Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor

Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang
{hbchen,fzzhang,chengchen,ziyeyang,chenrong,byzang}@fudan.edu.cn
Parallel Processing Institute, Fudan University

Pen-chung Yew
yew@cs.umn.edu
Department of Computer Science and Engineering
University of Minnesota at Twin-Cities

Wenbo Mao
mao_wenbo@emc.com
EMC China Research

Parallel Processing Institute Technical Report
Number: FDUPPITR-2007-08001
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

Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor

Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang
{hbchen,fzzhang,chengchen,ziyeyang,chenrong,byzang}@fudan.edu.cn
Parallel Processing Institute, Fudan University

Pen-chung Yew
yew@cs.umn.edu
Department of Computer Science and Engineering
University of Minnesota at Twin-Cities

Wenbo Mao
mao_wenbo@emc.com
EMC China Research

Fudan University, Parallel Processing Insitute, PPITR-2007-08001

August 2007

Abstract

Software applications today face constant threat of tampering because of the vulnerability in operating systems and their permissive interface. Unfortunately, existing tamper-resistance approaches often require non-trivial amount of changes to core CPU architectures, operating systems and/or applications.

In this paper, we propose an approach that requires only minimal changes to existing commodity operating systems running on commodity hardware without compromising their functionality and compatibility. The key idea is to use a trustworthy virtual machine monitor (VMM) to monitor and regulate the behavior of other untrustworthy processes including the underlying operating system that might have been compromised. We use the trusted VMM to compartmentalize a process that demands tamper resistant protection from OS kernels and other processes, by interposing security-sensitive operations (e.g. system calls) and isolating (and sealing) security-sensitive information (e.g. registers and memory).

We have implemented a working prototype, CHAOS [1], that supports tamper-resistant applications running on Linux and Xen VMM. Our prototype shows that it only requires minor changes (about 230 LOCs) to Linux and a small amount of code expansion to Xen (about 4200 LOCs). Performance measurements also show that CHAOS incurs a little performance degradation to the application software: about 3% for SPECINT-2000 and less than 15% for apache httpd and vsftpd.

1 Introduction

1.1 Motivation

Tamper-resistant software is vitally important to combat many challenges facing computer industry today [9, 16, 26]. Examples include software copyright protection, digital right management (DRM), secure remote execution (e.g. grid computing) and software security.

It has been noted that an untampered software execution should have at least the following two properties [23]: (1) *authenticity*: the code under execution should be authentic, and should not have been altered or changed. (2) *integrity*: runtime states (e.g. CPU registers, memory and sensitive I/O data) should not have been tampered with. To facilitate DRM and copyright protection, we believe that an additional property is also needed, i.e. (3) *privacy*: code, data and runtime states should not be observable to unauthorized processes or even underlying OS that might have been compromised.

Providing tamper-resistant protection can hardly be achieved without the support of operating systems. Unfortunately, despite continued efforts to improve modern operating systems, they are still essentially

[1] CHAOS stands for *c*onfidentiality and *h*igh-*a*ssurance equipped *o*perating *s*ystems.

untrustworthy in two aspects. First, they are big, complex and often developed using unsafe languages, thus are inherently error prone. They could also be tampered with or penetrated due to design flaws, security vulnerabilities and implementation bugs (e.g. [6, 22]). Second, they allow poor isolation among processes due to permissive OS interface [15, 8]. For example, some processes are often granted with high privileges, yet can easily be tampered with due to security vulnerabilities [4]. A tampered process with the root privilege can easily access private data and tamper with the execution of other processes. Meanwhile, using specially tailored operating systems can only have very limited success due to their restricted functionalities and compatibility to existing applications.

To remedy this situation, there have been numerous efforts aimed at providing high-assurance execution environments. Generally, those efforts can be classified into three categories. First, there are architectural enhancements [17, 26, 7] aiming to provide a private, tamper-resistant execution environment for high-assurance applications. XOMOS [16] is such a system. It utilizes architectural enhancements to support trustworthy applications running on an untrustworthy operating system. However, this approach requires nontrivial changes to the core processor architecture (e.g. tagging registers and caches, as well as adding crypto units). There are currently no commercially available processors supporting such functionalities.

Second, VMM-based machine partitioning schemes, such as Terra [9] and NGSCB [20, 12], try to solve the problem by multiplexing commodity OSes and specialized OSes on the same hardware platform. They rely on the specialized OSes to provide tamper-resistant capabilities. Essentially, they provide *all-or-nothing* trustworthiness at the operating system level: an operating system and its applications are either completely trustworthy or not at all. Applications that demand tamper-resistant protection can only run on specialized trustworthy OSes with restricted functionalities [2]). It may also require modification to the applications.

Third, micro-kernel based approaches [24, 25] try to reduce their trusted computing base (TCB) by running only limited code in the privileged mode. However, they require a redesign of operating systems and/or applications. There, two design options are usually available: (1) Allow the micro-kernel to retain the same interface of commodity OSes. However, this may give a hostile process the same permissive interface to tamper with other processes; or (2) Redesign the existing interface to the micro-kernel that provides tamper-resistant capability. However, it will then require nontrivial modifications to port existing applications [25].

1.2 Our Contributions

In this paper, we propose a VMM-based tamper-resistant scheme, called CHAOS, which could *transparently* provide a *tamper-resistant* execution environment using commodity (thus untrustworthy) kernels, to host *existing* applications that demand tamper-resistant protection. In contrast to existing systems, it does not require changes to existing processor architectures. It also avoids using specialized OSes to provide tamper-resistant capabilities, and it preserves the same OS interface to retain backward compatibility for existing applications.

The key idea is using a trusted VMM to monitor and regulate the behavior of its guest operating systems. It *compartmentalizes* an application that demands tamper-resistant protection from the kernels and other applications at runtime, thus preserves the *privacy* and *integrity* of the application. The VMM also uses cryptographic approaches to detect possible tampering of the code to ensure its *authenticity*.

Privacy and Integrity: In CHAOS, applications that demand tamper-resistant protection are executed as *trusted processes*. They are resistant to both inspection and tampering from other processes and even from the untrustworthy OS kernel. To achieve this, the trusted VMM *interposes* all kernel/user interactions such as system calls (section 2.2.1). Using interposition, CHAOS could then have the capabilities to *isolate* CPU context and memory owned by a trusted process by concealing them from the OS kernel and other processes (section 2.2.2). Also, I/O data owned by a trusted process are encrypted before being transferred to the OS kernel (section 2.2.3).

Authenticity: CHAOS uses cryptographic approaches to prevent an OS kernel from running alternative or modified code, as done in XOMOS [16]. The code and data in a trusted application are both encrypted using the available public key of the platform. They can only be decrypted with the assistance and attestation of the VMM since only the VMM knows the private key to decrypt them. The encrypted hash is used to

[2]Terra's architecture requires a high-assurance OS to be run as a close-box VM. While there is no such high-assurance OS, its prototype uses Linux to run high-assurance applications, which should obviously not be trusted.

verify the integrity of the application. Hence, the VMM can attest to the authenticity of the code, and even a compromised OS kernel cannot tamper with the code.

Normal (i.e. untrusted) processes that did not request tamper-resistant protection can co-exist with trusted processes in the OS kernel, yet with little or no intervention from the trusted VMM. Processes are uniquely identified by their page table root, as done in Antfarm [13]. In essence, CHAOS makes no restriction on which applications can execute as trusted processes, thus tamper-resistant.

