocates, deallocates and switches the process control structure. In Linux running on x86, process creations and exits are done through invoking *sys_fork* and *sys_exit* (as well as *sys_exit_group*) accordingly. A process context switch happens if there is a write operation to the page table root installed in CR3.

Shepherd tracks the *sys_fork* and *sys_execve* to maintain the parenthood among wild processes to maintain shepherding groups. The profile for a process is loaded during a *sys_execve* call. A child inherits the profile and the logged system call sequences from its parent process. However, if the child loads a new executable image via *sys_execve*, the inherited control structure is discarded and a new structure is created according to the new profile. During a context switch, i.e., a new page directory is loaded into *CR3*, Shepherd queries the new value in the process list and marks the corresponding process control structure as in use.

5.2.2 Regaining Control after System Calls

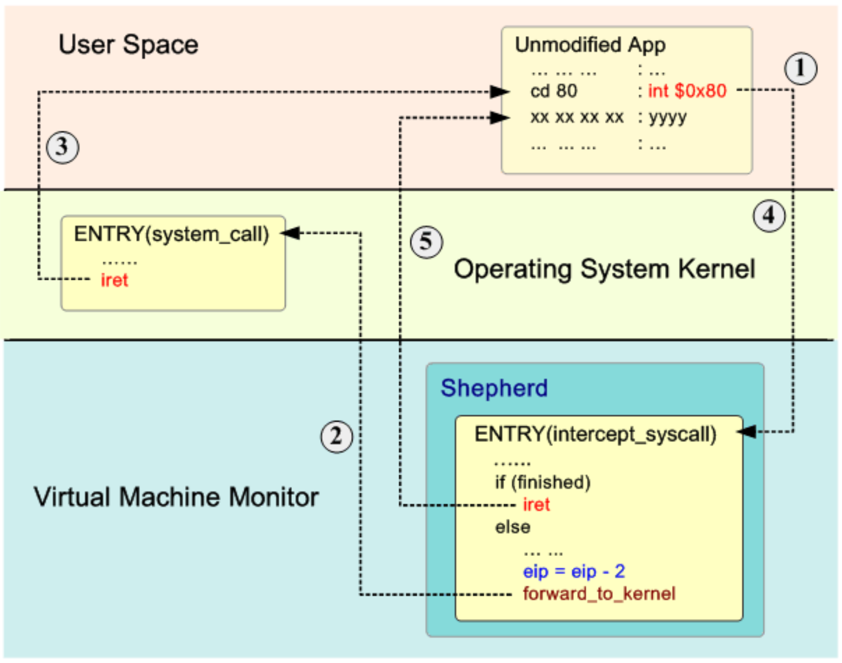

Figure 4: Regaining control after a system call. (1)a user process invokes a system call. (2) Shepherd adjusts the *eip* in the faulting context and forwards the system call to the kernel. (3)the process resumes execution in user mode after the system call is served. (4)the process re-executes the *int 0x80* instructions that traps to the VMM. (5)Shepherd does some additional work and resumes execution directly to user space.

In some cases, Shepherd needs to intercept the execution both before and after a system call. For example, to track the parenthood among processes, Shepherd needs to regain control right after a *sys_fork*. As shown in Fig. 4, Shepherd achieves this by modifying the execution context upon catching a system call. The faulting programming counter (EIP) is modified to point to the previous instructions of faulting EIP, which is the instruction that issues the system call (i.e. *int 0x80*). Thus, a system call is reissued again after the process resumes its execution in user space. Shepherd catches the system call and does some additional work here. Finally, Shepherd resumes the process by directly returning from Xen to user space through *iret*. For the case to track parenthood among processes, Shepherd only needs to examine the return value (in *EAX*) and the *EIP* of the *sys_fork*. If the return value is zero and the *EIP* is the invoking *EIP*, then the executing process is a child process of the process invoking *sys_fork*. Shepherd allocates a new process control structure for it and resumes its execution.

