# AppSec: A Safe Execution Environment for Security Sensitive Applicationst

Jianbao Ren
Xi'an Jiaotong University
renjianbao@stu.xjtu.edu.cn

Yong Qi
Xi'an Jiaotong University
qiy@mail.xjtu.edu.cn

Yuehua Dai
Xi'an Jiaotong University
xjtudso@stu.xjtu.edu.cn

Xiaoguang Wang
Xi'an Jiaotong University
mailwxg@foxmail.com

Yi Shi
Xi'an Jiaotong University
shiyi@mail.xjtu.edu.cn

## Abstract

Malicious OS kernel can easily access user's private data in main memory and pries human-machine interaction data, even one that employs privacy enforcement based on application level or OS level. This paper introduces AppSec, a hypervisor-based safe execution environment, to protect both the memory data and human-machine interaction data of security sensitive applications from the untrusted OS transparently.

AppSec provides several security mechanisms on an untrusted OS. AppSec introduces a safe loader to check the code integrity of application and dynamic shared objects. During runtime, AppSec protects application and dynamic shared objects from being modified and verifies kernel memory accesses according to application's intention. AppSec provides a devices isolation mechanism to prevent the human-machine interaction devices being accessed by compromised kernel. On top of that, AppSec further provides a privileged-based window system to protect application's X resources. The major advantages of AppSec are threefold. First, AppSec verifies and protects all dynamic shared objects during runtime. Second, AppSec mediates kernel memory access according to application's intention but does not encrypts all application's data roughly. Third, AppSec provides a trusted I/O path from end-user to application. A prototype of AppSec is implemented and shows that AppSec is efficient and practical.


*VEE '15*, March 14–15, 2015, Istanbul, Turkey..

http://dx.doi.org/10.1145/2731186.2731199

***Categories and Subject Descriptors*** D.4.6 [*Operating Systems*]: Security and Protection

***General Terms*** Design, Security, Performance

***Keywords*** Privacy, VMM, Kernel, Human-machine interaction

## 1. Introduction

Operating system controls all system resources and is the root of trust, so compromising the OS compromises everything on the system. Compromised OS can freely get our sensitive information through accessing main memory or intercepting human-machine interaction. If an application could remain safe even if OS were compromised, then OS exploits would no longer have the security threats as today.

Previous works on untrusted OS mainly focus on memory data and simply isolate trusted code and data from OS kernel [11, 27, 36, 37] or encrypts all data flowing to kernel roughly [15, 16, 31, 48]. Applications and OS either need to be re-designed or compiled statically. Besides, this arbitrary encryption may make applications unusable. For example, a web browser whose data is encrypted by the privacy protection mechanism can not communicate with web servers which does not run the corresponding decryptor. Although some applications can operate on encrypted data [10, 40, 46], cryptographic schemes for general-purpose computing [28, 29] have severe performance overhead and some data cannot be encrypted, like keyboard input and screen output.

Because human-machine interaction data is processed in plaintext, protections are still incapable without considering the security of human-machine interaction. Some dedicated work has been done to enforce the human-machine interaction [17, 23, 50]. However, they either need to modify the device drivers to eliminate any dependencies on the OS ker-

nel or hinder applications communicating with others. That is tedious and difficult to port on different systems.

This paper introduces AppSec, a hypervisor-based system in which **an untrusted operating system's behavior is verified according to application's intention and all human-machine interactions are enforced, without modifying OS kernel and applications**. To achieve these goals, we meet to solve the following problems on an untrusted OS:

1. How to infer application's intention and verify kernel's memory access transparently?
2. How to protect the security of all application linked dynamic shared objects (DSOes)?
3. How to protect human-machine interaction devices against a compromised OS?
4. How to enforce the security of application's X resource (e.g., clipboard, screen contents)?

AppSec provides several mechanisms to tackle these problems. AppSec intercepts every system call to infer application's intention. This includes the system call an application has invoked and the memory range that an application allows OS to access. During runtime, AppSec verifies whether a system call is invoked and whether the corresponding memory access is in accordance with application's intention. We know that all security sensitive applications pay much attention to their data security. So all application intended data transmission to a compromised kernel would not leak any privacy. To extract application's intention, AppSec intercepts security sensitive application's system calls. By analyzing the corresponding parameters, AppSec can validate OS kernel memory access at byte granularity. Verifying kernel memory access according to application's intention can protect user's privacy effectively and avoid encrypting all application data roughly.