We have implemented a prototype system using the Xen VMM [2] to provide a tamper-resistant environment on Linux. We have also conducted an experimental measurement on the performance of CHAOS using a set of benchmarks and real-life applications that demand tamper-resistant protection. The performance degradation for SPECINT-2000 is within 3%. The incurred overhead for two prevalent server software applications (vsftpd and apache httpd) is within 15%.

The rest of the paper is organized as follows. Section 2 describes the trust model as well as an overview of the system architecture of CHAOS. Then, we describe the implementation details of the prototype system in Section 3. Section 4 follows with an experimental evaluation on the relative performance of CHAOS. Section 5 compares CHAOS with existing systems. Finally, this paper ends up with a concluding remark in section 6.

2 CHAOS Overview

In this section, we first describe our trust model. Then, we present the overall system architecture.

2.1 Trust Model

The trust model in CHAOS is derived mainly from the perspective of user applications. That is, it focuses on parts in the hardware/software stack that could be trusted (or not trusted) by applications. Once the trustworthy parts are identified, CHAOS uses cryptographic approaches to enforce authenticity, integrity and privacy between applications and these parts (section 3.4.2).

As a VMM is much smaller and simpler than an OS kernel, it is reasonable to include VMM into the trusted computing base (TCB) [9, 20]. CHAOS, thus, chose not to trust the operating systems, but to trust an enhanced VMM and the underlying hardware instead. Hence, it can fend off sophisticated attackers even after they have gained full control of a compromised operating system. The integrity of the VMM could be ensured by authenticated boot using TPM (Trusted Platform Module) from the Trusted Computing Group (TCG) [28]. This could prevent an adversary from installing a rootkit below the VMM or launching a malicious VMM.

For practicality and cost-effectiveness, CHAOS does assume the trustworthiness of the underlying hardware (such as CPU, cache and memory). However, peripheral I/O devices are not trusted because they are shared among operating systems and processes and thus are vulnerable to exposure (i.e. violation of privacy) and tampering. From a user's viewpoint, we need to treat all machine owners as possible adversaries in order to enable copy protection and secure remote execution.

It is important to note though that CHAOS assumes that applications can correctly execute provided that the OS kernel is well-behaved. Hence, similar to other approaches, assuming the untrustworthiness of the operating system [16] and other legacy code [25] may subject trusted applications to denial of service attacks. A bug or a compromise in the OS kernel could also prevent trusted applications from functioning correctly, yet no secret in the applications can be divulged. CHAOS, like other approaches, makes no assumption that a trusted application is not buggy or malicious. CHAOS only ensures that a buggy or malicious application running as a trusted process cannot tamper with the execution of other trusted processes.

2.2 Approach Overview

CHAOS transparently creates a tamper-resistant environment by implementing a *trust management layer* in the VMM. This layer intercepts and monitors control and data transfer among trusted processes, the OS kernel and the underlying hardware. It ensures that the trusted processes cannot be observed and tampered with by a compromised OS kernel and untrustworthy applications.

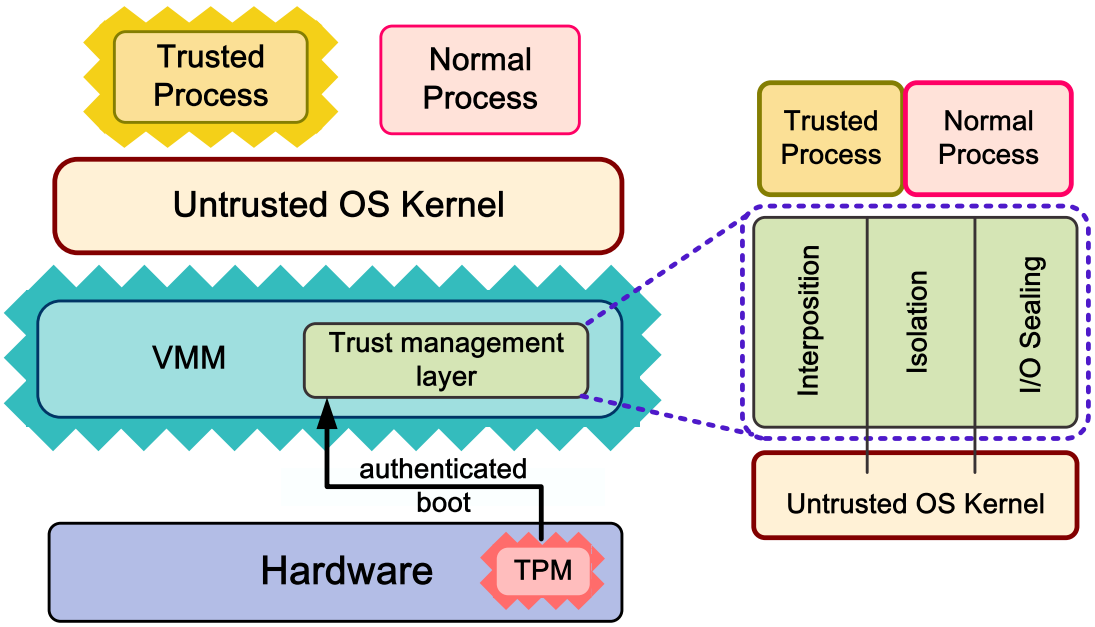

Figure 1: The physical and logical view of CHAOS, which is essentially a trust management layer that interposes kernel/user interactions.

An important property of this layer is *transparency*, that is, it must retain *backward compatibility* so existing applications could run without changes on code to accommodate this layer. Another property is that it is *mandatory* that no OS kernel, malicious or not, can bypass this layer. Some modifications to an OS kernel are needed (about 230 LOCs for Linux) to make it work with the trusted VMM in CHAOS. A malicious OS kernel could refuse to cooperate with the layer, but it cannot tamper with a trusted process.

Figure 1 depicts the overall system architecture of CHAOS. Unlike the conventional (i.e. physical, in the left figure) view of a VMM, CHAOS logically treats it as a *trusted layer* between an OS kernel and user processes, because a VMM is capable of intercepting all privileged operations and kernel/user interactions in an operating system.

CPU context	conceal it from OS kernel
memory pages	conceal user mappings from OS kernel
	track page usages to prevent unauthorized mappings
persistent data	encrypt it before flowing to OS kernel

Table 1: Methodologies to protect sensitive information.

For a trusted process, the sensitive information to be protected can be classified into three categories: CPU execution context, memory pages and I/O data. To protect that information, CHAOS mainly relies on three mechanisms: *interposition*, *isolation* and *I/O sealing*. Table 1 provides an overview of these three approaches. We describe each approach below.

2.2.1 Interposition

The interposition mechanism ensures that: (1) all user/kernel interactions pass through CHAOS trust management layer, (2) all data transfers between trusted processes and OS kernels are mediated by CHAOS, and (3) accesses from OS kernels to hardware resources are intercepted by the VMM. Through such an interposition mechanism, the VMM ensures that sensitive information will not flow to unauthorized parts, and a malicious process cannot create unauthorized mappings on pages owned by a trusted process.

Normally, system calls are made directly from processes to OS kernels without intervention from the VMM. To implement the interposition mechanism, CHAOS provides Trusted System Call (TSC) to handle communication between a trusted process and the OS kernel. The idea is to treat a system call as a remote procedure call (RPC) [3] between mutually untrusted peers. A TSC resembles a conventional system call except that the control and data transfers are interposed by the VMM. Such interpositions are completely transparent to the OS kernel and trusted processes.

