AppShield: Protecting Applications against Untrusted Operating System

Yueqiang Cheng, Xuhua Ding, Robert H. Deng

School of Information Systems
Singapore Management University, Singapore
{yqcheng.2008, xhding, robertdeng}@smu.edu.sg

11, 2013

SMU-SIS-13-101

AppShield: Protecting Applications Against Untrusted Operating System

Yueqiang Cheng, Xuhua Ding, Robert H. Deng
School of Information Systems, Singapore Management University
{yqcheng.2008, xhding, robertdeng}@smu.edu.sg

October 31, 2013

Abstract

Commodity operating systems are known to be vulnerable to rootkit attacks due to their enormous code base and complex logic. Since the OS runs with a higher privilege than user applications, the rootkit residing in the kernel can access the entire user application space, even when the application is designed and implemented with security considerations. Existing systems protecting user-space code have various drawbacks, such as high performance overhead, large Trusted Computing Base (TCB), hardware modifications, or with a restriction imposed on the protected code. Moreover, several newly identified threats in our paper are evidence that protecting applications from malicious OS is more challenging than previously realized.

In this paper, we present the design and implementation of AppShield, a hypervisor-based approach that efficiently and reliably safeguards code, data and execution integrity of a critical application. The protection overhead is localized to the protected application such that unprotected applications are not affected. We implement the prototype of AppShield with a tiny hypervisor. We experiment AppShield with several existing applications on a Linux platform and the results show that the performance costs on CPU computation, disk I/O and network I/O are insignificant.

1 Introduction

It is upheld as a norm that the operating system has a higher privilege than applications and therefore can access anywhere in the latter's address space. As a result, once the OS is compromised (which is commonly due to its enormous code size and complexity), the application and its data are under the threat of rootkit attacks. To cope with kernel level attacks, various mechanisms [23, 22, 29, 4, 7, 36, 32, 6, 30] have been proposed to protect those critical applications without trusting the OS.

Among them, the approaches like Flicker [23], TrustVisor [22] and Fides [29] simplify the problem setting by assuming that the protected code is self-contained with pre-defined inputs and outputs (e.g., inputs are the initial parameters and outputs are the final returns); and that the protected execution does not involve dynamic memory allocation or deallocation.

However, applications in practice usually interact with the kernel via system calls for memory management and data exchanges. To protect a full-fledged application, several systems [30, 21, 3, 11, 7, 36, 32, 6, 14, 17] are proposed. Among them, AEGIS [30], XOM OS [21], Bastion [3] and SecureME [11] require hardware modifications, which appear impractical for commodity platforms. Proxos [32] and Terra [14] introduce a dedicated trusted virtual machine for the protected applications, which dramatically increases the TCB size and therefore weakens their security strength. The systems like OverShadow [7], CHAOS [6], SP^3 [36], InkTag [17] aim to protect the whole process without requiring hardware modifications or a trusted VM. However, they incur high performance loss due to the costly encryption/decryption operations and are subject to the newly identified attacks described in Section 3.2.1. For example, the malicious OS may swap two address translation mappings to break the data integrity without directly accessing the data pages (i.e., mapping reorder attack). It may also illicitly return an allocated memory region with its virtual addresses occupied by the application stack (i.e., Iago attack [5]). By doing so, the application may happen to modify the control data (e.g., return address) in the stack, and thereby compromise the execution of the victim application. The newly identified threats are evidence that protecting applications against the malicious OS is more difficult than previously realized.

In this paper, we propose AppShield, a novel system which reliably and efficiently protects data secrecy and integrity of a critical application, as well as its execution integrity, against rootkit attacks. AppShield leverages the hardware-assisted virtualization techniques [19] to isolate the application's address space such that all accesses from the kernel are blocked except those explicitly authorized by the application through system calls. The protected application utilizes the main memory in the same fashion as in a normal (unprotected) setting. It accesses the memory with native speed, i.e. without encryption/decryption or being intercepted, and it can request the kernel to (de)allocate memory buffers. AppShield achieves performance isolation since those unprotected applications are not affected and do not have performance loss. We have implemented a prototype of AppShield which consists of a bare-metal hypervisor with roughly $29K$ SLOC and a tiny kernel module of around $2K$ SLOC. We have experimented the prototype with several applications (e.g., Apache and VIM) and run a suite of benchmark tests. The experiment results demonstrate that AppShield incurs insignificant performance costs in CPU computation, disk I/O and network I/O.

ORGANIZATION. In the next section, we define the problem by specifying the threat model, our objectives and the AppShield overview. In Section 3, we describe the dynamic address space isolation together with newly identified threats. The secure and efficient address space switch, and the support of legal data exchanges is described in Section 4 and Section 5, respectively. The implementation and evaluation are shown in Section 6. We discuss several issues and related work in Section 7 and Section 8, respectively. Finally, we conclude this paper in Section 9.

2 Synopsis

2.1 The Model

In this work, we consider an adversary who remotely controls the OS on the target platform by a rootkit and attempts to attack a critical application by tampering with its data and/or execution. The adversary can run arbitrary code and launch DMA operations in the victim platform. Nonetheless the adversary can not physically control it.

Our aim is to protect a critical application execution integrity and data security against such an adversary. We do not consider protection of its availability. Neither do we protect the application's raw I/O inputs and final data outputs[1]. Side channel attacks are also out of scope of our study.

We suppose that no malicious data input can subvert the control flow of the critical application. It is *orthogonal* to our objectives to enhance code security (e.g., fixing bugs) of the protected applications. The platform's chipset and all peripheral devices are trusted in the sense that they operate exactly following their specifications and do not contain Trojan-Horse circuits or microcode that respond to commands of the adversary.

In our model, a bare-metal hypervisor is trusted since it can be protected with secure boot/DRTM (e.g., Intel TXT) and hardware virtualization technology. Furthermore, the hypervisor can leverage some existing hypervisor protection schemes (e.g., HyperSafe [35], HyperSentry [1], HyperCheck [34]) to further enhance its security. Note that the hypervisor can intercept and emulate the SMM operations so that SMM-based attacks cannot subvert it.

[1]The critical application may encrypt its disk and network data. Existing secure I/O path schemes like [10, 37] can protect the raw I/O inputs

2.2 Desired Properties

It is desirable for a security solution for the stated problem to have the following properties. Firstly, the application's behavior should be preserved by the protection mechanism. The application is not assumed to be a piece of self-contained code and is entitled to issue system calls as in a normal setting. For instance, it can request the OS to allocate a memory buffer even though the OS is not trusted.

Secondly, the security mechanism should have minimum performance impact on the protected application and on the platform as a whole. The performance requirement has twofold implications. Ideally, the protected application should be able to access the main memory with the native speed. Therefore, hypervisor-based interposition and memory buffer encryption/decryption should be avoided since they take a significant toll on memory access delays. Moreover, the mechanism should only incur localized performance overhead, without affecting the performance of unprotected applications and the OS.

Lastly, the TCB of the security mechanism should small and simple, which ensures that the risk of subverting the TCB is kept minimal. This property precludes the approach of using a trusted virtual machine where the TCB encloses an operating system.

In this paper, we present the design and implementation of AppShield which is the first of its kind meeting the security requirement with all the aforementioned properties. AppShield uses a tiny hypervisor on the bare-metal machine to protect a critical application against the untrusted OS. Its overview is described in the next section.

2.3 AppShield Overview

The fundamental idea of AppShield is to isolate the target application's context (registers) and address space from the kernel and other applications, while allowing it to issue system calls and utilize the memory in a dynamic fashion. The rootkit cannot access its memory space, except those memory buffers explicitly exported to the kernel by its system calls. In the rest of the paper, we use CAP to denote the critical application that is under AppShield's protection.