Before verifying kernel memory access, we should identify which memory pages belong to a security sensitive application, but the dynamic memory allocation makes things a little more complicated. Previous work either instruments OS kernel or modifies applications to collect the page usage information explicitly [16, 31, 36]. While it may be bypassed when OS is compromised and it is not transparent to OS and applications. AppSec uses a skillful and non-bypassed page tracking based on hardware nested page table to deal with the dynamic memory allocation transparently. Its innovative features are that it does not instrument or modify OS kernel or applications and every physical pages once used by a protected application can be detected immediately.

DSOes are shared by all applications and their security is critical to user's privacy. Previous work compiles applications statically to avoid the security problem caused by DSOes [15, 16, 31, 48]. However, this loses the advantages of DSOes (e.g., updating during runtime). Besides, some close-source applications cannot be compiled statically. AppSec introduces a safe loader to check the code integrity of DSOes. During run-time, AppSec write-protects DSOes code and uses a skillful page tracking to verify the DSOes memory access.

For the security of human-machine interaction, AppSec uses an isolated dedicated customized OS to host the related input/output devices drivers and the X window server. All data transmissions between sensitive applications running on an untrusted OS and X server are encrypted. IOMMU and IOAPIC are used to protect the human-machine interaction devices from being attacked by the untrusted OS. On top of human-machine interaction devices protection, AppSec introduces a privilege-based window access control policy to prevent various attacks towards X window system, such as accessing chipboard and taking screenshot. All these protections are compatible with traditional applications and OS kernel.

In summary, we make the following contributions:

1. AppSec proposes a kernel memory access verifying mechanism based on applications' intention without redesigning or modifying applications and OS kernel.
2. AppSec introduces a safe loader to verify the code integrity of protected applications and DSOes which avoids compiling applications statically and keeps the advantages of DSOes.
3. AppSec proposes a security human-machine interaction channel and a privileged-based window system to protect all human-machine interaction data. Both mechanisms do not need to modify OS kernel or the corresponding drivers.
4. We have implemented a prototype of AppSec. All protections provided by AppSec are transparent to applications and OS kernel. An extra benefit of AppSec is that it can prevent ret2user [32] and ret2dir [33] attacks effectively. The experiments show that, with all protections, AppSec only incurs 6%$\sim$10% performance overhead.

In the next section, we give the threat model and some assumptions of AppSec. In Section 3, we give an overview of AppSec. In Section 4 and Section 5, we show how AppSec ensures the memory data security and human-machine interaction security respectively. The evaluation of AppSec is presented in Section 6.We discuss the limitation and our future work in Section 7. In Section 8, we compare AppSec with other work. In Section 9, we conclude our work.

## 2. Threat Model and Assumptions

In this section, we describe the threat model and our assumptions.

### 2.1 Threat Model

We consider an attacker who exploits the OS kernel vulnerabilities and has the full control of computer system but

the CPU, the memory controller, system memory chips and the corresponding system bus. Example attacks are: accessing any memory pages, injecting code into OS kernel and the sensitive application, malicious DMA accessing, surreptitiously obtaining sensitive user-input data by recording key strokes, screenshot of sensitive application display etc.

### 2.2 Assumptions

We assume that all hardware is trusted, no bus traffic interception, no Trojan-Horse circuits or malicious microcode. We also assume the system firmware (e.g., BIOS) is trusted.

The protection object of AppSec is security sensitive applications. We assume they are conscious of flowing out data and encrypt it timely, for example, encrypting the file contents before writing them to disk and encrypting the network connection when communicating with others. AppSec does not prevent a private data leakage because of the vulnerabilities of an application itself. Denial-of-service attacks caused by a compromised kernel are also out of our scope. We also do not consider any side-channel attacks like timing, cache-collision.

AppSec is based on our previous work, a lightweight hypervisor whose interface has been verified [20, 21], we assume the hypervisor is trusted. And we assume that the CPU supports hardware virtualization, such as AMD Secure Virtual Machine (SVM) or Intel Trusted eXecution Technology (TXT) [18, 47].

## 3. System Overview

In this section, we start off by stating the desired properties to enforce the security of private data. Then we describe the challenges involved and present an overview of AppSec at last.

### 3.1 Desired Properties

In order to provide a safe and practical execution environment, AppSec seeks to meet the following properties.

**Memory Access Verification.** This is the cornerstone of a safe execution environment. AppSec must ensure that every memory access from kernel to a protected application should be verified according to application's intention.