The above technique causes additional domain crossing among the user space, the kernel space, and the VMM, which may incur a little performance degradation. It may be more straightforward to modifying the operating system kernel to do these things. However, as one of the goals of shepherd is to retain OS-transparency, we believe it is valuable for the incurred performance overhead. Moreover, according to our performance benchmark, the overall performance degradation is still modest.

5.3 Implementing a Shadow File System

To prevent wild processes from tainting the system, Shepherd only grants them with read accesses to the main file system. For write accesses, Shepherd creates a shadow file system in a copy-on-write manner. A shadow file system is created in a group base. Processes from the same shepherding process access the same shadow file systems. Only if all processes in a group have finished its execution, can the shadow file systems been merged back to the main file system.

Implementing a shadow file system in VMM poses additional challenges as opposed to that in user space.

Table 2: A breakdown of code size in lines of code for various components in Xen: PA, AD, IM, MISC are short for Permission Auditor, Anomaly Detector, Isolation Manager and other miscellaneous code accordingly.

Components	PA	AD	IM	MISC	Total
LOCs	185	1,600	1,410	375	3, 570

A VMM usually provides little abstraction on system resources such as file systems. Shepherd resolves this by partitioning the role between the operating system and the VMM. The VMM is mainly responsible to manage and organize control commands, while the operating system serves the control commands. Thus, no additional interface is required to be added to the operating system kernel, since the VMM only relies on existing kernel services. This is possible because logically the VMM and user space can invoke the equivalent services from the kernel, although in different levels. The VMM only implicitly issues system calls during regaining control before and after a system call, as described in section 5.2.2. These system calls are called *compensative system calls*.

Shepherd mainly relies on compensative system calls to implementing a shadow file system. For example, on intercepting an open system call with write permission, Shepherd may need to issue a number of system calls to copy the original files to the shadow file system to create a shadow file, and redirects the write system calls to the shadow file. To copy a file, Shepherd may need to issue a number of system calls including sys_open, sys_read, sys_write, sys_stat, sys_chmod and sys_close. However, these do not incur additional performance degradation compared to those implemented in user mode, as they require exactly the same number of system calls.

To manage the relationships between isolation file system and main file system for wild processes, Shepherd needs to track the mapping between a file in the main file system and that in the shadow file system. One naive approach is to maintain a table containing the original file name and the new file name (the full path is included), which is rather space-consuming and incurs a high overhead to search the mapping (e.g. a number of memory comparisons using *memcmp* are required.). To reduce such overhead, Shepherd only maintains a hash table recording the hash of the file names whose files has already in the shadow file system. On opening a new file with read or write permission, a compared-by-hash is used to find the hash of the file name in the hash table. If it is found, then Shepherd should open the file in the shadow file system. Otherwise, the file in the main file system should be opened instead. It is possible that a collision may happen, that is, a file is opened as a shadow file even if it not in the shadow file. To handle the case, Shepherd regains control over all open operations to the shadow file system, as described in section 5.2.2. If the open fails, Shepherd will reissue it again to the main file system. Note that it is disallowed for any open operation with write permission to the main file system. If it does happen, Shepherd handles it in a copy-on-write manner. Compare-by-hash tends to be efficiency because a collision rarely happens.

5.4 Implementation Effort and Impact on TCB size

Implementing Shepherd in a VMM inevitably increases the size of trusted computing base (TCB) in the system. Our implementation only keeps the security critical parts such as auditing and detecting functionalities in a VMM. Most non-security-critical functionalities such as the shadow file system mostly reuse the code of operating system kernel. Thus, the brought code size expansion is really small.

Table 2 provides a breakdown of the total code added to Xen, in non-comment lines of code (LOCs). The total added code is about 3,570-LOCs, which is rather small compared to about 212,000-LOCs for the total code in Xen. Moreover, the code for process shepherding is modularly grouped and has little interference with other code. Thus, Shepherd incurs only a small amount of implementation complexity, thus has little impact on the TCB of the system.

