Efficient Virtualization-Based Application Protection Against Untrusted Operating System

Yueqiang Cheng*
CyLab, Carnegie Mellon University
4720 Forbes Ave
Pittsburgh, PA, USA
chengyue@andrew.cmu.edu

Xuhua Ding
Singapore Management University
80 Stamford Rd
Singapore, 178902
xhding@smu.edu.sg

Robert H. Deng
Singapore Management University
80 Stamford Rd
Singapore, 178902
robertdeng@smu.edu.sg

ABSTRACT

Commodity monolithic operating systems are abundant with vulnerabilities that lead to rootkit attacks. Once an operating system is subverted, the data and execution of user applications are fully exposed to the adversary, regardless whether they are designed and implemented with security considerations. Existing application protection schemes have various drawbacks, such as high performance overhead, large Trusted Computing Base (TCB), or hardware modification. In this paper, we present the design and implementation of AppShield, a hypervisor-based approach that reliably safeguards code, data and execution integrity of a critical application, in a more efficient way than existing systems. The protection overhead is localized to the protected application only, so that unprotected applications and the operating system run without any performance loss. In addition to the performance advantage, AppShield tackles several newly identified threats in this paper which are not systematically addressed previously. We build a prototype of AppShield with a tiny hypervisor, and experiment with AppShield by running several off-the-shelf applications on a Linux platform. The results testify to AppShield's low performance costs in terms of CPU computation, disk I/O and network I/O.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Isolated Execution Environment

General Terms

Security

Keywords

Address Space Isolation, Untrusted OS, Application Protection, Isolated Execution Environment

*The work was done at Singapore Management University.

ASIA CCS'15, April 14–17, 2015, Singapore, Singapore.

http://dx.doi.org/10.1145/2714576.2714618 .

1. INTRODUCTION

With a superior privilege than user applications, the commodity monolithic operating systems are often regarded as the security basis of systems. A conundrum facing the end users is that the commodity OSes are not always trustworthy as expected. Their enormous code size and broad attack surfaces make them vulnerable to attacks. Once an OS is subverted, all applications and sensitive data are at the mercy of the attacker. Moreover, some high-profile end users and organizations are even concerned about whether the commodity operating systems in their use are purposely implemented with trapdoors to invade their privacy or data secrecy.

To cope with OS level attacks, various mechanisms [23, 22, 28, 3, 6, 34, 31, 5, 29] have been proposed to protect critical applications without trusting the operating system. self-contained code Among them, the approaches like Flicker [23], TrustVisor [22] and Fides [28] are only applicable to self-contained code with predefined inputs and outputs (e.g., inputs are the initial parameters and outputs are the final returns). Those code cannot even make the basic system calls for dynamic memory allocation or deallocation. Although MiniBox [20] extends the functionality by supporting system calls of the self-contained code, it still has several limitations, such as lack of multi-thread support and limiting to sandbox-capable modules.

To protect a full-fledged application, several systems [29, 21, 2, 9, 6, 34, 31, 5, 14, 17] are proposed. Among them, AEGIS [29], XOM OS [21], Bastion [2] and SecureME [9] require hardware modifications, which is apparently impractical for current commodity platforms. Intel's upcoming Software Guard Extensions (SGX) [10] technology provides a suite of hardware extension for software protection which requires significant changes on the software level, and therefore is not compatible with legacy applications. Proxos [31] and Terra [14] introduce a dedicated trusted virtual machine for the protected applications, an approach with a dramatically expanded TCB size and therefore a weaker security assurance.

The systems like OverShadow [6], CHAOS [5], SP^3 [34], InkTag [17] aim to protect the whole process without requiring hardware modifications or a trusted VM. However, they all rely on the costly encryption/decryption operations and are subject to the newly identified attacks (as described in Section 3.2.1) whereby the kernel manipulates the address mapping, e.g., the malicious OS could swap two address translation mappings to break the data/code integrity without directly modifying the data/code of the protected application. Virtual Ghost [12] prevents the kernel from illicitly accessing application memory by instrumenting memory access instructions and enforcing a complete control flow integrity in the

kernel, which are usually not applicable for commodity OSes (e.g., Windows) in practice.

In this paper, we propose AppShield, a novel system which *reliably*, *efficiently* and *transparently* protects data secrecy and integrity of a critical application, as well as its execution integrity, against OS-level malware attacks (Figure 1). AppShield leverages the 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 setting, since it can request the kernel to (de)allocate memory buffers. Its memory accesses are in the native speed without computation-heavy encryption/decryption or being intercepted. Furthermore, AppShield also achieves *performance isolation* which isolates the performance loss only to the protected application, keeping those unprotected applications not affected. Our scheme is complementary to secure I/O (e.g., DriverGuard [8] and Trusted Path [35]) and encrypted I/O (e.g., SSL for network data) schemes such that they can jointly provide a holistic protection on the application and its I/O data.

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 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.