2.2.2 Isolation

The isolation mechanism is built on the interposition mechanism and provides *compartmentalization* among trusted processes, normal processes, and OS kernels. It conceals CPU execution context and memory pages while a trusted process is not running in user mode. CHAOS follows two policies to facilitate memory isolation. First, the user-level mapping on a page table owned by a trusted process is hidden from the OS kernel and other processes. Second, pages owned by a trusted process are not allowed to be mapped by other processes and the OS kernel. However, concealing user-level mappings could disrupt the required communication between an OS kernel and a trusted process. For example, to handle I/O related system calls, an OS kernel needs to get/put the data to be written/read from/to a process. To handle this, CHAOS provides a *TSC layer* in the VMM to transparently handle control transfers and data exchanges between a trusted process and the OS kernel.

In summary, the major tasks of the isolation mechanism are: (1) to save and restore CPU context and the page table during control transfer between kernel and trusted processes, (2) to track pages owned by a trusted process, and (3) to implement a TSC layer to handle kernel/user communication.

2.2.3 I/O Sealing

The sealing mechanism protects data transfer (e.g. sensitive files) between memory and disks for a trusted process. As disks and OS kernel are not trustworthy, data passed to them should be protected. CHAOS uses cryptographic approach to protect sensitive I/O data from inspection and tampering. The sealing code handles three types of memory-disk interactions: (1) system-call based I/O, (2) memory-mapped I/O, and (3) paging (i.e. virtual memory) related I/O.

For I/O related system calls, a trusted process may need to pass I/O data to an untrustworthy OS kernel, which is capable of inspecting sensitive information in the data. To prevent this, the sealing mechanism in VMM uses encryption to protect the I/O data. VMM also decrypts the I/O data handed over from the OS kernel before passing it to the trusted process. The I/O sealing is built upon the TSC layer and completely transparent to user processes and the OS kernel.

CHAOS currently only handles file related I/O. CHAOS does not seal network I/O since most security-sensitive applications already use cryptographic approach (e.g. SSL) to ensure data privacy. Nevertheless, CHAOS does need to provide security-related services such as random number generation and time services to protect cryptographic mechanism. For example, a malicious OS cannot fake a pseudo-random number to make the application reveal cryptographic keys.

Memory-mapped I/O (MMIO) occurs without explicitly issuing system calls. CHAOS relies on the demand paging mechanism to handle MMIO. On intercepting a page fault, CHAOS determines whether the faulting address falls within the MMIO range. It unseals data when it is copied from disks to memory. An untrustworthy OS kernel cannot access the unsealed data because its mapping will only be available when the trusted process is resumed in user mode.

3 Detailed Design and Implementation

We have implemented a prototype system based on Linux 2.6.16 running on Xen 3.0.2. The hardware platform is x86 architecture. However, we believe the CHAOS architecture and its approach are not Xen-specific and its concept and architecture should be applicable to other VMMs and operating systems.

Figure 2 depicts the working model of CHAOS based on Linux and Xen. All kernel/user interactions of a trusted process are monitored and protected by the interposition, sealing and isolation modules. For interrupts with no data exchanges, only the interposition and isolation modules are involved. We use the flow

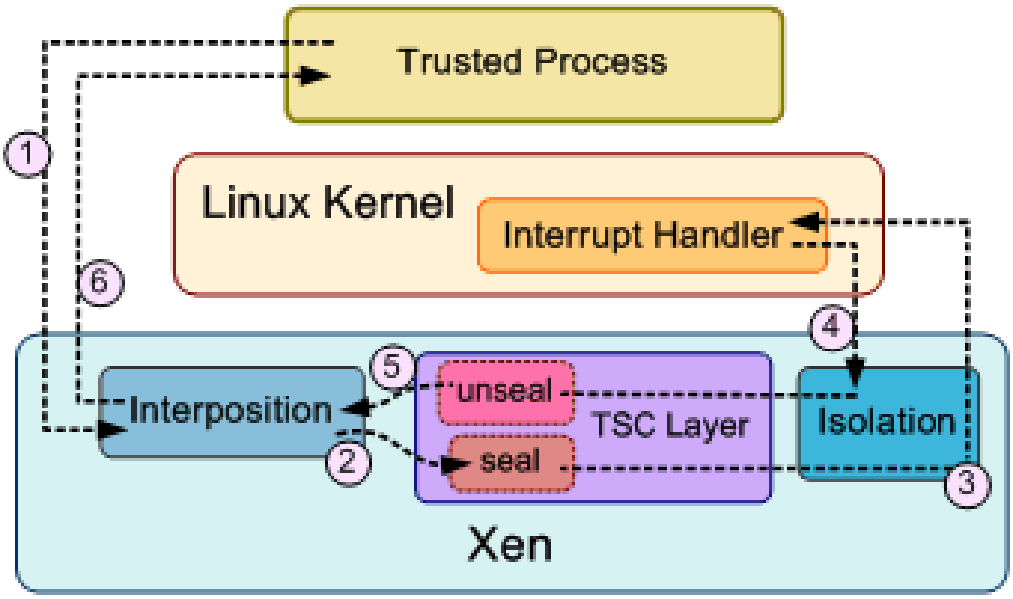

Figure 2: The CHAOS layer in Xen and Linux and its processing stages.

of a trusted system call (TSC) to illustrate the processing stages in CHAOS. In Step 1, CHAOS interposition module intercepts a TSC from a trusted process and forwards it to the TSC layer. The TSC layer fetches the data associated with the TSC and encrypts the data if the TSC is write-related (Step 2). In Step 3, the isolation module conceals the CPU-context and user-level page table mappings before transferring the execution to the Linux kernel. The interrupt handler in Linux serves the TSC request and then invokes Xen, where the isolation module restores the concealed CPU-context and page table (Step 4). Afterwards, the TSC layer retrieves the returned data and decrypts it if necessary in Step 5. Finally, CHAOS resumes the execution of the trusted process in user mode (Step 6).

The following subsections describe the implementation detail of the major mechanisms in CHAOS: interposition, isolation, I/O sealing, and management of trusted processes. Then, we present the implementation complexity of CHAOS in terms of lines of code (LOC). We also discuss the implementation limitations and possible future work.

3.1 Interposition

The interposition mechanism in CHAOS intercepts all operations that trigger control transfers among processes, OS kernels and the VMM. These transfers include: (1) system calls between user mode and kernel mode; (2) hardware traps and interrupts; (3) hypercalls [3] to Xen-provided interfaces.

Normal interrupts and traps will be captured by Xen since Xen is the sole resource manager. However, it requires special handling for system calls and resumption from interrupts. Figure 3 depicts the control flow transitions for a *trusted system call* (TSC) and a normal system call in CHAOS. Normally, a system call is basically an interrupt [4] (0x80 in Linux) intercepted and forwarded by the VMM. However, to improve performance, Xen optimizes the system call forwarding mechanism that allows user processes to directly call operating system kernel. CHAOS forces Xen to regain control by using *trusted system calls* (*TSC*). To do so, it simply replaces the system call entry in the *interrupt description table* (IDT) with a routine provided by CHAOS. The routine performs necessary operations such as isolation and sealing to protect sensitive data.

Further, to avoid the performance loss for normal processes, CHAOS utilizes another unused interrupt line (i.e. 0x81) for TSCs. We provide an off-line binary rewriting tool to rewrite all system calls in a trusted application to utilize the new interrupt line, i.e., to change the system call code from *int $0x80* to *int $0x81*.

[3] A hypercall is essentially an interrupt that allows a guest OS to invoke the services provided by the Xen VMM.
[4] Other forms such as *sysenter* (Intel) and *syscall* (AMD) can be similarly handled in CHAOS.