5.5 Discussion and Future Work

Ascertain the Correctness of OSes: Assuming the untrustworthiness of OS kernel may make the processes vulnerable to denial-of-service (DoS) attacks. As Shepherd still relies on parts of the functionalities in the OS kernel, this may cause a problem called liability inversion (Heiser et al. , 2006). This is in essence a general problem for system assuming the untrustworthiness of OS kernels.

To ascertain execution *correctness*, the VMM may need to constantly verify if the OS kernel is correct functioning. Upon system call interposition, Shepherd can randomly and automatically issues some verifying system calls (VSCs) to ascertain that an OS kernel can provide correct services. There can be two types of VSCs: (1) context-aware VSCs, that are dependent with the process execution context. For example, after a process writes some chunks to a file, the VMM can selectively keep the data and issues a read system call to fetch the data back and make a comparison. (2) context-free VSCs, that are self-contained call sequences that are independent of execution context. For instance, a VMM can issue a system call trace that creates a file, writes some data, fetches data back and then delete the file. The intermediate state is checked to see if the file-system related system call services are correct. We intend to integrate this scheme to Shepherd in our future work.

Accuracy of Shepherding: Shepherd aims to provide a framework that uses VMM as a security building block to shepherd wild processes. The current implementation may have several limitations and restrictions. For example, the ability of the Shepherd directly relies on the accuracy of the permission auditor and anomaly detector. Inaccurate specifications may allow malicious applications to pass the auditing. A poor-quality profile may either incur false positives or false negatives. However, we believe our main goal is to provide a framework to monitor processes. More sophisticated and accurate policies and detectors could be similarly integrated into Shepherd to make it more powerful.

Networking Applications: There are some limitations for Shepherd in monitoring networking applications. Shepherd only considers isolating and monitoring modifications to the local system. On detecting an anomaly, it discards changes to local file system and kills the offending process. However, for network operations, Shepherd can not discard their effects since it requires coordination among each communicating machines. Completely disallowing network operations is not a good solution either, because it limits the applications of Shepherd to monitor networked applications. It may be more suitable to log the network operations and let operators to decide if they are malicious or not.

OS-transparency and Performance: At present, our implementation aims to maintain the OS-transparency, at the cost of some possible performance overhead, for example, a lot of cross-domain switches. Making the kernel more cooperative will surely reduce such overhead, yet at the cost of performance degradations. Currently, we only implement our system in the former approach.

Isolation of Logs and Shadow File System Currently, Shepherd writes the logs and some security-critical information directly into the file system of the operating system where a shepherded process executes. The shadow file system also lies within the shepherded operating system. Thus, they are visible and may be vulnerable to malicious attackers. One way to secure them is to create a private virtual machine and redirect all such operations to another private virtual machine, similar to the approach used by Proxos Ta-Min et al. (2006). Such an approach can increase the level of isolation and security to a higher extent.

Automatic Checkpoint and Restart: On detecting an anomaly, Shepherd simply discards the changes to file systems or suppresses it and returns silently. Such a policy is appropriate for trying untrusted applications or unknown code. However, for monitoring sever applications during production runs, the policy may cause absence of services. Recent proposals demonstrate that rollback and retrying are promising approaches for applications to survive from transient anomalies and even attacks Qin et al. (2005). We intend to provide Shepherd with the ability of process checkpoint and restart. Shepherd periodically take a checkpoint on a running process. On detecting an anomaly, Shepherd automatically rollback the offending process to a recent checkpoint and restarts it again. If the anomaly happens again, Shepherd then kills the offending process or returns silently.

Checkpoint can be rather straightforward implemented in VMM. General VMMs use shadow page tables Waldspurger (2002) for memory virtualization. In shadow page table mode, a VMM can intercept any write accesses to the memory of a process, during which the VMM can make a shadow copy of the original page. Thus, checkpoint the in-memory state can be done using copy-on-write on the pages of the process. Such checkpoint is completely transparent to operating systems and incurs very little disruption on the running process. For file systems checkpoint, Shepherd can also use a copy-on-write approach to modifications to files. Shepherd creates a new copy of a file on any write accesses to the file.