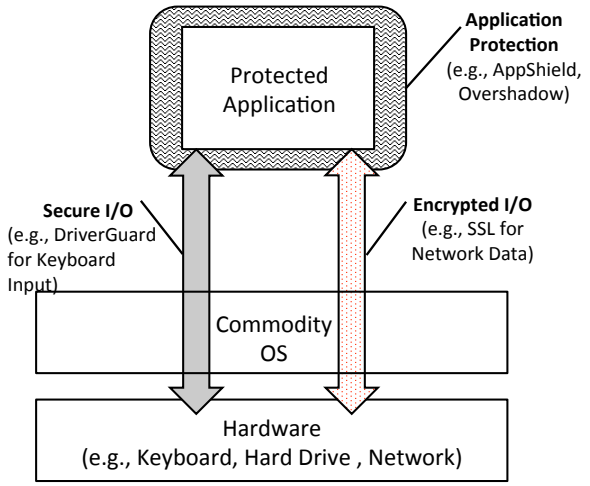

Figure 1: Protections for application and its data. Our scheme is complementary to secure I/O and encrypted I/O schemes.

To summarize, our contributions of this paper are listed below.

- We discover several address mapping attacks which are not addressed by existing application protection schemes. Through these attacks, the malicious kernel can tamper with the protected application's data secrecy and execution integrity.
- We design AppShield to protect execution integrity, data secrecy and integrity of an off-the-shelf application which is allowed to issue system calls.
- Compared with existing work in the literature, our AppShield design achieves better performance by avoiding computation-heavy cryptographic operations and enforcing performance isolation in the sense that the performance loss is incurred to the protected application *only*.
- We build a prototype of AppShield and evaluate it with several commonly used applications and benchmark tools. The experiments show that AppShield does not introduce high overhead to the system.

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

2. SYNOPSIS

2.1 The Model

In this work, we defend against kernel-level malware attacks on a critical application by tampering with the latter's data and/or execution. The adversary can run arbitrary code and launch DMA operations in the victim platform. Nonetheless the adversary does not have physical control over the platform. All hardware and peripheral devices, together with their firmware, are considered as trusted. In other words, the platform's chipset and all peripheral devices operate as expected, namely, following their specifications and not containing Trojan-Horse circuits or microcode that respond to commands of the adversary. In our model, we trust the a bare-metal hypervisor in use, which has a tiny code size and limited number of interfaces. Moreover, we assume our hypervisor intercepts and emulates the System Management Mode (SMM) operations in order to tackle SMM-based attacks by leveraging SMM containerization. In fact, Intel has supported such mechanism [19].

Neither side channel attacks nor application availability is in the scope of our study. We also suppose that no ill-formed inputs can subvert the control flow of the critical application which can be achieved by input sanitation and proper code development. It is *orthogonal* to our work to enhance code security (e.g., fixing bugs) of the protected applications.

Our goal is to protect a critical application execution integrity and data security within the application memory space. The critical application may use cryptographic techniques to protect its derived data for disk and network I/O, and may leverage existing secure I/O path schemes like [8, 35] to protect the raw I/O data for peripheral devices such as a keyboard and a fingerprint reader.

2.2 Design Principles

In the design of AppShield, we follow the four principles described below. Firstly, it should support and protect the application's system calls. The critical application can safely issue system calls to request the services (e.g., memory allocation) from the operating systems even though the latter is not trusted.

Secondly, no significant performance impact should be inflicted by AppShield on the protected application and on the platform as a whole. Ideally, the protected application accesses the main memory in the native speed without being interposed on or going through an encryption/decryption procedure. Moreover, the mechanism should take a limited performance toll on the protection application, whereas other unprotected applications and the OS are not affected. We term this property as *performance localization* in this paper.

Thirdly, out of the practicality consideration, we intend to design AppShield to be compatible and even transparent to legacy off-the-shelf applications. The requirement of significant source

code or binary code modifications hinders the adoption of AppShield in practice. Note that the compiling and installation of the kernel module do not rely on the source code of the operating system.

Lastly, as a widely accepted design principle, the TCB of the security mechanism should be kept small and simple, which ensures that the risk of subverting the TCB is minimal. Therefore, it excludes the approach of using a trusted virtual machine where an operating system is part of the TCB.

2.3 AppShield Overview

The high level idea of AppShield is to dynamically isolate the target application's context (registers) and address space from the rest of the platform (including the kernel) in an exclusive fashion, while its system calls are securely mediated by the hypervisor to fend off attacks. For the easiness of presentation, we use CAP in the rest of the paper to denote the critical application under AppShield's protection.

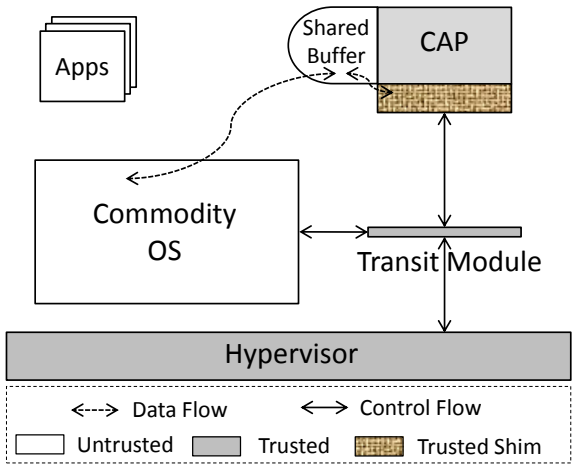

Figure 2: The architecture of AppShield. The data flows (dotted lines) between the protected Critical APplication (CAP) always go through the shared buffer and mediated by the *shim* code. The control flows (solid lines) between CAP and the OS are mediated by the Transit Module. The executions of transit module and the trust shim are protected by the hypervisor.