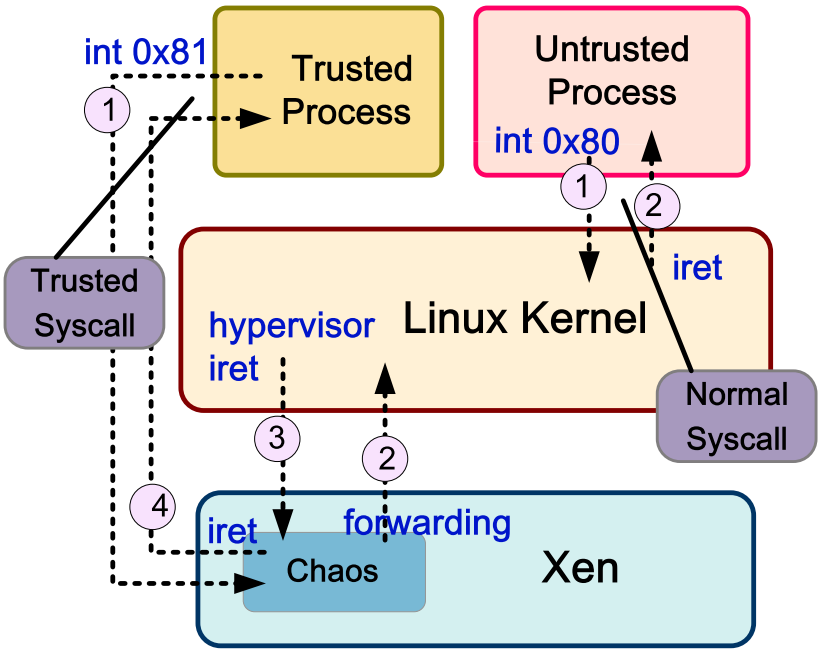

Figure 3: System call control flow for trusted system calls and normal ones in CHAOS.

To prevent normal processes from triggering this interrupt line, the service routine denies all accesses from an untrusted process.

Normally, Linux kernel directly resumes the execution of user processes using *iret* without the intervention of Xen. To interpose such transitions, CHAOS modifies the interrupt handlers in Linux to check if the running process is a trusted process or not before transferring control to the user process. If it is a trusted process, a hypercall (*hypervisor_iret*) to Xen is invoked instead. Note that the interposition mechanism is mandatory and cannot be bypassed. The isolation mechanism ensures that if the OS kernel does not behave correctly the process will not have the necessary information, such as execution context and memory mapping, and cannot be resumed.

3.2 Isolation

It is relatively easy to conceal CPU context. In contrast, concealing memory data is more complicated and requires four parts: (1) page table management - to conceal user-level mapping from the kernel; (2) page usage tracking - to forbid unauthorized mapping to pages owned by a trusted process; (3) paging handling - to handle swapping in and out of pages owned by a trusted process, and (4) *Trusted System Call (TSC) Layer* - to implement a RPC layer in Xen that facilitates secure data exchange between TSCs and the OS kernel.

3.2.1 CPU context Isolation

Upon an interposition of an interrupt to a trusted process, Xen saves the CPU context and clears register information that is not needed to limit data exposed to the Linux kernel. This includes general-purposed registers (GPRs). For system calls, some GPRs cannot be concealed and must be available to the Linux kernel in order to service it. Yet, an operating system can derive very limited information from them [16]. The saved context is restored when the kernel transfers control to the trusted process. Here, a replay attack by a malicious kernel is not possible because the VMM does not use the context provided by the untrustworthy kernel before resuming the trusted process.

Signal Handling: User-defined signal handlers permit applications to handle specific kernel events. A process receiving a signal will save its current context and transfer its execution to the signal handler. The

kernel could alter the program counter of the process to an arbitrary user instruction, which should be considered as a potential threat. To defend this, CHAOS intercepts system calls that register user-defined handlers, and records the signal numbers and handlers. The saved addresses of the handlers will then be used to verify the CPU context before a trusted process resumes its execution in user mode. Hence, even if a malicious kernel alters the process execution by sending arbitrary signals, it can hardly cause a trusted process to expose sensitive information because only the user-mode handlers provided by the trusted process can be used.

3.2.2 Page Table Concealing

CHAOS conceals the user-level memory mapping of a trusted process. A malicious kernel thus cannot inspect or tamper with the memory owned by a trusted process. To achieve this, Xen saves the page table of the trusted process and installs an *idle page table* (*swapper_pg_dir* in Linux) that contains only the kernel-level mapping of memory pages. The page table will be restored before the trusted process is resumed. Since the installation of a page table will be validated by the VMM, a malicious kernel cannot trick CHAOS with a tampered page table.

3.2.3 Page Usage Tracking

Xen already tracks the usage of all pages to ensure the strict isolation among virtual machines [2]. To further enforce the isolation among processes in an OS kernel, CHAOS extends this mechanism by tracking each page at the process level, and disallowing page sharing between a trusted process and an untrusted one. CHAOS maintains owner ID and group ID for each page owned by a trusted process. Only trusted processes in the same group (i.e. executing the same binary image) are allowed to share pages. Moreover, CHAOS maintains a bit to indicate whether a page needs to be encrypted or not, i.e. the page is *sensitive* or not. By default, all pages owned by a trusted process are sensitive and need encryption. CHAOS tracks the virtual address range of insensitive memory-mapped files by tracking the related system calls (e.g. *sys_mmap*). Pages mapped to the above memory range need not be encrypted.

CHAOS transparently isolates process memory by intercepting the page table updating requests (e.g. *set_pte()*). Whenever a page is allocated to a trusted process, CHAOS updates the table entry that corresponds to the allocated page. CHAOS then forbids further unauthorized mappings to the allocated table entry by validating each page table updating request. When a page is de-allocated, its previous mapping will be restored to make this page accessible by the Linux kernel again.

3.2.4 Paging of a Trusted Process

CHAOS marks all pages as insensitive if they are still being mapped by at least one trusted process. As mentioned before, CHAOS tracks the mapping of a page by intercepting page table updating requests. When the reference count of a page decreases to zero, i.e. the page is to be de-allocated and swapped out to disk, CHAOS encrypts the page and treats this page as a normal one. Thus, the page swapper will be able to map this encrypted page and swap it out. When the page is swapped in again, CHAOS will decrypt the page, increase its reference count, and mark it as insensitive before handing it over to the trusted process.

3.2.5 Trusted System Call Layer

Since the Linux kernel and a trusted process are considered mutually untrustworthy in CHAOS, strict isolation prevents most kernel-user data transfer. However, parameters and return values still need to be passed between them during system calls. Some of them are stored in user space and passed through memory, e.g. user buffers for I/O related system calls.

As the Linux kernel cannot access the memory space of a trusted process, CHAOS implements a RPC (Remote Procedure Call) layer for trusted system calls in Xen to monitor and handle data exchanges between the Linux kernel and trusted processes. CHAOS first validates a system call and finds the memory locations of the parameters. Then, CHAOS copies the parameters from the user space to a pre-allocated buffer, and adjusts the system arguments to make sure the kernel can correctly fetch them. Cryptographic approaches are used to conceal sensitive information, as described in section 3.3. On returning from a system call,

CHAOS also checks the data and copies them to proper memory locations indicated by the system call arguments. The TSC layer uses the POSIX (Portable Operating System Interface for uniX) standard to interpret and pass parameters for a TSC, thus should be portable to all operating systems conforming to the POSIX standard.

3.3 I/O Sealing

Like other trusted computing systems, CHAOS uses cryptographic approach to ensure the security of persistent data, but in a *transparent* way. CHAOS supports both system-call based I/O and memory mapped I/O.