6 Experimental Evaluation

In this section, we evaluate both the functionalities and the performance of Shepherd, using several real-life applications and benchmarks.

6.1 Security Test

We used several recent real vulnerabilities to test the effectiveness of Shepherd to detect possible attacks. To measure the effectiveness of each components in Shepherd, we run the vulnerable applications with each component being *selectively* turned on. As shown in Table 3, all these attacks are detected or tolerant, by some or all components of Shepherd.

As other intrusion detection systems, Shepherd also needs to balance false positives against false negatives. The rate of false negatives directly relies on the accuracy of the anomaly profiles, and an accurate profile can effectively reduce false negatives. For false negatives, since our current anomaly detector is based on system-call sequence, it may be vulnerable to mimicry attacks (Wagner and Soto, 2002). However, although the isolation manager essentially can not detect attacks or intrusions, it can isolate the modifications made by adversaries. Thus, it prevents adversaries from successfully attacking the system by "tolerating" them.

For GNU Tar 1.13, there is a vulnerability allowing an adversary to replace normal files (e.g. .bashrc) with malicious files in the hosting system. All three components in Shepherd can detect or tolerate such an attack. The permission auditor detects it because the attack program needs to invoke *sys_symlink* to link an directory with the root directory "/". The anomaly system call sequence for the attack is "*sys_mmap2, sys_read, sys_write, sys_munmap,* **sys_symlink**, *sys_open, sys_fcntl64, sys_fstat64*", because in normal operations the *sys_symlink* is only performed after file or directory operations such as *sys_open* and *sys_mkdir*.

Xpdf 3.0.1 is vulnerable to multiple remote buffer-overflow attacks due to its absence of checking the boundary of user-supplied data. As Fedora core 2 uses address space randomization to harden buffer overflow attacks, we currently did not succeed in forcing it to execute our supplied code, but only caused it to run in an infinite loop with a denial of service. The permission auditor failed to detect such attack because it does not involve in accessing to privileged resources. The anomaly sequence is "*sys_llseek, sys_fstat64, sys_llseek,* **sys_llseek, sys_write,** *sys_llseek, sys_read, sys_fstat64*", because there is no sequence with *sys_llseek* before *sys_write*, but only that *sys_llseek* before read in normal operations. After such an anomaly sequence, the program entered into an infinite loop executing *sys_llseek*, thus Shepherd should gracefully terminate the offending process.

Sendmail 8.13.6 is susceptive to remote code-execution attacks due to a race condition in asynchronous signal handling. The anomaly detector found that the *sys_select* and *sys_umask* are not within the normal system call sequences when combining them with previous system calls. As the case for a2ps 4.13b, it is prone to a command execution attack since it can execute the commands embedded in the file name passed to a2ps. As executing the embedded commands requires a2ps invoking *sys_execve*, our anomaly detector finds that the root cause for the attack is the sequence *sys_waitpid*, *sys_rt_sigprocmast,sys_rt_sigaction*, *sys_rt_sigaction*, *sys_rt_sigaction*, *sys_rt_sigaction*, **sys_execve**, *sys_rt_sigprocmast.*

For all these vulnerabilities, even if the anomaly detector fails to detect these attacks (we simulated this situation by turning off the permission auditor and anomaly detector), the isolation manager can still tolerate the attack since the changes to files can only be placed in the shadow file system. Users are aware of the files added to the system when the shadow file system is integrated to the main file system.

Table 3: Several recent real vulnerabilities tested with Shepherd. The "Results" columns indicate whether the three components in Shepherd can detect or tolerant the attacks. PA, AD and IM are short for Permission Auditor, Anomaly Detector and Isolation Manager, accordingly.

Application	BID	Class	Vulnerability	Testing Type	Results		
					PA	AD	IM
tar 1.13	21235	Input Validation Error	Directory Traversal	Remote	Y	Y	Y
Xpdf 3.01	21910	Boundary Condition Error	Mult. Remote Buffer Overflow	Local	N	Y	Y
sendmail 8.13.6	17192	Race Condition Error	Code Execution	Remote	N	Y	Y
a2ps 4.13b	11025	Input Validation Error	Command Execution	Local	N	Y	Y

6.2 Performance Evaluation

One important issue for Shepherd is that it should not incur too much overhead to the shepherded applications. Meanwhile, it should have little performance impact on normal applications.