Figure 2 depicts the architecture of AppShield. It consists of a bare-metal hypervisor, a *transit module* in the guest kernel[1] space mediating *control flow* transitions between CAP and the kernel, and a *shim code* in the user space assisting inbound and outbound *data flows*. Both the transit module and the shim code are self-contained and safeguarded by the hypervisor to defend against attacks from the kernel and malicious DMA requests as in [8, 22]. The AppShield hypervisor as the root of trust in our system could boot up using SRTM [32] (Static Root of Trust for Measurements) and DRTM [18] (Dynamic Root of Trust for Measurements) techniques. Thus, we could install AppShield during runtime, even if the guest operating system is already infected before installing the hypervisor, because any integrity violation could be verified.

CAP runs in an address space isolated from the rest of the guest domain, while the guest OS and other unprotected applications on the platform run as usual without being affected. The page table of CAP is managed by the guest OS, but its updates are intercepted and verified by the hypervisor to defend against various attacks including the new ones introduced in Section 3.2.1. Data flows in system calls are mediated by the shim code which is essentially a wrapper of `libc` libraries. Thus, it does *not* require source code of libc and any modification on the protected application. 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.

The transit module regulates the control flow transitions between CAP and the guest kernel. The transitions are triggered by the events including system calls, exceptions and interrupts. In order to respond to those events before the guest kernel, the new handlers within the transit module are invoked before the handlers in the kernel so as to prevent the context switch from being manipulated.

3. DYNAMIC ADDRESS SPACE ISOLATION AND VERIFICATION

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

3.1 Dynamic Address Space Isolation

In a nutshell, the physical memory assigned to the guest is divided into two separated regions by the hypervisor. One (trusted) region is used for CAP while another (untrusted) region is for the guest OS and other applications. The hypervisor configures IOMMU to prevent malicious DMA requests to access the trusted region. To prevent illicit software access, the memory dichotomy as depicted in Figure 3 is realized by two suites of Extended Page Tables (EPTs) maintained by the hypervisor, respectively. The EPT enforced address space isolation ensures that the guest OS and other untrusted applications can never access the memory regions assigned to CAP; on the other hand, CAP cannot access memory regions belonging to the guest system either. 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.

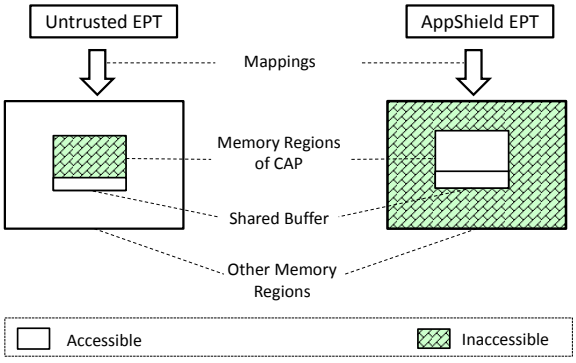

Figure 3: Address Space Isolation. With the AppShield EPT, only the memory regions of CAP and the shared buffer are accessible and other memory regions are inaccessible. In contrast, with the original EPT, CAP's regions except the shared buffer are inaccessible.

[1]Guest kernel refers to the kernel running in the guest VM.

3.1.1 Activation and Deactivation

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 page table base address from `CR3` register and traverses the page table entries (PTEs) belonging to the application, so that it locates all pages within the address space, including the shared libraries. Note that the transit module copies the shared libraries into the isolated space region, and keeps all original shared libraries to be used by the untrusted applications. 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 entries 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 the integrity of the protected application has been priorly authenticated by a signature or an HMAC tag. In addition, the hypervisor measures its integrity and ensures that its memory region does not overlap with existing ones.

With the deactivation hypercall, CAP notifies the hypervisor to disable the protection. In response, the hypervisor first ensures that it is issued by CAP, and then destroys the AppShield EPT and restores the entries in the original EPTs.

3.1.2 Tracking Address Space Updates

One of the main tasks of isolating a full-fledged 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 invoked by the corresponding memory usage functions in the `libc` library, such as *malloc* and *free*.

The details of how AppShield mediates system calls are described in the next section. Here we focus on explaining how the hypervisor dynamically maintains address space isolation, which demands the hypervisor to track memory space updates at runtime.

To track page table updates, one design option is to let the hypervisor directly manages a dedicated guest page table for the CAP. Obviously, it significantly increases the hypervisor complexity which weakens the security strength. The paraverfication technique approach used by InkTag [17] is another alternative. However it requires massive modifications of the OS. To follow the design principles we put forth in Section 2.2, our design utilizes paravirtualization as Xen [1] whereby the guest page tables, being set as read-only, are managed by the kernel and any updates are trapped into the hypervisor and conducted by the hypervisor.