For system-call based I/O, CHAOS mainly relies on the *TSC Layer* to do I/O sealing on security-sensitive files. Upon interceptions of I/O related system calls, CHAOS encrypts/decrypts the associated data before resuming the execution in the kernel/trusted process. The system call arguments are modified accordingly to ensure correct data transfer. In contrast, memory-mapped I/O is handled using the page fault mechanism. When a page is allocated (i.e. *set_pte()*) to service a *no_page* fault incurred by a trusted process, CHAOS decrypts the content of the page before it is handed over. The memory-mapped virtual address range is recorded during each memory map system call (i.e. *sys_mmap()*).

3.3.1 CHAOS File Format

In CHAOS, files are classified into two categories: (1) public files shared among all processes, including system configuration files (e.g. /etc/mtab), *proc* files and device files; (2) private files exclusively owned by a trusted process, which usually contain sensitive information. CHAOS protects private files by encrypting them using symmetric cipher (e.g. Advanced Encryption Standard, AES). As the public files should be accessible by multiple mutually untrusted processes, they should not contain sensitive information and should not be encrypted.

Public files can usually be identified by the file names, such as (/etc/∗, /proc/∗, /dev/∗, etc.). CHAOS should differentiate the private (encrypted) files from the public (plain) files. It is important that an encrypted file is self-evident and compatible. It should contain information on the permission and ownership of this file, i.e., which application on which platform could manipulate the file. It should also require no adjustment of existing file formats. To satisfy these two requirements, a CHAOS file is encrypted using a symmetric key and the key is also sealed together with a platform tag using the public key of the platform owning the file. The tag resembles UUID [5] and is a number that uniquely identifies the platform. The sealed key and tag are appended to encrypted file and will be used by CHAOS to distinguish between an encrypted file and a plain file. Hence, CHAOS does not need to know the detail of the file format, and thus maintains compatibility with existing files. We also provide a simple tool to encrypt the file.

3.3.2 Key Management and I/O Sealing

CHAOS uses both asymmetric and symmetric cipher to protect files owned by trusted processes. CHAOS uses the storage root key (SRK) in the TPM [28] as the platform key and utilizes it to generate other keys. The SRK consists of a public/private key pair (*PK-SRK* and *SK-SRK*). The *PK-SRK* is used by third parties to sign data to the platform, while the *SK-SRK* is used to decrypt the data. To secure software distribution, CHAOS requires a binary file to be encrypted using a random symmetric key. The symmetric key is also encrypted using the *PK-SRK* of a platform and appended in the binary file.

As shown in Figure 4, CHAOS relies on a group-based key management scheme to handle file I/O for trusted processes. The file I/O is sealed and unsealed using symmetric cipher. All trusted processes using the same binary image share the same encryption key provided by the image. Thus, an *execve()* after a *fork()* should require the child process to use a different encryption key, which can be retrieved from the image file. When an opened file is closed, CHAOS appends the encryption key together with a *platform tag* [6](encrypted together using the public key of the SRK) to the end of the file. On opening a file, CHAOS first retrieves the encryption key and tried to match the tag. If the tag does not match, we know it is not

[5] http://en.wikipedia.org/wiki/UUID

[6] A platform tag is essentially a unique ID that represents a machine.

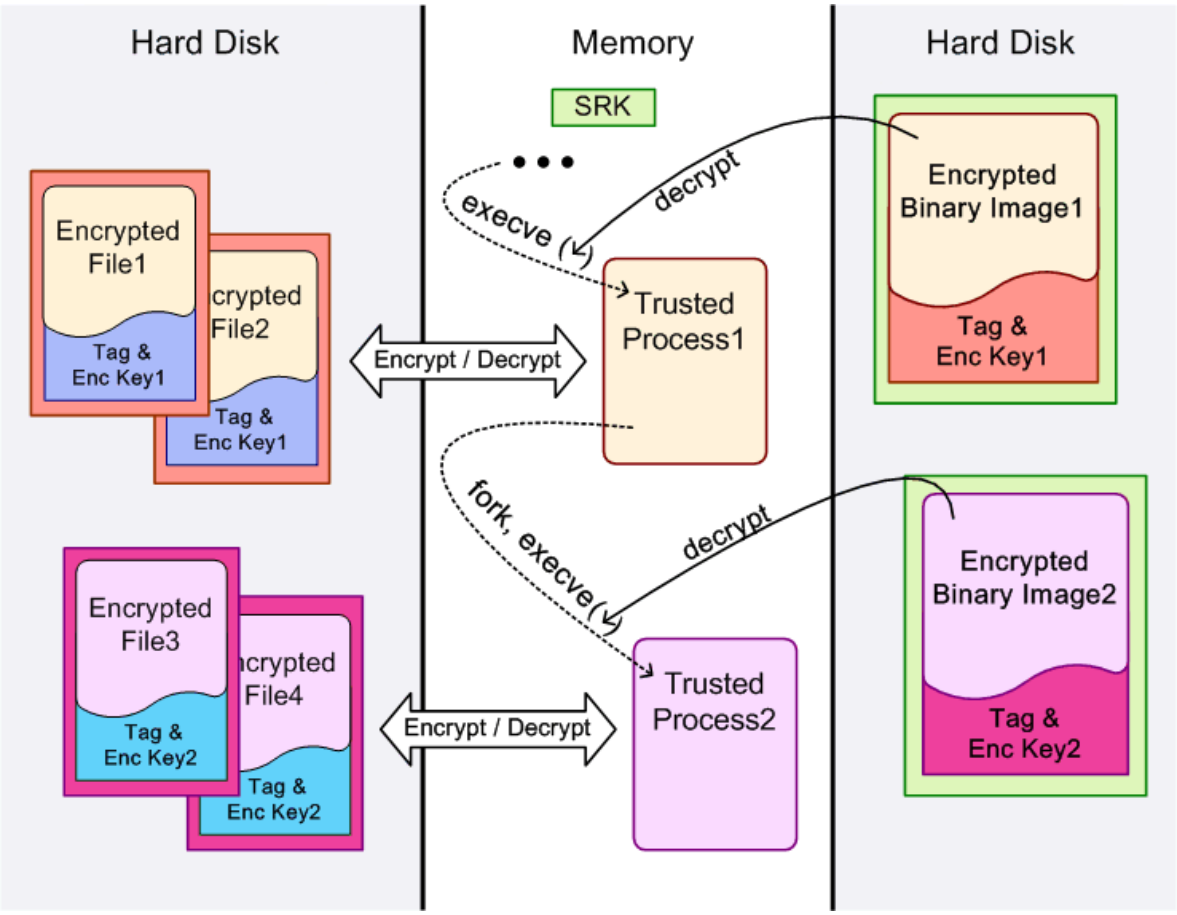

Figure 4: Key management in CHAOS.

an encrypted file. However, a trusted process is not allowed to write unencrypted data to a regular file. Therefore, even if a malicious kernel could trick CHAOS by disguising an encrypted file as a normal file (e.g. system configuration files), no secret will be divulged because subsequent data written to the file are encrypted. On closing an encrypted file, CHAOS appends the encrypted key together with the tag to the end of the file. Newly created files should also be encrypted and appended with the key and tag on file closing.

Sometimes an I/O operation is not aligned, i.e., it does not conform to the block size of a ciphering algorithm. To handle this, CHAOS transparently adds some *compensating system calls* to perform compensative work for the user-issued system call. For instance, for an unaligned write operation, CHAOS needs to fetch the unaligned chunk, decrypt the content, combine the data to make it be aligned to the block size, encrypt them, and finally write back the sealed data. To mitigate possible overhead, CHAOS uses *system call clustering* [21] to batch independent calls to a *multicall*. It reduces the number of transitions between the VMM and the kernel. The clustering is completely transparent to user applications. Fortunately, trusted processes using buffered I/O could largely mitigate the overhead by reducing the number of system calls.