The performance of Xen against native operating systems has been well studied in (Barham et al., 2003), and the measurements indicate that Xen only-incurs very little performance degradation. To measure the incurred overhead, we compared the performance of several applications running on the following systems and configurations: Unmodified Xen-Linux (X-Linux); normal processes in Xen-Linux running on Shepherd (S-Normal); shepherded processes on Xen-Linux running on Shepherd. Further, to get the detailed distribution of overall overhead incurred by Shepherd, we provided the performance for the three configuration of Shepherd: only permission auditor is turned on (S-Audit); both the permission auditor and the anomaly detector are turned on (S-Both); all three components are turned on (S-All).

All the experiments were conducted on a system equipped with a 3.0GHz Pentium IV with 1GB RAM, a Intel Pro 100/1000 Ethernet NIC in a 100M LAN, and a single 250GB 7200 RPM SATA disk. The version of Xen-Linux is 2.6.16 and the version of Xen VMM is 3.0.2. The Fedora Core 2 distribution was used throughout. It is installed on ext3 file system. We configured 900,000KB of memory for Linux.

Several benchmarks were used to characterize the performance overhead of Shepherd. We used two types of benchmarks: (1) application-level benchmarks to study the performance overhead of Shepherd against other mentioned systems. (2) micro-benchmarks to gain more precise performance results of some particular subsystems in all the test systems.

6.2.1 Application Benchmarks

Table 4: Test methodologies for the three applications.

Apps	connection time	transfer rate
vsftpd	average time of requesting 1,000 empty files using *wget*	download rate of a single 222MB file
sshd	total elapsed time of 1,000 requests divided by 1,000	use *scp* to copy a single 138MB file
Apache httpd	use ab (apache benchmark) with "ab -n 50000 -c 500 http://localhost/apache_pb2.gif"	

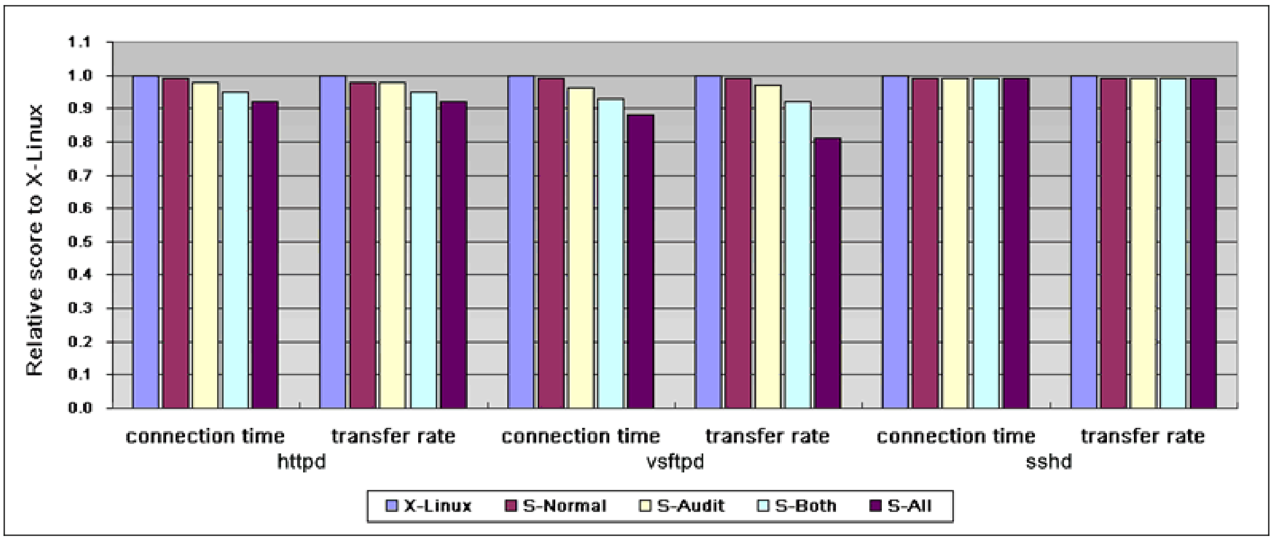

Figure 5: Connection time and throughput for the three server systems.