3.2 Address Space Verification

Although the malicious kernel does not have a direct write access to CAP's guest page table, it may manipulate the virtual and/or physical address of the newly allocated memory regions to compromise CAP's security without accessing the latter's memory space. One such example is the Iago attack [4] which relies on a vulnerability in libc libraries. In the following, we first show several newly discovered attacks in the same vein, but in a more generalized setting, and then we show how the hypervisor in AppShield verifies an address space update (before isolating it) to counter these attacks.

3.2.1 Address Mapping Manipulation

In general, address mapping 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.

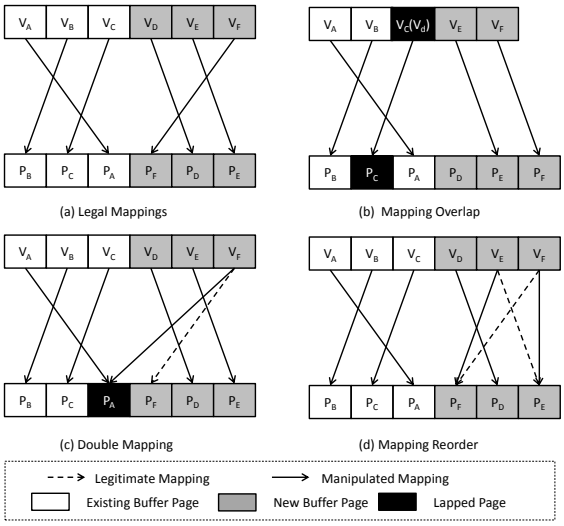

Figure 4: Address mapping manipulation attacks. $V_A, \cdots, V_F$ are regions in the virtual addresses and $P_A, \cdots, P_F$ are their guest physical pages respectively.

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. When there is no attack, the newly allocated buffer's virtual address and physical addresses do not overlap with any existing regions, as illustrated in Figure 4-(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 both virtual and physical address space. As illustrated in Figure 4-(b), the new buffer is set to the pages located at V_D to V_E. The overlapping of V_C and V_D leads to undesired modifications of data in P_C when the application attempts to update the first page of the allocated buffer (through the mapping from V_D to P_C). Obviously this attack breaches data integrity. It can also subvert the control flow of CAP when the overlapping memory is in the application stack and the modifications change the stored return address(es).

Double Mapping Attack. This attack maps two or more virtual pages to one physical page in the user space. In double mapping attack, there are only overlaps in the physical address space but no overlap in the virtual address space, which is the main difference with the mapping overlap attack. As shown in Figure 4-(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 Re-order Attack. The mapping re-order attack is to reorder the existing address mappings between the virtual addresses and the physical addresses. As shown in Figure 4-(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. In this attack, the malicious kernel release one or more existing mappings without any system call requests from the protected application. The mapping-release could induce the hypervisor to give up the protection on those pages since they are not considered in CAP's addresses space. By doing so, the guest OS can freely access the data on those released pages.

Essentially, these attacks can be neutralized by the hypervisor via monitoring and verifying changes to the guest page tables. To the best of our knowledge, no existing work precisely describe the verification procedures and many of them [6, 34] suffer from one or more aforementioned attacks. Moreover, it is not easy to efficiently verify them due to frequent page table updates.

3.2.2 Context Information Collection

In order to determine whether an address space change is legitimate, the hypervisor needs to be aware of the present memory layout, the application's intent to memory updates and the resulting page table updates following the system calls.

The existing memory layout (the mapping relationship between guest virtual addresses to guest physical addresses) is collected by traversing CAP's guest page table. The collected information is trustworthy since it is collected by the hypervisor and the guest page table is set as read-only so that the kernel cannot directly update it.

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 securely pass such information to the hypervisor.

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 overlap with any existing memory region, in both the virtual address space and the guest physical address space.

3. Once the mappings between the virtual addresses and the guest physical addresses are fixed, they are not allowed to be re-mapped.

4. The memory regions can be only released upon CAP's requests, and the page data should be cleaned before allowing the guest OS to manage/access it.

The shim code checks the overlap in the virtual address space after system calls, because it wraps all libc functions related to system calls and therefore has the entire virtual address layout. Taking *mmap* as an example, the trusted shim stores the size of the memory-mapped region through the second parameter of *mmap* and the base address through the return value. Such information are securely deposited in an ordered list which is inaccessible from the kernel 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 interprets the present mapping (denoted as M) and the resulting mapping M', and analyzes the intent of this update. If the guest kernel is to build a new mapping (i.e., M is empty and M' 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 and the request is denied; otherwise the update is approved. If the guest OS aims to remap/reorder the mappings (i.e., both M and M' point to the guest physical pages), the hypervisor directly rejects it.

If the guest kernel aims to free an existing mapping (i.e., M points to a guest physical page while M' is empty), the hypervisor verifies whether it has been priorly informed with CAP's such requests via the shim code's hypercall. The addresses and sizes of those memory pages to be freed 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 zeroed by the trusted shim once it gets the release requests from the CAP.