Figure 1 depicts the architecture of AppShield. It consists of a bare-metal hypervisor, a *transit module* in the guest kernel space and a *shim code* in the user space. Both the transit module and the shim code are safeguarded by the hypervisor to defend against attacks from the guest kernel. CAP runs in an address space isolated from the rest of the guest domain, while the guest OS and those unprotected applications on the platform run unaffected. The page table of CAP is managed by the guest OS, but the updates are intercepted and verified by the hypervisor to defend against various attacks (Section 3.2.1). CAP's system calls

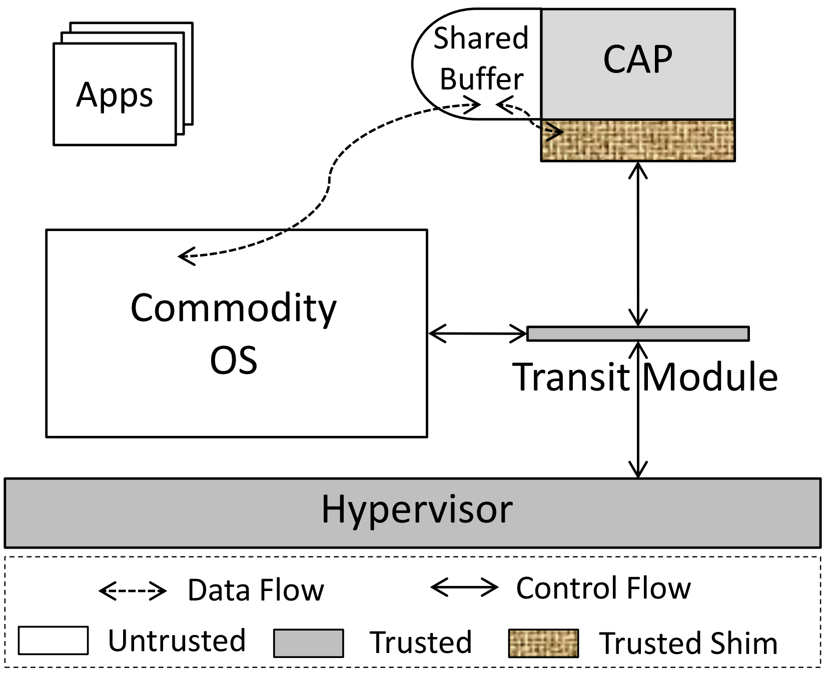

Figure 1: **The architecture of AppShield.** The data flows (dotted lines) between the protected Critical APplication (CAP) always go through the shared buffer and mediated by the *shim* code. The control flows (solid lines) between CAP and the OS are mediated by the Transit Module (TraMod). The execution of transit module are protected by the hypervisor.

are mediated by the trusted shim code which is essentially a wrapper of `libc` libraries. The main task of the shim is to marshal the system call parameters by exporting the data needed by the system call routine into the shared buffer accessible to the kernel. Since events like interrupts and system calls causes context switches between CAP and the guest kernel, the transit module responds to the event, facilitates the context switches, and prevents the context switch from being manipulated by the rootkit.

3 Dynamic Address Space Isolation

Dynamic address space isolation is the bedrock of AppShield. In this section, we first elaborate how the hypervisor isolates a pre-defined address space of CAP. Then, we explain how the isolation is dynamically adapted to the changes of the memory boundary at runtime. While our description follows Intel virtualization technology, the approach is applicable with AMD's as well.

3.1 Address Space Isolation

In a nutshell, the physical memory assigned to the guest is divided into two separated regions by the hypervisor. One region is used for CAP while the other is for the guest OS and other applications. The memory dichotomy as depicted in Figure 2 is realized by two suites of EPTs maintained by the hypervisor, respectively. In this way, the virtual addresses of the guest OS and other applications are never mapped to a physical address dedicated to the protected application, and vice versa. The hardware enforced address space isolation ensures that the guest OS and the protected application cannot directly access each other, provided that all EPTs are properly set and applied. For the sake of clarification, we use *AppShield EPT* to refer to the ones dedicated for CAP. In the following, we only focus on the EPT configuration. The details of applying the proper EPT are described in Section 4 which elaborates the context switches between CAP and the guest OS.

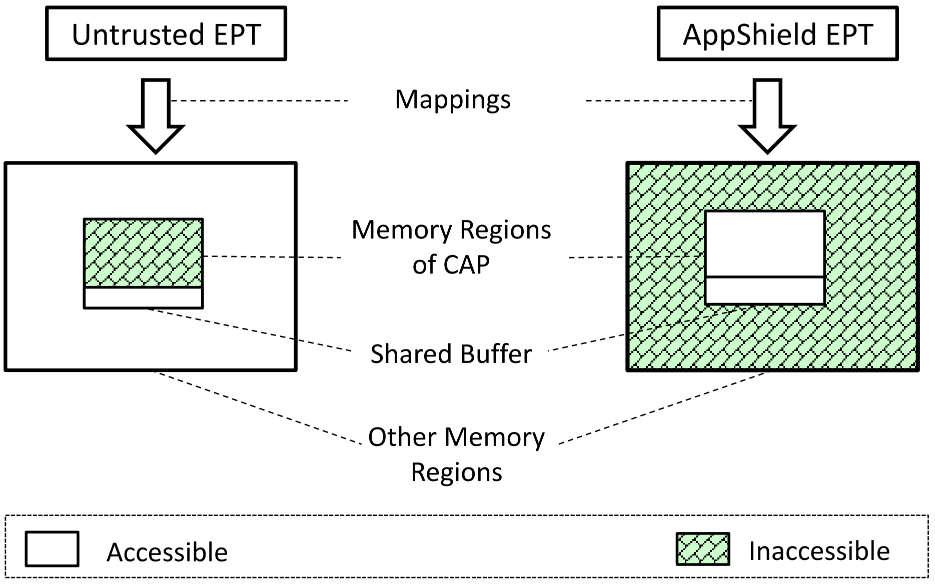

Figure 2: **Address Space Isolation.** With the trusted AppShield EPT, only the memory regions of CAP and the shared buffer are accessible, while with the original EPT, the memory regions except the shared buffer are inaccessible.

The hypervisor exports two hypercalls for CAP to activate and deactivate the protection. The activation hypercall is issued before CAP's main function is entered. In response, the hypervisor obtains the `CR3` register value from the VMCS and traverses the page table entries belonging to the application, so that it locates all pages within the address space, including the shared libraries. (Note that the page fetching during the traversal forces the guest kernel to load share libraries into the memory.) Both the traversed guest PTEs and the pages pointed by them constitute the physical memory region that needs to be separated from the guest. The hypervisor creates the AppShield EPT for this region and marks the corresponding entires in the original EPT as inaccessible, so that the guest cannot visit the isolated region. Once the application's code and data are isolated, the hypervisor can validate its launch-time integrity, supposing that it has been priorly authenticated by a signature or an HMAC tag.

Through the deactivation hypercall, CAP notifies the hypervisor to disable the protection. In response, the hypervisor destroys the AppShield EPT and restores the entries in the original EPTs. Note that the deactivation hypercall can only be issued by CAP. Any deactivation requests from malicious guest OS and other unprotected applications will be rejected by the hypervisor.

3.2 Dynamic Isolation

One of the main challenges of isolating a full-fledge application is that its memory region evolves over time, due to dynamic memory allocation and deallocation as a result of relevant system calls (e.g., *brk*) which are in turn invoked by the corresponding memory usage functions (e.g., *malloc* and *free*) in the `libc` library.