3.4 Trusted Process Management

3.4.1 Identifying a Trusted Process

CHAOS needs to track the execution of a trusted process, thus it needs to uniquely identify a trusted process. As a page table is generally unique for each trusted process, the VMM uses the *page table base* (PTB) to identify a trusted process [13]. Because loading the PTB into hardware is done by the VMM, a malicious

Source Components	Lines of Code
Interposition	217
TSC Layer	2,281
Page Tracking	605
I/O Sealing	654
Process Manager	256
Miscellaneous	173
Total changes in Xen	4186
changes in Linux	227
Total line of code	4,413

Table 2: The number of non-comment lines of source code in CHAOS for Xen-Linux-2.6.16 on Xen-3.0.2.

OS cannot trick the VMM with a faked trusted process.

3.4.2 Running a Trusted Process

Many applications use dynamically linked library (DLL) to minimize the code size. However, this could undermine the security concern since DLLs are shared among processes. The DLLs may be tampered with by other processes to inject malicious code to observe and steal secrets. To prevent such attacks, CHAOS requires a trusted application be statically linked and encrypted using the public key of this platform. We believe static linking should not be a problem for compatibility, and there are also tools available to convert a dynamically linked executable to a statically linked one [5]. Further, static linking can avoid "DLL Hell" [7] problem and making programs mostly independent of the system environment, although at the cost of increased memory usage. Besides, the performance could be slightly improved due to reduced indirection and increased locality [10].

A trusted application can run as a trusted process by simply appending an extra flag to the argument list to inform the OS kernel and the VMM. As the OS kernel does not know the private key required to decrypt the program code and data, launching a trusted process should be in cooperation with the VMM. Thus, even if a user or the operating system has maliciously tampered with the flag, it cannot run the program correctly. Here, standard process launching (e.g. *do_execve()* in Linux) is adjusted to invoke a hypercall to Xen, and register a control structure to monitor the process.

A trusted process follows the scheduling policies of OS and time-shares the CPU with others. A process context switch is intercepted by the VMM since it is required to update the PTB (i.e. CR3 in x86). Upon a context switch, the VMM examines the PTB to determine whether the process is trusted or not.

3.4.3 Interprocess Communication

Interprocess communication (IPC) allows processes to communicate and exchange data. As IPC requests are usually delivered using system calls, the VMM can intercept these requests and make decisions on whether to block or to proceed. Generally, as IPCs require collaboration between two communicating parties, a trusted process could be aware of its communication with an untrusted counterpart. Therefore, CHAOS allows the IPC requests issued from a trusted process, but denies IPCs initiated from processes not in the same trusted group.

3.5 Implementation Complexity

Table 2 lists the implementation complexity of CHAOS for Linux-2.6.16 and Xen-3.0.2. A relatively small amount of source code is added for CHAOS. As illustrated in the table, CHAOS is made of 4,413 non-comment lines of code. The code changes consist of 227 lines of code, most of which are changes to the interrupt handling routines and file I/O (to implement I/O system-call clustering (in section 3.3.2)). Thus, CHAOS introduces only a small amount of changes to existing systems, and incurs only a modest increase in the system complexity and little degradation to the trustworthiness of the Xen VMM.

[7] http://en.wikipedia.org/wiki/DLL_hell

3.6 Discussions and Limitations

DMA Access: Currently, for platforms without an I/O MMU (e.g. most x86-based platforms), CHAOS is still vulnerable to unrestricted DMA accesses. A hardware device can perform arbitrary memory accesses, including accesses to memory owned by a trusted process. Fortunately, hardware vendors will soon release hardware extensions to restrict the memory accesses by DMA devices [1]. We plan to incorporate the extension to close such known vulnerability.

Covert Channels: While CHAOS explicitly protects CPU context, memory and I/O data, there could be some implicit information leaking to the OS kernel and other process. Typical covert channels include timing and execution statistics. As in other similar trusted systems, CHAOS does not aim to prevent all covert channels, but instead tries to make it hard enough to derive sensitive information.

Secure Human Computer Interaction: In light of trusted computing metrics, CHAOS has achieved code identity (section 3.4.2), curtained memory (section 3.2), Sealed I/O (section 3.3) and Attestation (using TPM). Yet, secure human computer interaction is an extremely hard problem for most existing systems using commodity hardware devices (e.g. mouse and keyboard). Recently, Intel Trusted eXecution Technology [12] is considering hardware devices with protected I/O paths. We intend to use these security enhanced I/O devices in CHAOS when they are available for highly interactive applications.

DoS Attacks: Assuming the distrust of an OS kernel or legacy code may make software vulnerable to denial-of-service (DoS) attacks from the OS kernel, due to its role as the service provider [16, 26, 25]. For example, a malicious OS kernel can provide unintended services for a system call requested by software or completely refuse to serve it. Although this is still an open problem and there is no general solution, we intend to provide verifying system calls (VSCs) to ascertain the correctness of OS services. VSCs are similar to *Proof-Carrying Code* but are issued by the VMM, thus are completely transparent to user processes. They are automatically generated by the VMM according to the system call trace of a running trusted process. CHAOS verifies the results of VSCs according to the execution context, and determines if the OS kernel is functioning correctly. For example, after a process writes some data 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. Since the VSCs can be selected to mimic the normal sequences of a process, the OS kernel can hardly distinguish it from normal ones.

Platform Mobility: Like many other trusted computing platforms, CHAOS also restricts the mobility of resources including files as they are tightly bound to the platform key (SRK). However, the TPM V1.2 standard does provide key migration between TPMs and the key migration has been implemented [19]. This can largely facilitate the migration of files among trusted peers.

Adapting CHAOS for Other OSes: CHAOS is currently implemented for Linux. Providing tamper-resistance to trusted processes requires a VMM that understands processes to some extent. Fortunately, modern OSes share a lot of similarities in their design and implementation, as well as interfaces. For example, many operating systems conform to the POSIX interface. Thus, the TSC layer, which contributes to the major source of implementation complexity in CHAOS, can be reused for POSIX-compliant OSes. We believe that adapting CHAOS to other OSes (e.g. NetBSD, OpenBSD and OpenSolaris) should require only minimal coding effort since the core mechanisms are very similar.

4 Evaluation

One important issue for CHAOS is that it should not incur too much overhead to the trusted applications. Meanwhile, it should have minimal performance impact on normal applications that do not require tamper resistant protection.

To measure the performance overhead of CHAOS, we compared the performance of several applications running on the following systems and configurations: original Linux-2.6.16; Xen-Linux on unmodified Xen-3.0.2; normal processes in Xen-Linux running on CHAOS; trusted processes on CHAOS. Besides, to understand the benefit of TSC rewriting (using *int 0x81*) since it saves the kernel/user crossing for normal system calls, we also provided the results for normal processes running on CHAOS without TSC rewriting.

All the experiments were conducted on a system equipped with a 2.8GHz Pentium IV with 1GB RAM, a Realtek 8169 Gigabit Ethernet NIC in a 100M LAN, and a single 160GB 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.

As we are currently not aware of general trusted computing benchmarks, we use traditional system-related benchmarks and typical security-critical applications. Two types of benchmarks were used to characterize the performance overhead of CHAOS: (1) application-level benchmarks to study the performance overhead of CHAOS against other mentioned systems and configurations. (2) micro-benchmarks to gain more precise performance results of some particular subsystems in all the test systems. Since applications that demand tamper-resistance protection should be statically linked in CHAOS, all tested benchmarks are statically linked for fairness. The results presented are the median of five trials.