Note that all mapping updates should be driven by the requests from the application itself. Thus, the above verification algorithm does not prevent normal memory sharing within user space, e.g., a JIT compiler may request two virtual address for its code, one read/execute only, the other for writing.

4. SECURE ADDRESS SPACE SWITCH

Events like system calls, interrupts and exceptions lead to context switches between CAP and the kernel. Different from the conventional user-kernel context switch, the switch between CAP and the kernel involves address space switches, since they run in two exclusively separated 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 5, 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.

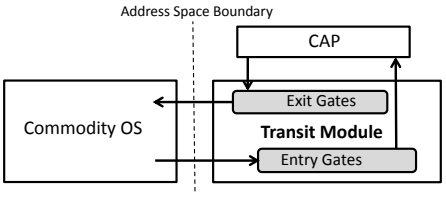

Figure 5: Control flow between the CAP and the guest kernel. Each control flow of CAP starts from an entry gate and ends with an exit gate.

4.1 Components of Transit Module

The transit module is a self-contained kernel module with its execution being protected by the hypervisor using the mechanism described in [27]. 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 predefined addresses called *entry/exit gates* as in Figure 5.

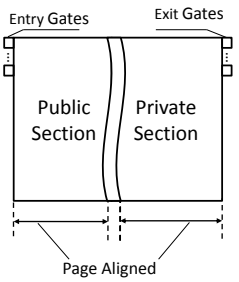

Figure 6: The format of transit module

The transit module has two sections (Figure 6), 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 comprise an AppShield Interrupt Descriptor Table (IDT) which points to a set of its own interrupt handlers called *AppShield interrupt handlers*. Note that the interrupts are still handled by the guest kernel as in the normal setting, not by the transit module. As explained in subsequent sections, the transit module is for AppShield to capture events and protect the context switch without having performance effect on the kernel or other applications.

An AppShield interrupt handler is composed of two code stubs (Figure 7): the *entry gate* in the public section and the *exit gate* 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 of their working mechanisms are presented in Section 4.3.

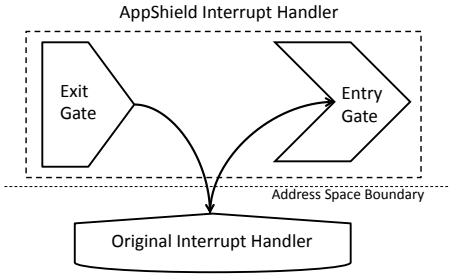

Figure 7: An AppShield interrupt handler consists of one pair of gates, which invokes the original interrupt handler to response the corresponding interrupt and mediates the return from the kernel to CAP.

4.2 Event Capture

The hypervisor in AppShield does not intercept interrupt events since it will significantly affect the platform performance. When AppShield is activated, the hypervisor loads and protects the transit module which captures events within CAP, so that unrelated applications are not involved.

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 in the VMCS structure. Consequently, the AppShield interrupt handlers become the first responders to interrupts on the CPU occupied by CAP. They use hypercalls to notify the hypervisor when necessary. When the guest OS is running, it still uses the original IDT and interrupt handlers. The switch of the two IDTs follows the switch of the address space. As illustrated in Figure 8, the original IDT is uninstalled and the secure IDT is installed for the CAP execution.

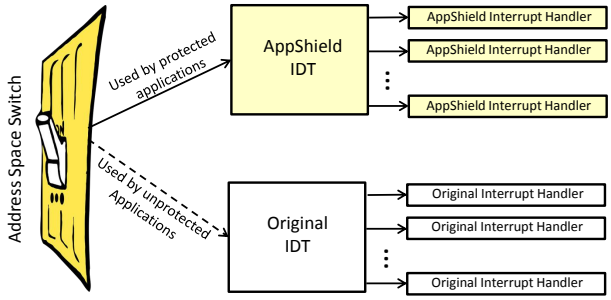

Figure 8: Performance Overhead Localization. When the context switches to CAP, the normal IDT is uninstalled and the secure IDT is installed.

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 with the executions of other applications or the guest OS.

4.3 Context and Address Space Switch

Figure 9 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 sets out to the context. 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 for 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.

To allow the execution flow to securely 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 and guest page tables. 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.

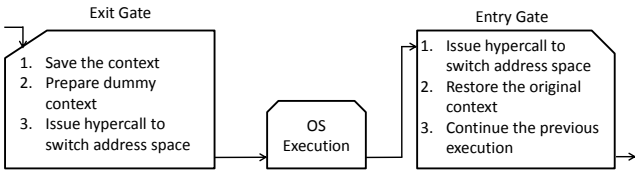

Figure 9: A typical address space switch always starts with an exit gate and ends with an entry gate. The commodity OS handles the events that trigger the address space switch.

4.4 Special Considerations

4.4.1 Fast-System-Call Cost Localization

Platforms with modern processor and chipset support fast system call mechanisms by introducing new instructions. The SYSENTRER(SYSCALL) instruction traps the CPU to the kernel mode while the SYSEXIT(SYSRET) instruction transfers the CPU back to the user mode. In this paper we use SYSENTER and SYSEXIT instruction pair to illustrate the localization mechanism.