The semantics of these system calls are preserved in AppShield as the guest OS still manages the memory resources for CAP through the guest page table. Although the hypervisor protects the guest page table used by CAP, the guest kernel may manipulate the virtual and/or physical address of the new buffer to attack CAP without direct access to the latter's memory space. We identify several such attacks below.

3.2.1 Address Manipulation

In general, address manipulation attacks can be launched by the kernel in response to any system calls that result in page table updates. Without loss of generality, we use buffer allocation as an example to illustrate the attacks.

Suppose a CAP's buffer contains three consecutive pages at virtual address V_A, V_B and V_C respectively and CAP requests a new buffer. In a normal scenario, the newly allocated buffer's virtual address and physical addresses do not overlap with any existing one, as illustrated in Figure 3-(a), where they are at virtual address V_D, V_E and V_F. In the following, we show four types of manipulation attacks.

Mapping Overlap Attack. The malicious kernel may overlap two memory regions in the virtual address space. As illustrated in Figure 3-(b), the new buffer is set to the pages located at V_C to V_E. The overlapping leads to undesired modifications of data in P_C when the application attempts to update the first page of the

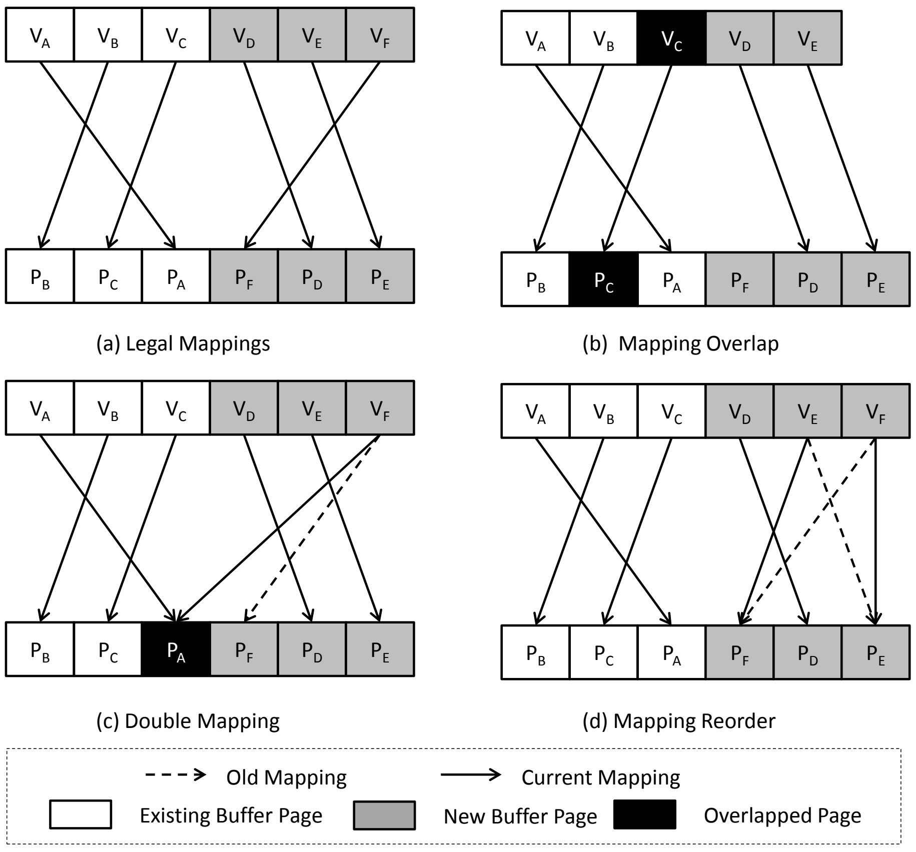

Figure 3: **Threats for address space isolation.**

allocated buffer. Obviously this threat could break the data integrity, and it may also subvert the control flow of the application, if the overlapping memory is in the application stack and the modifications change the stored return address(es). In fact, the mapping overlapping is a type of Iago attack [5].

Double Mapping Attack. The double-mapping attack maps two or more virtual pages to one physical page in the user space. As shown in Figure 3-(c), a write to V_A affects the result of a read operation at V_F. This attack is more stealthy than the mapping-overlap attack, as the physical addresses are transparent to the code running in the virtual space which is not tampered with at all.

Mapping Reorder Attack. The mapping-reorder attack is to reorder the existing address mappings between the virtual addresses and the physical addresses. As shown in Figure 3-(d), CAP retrieves wrong data when it reads from V_F. As a result, CAP's data or control flow can be manipulated by the malicious kernel.

Mapping Release Attack. The mapping-release attack is to release one or more existing mappings without any system call requests driven by the protected application. The mapping-release could induce the hypervisor to give up the protection of those pages since they are not in the protected addresses space any more. By doing so, the guest OS can freely access the data on those released pages.

3.2.2 Information Collection

Most applications and shared libraries trust OS by default, and they all miss the verification of the OS behaviors in the memory allocation and deallocation. To fix this loophole, AppShield has to verify if the memory updates follow the requests of the application and are not manipulated.

To verify the OS behaviors in memory updates, we should know the existing memory layout, determine the intent of the application relevant to memory updates and interpret the page table updates operated by the untrusted OS. The existing memory layout (the mapping relationship between guest virtual addresses to guest physical addresses) can be collected from the guest page table of the CAP. The collected information is reliable since it is collected by the hypervisor and the page table has been protected to prevent any update.

To determine the intent of the application relevant to memory updates, one possible way is to allow the hypervisor to intercept all system calls that are potentially used by the CAP to allocate or deallocate memory. In order to correctly interpret the memory updates information (i.e., the based address and the size), the hypervisor has to know the exact semantic meaning of all parameters and return values. It inevitably increases the complexity of the hypervisor and thereby dampens its security. In our paper, the trusted shim running in the user space closely works with the CAP. Thus, it knows the system calls used by the CAP and their semantic meanings, e.g., the parameter of the *malloc* is the memory size and the return value is the based address of the new allocated buffer. Through several hypercalls, the trusted shim is able to securely synchronize such information with the hypervisor.

To intercept and interpret page table updates, one possible solution is to allocate a dedicated Guest Page Table (GPT) in the address space of the CAP, and the hypervisor or transit module manages its updates. By doing so, the security of the page table is guaranteed but the complexity of the hypervisor or transit module will dramatically increase, which further takes a toll on the overall system security. Another possible way is the paraverfication technique [17]. However it requires costly modifications of OS code. To achieve good compatibility and make the hypervisor and transit module small and simple, we choose the solution that is similar to the management of the page table in paravirtualization, e.g., Xen [2]. Specifically, the page table created by the untrusted OS for the CAP is set read-only, but the management is still handled by the untrusted OS. The updates referring to the CAP memory regions are intercepted and verified by the hypervisor. Note that during the validation procedure, the hypervisor gets the original and the new value of the page table slot, together with the virtual address according to the slot position. Considering the updated address and comparing the original value with the new one, the hypervisor obtains the meaning of the current update, and thereby can validate the page table update.

3.2.3 Verification Details

In page table update verification, the hypervisor and the shim code jointly enforce the following policies for protecting the address space of a CAP.

1. The page table of CAP should be non-writable for the untrusted guest OS. Any update should be intercepted by the hypervisor.
2. The newly added memory region should not conflict/overlap with any existing memory regions, no matter the conflicts happen in virtual address space (no mapping overlap) or physical address space (no double mapping).
3. Once the mappings between guest virtual addresses and guest physical addresses are fixed, they are not allowed to re-map (e.g., no mapping reorder).
4. The memory regions can be released only if the CAP requires to release them (no malicious release), and the page data should be cleaned before allowing the guest OS to manage/access it (no data leakage).