4.1 Application Benchmarks

For application benchmarks, we report the performance results for SPECCPU-INT 2000 testsuite. we also use two prevalent server applications demanding tamper-resistance protection: the very secure FTP daemon (*vsftpd*), which is a de facto FTP server in UNIX environments; and Apache http daemon, the prevalent http server (httpd). Table 3 describes the methods used to measure these two applications.

Apps	connection time	transfer rate
vsftpd	average time of requesting 100 empty files using *wget*	download rate of a single 165MB file
httpd	use ab (apache benchmark) with "ab -n 20000 -c 500 filename" to get a 2KB file	

Table 3: Test methodologies for the two server applications.

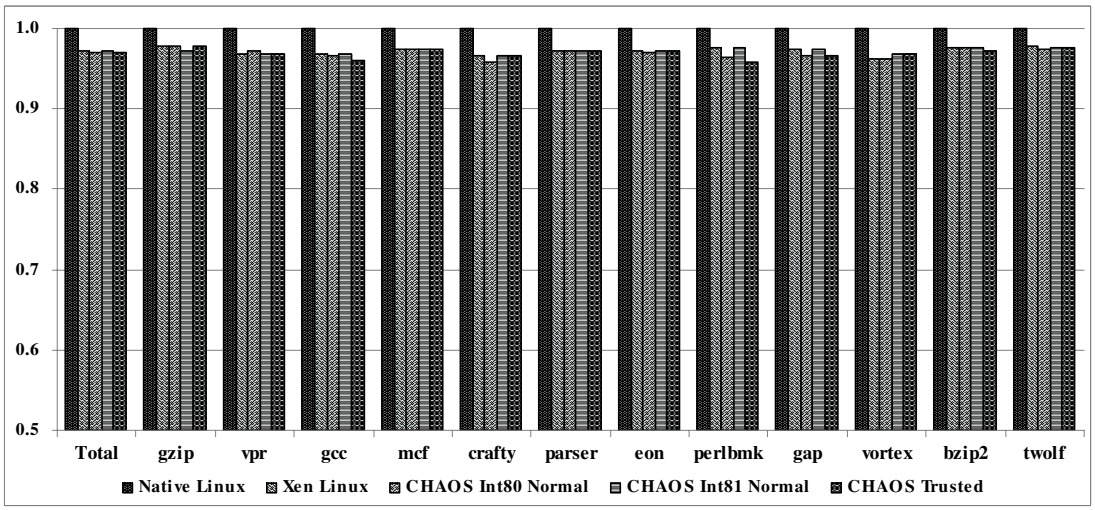

Figure 5: Relative results of SPECINT-2000 for Native-Linux, unmodified Xen-Linux, normal process (without and with system call rewriting) and trusted process in CHAOS.

Figure 5 shows the performance overhead for SPECINT-2000. As shown int the figure, CHAOS incurs only undetectable performance overhead to normal applications and trusted applications in CHAOS. This is because the incurred overhead mainly lies in the TSC layer and I/O sealing, without impacts on user-level activities. For CPU-intensive applications, most of their execution is in user space. As the overhead from system calls are amortized in application benchmarks, the incurred performance overhead is nearly undetectable. Although not significant, it can still tell from the figure that the results for normal applications without system call rewriting degrade a little compared to those with system call rewriting.

Figure 6 depicts the performance overhead for vsftpd and apache httpd, including the data for both connection time and transfer rate. For transfer rate of vsftpd, the incurred overhead is nearly undetectable because most operations are network-bounded, without lots of system calls. For the connection time of vsftpd, the incurred overhead is a bit higher (about (9%) due to a number of file creation, open and close

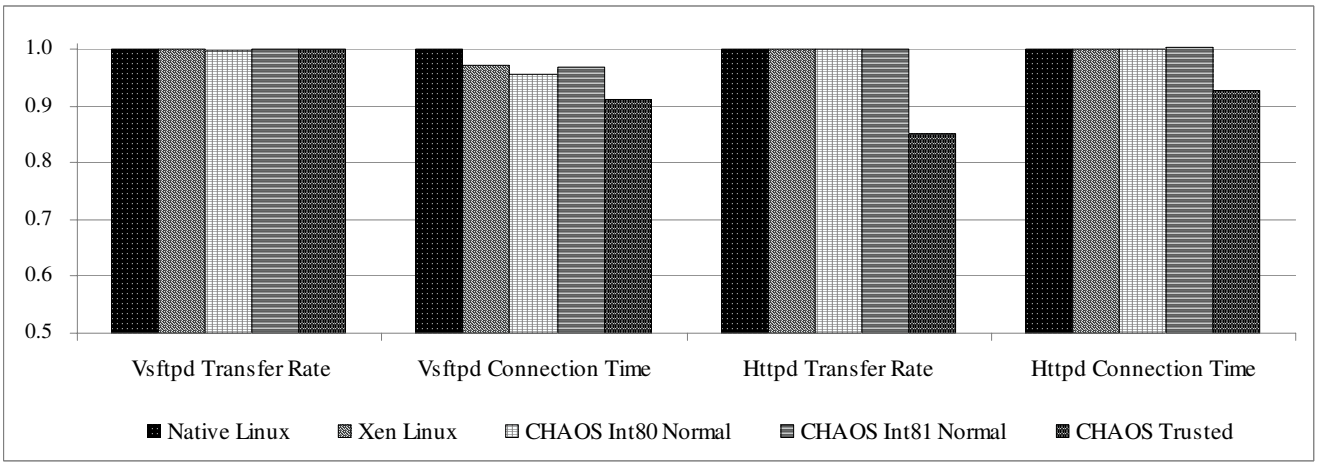

Figure 6: Relative results of vsftpd and apache httpd for Native-Linux, unmodified Xen-Linux, normal process (without and with system call rewriting) and trusted process in CHAOS.

tests	L-O	X-N	C-W	C-N	C-T	Overhead	
Null.call	0.39	0.40	0.75	0.40	2.4	2.01	6.2X
stat	1.60	1.65	1.95	1.66	9.41	7.81	5.9X
open&close	2.40	2.67	3.17	2.64	11.97	**9.57**	4.9X
read	0.50	0.50	0.86	0.51	3.70	3.20	7.4X
write	0.46	0.47	0.84	0.48	3.51	3.05	7.5X
fstat	0.63	0.64	0.99	0.63	3.98	3.35	6.4X

Table 4: lmbench system call results for Native-Linux (L-O), unmodified Xen-Linux (X-N), normal process (without (C-W) and with (C-N) system call rewriting) and trusted process in CHAOS (C-T) in μs.

operations. The incurred overhead for apache httpd is within 15% for connection time and 7.2% for transfer rate. The major incurred overhead also comes from the large numbers of disk I/O operations. However, the incurred overhead is still tolerable for applications demanding tamper-resistant protection.

4.2 Micro-benchmarks

Micro-benchmarks provide a more intimate understanding of the incurred performance overhead. The benchmarks were chosen from LM-bench 3.0-a5 micro-benchmark suite, with five tests as shown in Table 4. As shown in the figure, CHAOS incurs negligible overhead over normal processes, because the system calls are made directly between OS kernel and user applications without the intervention from CHAOS. In contrast, CHAOS incurs a relative high overhead for trusted processes. The performance of CHAOS without TSC rewriting for normal process is high due to the unnecessary system call interceptions. This also reflects the proportion of TSC interception in the total incurred overhead.