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 uses an additional set of `MSR` dedicated for CAP, where `EIP` value in the corresponding `MSR` (i.e., `SYSENTER_EIP_MSR`) is set to point to the corresponding exit gate within the transit module. By doing so, all fast system calls will be intercepted the transit module. The guest kernel uses its own `MSRs`. 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. ItâĂŹs a well known fact that when multiple processors are present in a system, every processor has its own set of MSRs. Thus, the modifications of MSRs for the protected application do not affect other applications running on other processors.

4.4.2 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 which share the same address space with their parent and are scheduled by the guest OS.

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., the 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 MEDIATION

The system call from CAP to the guest kernel reveals some application data when they are passed to the kernel as parameters. AppShield provides a *spatial-temporal* protection [9] 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. According to the security risks, we adapt those low risk system calls and emulate those high risk ones.

5.1 System Call Adaption

In the majority of system calls, the application information needed by the kernel, if any, is passed as parameters and there is no need for the kernel to access the application address space. These calls are not adapted in AppShield.

Nonetheless, system calls with parameters of the pointer type (e.g., a pointer pointing to the file name in *open*), requires the kernel to access the application's space to acquire needed information. In order to prevent the kernel misuse such accesses, it is desirable to adapt those system calls with parameter marshaling.

To ensure spatial protection, two approaches of parameter marshaling have been proposed in the literature. One approach as used in [6] is to interact with the hypervisor eight times to safely move the decrypted data into a newly allocated shared/public buffer. Obviously, the multiple round interaction with the hypervisor is detrimental to the system performance. The other approach as in [9] incurs less context switches as it decrypts the data and overwrites the cipher text using the same buffer. Nonetheless, both approaches use encryption algorithms which consume an order of magnitude more CPU cycles than conventional system calls. Therefore, the performance deteriorates significantly when CAP frequently issues system calls. We summarize the performance cost of the parameter marshaling in a system call in these two approaches (i.e., Overshadow and SecureME) together with our scheme in Table 1.

	Crypto. Operations	Data Movement	Context Switch (#)
OverShadow [6]	yes	yes	8
SecureME[9], InkTag[17]	yes	yes	2
AppShield	no	yes	2

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

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

To adapt system calls, we develop the shim code with the semantics of each system call, i.e. the parameter semantics and the return values. In addition, the semantics also includes the data flow direction, i.e. whether the memory buffer referred to by the parameter is to receive data from the guest kernel, or store the data to be sent out to the kernel.

Specifically, for the data that the CAP attempts to send out, the shim simply copies 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.

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.

5.2 System Call Emulation

There are several system calls whereby the system call adaption technique is not applicable to resolve the conflict between the system call service and the security requirement. Specifically, such system calls are not designed for exchanging data. Instead, they are used to introspect or manipulate the application by accessing or modifying its internal state. For such system calls, we have to emulate them in the transit module.

The fast user mutex system call (*futex*) allows an application to wait for a value at a given address, and to wake up other applications waiting on a particular address. The handler of *futex* not only directly accesses the process's space, but also binds some information (e.g., a hash bucket) with the address. Therefore, the system call adaption technique described previous leads to the failure of *futex* as the information is bound to an incorrect address.

Other system calls requiring emulation are for signal handling, where the guest kernel needs to prepare a temporary execution context for the application and transfers the execution control to a preregistered handler to handle the corresponding signal. The critical security issue here is that the guest kernel 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.

CAVEAT Note that the *ptrace* system call is not allowed in AppShield 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 cannot be reconciled with the security requirements. We do not emulate this system call.

6. IMPLEMENTATION AND EVALUATION

We have implemented a prototype of AppShield on a PC with Intel i7-2600 (3.4GHZ), 3GB memory and Ubuntu 10.04 with kernel 2.6.32.59 (Table 2). The prototype consists of a dedicated hypervisor [7] 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 or 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 run-time, and the dynamic linker loads those shared libraries in whatever order it needs them. However, when LD_PRELOAD is set for a shared library, it will be loaded before any other libraries, including the libc library. Pre-loading a library means that its functions will be used before others of the same name. We use this feature in our implementation, saving the cost of the source code modification.

The trusted shim needs to do some initialization and preparation 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 that they do not execute until the application explicitly calls 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 in our implementation supports 56 most commonly used system calls as listed in Table 3 below.

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, socketcall
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.

Implementation Challenges. The techniques used in the system call interception and parameter marshaling are not as trivial as they seem. The operations of each system call and the related data structures are rather complex. For example, *socketcall* supports many possible operations on the selected file. The operation is determined by a *command* parameter. There are up to 20 command options, and the commands could impact the meanings of other parameters and invoke different data structures. Handling all these variations requires both deep understanding and careful implementation. The challenges of implementing performance isolation are related to the installation of interrupt handlers which involve special steps (e.g., saving context) in the assembly code before invoking the corresponding native C-code handler. The assembly code needs to prepare the stack and registers (e.g., as parameters) for the native handler. Since this piece of code usually breaks the stack layout and alters the register values, those information have to be stored properly right before executing the code and are restored by the code right before invoking the C-code handler. All these operations are further complicated by the requirement that they should be finished in an atomic way. Any interrupt during the operations overwrites the saved context and/or breaks the stack layout.