Essentially, the mapping overlap attack is the conflicts in the virtual address space. Thus, the verification algorithm can be put into the trusted shim, since it is protected and aware of virtual addresses. Specifically, the trusted shim is able to know all memory regions used by the CAP by collecting such information in the memory-related system calls. For example, the trusted shim can know the size of the memory-mapped region through the second parameter of *mmap* and the base address through the return value. Such information stored in an ordered list is inaccessible for the untrusted guest OS since the address space of the CAP is isolated by the hypervisor. For each new allocated memory region, the trusted shim verifies it with existing ones. If there is no overlap, it then updates the maintained list and passes the execution flow to the CAP; otherwise it will issue a hypercall to the hypervisor to inform the policy violation.

To defend against double mapping and mapping reorder attacks in the page table updates, the hypervisor has to interpret the old mapping M_o and the new mapping M_n, and analyze the intent of this update. If the guest OS is to build a new mapping (i.e., M_o is empty and M_n points to a guest physical page), the hypervisor verifies if the new pointed physical page is occupied before. If it is already occupied, it is a double mapping attack; otherwise the update is approved.

If the guest OS aims to remap/reorder the mappings (i.e., both M_n and M_o point to guest physical pages), the hypervisor directly rejects it.

If the guest OS aims to free an old mapping (i.e., M_o points to a guest physical page while M_n is empty), the hypervisor verifies if CAP requires the guest OS to release this memory page. The information about the released memory pages is provided by the trusted shim through hypercalls. Those potentially released memory pages are stored in a list in the hypervisor space. By searching the list, the hypervisor decides if the current page is the one that CAP aims to release. If it is not, the hypervisor rejects the update; otherwise it approves it and updates the list by deleting the corresponding record. Note that the data on the releases memory page is cleaned by the trusted shim once it gets the release requests from the CAP.

4 Secure Context Switch

Events like system calls, interrupts and exceptions, lead to context switches between CAP and the kernel. Different from traditional user-kernel context switch, the switch between CAP and the kernel involves address space switches, since they run in two address spaces.

When CAP is in execution, the transit module in AppShield handles all interrupts and prevents the kernel from exploiting the context switch to attack CAP. Its main tasks are to facilitate the context switch and to safeguard CAP's context information. It also notifies the hypervisor to perform address space switch. As shown in Figure 4, when an interrupt is raised, the control flow leaves from CAP to the kernel. Once the event is processed by the kernel, the flow goes back to CAP. We proceed to elaborate the details of context switch.

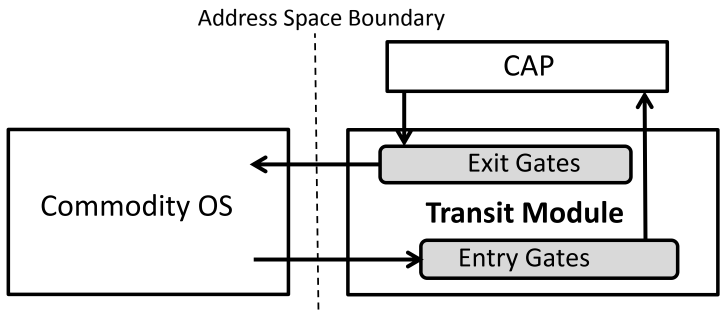

Figure 4: Control flow between the CAP and the guest kernel.

4.1 Transit Module

The transit module is a self-contained kernel module with its execution being protected by the hypervisor using the mechanism described in [28]. Specifically, the memory regions occupied by the transit module is isolated by the hypervisor, such that the untrusted commodity OS can not modify the data and the code. The control flows of the transit module execution *always* start from the pre-defined addresses.

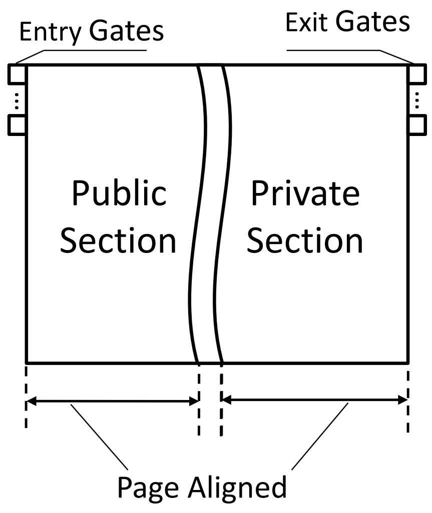

Figure 5: The format of transit module

The transit module has two sections (Figure 5), which are page aligned for facilitating memory protection. The first section is the *public section* which contains information that is read-only for the transit module and the commodity OS. The second section is the *private section* which contains private data. Accesses to the private section are only allowed if they are from the transit module; other accesses originated from outside of the transit module are blocked by the hypervisor. The transit module consists of a set of interrupt handlers called *AppShield interrupt handlers*. An AppShield interrupt handler is composed of two code stubs (Figure 6). One is called the *entry gate* which is located in the public section and the other is called the *exit gate* which is in the private section. The control flow of the transit module always starts from one of the entry/exit gates. The exit gate handles the context switch from CAP in protection to the guest kernel while the entry gate handles the switch back to CAP. More details are presented in Section 4.3.

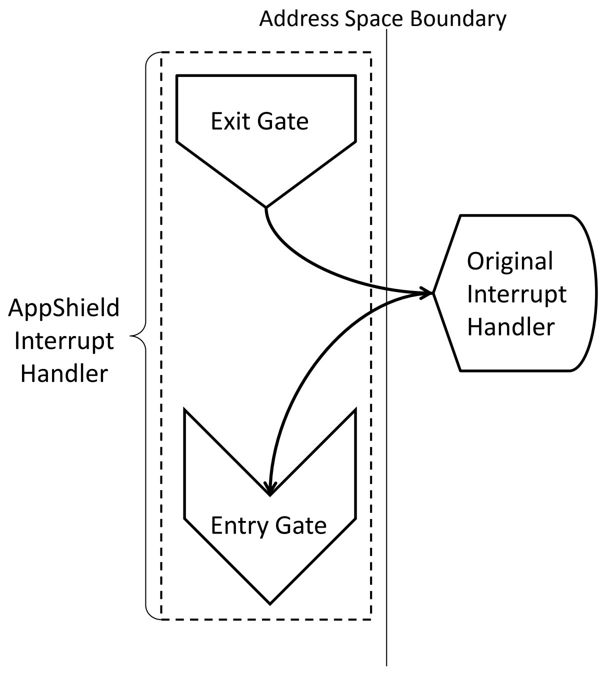

Figure 6: The AppShield interrupt handler.

4.2 Event Capture

We do not use the hypervisor to intercept the interrupt events as this method significantly affects the platform performance. When AppShield is activated, the hypervisor loads the AppShield Interrupt Descriptor Table (IDT) which is dedicated to CAP under protection, and protects it from be modified by the guest kernel by setting its region as read-only.

The AppShield IDT contains the pointers pointing to the AppShield interrupt handlers. The hypervisor installs the AppShield IDT to the CPU occupied by CAP by setting its IDTR register. Consequently, the AppShield interrupt handlers become the first responders to interrupts on the CPU. When the guest OS is running, it uses the original IDT and interrupt handlers. The switch of the two IDTs follows the switch of the address space. As illustrated in Figure 7, the original IDT is uninstalled and the secure IDT is installed for the CAP execution.

By using two sets of interrupt handlers, our design achieves performance overhead localization, because the transit module is only invoked when CAP is interrupted. AppShield is not involved when other applications and the guest OS are running.