**Code and Control Flow Integrity.** AppSec needs to guarantee the code integrity of sensitives application and all DSOes they linked during the whole life-cycle. During runtime, application may be interrupted at any time by OS kernel. AppSec must ensure the execution of an interrupted application begins at the interrupted site to prevent compromised kernel injecting malicious code into the protected application.

**Trusted Human-Machine Interaction Path.** Because all human-machine interaction data is plaintext, AppSec must protect it from being stolen or peeked by compromised OS and other applications.

**Device Operation Verification.** This consists of two different aspects. First, AppSec must prevent compromised OS accessing the human-machine interaction devices or intercepting their events. Second, AppSec must prevent the malicious device DMA accessing to sensitive application memory.

**Compatibility and Transparency.** Although these are not security properties, but are very important in practice. AppSec should not modify OS kernel and sensitive applications.

**Low Performance Overhead.** The safe execution environment provided by AppSec should incur acceptable overhead. And the performance overhead incurred to the other normal applications should be as little as possible.

### 3.2 Challenges

We now discuss the challenges we are facing in designing such a safe execution environment. The very first problem is the ***semantic gap between guest OS and AppSec***. Because AppSec does not instrument the OS kernel or applications, it cannot get the exact information about the interactions between OS kernel and applications. This makes AppSec difficult to judge whether a memory access from kernel to application is application intended. The dynamic memory allocation makes things more complicated. It is difficult for AppSec to know which pages belong to a protected application and whether one page of a protected application is freed.

The next issue is raised by the ***dynamic shared objects***. A tampered DSO may steal user's private data by changing application's original semantics. For example, the *strcpy* function can be modified to pass private data to a compromised kernel. So AppSec must ensure the code integrity of DSOes. Unfortunately, the random loaded location of DSOes makes this verification difficult. Besides, DSOes are shared by all applications. AppSec must distinguish whether the DSO code execution is in the context of a protected application or not. This is a premise for AppSec to judge if a memory access is from sensitive application itself.

Another problem we are facing is the ***human-machine interaction devices security***. All devices drivers are hosted by OS kernel. Thus, once OS kernel is compromised, the adversary can capture all data passing through the human-machine interaction devices. What's more, the interrupt vector's mis-configuration can also cause a benign drivers' misbehavior and leak user's privacy.

The last issue is the ***innate problem of X system***. The design philosophy of X system makes the trusted-path failure even if OS kernel and device drivers are trusted. For example, a malicious application can access the private data stored by a sensitive application in clipboard without any restriction. Applications can also use the corresponding functions (e.g., XGetImage) to get the screen contents. This exposes a sensitive application to risk.

### 3.3 AppSec Architecture Overview

Figure 1 illustrates the architecture of AppSec. The light weight VMM [20, 21] runs on bare-machine and provides a strong isolation to run two different OSes simultaneously.

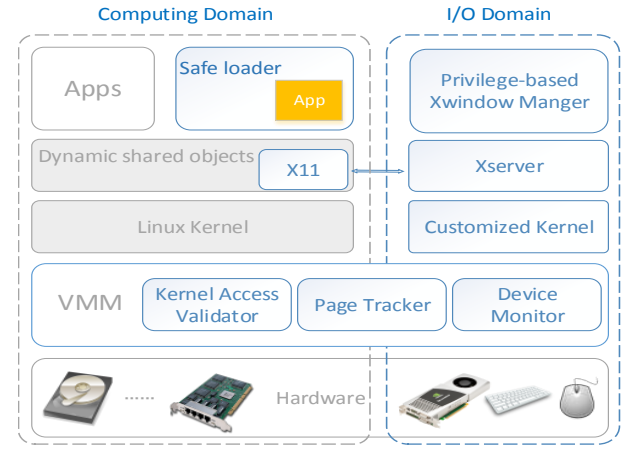


**Figure 1.** The architecture of AppSec.

One OS provides a normal computing environment and is called *computing domain*. The other one, running a customized kernel and only providing the X system service, is called *I/O domain*. I/O domain controls all human-machine interaction devices. Because I/O domain just serves the human-machine interaction requests and can only communicate with outside through X server's interface, we assume it is hard to be compromised.

A security sensitive application is loaded by the safe loader as depicted in Figure 2. The safe loader checks the integrity of DSOes and applications according to the hash values which are calculated in advance. Safe loader can distinguish which applications and DSOes are needed to check and invokes hypervisor to get the correct hash values transparently. We expect some third-parties could manage these hash values and provide them through Internet in the future. And now, we just store these hash values in hypervisor.