For application benchmarks, we tested three prevalent server software: the Very Secure FTP daemon (*vsftpd*), which is the *de facto* FTP server in UNIX environments; the ssh daemon (*sshd*) from the OpenSSH suite, which is a widely-used secure shell daemon; the *apache* HTTP server (*httpd*) , which is a most

Table 5: lmbench system call results - time(ms).

System calls	X-Linux	S-Normal	S-Audit	S-Both	S-All
null	0.32	0.59	0.59	0.74	0.76
read	0.42	0.71	0.72	0.87	0.90
write	0.37	0.67	0.67	0.82	2.0
open	2.43	2.90	2.92	3.24	*16.03*
state	1.63	1.83	1.84	2.02	2.03

prevalent HTTP server used nowadays. For the three applications, we measured their performance using two metrics: connection time and transfer rate. The test methodologies are shown in Table 4. To measure the performance of apache, we used *ab* (apache benchmark) to issue 50,000 requests for a single 2.4 KB file, with 500 simultaneous threads.

As shown in Fig. 5, Shepherd incurs only undetectable performance overhead to applications not shepherded. For applications shepherded with only permission auditor and anomaly detector, the performance overhead is rather small. The incurred performance overhead by isolation manager is somewhat larger as it requires extra file operations. Also, the incurred performance overhead for *vsftpd* is relative higher than *sshd* and *httpd* since it involves much more file system operations.

6.2.2 Micro Benchmarks

For micro-benchmarks, we used lmbench (McVoy and Staelin, 1996) benchmark to understand the low level overhead relative to each individual system call. Table 5 shows the decomposed overhead incurred by Shepherd. As shown in the table, for normal processes running on Shepherd, the incurred overhead is relative small, which is mainly incurred by system call interposition. The isolation mode of Shepherd incurs a relative large overhead because Shepherd has to create new files in a copy-on-write fashion and maintain the relationship between the main file system and shadow file system. For example, to open a existing file from the main file system in write mode, Shepherd needs to create a shadow copy of the file in the shadow file system. This involves file creation and copying, which tend to be time-consuming. Thus, the incurred overhead for read is relatively high. However, we believe it is valuable for trying new software in Shepherd. Further, the amortized overhead in application software is relatively small. In Shepherd, operators can also freely specify whether such isolation is turned on for each application.

7 Conclusion

This paper has described a VMM-based process shepherding system to monitor and detect malicious actions made by wild processes. Shepherd integrates three techniques as security building blocks to prevent, detect and isolate possible intrusions, attacks and misbehavior from adversaries. Compared to existing kernel-level and user-level monitoring system, Shepherd shows its advantages in superior security and OS-transparency. Currently, Shepherd requires no modifications to operating system thereon, resulting in good portability and maintainability. To demonstrate the effectiveness of Shepherd, we have tested several recent real-life vulnerabilities and attacks against Shepherd. According to our measurements, Shepherd is resistant to all these attacks, through prevention, detection and isolation the effects of these attacks. Performance measurements indicate that the overall incurred performance overhead is still relatively small. We plan to further enhance Shepherd to make it more robust and powerful, yet more efficient, through adding and trying different design and implementation strategies.

References

Acharya, A., Raje, M., 2000. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In: Proceedings of the 9th USENIX Security Symposium, pp. 1-17.

Asrigo, K. et al., 2006. Using VMM-based sensors to monitor honeypots. In: Proceedings of the 2nd international conference on Virtual execution environments, pp. 13-23.

Barham, P. et al., 2003. Xen and the art of virtualization. In: Proceedings of the 19th ACM symposium on Operating systems principles, pp. 164-177.