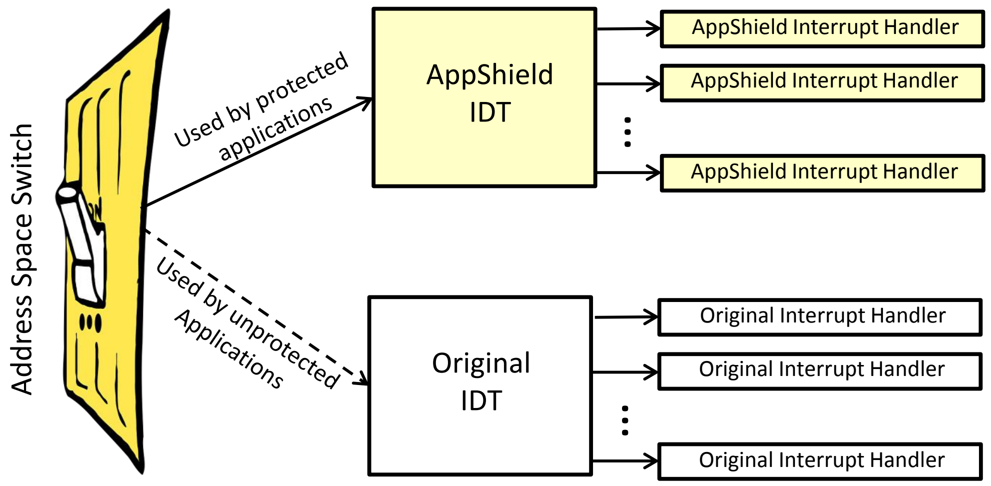
Address Space Switch
Used by protected applications
Used by unprotected Applications
AppShield IDT
Original IDT
AppShield Interrupt Handler
AppShield Interrupt Handler
AppShield Interrupt Handler
Original Interrupt Handler
Original Interrupt Handler
Original Interrupt Handler

Figure 7: **Performance Overhead Localization.** When the context switches to CAP, the normal IDT is uninstalled and the secure IDT is installed.

4.3 Context and Address Space Switch

Figure 8 depicts the control flow of event handling with two context switches at the exit gate and the entry gate. When an interrupt is raised during CAP's execution, the exit gate of the AppShield interrupt handler kicks off the context switch. Under the protection of the hypervisor, the exit gate first prepares a buffer and saves CAP's context in the transit module's private section. It then creates a dummy context for the kernel to execute within. Note that the dummy context should *not* be randomly generated since some context information is used by the kernel to serve the application. For instance, the EIP should point to the corresponding interrupt handler so that the original handler can serve the interrupt. Specifically, we only need to hide the information in the general registers (i.e., EAX, EBX, ECX, EDX, ESI, EDI, EBP) since they may contain sensitive CAP data. In the case of system call context switch, we also need to keep the parameters in the corresponding registers. Moreover, to allow the execution flow to come back to the transit module, the return address of the dummy context is set to point to the corresponding entry gate. In the end, the exit gate then issues a hypercall to inform the hypervisor to restore the original page tables so that the interrupt handler in the guest kernel can properly execute.

Once the guest interrupt handler finishes its process, the control is returned to the entry gate. The entry gate issues a hypercall to request the hypervisor to restore the AppShield EPT. After ensuring that the request is indeed from the legitimate entry gate, the hypervisor restores the AppShield EPT and installs the AppShield

IDT, so that the entry gate can properly restore the saved context and resume the interrupted CAP execution.

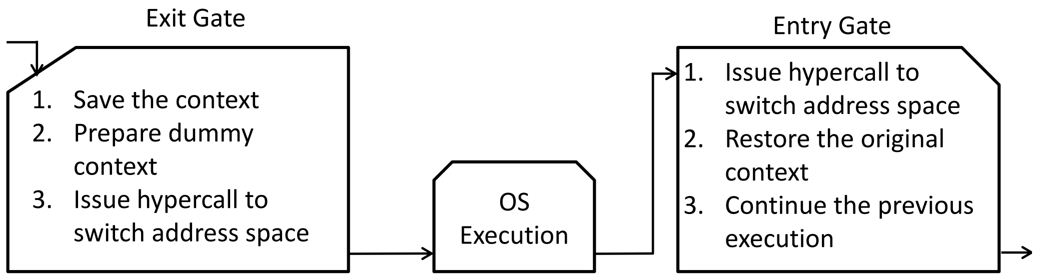
Exit Gate
1. Save the context
2. Prepare dummy context
3. Issue hypercall to switch address space
OS Execution
Entry Gate
1. Issue hypercall to switch address space
2. Restore the original context
3. Continue the previous execution

Figure 8: A typical address space switch always starts with an exit gate and ends with an entry gate. The commodity OS handles the events that trigger the address space switch.

4.4 Special Considerations

Fast-System-Call Cost Localization The platforms equipped with new processor and chipset support fast system call mechanisms (i.e., SYSENTER/SYSCALL, SYSEXIT/SYSRET), which are independently proposed by Intel and AMD, respectively. The SYSENTRER/SYSCALL traps the CPU to the kernel mode and the SYSEXIT/SYSRET transfers the CPU back to the user mode. In this paper we choose one pair (i.e., SYSENTER/SYSEXIT) to illustrate.

The $SYSENTER$ instruction sets the registers (i.e., CS, EIP, SS and ESP) according to values specified by the operating system in certain Model-Specific Registers (MSR), and triggers the CPU to trap into the kernel mode. To localize the performance overhead to CAP, the hypervisor also prepares two sets of MSR registers. One set is used for CAP, where the EIP value in the corresponding M-SR (i.e., *SYSENTER_EIP_MSR*) is modified to point to the new system call handler prepared by the transit module. By doing so, all fast system calls will be intercepted the transit module. Another set is used for the unprotected applications as usual. The two sets of registers are switched following address space switches. Note that the context backup and restoration are still handled by the pairs of the exit and entry gates.

Multi-Thread Execution AppShield supports multi-thread execution of CAP. The child threads could be user threads, which are completely maintained by CAP in user space, or light weight processes scheduled by the guest OS and sharing the same address space with their parent.

The user threads do not have their own contexts since they do not have the kernel structure for scheduling. Therefore, they are transparent to AppShield. In contrast, light weight process threads may have multiple user contexts for CAP, since each of them has its own corresponding structures (e.g., kernel stack) for scheduling. These threads may run in parallel and trap into the guest OS simultaneously. Therefore, by using the base addresses of their kernel stacks as the identifiers, the transit module can distinguish each of them, and save/restore the respective contexts.

5 System Call Adaption

The system call from CAP to the guest kernel exposes some CAP data since they are passed to the guest as parameters. AppShield provides a *spatio-temporal* protection [11] for the data involved in the system call. It ensures that the guest OS can only access the authorized data (spatial protection) during the execution of the system call (temporal protection). The previous sections have explained that temporal protection is achieved by address space isolation and secure context switch. In this section, we describe how AppShield enforces spatial protection through system call adaption.

A majority of system calls do not need the OS to access the application address space to get further information, since all needed information in in the parameters (e.g., *close*) or even without needing any parameters (e.g., *getpid*). These calls are passed to the guest OS without any adaption.

Those system calls whose parameters contain pointers (e.g., a pointer pointing to the file name in *open*), need adapting. To ensure spatial protection, researchers have proposed two possible solutions. The first approach [7] is to interact with the hypervisor multiple rounds to safely move the *decrypted* data into a shared/public buffer. The other approach [11] does not allocate a new buffer. Instead, it decrypts the data in the original buffer and allows the OS to directly access the buffer. Both solutions use expensive cryptographic technique, which dramatically reduces the application performance if the application frequently issues system calls. In addition, the multiple round interaction (with the hypervisor) is another source of the performance loss. We summarize the time cost of the parameter marshalling in a system call in two typical schemes (i.e., Overshadow and SecureME) together with our scheme in Table 1. Overshadow needs 8 context switches and costly cryptographic operations, and SecureME also needs cryptographic operations together with 2 context switches. For our scheme, we only need 2 context switches for one system call.