The hardware nested paging technique is leveraged by AppSec to construct a strong isolation between sensitive applications and OS kernel transparently. Sensitive applications can store their encryption keys in user space as normal. The un-bypassed and transparent page tracker collects application memory pages information in real-time. Everytime when kernel access sensitive applications' memory, a nested page table (NPT) fault is raised. In the NPT page fault handler, AppSec verifies kernel memory access according to application's intention. Access which is not in accordance with application's intention is denied.

AppSec encrypts all X connections to prevent computing domain kernel picking up the communication between sensitive applications and X server. Moreover, we retrofit the X system to support a privilege-based window management. The protected application is in the highest privilege level. AppSec ensures that the low privilege group windows cannot access the resource of high privilege group windows. This can effectively solve the aforementioned innate problem of X system.

# 4. Memory Data Security Enforcement

In this section, we discuss how AppSec tackles the aforementioned challenges to guarantee the memory data security of a security sensitive application. We start off by stating how AppSec constructs a trusted initial environment. And then, we show how AppSec protect application's private data during runtime. This includes tracking application's page in real-time, protecting application context switch and verifying kernel memory access.

## 4.1 Environment Initialization

### 4.1.1 Safe Loader

The very first problem to protect a security sensitive application is how to integrate it with AppSec transparently and ensure the code integrity of itself and all DSOes it linked. The easiest way is modifying the application to invoke hypervisor explicitly. However it conflicts with our transparency requirement. As we know, OS alway invokes the ELF (Extensible Linking Format) loader as a interpreter before transferring control flow to ELF applications. AppSec introduces a safe loader to replace the Linux traditional standard ELF loader.

The safe loader is aware of which application will run and invokes AppSec to verify its code integrity. It uses the hypercall to communicate with the core components of AppSec which run in the VMM. All these communication is imperceptible to OS kernel. The verification result is presented to user via a trusted dialog box. Such a dialog box, which executes under the control and with the privileges of hypervisor (not the OS), is often called a "powerbox" [41]. Except the verification results, the trusted dialog box also contains a user personal reserved information like a bank counterfoil. This can avoid OS faking the dialog box and bypassing the safe loader to run security applications.

The random loaded address of DSOes confuses AppSec with their code integrity checking. Previous works require the application to be compiled statically to eliminate this confusion. These conflicts with our transparency requirement and loses the advantage of DSOes. Because ELF loader is responsible for loading all DSOes into application's address space, the safe loader knows the memory mapping information of each linked DSO exactly. When the safe loader loads DSOes into application's address space, it records their virtual address and the corresponding inner offset.

As the demand paging mechanism used by OS kernel, before invoking AppSec to verify the code integrity, the safe loader touches all virtual pages to force OS loading the corresponding physical pages into memory. Combining with the address space isolation mechanism, AppSec can ensure the application and all DSOes it linked cannot be modified after they are verified.

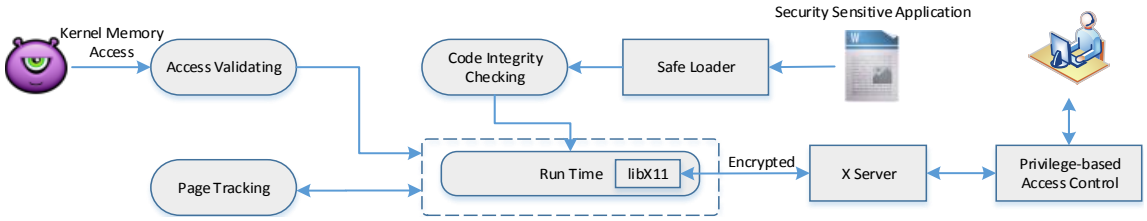


**Figure 2.** Overview of AppSec to protect a sensitive application for while life-cycle.

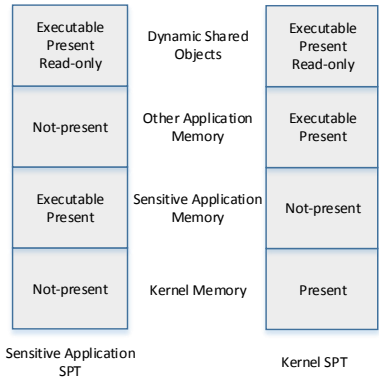


**Figure 3.** The SPT difference between the sensitive application and OS kernel.

#### 4.1.2 Address Space Isolation

After the code integrity verification, AppSec isolates sensitive applications from OS kernel and other applications. The *nested paging* hardware virtualization is used to construct such isolation environment. Nested paging technology provides two levels of address translation (i.e., translation from guest liner address to guest physical address and translation from guest physical address to machine address). The page table which translates guest address to machine address is called *SPT (Shadow Page Table)*. The corresponding implementation of AMD and Intel processors are called *NPT* and *EPT* respectively.