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 registers (including general, flag, control registers, and MMX/SSE/AVX registers) and creating a dummy context. The context restoration is to load all the saved registers. The cost of address space 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 composed of the address-space switch cost and the parameter marshaling cost. The latter varies with different system calls. For instance, there is no such cost for *getpid*, while *write* involves a data copy. Thus, we do not provide individual evaluation. However, they are reflected in macro

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.

benchmarking which evaluate the whole application performance overhead.

6.2 Macro Benchmark

In macro benchmarking, we apply AppShield on several applications (including Apache, and *ls*, *vim* on Linux) as well as benchmark tools, and measure their performance effects on computation, disk and network I/O.

6.2.1 AppShield Impacts on Performance

SPEC CINT2006 [11] is an industry-standard benchmark intended for measuring the performance of the CPU and memory. We executed SPEC CINT2006 in two setups: with and without AppShield protection. Without AppShield protection, the performance overhead is due to the virtualization itself. The full evaluation has been reported in [7]. Generally, it only introduces 0.2% to 10.3% performance overhead. In addition to the virtualization cost, AppShield has 0.01% slowdown on average. The primary source of virtualization overhead is VM exits due to interrupts and privileged instructions [15]. Figure 10 shows the results.

6.2.2 Computation centric programs

We measure the AppShield's protection on computation-intensive programs. In our experiment, we measure three encryption algorithms (i.e., AES, RC4 and RSA) from *OpenSSL 0.9.8k* package. We run these algorithms to encrypt/decrypt messages with different lengths, from 32bytes to 2048 bytes. The measurement results in Figure 11 shows that the protection effects on the computation programs is quite small.

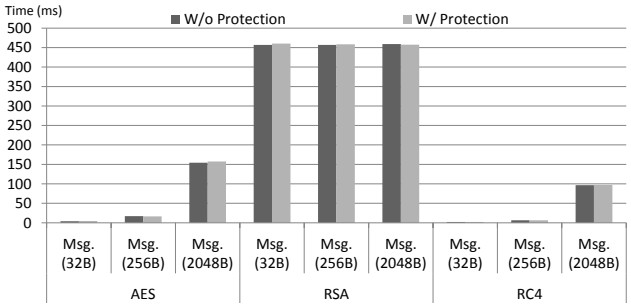

Figure 11: The effects of AppShield protection on computation

6.2.3 Disk I/O centric programs

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 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 12. For example, the performance overheads with 4KB-granularity are quite high, and have (81.91%, 71.84%, 74.57%) for (read, write, copy) respectively, while the performance overhead with 256KB-granularity are very small, only has (0.68%, 4.52%, 0.00%) for (read, write and copy) respectively. Note that the overhead is mainly introduced by data copy and context backup/restoration.

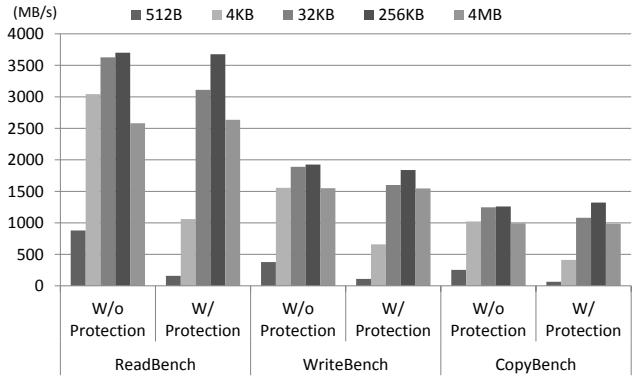

Figure 12: The disk I/O Benchmark

6.2.4 Network I/O Benchmark

We measured the network performance with the Apache web server. The server is configured in worker mode with one main process and 20 threads. We run the standard *ApacheBench* 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. With AppShield, 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. We also compare AppShield against Overshadow[6] and InkTag [17] in Table 6. Note that Apache may cache the frequently requested pages, without issuing disk I/O for each request, which helps to reduce the overhead. We also compare the network performances under Overshadow, InkTag and AppShield protections, The results that are listed in Table 6 also indicate that our scheme have the lowest performance overhead and latency on network I/O.

	Linux	AppShield	Overhead
Throughput (req/s)	321	317	1%
Conn. Processing (ms)	160	163	2%
Conn. Waiting (ms)	131	135	3%

Table 5: The benchmark results of Apache performance

	Overhead		
	OverShadow	InkTag	AppShield
Req. Throughput	100%	2%	1%
Conn. Latency	–	13%	3%

Table 6: Network performance comparisons with Overshadow and InkTag.