In our scheme, the trusted shim creates a shared region (buffer) in its user space, and issues a hypercall to inform the hypervisor that the shared region is

	Crypto. Operations	Data Movement	Context Switch (#)
OverShadow	$\surd$	$\surd$	8
SecureME	$\surd$	$\surd$	2
AppShield	X	$\surd$	2

Table 1: The time cost of the parameter marshalling in a system call. Our scheme is relatively efficient because we give up the costly cryptographic operations and reduce the switch times.

accessible for the guest OS. In this way, the guest OS can only access the data within the shared region, but cannot access any other regions within the user space of the CAP, achieving the spatial protection.

To adapt system calls, the shim developers should understand the semantic meaning of each system call. Specifically, they should know the meaning of the parameters and the return values. In addition, they should know the direction of the data exchange, e.g., from application perspective, the buffer referred to by the parameter is for receiving data from the guest OS, or caching the data that will be sent out. Getting such semantic information, they are ready for the system call adaption. Specifically, for the data that the CAP attempts to send out, the shim simply moves the data into a buffer allocated in the shared region, and updates the corresponding parameter to refer to the new buffer. To receive data from the guest OS, the shim should reserve a buffer in the shared region. The shim then saves the base address of the original buffer, and updates the corresponding parameter to refer to the reserved one. When the system call returns, the shim copies the received data into the original buffer and continues the execution.

5.1 System Call Emulation

There are several system calls whereby the system call adaption technique is not applicable to resolve the conflicts between the system call purpose and our security requirements. Specifically, such system calls are not designed for exchanging data. Instead, they are for introspecting or manipulating the application by accessing or modifying internal status.

The first is *Futex* (i.e., fast user mutex), which provides a method for an application to wait for a value at a given address, and a method to wake up other applications waiting on a particular address. The implementation of Futex not only directly accesses the process memory, but also binds some information (e.g., a hash bucket) with the address. Therefore, if we simply apply the system call

Configurations	Descriptions
CPU	Intel i7-2600 with 3.40GHZ
Memory	3GB DDR3 1333MHZ
Network Card	Intel Device 1502 with 1Gbps
Disk	ATA 7200RPM
OS	Ubuntu 10.04 with Kernel 2.6.32.59

Table 2: The configurations of the experiment machine.

adapting technique to Futex, the semantic information may be bound to a wrong address, which may lead to the failure of Futex.

The rest are the system calls used in the *signal-handling*, where the guest OS needs to prepare a temporary execution context for the application and transfers the execution control to a pre-registered handler to handle the corresponding signal. The critical security issue here is that the guest OS needs to be authorized to manipulate the application context. Such authorization may be exploited to reveal and tamper with the application data, e.g., involve a function to send plain text outside. To revoke the authorization from the guest OS, we have to emulate it.

5.2 Ptrace

The *ptrace* system call is not allowed for CAP since its working mechanism requires the guest OS to directly read the content of the user space, or to modify the data or even code of the specific addresses, which is *completely* conflicts with our security requirements. We should not emulate this system call since it opens the door for the malicious guest OS to read/write the whole address space of the CAP.

6 Implementation and Evaluation

We have implemented a prototype of AppShield on a PC whose specification is listed in Table 2. The prototype consists of a dedicated hypervisor running on the bare-metal hardware, and a Linux loadable module as the transit module. The code base of the hypervisor is around $29K$ SLOC with $218KB$ binary size. The transit module consists of around $2K$ SLOC, and the trusted shim is around $1K$ SLOC.

Trusted Shim We do not modify the source code of the application and the shared libraries, instead we create the shim as a wrapper of libc, and allow it to intercept the function calls that are supposed to call the libc functions. Specifically, on the Linux system, an application usually needs shared libraries at runtime, and

System Calls	
Files	open, close, read, write, chdir writev, access, fstat64, uname, poll, fcntl statfs64, fstatfs64, getdents64, getdents stat64, lseek, _llseek, getcwd, fchdir, ioctl
Network	bind, listen, accept, sendto, recvfrom, accept4, select connect, send, recv, getsockname
Memory	mmap2, munmap, mremap, brk, mprotect
Process	getpid, gettid, getgroups32, set_thread_area getuid, geteuid, getgid, getegid exit_groud, tgkill, getrlimit, exit
Time	time, clock_gettime, gettimeofday
Others	futex, rt_sigaction, rt_sigprocmask, sigaltstack

Table 3: Supported system calls.

the dynamic linker loads those shared libraries in whatever order it needs them. However, when you set $LD\_PRELOAD$ to a shared library, that file will be loaded before any other libraries, including the libc library. Preloading a library means that its functions will be used before others of the same name in later libraries, allowing a function to be intercepted. We use this feature in our implementation, saving the cost of the source code modification.

The trusted shim needs to do some initializations and preparations for the protection and the interception, such as allocating the shared buffer, and informing the hypervisor to protected the application. However, those functions for intercepting system calls are passively invoked, meaning those functions will not execute until the application explicitly call them. To solve this problem, we resort to another feature - constructor function. A constructor function marked with *.init* will be called by the dynamic linker when the library is loaded. The trusted shim supports 56 system calls (listed in Table 3) in the current implementation.

We evaluate the impacts of AppShield by running both macro- and micro-benchmark kits.

6.1 Micro Benchmark

In the micro benchmark, we evaluate the cost of the address space switch (Table 4). An address space switch event can be divided into three parts: protection mode switch, context backup and restoration. The protection mode switch includes a hypercall, IDTR and EPT switching. The context backup consists of saving 17 registers (including general, flag and control registers) and creating a

Operation	**Time** (μs)
Out of Protected Address Space	1.72
Back to Protected Address Space	1.33
Context Backup	0.11
Context Restoration	0.08

Table 4: The micro-benchmark results for address space switch.

dummy context. The context restoration is to load all the saved registers. The cost of domain switch is relatively high, because it contains the costly memory access from hypervisor space to guest space, i.e., inserting the return address to the kernel stack. All three costs constitute the latency for the system to handle a particular interrupt or exception. The cost for a system call is for address-space switch cost and parameter marshalingt. The parameter marshaling cost varies for different system calls. For instance, there is no such cost for *getpid*, while we need it to copy data from user space to the shared region in *write*. Thus, we do not measure them individually, but choose to evaluate the whole application performance overhead in macro benchmark.

6.2 Macro Benchmark

AppShield Impacts on Performance SPEC CINT2006 [12] is an industry-standard benchmark intended for measuring the performance of the CPU and memory. We executed SPEC CINT2006 in two setups: system with virtualization, and the system with AppShield. Figure 9 shows the results.

Comparing the impact of running the workload in a system with a bare-metal hypervisor, we calculate the overhead added by the additional virtualization layer. Based on the virtualization impacts, AppShield imposes an additional 0.01% slowdown on average. The primary source of virtualization overhead is VM exits due to interrupts and privileged instructions [15].

Computation Effects We measure the AppShield protection on computation programs. In our experiment, we measure three encryption algorithms (i.e., AES, RC4 and RSA), which is adopted from *OpenSSL 0.9.8k* project. We run these algorithms to encrypt/decrypt messages with different lengths, from 32bytes to 2048 bytes. The measurement results in Figure 10 show that the protection effects on the computation programs is quite small.