From Table 4, we know the performance overhead for a read/write system call is relatively high since it involves the data forwarding and decryption/encryption operations, which tends to be time-consuming. The overhead for an open/close system call is a bit high since opening an encrypted file involves matching the tag and key appended to the file. For all system calls, the TLB flushes incurred by reloading page tables cost some performance overhead. Although the performance penalty for system calls is relatively high, however, we believe it is quite justifiable for applications demanding tamper-resistance. Yet, as shown in the application benchmark, the amortized overhead is still modest.

5 Related Work

While there are many schemes and innovations proposed for a tamper-resistant execution environment, our work differs from previous efforts in that it relies on existing programming languages, conventional operating systems and commodity hardware to retain backward compatibility for existing applications. We only discuss the closely related work here.

XOM and Aegis: XOM [17] and Aegis [26] use core CPU architectural enhancements to support tamper-resistant software. XOMOS [16] examines these enhancements to support an untrusted commodity OS (IRIX in their case). XOMOS and Aegis do not trust main memory and encrypt off-chip data, which results in a strong tamper-resistant environment even to physical attacks. Yet, it could also incur high CPU overhead due to encryption. By contrast, CHAOS uses a VMM with a commodity OS to support tamper-resistant protection, which can be more flexible and practical. It relies only on commodity hardware and does not require changes to the core CPU architectures.

Pioneer: Pioneer [23] is a trusted computing primitive that utilizes time-difference as well as cryptographic approaches to ascertain untampered software execution. Functions executing longer than normal indicates possible tampering. It aims at preserving the integrity of the code and execution, but not privacy. Besides, the approach is applicable only to a restricted execution environment (e.g. no SMP, no SMI and should execute at most privileged level) and may not work correctly if the OS becomes malicious or compromised.

Proxos: Proxos [27] aims to provide tamper-resistant protection to application by partitioning the system calls between a private OS and a commodity OS, thus reusing the functionality in a commodity OS. However, the private OS in Proxos can only support limited functionalities (e.g. single address space and single-threaded). If the private OS is full-fledged, then its code size will inevitably be comparable to a commodity operating system, which degrades its trustworthiness. Moreover, Proxos requires programmers to customize the sensitivity of system calls. It incurs a relatively high overhead due to the need to create a VM for each process and inter-VM switches during system calls. Besides, Proxos cannot prevent attacks from the machine owner.

Type-safe OS: There is a recent trend that uses type-safe languages to build new operating systems to host security-sensitive applications. Singularity [11] employs safe languages (e.g. C#) to build a new operating system with the primary goal of dependability, but not compatibility. Instead of heavily relying on runtime monitoring, Singularity utilizes type-safety of a program and uses compilers to statically verify its safety and trustworthiness, which differs from CHAOS in the design goals and approaches.

MAC-based OS: In the past, mandatory access control (MAC) systems have been widely used to enhance system security. One typical system is SELinux [18]. It integrates fine-grained access control policies to many system resources in Linux. However, the policies are usually rather big and complex. They are hard to derive and to verify their completeness. Other MAC-based systems such as Eros [24], Asbestos [8] and Histar [29] are not designed to retain backward compatibility. They build new operating systems from scratch and require modifications to existing applications to be ported.

VMM-based Resource Tracking: Tracking processes and resources in a virtualized environment [13, 14] has recently been investigated. They aim to bridge the semantic gap between a VMM and the operating systems thereon by deriving useful information from hardware events (e.g. page faults, context switches). However, these systems are mainly designed to improve the efficiency of the resource utilization. In contrast, CHAOS monitors and tracks processes and resources mainly for tamper-resistance.

6 Conclusion

In this paper, we present CHAOS, a trusted computing infrastructure that transparently provides a tamper-resistant execution environment for a commodity OS kernel. CHAOS uses a trusted VMM to *interpose* privileged operations, *isolate* sensitive information and *seal* persistent data, thus protects a trusted process from exposing its private data and prevents tampering from a compromised OS kernel and other processes. It also utilizes cryptographic approaches to securing software distribution and process launching. In contrast to other schemes, CHAOS embraces both functionalities and tamper-resistance, yet is cost-effective. It also retains backward compatibility.

References

[1] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schionas, I., Uhlig, R., Vembu, B., and Wiegert, J. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal 10*, 3 (2006), 179–192.

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. SOSP* (2003), pp. 164–177.

[3] BIRRELL, A., and NELSON, B. Implementing Remote Procedure Calls. *ACM TOCS 2*, 1 (1984), 39–59.

[4] CERT Coordination Center. http://www.cert.org, 2007.

[5] Collberg, C., Hartman, J., Babu, S., and Udupa, S. Slinky: Static Linking Reloaded. In *Proc. USENIX ATC* (2005), pp. 309–322.

[6] CVE-2006-0038. Linux kernel netfilter do_replace local buffer overflow vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-0038.

[7] Duc, G., and Keryell, R. Cryptopage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In *Proc. ACSAC* (2006).

[8] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. Labels and event processes in the asbestos operating system. In *Proc. SOSP* (2005), pp. 17–30.

[9] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: a virtual machine-based platform for trusted computing. In *Proc. SOSP* (2003), pp. 193–206.

[10] Ho, W., Chang, W., and Leung, L. Optimizing the performance of dynamically-linked programs. In *Proc. USENIX* (1995), pp. 19–19.

[11] Hunt, G., Hawblitzel, C., Hodson, O., Larus, J., Steensgaard, B., and Wobber, T. Sealing os processes to improve dependability and safety. In *Proc. EuroSys* (2007).

[12] Intel. Intel trusted execution technology. http://www.intel.com/technology/security/, 2006.

[13] Jones, S., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proc. USENIX ATC* (2006).

[14] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proc. ASPLOS* (2006), pp. 14–24.

[15] Krohn, M., Efstathopoulos, P., Frey, C., Kaashoek, F., Kohler, E., Mazieres, D., Morris, R., Osborne, M., VanDeBogart, S., and Ziegler, D. Make least privilege a right (not a privilege). In *Proc. HotOS* (2005).

[16] Lie, D., Thekkath, C., and Horowitz, M. Implementing an untrusted operating system on trusted hardware. In *Proc. SOSP* (2003), pp. 178–192.

[17] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. Architectural support for copy and tamper resistant software. In *Proc. ASPLOS* (2000), pp. 168–177.

[18] Loscocco, P., and Smalley, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proc. FREENIX* (2001), pp. 29–42.

[19] Mao, W., Yan, F., and Chen, C. Daonity: grid security with behaviour conformity from trusted computing. In *Proc. ACM STC* (2006), pp. 43–46.

[20] Peinado, M., Chen, Y., England, P., and Manferdelli, J. NGSCB: A Trusted Open System. In *Proc. ACISP* (2004), pp. 86–97.

[21] Rajagopalan, M., Debray, S., Hiltunen, M., and Schlichting, R. Cassyopia: Compiler assisted system optimization. In *Proc. HOTOS* (2003), pp. 103–108.

[22] securityfocus. Windows vista voice recognition command execution vulnerability. http://www.securityfocus.com/bid/22359/, 2007.

[23] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proc. SOSP* (2005), pp. 1–16.

[24] Shapiro, J., Smith, J., and Farber, D. EROS: a fast capability system. In *Proc. SOSP* (1999), pp. 170–185.

[25] Singaravelu, L., Pu, C., Hartig, H., and Helmuth, C. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proc. EuroSys* (2006), pp. 161–174.

[26] Suh, G., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proc. Supercomputing* (2003), pp. 160–171.

[27] Ta-Min, R., Litty, L., and Lie, D. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proc. OSDI* (2006), pp. 279–292.

[28] Trusted Computing Group. http://www.trustedcomputing.org.

[29] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazieres, D. Making Information Flow Explicit in HiStar. In *Proc. OSDI* (2006), pp. 279–292.