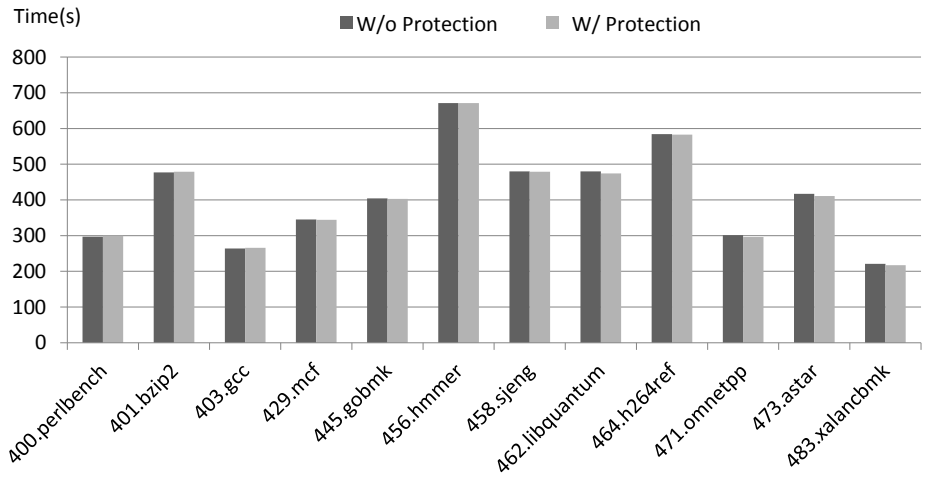

Figure 10: SPECint 2006 Result. AppShield introduces insignificant slowdown.

7. 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.

7.1 Self-contained Code Protection

Flicker [23] system built on the TPM-based Dynamic Root Of Trust (DROT) technology can create 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 TPMs (VTPMs) and migrate them into hypervisor space. Note that both schemes focus on the protection of a small piece of code and data. Both schemes only protect self-contained code with pre-defined inputs and outputs (e.g., inputs are the initial parameters and outputs are the final returns), and they do not support the protection with dynamic memory allocation and system calls. The increasing of the protection scope, such as protecting the whole application, may lead both schemes to failure. MiniBox [20] attempts to extend the functionality of the self-contained code by combining virtualization-based memory isolation and user-space sandbox (e.g., Google Native Client) techniques. But it still have several limitations to support a whole legacy application, such as lack of multi-thread support and limiting to sandbox-capable applications.

7.2 Whole Application Protection

Secure-Processor-Based Protection. AEGIS [29] 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 [29] also provide an alternative implementation, which requires to build security into the OS.

Bastion [2] and SecureME [9] 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 [25] 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.

Intel Software Guard Extensions (SGX) technology [10] is able to to protect an application by extending hardware processors. It introduces Enclave - isolated memory of code and data within an application's address space. It enforces that only code executing within the enclave can access data within the same enclave. Any accesses even they are from privileged software or SMM will be rejected. The exchanged data between processor cache and main memory are encrypted. Thus, bus sniffing attack does not work. Comparing with AppShield, Intel SGX could achieve stronger security (e.g., it is able to defend against the bus sniffing attack which would work in AppShield). However, its performance would be slower as it requires lots of encryptions and decryptions on data exchange between processor cache and main memory. Note that all memory accesses in AppShield setting is native speed, without any encryption or decryption.

Microkernel-Based Protection. EROS[26], Perseus[24], 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 [31] 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 [6], CHAOS [5] and SP^3 [34] 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 mapping reorder attack. Thus, the data and code integrity may still be broken by potentially compromised OS. InkTag [17] is an approach that 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. Virtual Ghost [12] provides application security by providing *ghost memory*. However, it requires compiler instrumentation on kernel code that is not always available for commodity platforms. In addition, it requires *complete* control-flow integrity checking at runtime, which is extremely hard to achieve in reality.

BIOS-Based Protection. Lockdown [33] system relies on a BIOS-assisted lightweight hypervisor and an 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 [30] 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. Essentially, both approaches needs to shut down one world to run another one, meaning that they can not simultaneously execute two worlds. However, our AppShield allows the coexistence, meaning the protected applications can simultaneously executed with the untrusted/unprotected applications in a system.

8. CONCLUSIONS

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. The design of AppShield has taken into consideration several newly identified threats where the kernel manipulates the address mapping. 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 that AppShield is lightweight and efficient.

9. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments that greatly helped improve the paper. This research was funded in the Singapore Management University through a research grant *C220/MSS13C005* from the Ministry of Education Academic Research Tier 1.

10. REFERENCES

[1] 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.

[2] 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.

[3] David Champagne and Ruby B. Lee. Scalable architectural support for trusted software. pages 1–12. IEEE Computer Society, 2010.

[4] 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.

[5] 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.

[6] 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.

[7] Yueqiang Cheng and Xuhua Ding. Guardian: Hypervisor as security foothold for personal computers. In *Trust and Trustworthy Computing*, pages 19–36. Springer Berlin Heidelberg, 2013.

[8] 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.

[9] 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.

[10] Intel Corporation. Innovative instructions and software model for isolated execution. http://privatecore.com/wp-content/uploads/2013/06/HASP-instruction-presentation-release.pdf.

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

[12] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. ASPLOS '14, pages 81–96. ACM, 2014.

[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] Yanlin Li, Adrian Perrig, Jonathan M McCune, James Newsome, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code (cmu-cylab-14-001). 2014.

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Trusted Computing Group. TPM main specification. Main Specification Version 1.2 rev. 85, February 2005.

[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] 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.

[35] 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.