Disk I/O Benchmark The disk I/O benchmark includes three sub-benchmarks to evaluate the overhead in disk reading, writing and copying. Disk I/O benchmark reads/writes data from/to files with different sizes. In our experiments, the

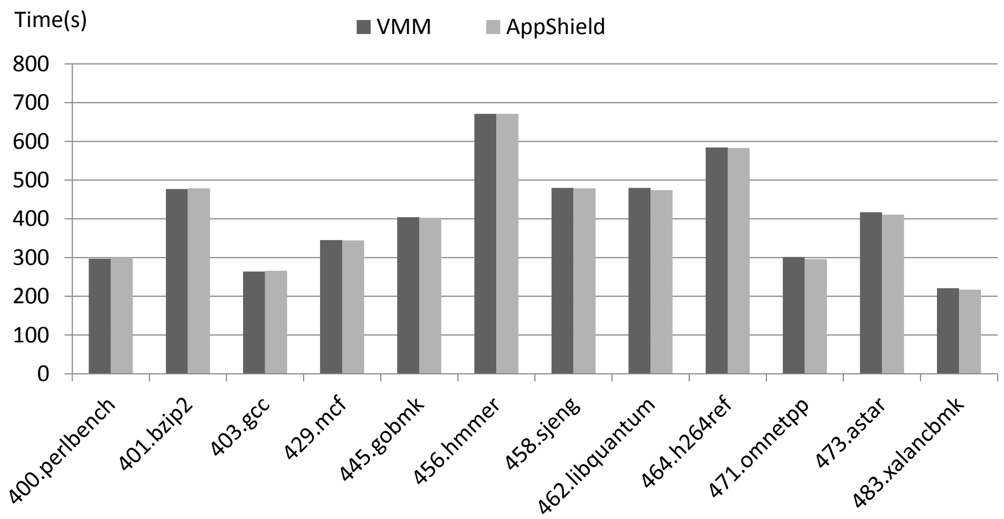

Figure 9: **SPECint 2006 Result.** AppShield introduces insignificant slowdown comparing with virtualization.

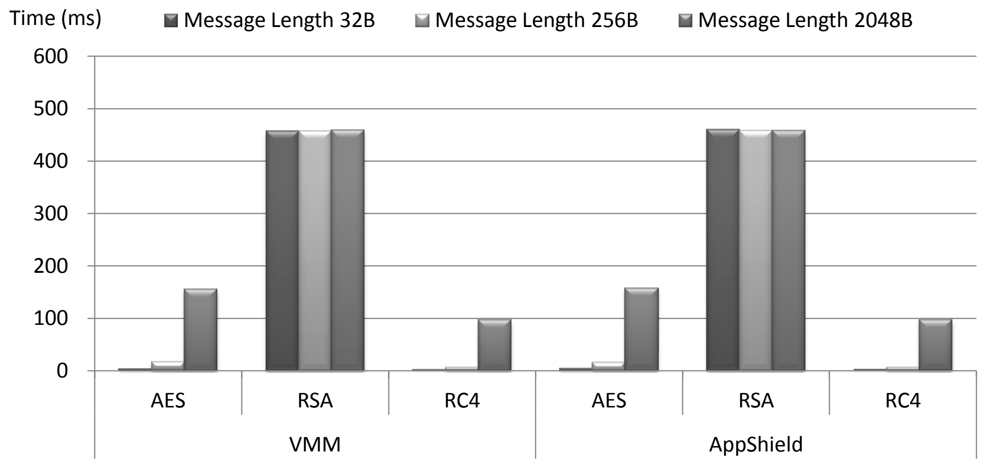

Figure 10: **The effects of AppShield protection on computation.**

file size is 64MB, and the read/write granularity is from 512B to 4MB. Experiments with a larger file and a smaller buffer result in more system calls, and consequently introduce more context switches. However, with the increasing of the buffer size, the performance is better, which is also proved by the experiment results in Figure 11. Note that the overhead is mainly introduced by data copy and context backup/restoration in parameter marshaling.

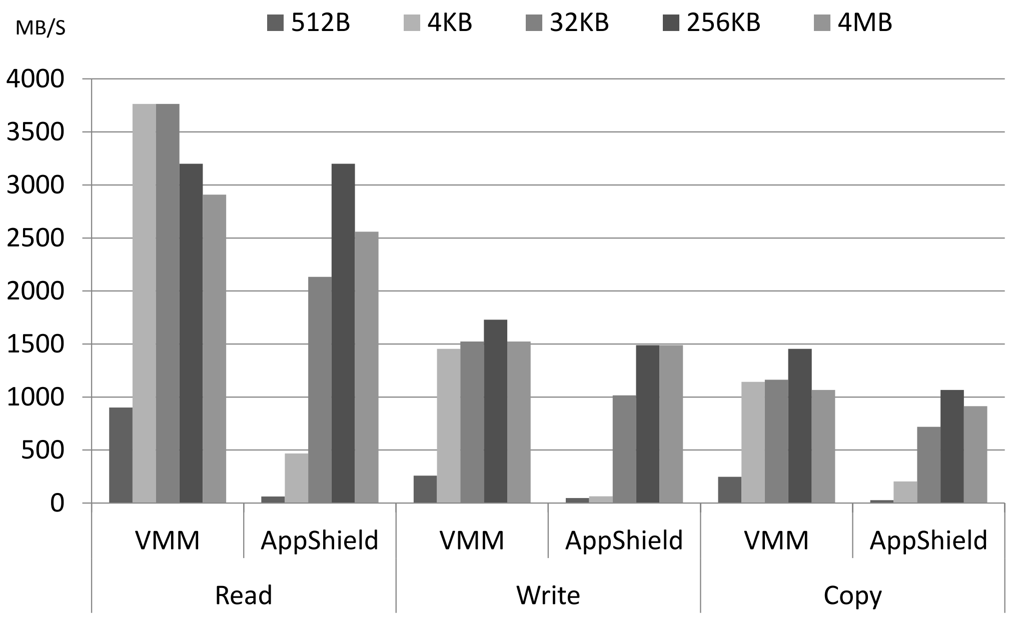

Figure 11: **The disk I/O Benchmark.**

		Linux	AppShield	Overhead
Apache Throughput		320.65 req/s	316.84 req/s	1.01x
Connection Time (ms)	Processing	160	163	1.01x
	Waiting	131	135	1.03x

Table 5: The benchmark results of Apache.

Network I/O Benchmark We measured the network performance with the A-pache web server. The Apache is configured as worker mode with one main process and 20 threads. We run the standard *ab* matchmarking tool included in the Apache utility tools. We execute 10,000 web requests, at the concurrency level of 100 to fetch the default *index* page. The web client and the Apache server are in the same LAN. The Apache web server serves requests with 1.20% overhead in throughput, and about 3.05% overhead in waiting time and 1.86% overhead in processing time. The overhead is reasonable since Apache may cache the frequently requested pages, without issuing disk I/O for each request.

7 Discussions

I/O Data Protection To keep the hypervisor and the transit module small and simple, we do not integrate the device drivers into them, while choose to reuse the legacy ones in the untrusted OS. It means that the untrusted OS have the chance to reveal or tamper with the input data of the CAP. For the file and network data, we can use cryptographic techniques to protect them, e.g., all file and network data are encrypted with the CAP private key. However, the data like keyboard input and mouse input are passed from the devices in plain text, meaning that the untrusted OS is able to get the content or even arbitrarily modify them. To protect such data, we can integrate the trusted path [10, 37] with our AppShield technique.

Fine-Grained Protection Currently, we focus on the whole application protection, while the AppShield technique can be adapted to protect self-contained high-assurance components. To achieve this, we can split the application into low- and high-assurance partitions, and only protect the high-assurance components. In addition, the high-assurance components that are aware of the existence of the AppShield can explicitly communicate with the hypervisor to request more fine-grained protections or certain special services, e.g., the online transaction service.