CERT Coordination Center, 2006. Available from: <http://www.cert. org>.

Chen, P.M., Noble, B.D., 2001. When virtual is better than real. In: Proceedings of 8th Workshop on Hot Topics in Operating Systems (HOTOS-VIII), pp. 133-138.

CVE-2006-0038, 2006. Linux kernel netfilter do_replace local buffer overflow vulnerability. Available from: <http://cve.mitre.org/cgi -bin/cvename.cgi?name=CVE-2006-0038>.

Dunlap, G.W. et al., 2002. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In: Proceedings of the 2002 Symposium on Operating Systems Design and Implementation, pp. 211-224.

Efstathopoulos, P. et al., 2005. Labels and event processes in the as-bestos operating system. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP-20), pp. 17-30.

Forrest, S. et al., 1996. A sense of self for Unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp. 120-128.

Gao, D. et al., 2004. On gray-box program tracking for anomaly detection. In: Proceedings of 13th Usenix Security Symposium, pp. 103-118.

Garfinkel, T., 2003. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In: Proceedings of the Network and Distributed Systems Security Symposium.

Garfinkel, T., Rosenblum, M., 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the 2003 Network and Distributed System Security Symposium(NDSS).

Goel, A. et al., 2005. The taser intrusion recovery system. In: Proceedings of the 20th ACM sysmposium on Operating systems principles, pp. 163-176.

Goldberg, R.P., 1974. Survey of Virtual Machine Research. IEEE Computer 7(6), 34-35.

Hu, W. et al., 2006. Secure and practical defense against code-injection attacks using software dynamic translation. In: Proceedings of the 2nd international conference on Virtual execution environments, pp. 2-12.

Hsu, F. et al., 2006. Back to the Future: A Framework for Automatic Malware Removal and System Repair. In: Proceedings of the Annual Computer Security Applications Conference, pp.257-268.

Jones, S.T. et al., 2006. Antfarm: Tracking Processes in a Virtual Machine Environment. In: Proceedings of the USENIX Annual Technical Conference (USENIX'06).

Keahey, K. et al., 2004. From sandbox to playground: dynamic virtual environments in the grid. In: Proceedings of 5th IEEE/ACM International Workshop on Grid Computing, pp. 34-42.

Kiriansky, V. et al., 2002. Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, pp. 191-205.

Loscocco, P., Smalley, S., 2001. Integrating Flexible Support for Security Policies into the Linux Operating System. In: Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference table of contents, pp. 29-42.

McVoy, L., Staelin, C., 1996. lmbench: Portable tools for performance analysis. In: Proceedings of Usenix Technical Conference, 1996.

Securityfocus, 2007. Windows vista voice recognition command execution vulnerability. Available from: <http://www.security focus.com/bid/22359/>.

Somayaji, A., Forrest, S., 2000. Automated Response Using System-Call Delays. In: Proceedings of the 9th USENIX Security Symposium, pp. 185-198.

Sun, W. et al., 2005. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In: Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS), pp. 265-278.

Wagner, D.A., 1999. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056. University of California, Berkeley.

Wagner, D.A., Soto, P., 2002. Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM conference on Computer and communications security, pp. 255-264.

Yu, Y. et al., 2006. A feather-weight virtual machine for windows applications. In: Proceedings of the 2nd international conference on Vritual execution environments, pp. 24-34.

Zeldovich, N. et al., 2006. Making Information Flow Explicit in HiStar. In: Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation, pp. 279-292.

Waldspurger, C.A., 2002. Memory resource management in VMware ESX server. In: Proceedings of the 5th symposium on Operating systems design and implementation. pp. 181–194.

Qin, F. et al., 2005. Rx: treating bugs as allergies—a safe method to survive software failures. In: Proceedings of the 20th ACM symposium on Operating systems principles. pp. 235–248.

Ta-Min, R. et al., 2006. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation pp. 279–292.

Heiser, G. et al., 2006. Are virtual-machine monitors microkernels done right?. ACM SIGOPS Operating Systems Review 40(1), pp. 95–99.