Verifiable Protection End users usually require a proof (verifiable protection) to indicate the state of the protection. There are several approaches proposed to provide a secure feedback channel. Bumpy [24], ZTIC [20], Lockdown [33] and Trusted Path [37] attempt to use a dedicated (extra) hardware device (e.g., USB token and mobile) as the trusted monitor, while KGuard [8] and Guardian [9] build a visual verification on the display. We can integrate the the visual verification into our hypervisor since it does not need dedicated devices. Note that the visual verification is only involved when the first time the CAP is isolated.

8 Related Work

There are several approaches proposed to protect application code and data, and all of them attempted to remove the OS out of TCB to provide a higher-assurance execution environment.

8.1 Self-contained Code Protection

Flicker [23] system built on the TPM-based Dynamic Root Of Trust (DROT) technology can build an isolation environment to protect a piece of code and data. Due to the limitation of the TPM, the latency of the Flicker system is quite high. To minimize the latency, TrustVisor [22] scheme are proposed. By leveraging virtualization technology, TrustVisor virtualizes the physical TPM into Virtual TPM (VTPMs) and migrate them into hypervisor space. Note that both of them focus on the protection of a small piece of code and data. The increasing of the protection scope, such as protecting the whole application or device drivers, may lead both schemes to failure.

8.2 Whole Application Protection

Secure-Processor-Based Protection. AEGIS [30] and XOM OS [21] are secure-processor based approaches that provide compartments to isolate one application from others. Both of them incur poor computability since they require substantial modifications on the OSes and applications. AEGIS [30] also provide an alternative implementation, which requires to build security into the OS.

Bastion [3] and SecureME [11] aim to deal with untrusted OS and untrusted hardware attacks simultaneously with the assistance of a secure processor. Bastion focuses on the protection of a security module, while SecureME attempts to provide privacy and integrity for data and code of the application. SecureME requires modifications on both OSes and applications.

In addition, a Processor-Measured Application Protection Service P-MAPS [26] is announced by Intel, which is built upon Intel TXT [18] and Intel VT [19] hardware capabilities. P-MAPS provides runtime isolation to protect standard applications with small TCB. P-MAPS is quite similar to our scheme at a high level. However, the details of P-MAPS are unavailable for public to conduct an in-depth comparison.

Microkernel-Based Protection. EROS[27], Perseus[25], Microsoft's NGSCB [13] and Nizza [16] are microkernel(or small kernel) based solutions. They attempt to run commodity OS and untrusted applications in the low-assurance partitions, and run the applications with higher security requirements in the high-assurance partitions, which are isolated and protected by the microkernel itself. However, all of them incur compatibility issue since they may require splitting or even redesigning on the applications.

Virtualization-Based Protection. The approaches like TERRA [14] and Proxos [32] are hypervisor-based trust partitioning systems. They protect applications by isolating them into trusted domains with application-specific OSes. These systems incurs large TCB since they include all secure domains inside. In addition, they are still vulnerable once the application-specific OSes are compromised.

OverShadow [7], CHAOS [6] and SP^3 [36] aim to protect the whole application execution against malicious application and OSes. However, all of them need complex encryption and decryption operations on the application data. Obviously, these additional costly cryptographic operations may reduce the performance and increase the latency of the whole system, especially for the protected application. In addition, none of them claims that they protect applications from the MS attack. Thus, the data and code integrity may still be broken by potentially compromised

OS. InkTag [17] is a new proposed approach, which also protects the whole application and verifies the OS behaviors through paraverfication technique. The paraverfication technique needs to modify the source code of the kernel, which is not always available. Thus, it may lead to the failure of the protection on the close-source OSes, e.g., Windows.

BIOS-Based Protection. Lockdown [33] system relies on a BIOS-assisted lightweight hypervisor and a ACPI-based mechanism to provide two switchable worlds - green world for trusted applications and red world for untrusted applications. Lockdown uses a trusted path built upon LEDs to provide a verifiable protection. The main drawback of the Lockdown system is the switch latency is too high, roughly 40 seconds. SecureSwitch [31] system that is quite similar to Lockdown also leverages a BIOS-assisted mechanism for secure instantiation and management of trusted execution environments. The switch latency is relatively smaller, roughly 6 seconds. Both approaches needs to shut down one world to run another one, meaning they they can not simultaneously execute two worlds.

9 Conclusion

In this paper, we have presented the designed and implementation of AppShield, which reliably and flexibly protects critical applications with complete isolation, rich functionalities and high efficiency. We have implemented the prototype of AppShield with a small bare-metal hypervisor. We have evaluated the performance impacts on CPU computation, disk I/O and network I/O using micro and macro benchmarks. The experiments show thatAppShield is lightweight and efficient.

References

[1] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[3] D. Champagne and R.B. Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[4] David Champagne and Ruby B. Lee. Scalable architectural support for trusted software. In Matthew T. Jacob, Chita R. Das, and Pradip Bose, editors, *HPCA*, pages 1–12. IEEE Computer Society, 2010.

[5] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. ACM.

[6] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P.C. Yew, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Technical Report FDUPPITR-2007-0801, Parallel Processing Institute, Fudan University, August 2007.

[7] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 2–13, New York, NY, USA, 2008. ACM.

[8] Yueqiang Cheng and Xuhua Ding. Virtualization based password protection against malware in untrusted operating systems. In *Proceedings of the 5th International Conference on Trust & Trustworthy Computing*, Vienna, Austria, 2012. Spring.

[9] Yueqiang Cheng and Xuhua Ding. Guardian: Hypervisor as a security foothold for personal computers. In *Proceedings of the 6th International Conference on Trust & Trustworthy Computing*, London, UK, 2013. Spring.

[10] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. Driverguard: a fine-grained protection on I/O flows. In *Proceedings of the 16th European conference on Research in computer security*, ESORICS'11, pages 227–244, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 108–119, New York, NY, USA, 2011. ACM.

[12] Standard Performance Evaluation Corporation. Spec cint2006. http://www.spec.org/.

[13] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *Computer*, 36(7):55–62, July 2003.

[14] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 193–206, New York, NY, USA, 2003. ACM.

[15] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 411–422, New York, NY, USA, 2012. ACM.

[16] H. Hartig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The nizza secure-system architecture. In *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*, pages 10–pp. IEEE, 2005.

[17] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: secure applications on an untrusted operating system. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM.

[18] Intel. Intel Trusted Execution Technology (Intel TXT) software development guide. Dec 2009.

[19] Intel. Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. October 2011.

[20] IBM Zurich Research Lab. Security on a stick, October 2008.

[21] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 178–192, New York, NY, USA, 2003. ACM.

[22] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and

attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[23] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[24] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, February 2009.

[25] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, and A. Weber. The perseus system architecture. In *VIS*, pages 1–18, 2001.

[26] R. Sahita, U. Warrier, and P. Dewan. Dynamic software application protection. *Intel Corporation, Apr*, 2009.

[27] J.S. Shapiro. *EROS: A capability system.* PhD thesis, University of Pennsylvania, 1999.

[28] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 477–487, New York, NY, USA, 2009. ACM.

[29] Raoul Strackx and Frank Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM.

[30] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[31] Kun Sun, Jiang Wang, Fengwei Zhang, and Angelos Stavrou. Secureswitch: Bios-assisted isolation and switch between trusted and untrusted commodity oses. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[32] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 279–292, Berkeley, CA, USA, 2006. USENIX Association.

[33] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig. Lockdown: towards a safe and practical architecture for security applications on commodity platforms. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 34–54, Berlin, Heidelberg, 2012. Springer-Verlag.

[34] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: a hardware-assisted integrity monitor. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 158–177, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[36] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[37] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.