AppSec provides different SPTs for sensitive applications and OS kernel. As Figure 3 shows, the SPT of sensitive applications contains all page mappings of their user space. Oppositely, all pages of sensitive applications are masked not-present in the SPT of OS kernel. Other applications share the same SPT with OS. As DSOes are shared by all applications, we mask all their pages present and read-only in every SPT. Because SPTs are maintained by hypervisor, OS kernel is unaware of the SPT page mapping and it can still manage the guest physical address mapping with its own page table. Everytime when OS access the protected application, a SPT fault will be triggered. In the SPT fault handler, AppSec can verify the kernel memory access transparently according to application's intention and the details are stated in Section 4.2.3. Because the SPT is transparent to OS, the memory access verification cannot be bypassed no matter what the attacker does to the OS kernel.

Currently, AppSec uses group-based policy to manage applications. All applications in the same group share one SPT and can communicate with each other as normal. Child processes belong to the same group with their parent in default. So child processes and parent process can communicate with each other based on shared memory freely. For unshared memory communications like pipe, socket, AppSec assumes that applications encrypt private data by themselves or use an encrypted connection and does not intervene these communications.

Only isolating applications from OS kernel is inadequate to protect their private data. Compromised OS may map a page containing malicious codes into protected application's user space or replace a legal virtual page mapping with a malicious physical page and then transfers control flow to this page when a system call returns. As the malicious code resides in the same address space with the sensitive application, they can access the sensitive application memory data straightly. What's more, compromised OS kernel can also fork a fake child process to get it sharing the same SPT with a protected application. OS kernel can map the protected application's physical pages into the fake child process address space and access the protected application's memory indirectly through the fake child process. We will show how to prevent these attacks in Section 4.2.2.

### 4.2 Runtime Protection

During runtime, a sensitive application interacts with OS kernel frequently. This includes memory allocation, data transmission, context switch and so on. AppSec has to intercept these interactions and verifies the validity of these interactions according to application's intention.

#### 4.2.1 Page Tracking

Before verifying OS memory access, we must know which memory pages belong to a protected application first. The very first problem to get the memory page usage information during runtime is caused by the dynamic memory allocation. Because of the semantic gap, it is difficult for AppSec to know if one page is allocated to an application or if one page is freed by an application. Instrumenting all memory allocation interfaces is the most obvious way to notify AppSec about the dynamic memory usage information. However, this is not transparent to OS kernel and may be bypassed

if OS kernel is compromised. An optimum solution is implementing the page tracking into hypervisor.

AppSec uses a lazy memory page usage tracking mechanism to deal with the dynamic memory allocation. When a page is allocated to sensitive application by OS, AppSec does not know this allocation and the physical page is still not-present in the application's SPT. So, when the application touches this page for the first time, an SPT fault would be raised. In the SPT fault handler, AppSec knows a memory page allocation has occurred passively and updates the corresponding SPT (i.e., mask the page present in protected application's SPT and not-present in OS kernel's SPT). In order to make sure that all CPU cores have a coincident page mapping, AppSec must flush other CPU's TLB cache.

DSOes make things a little more complicated. As aforementioned, all applications share the same physical pages of DSOes. So when an SPT fault occurs in DSOes, we must identify if it is caused by a sensitive application. It is straightforward to use the different SPTs to identify the context. When an SPT fault occurs during the usage of sensitive application's SPT, we consider this SPT fault is caused by a sensitive application, otherwise it is caused by OS or other applications.

Unfortunately, it is likely that a sensitive application may execute with the OS kernel SPT sometimes. For example, when a sensitive application executes in DSOes code, OS may schedule other applications to run. AppSec switches the SPT and uses OS kernel SPT to run other applications. As all DSOes are masked present in OS kernel SPT, when the sensitive application is scheduled back, it will run with kernel SPT. To solve this problem, we compel the control flow to be transferred into a fixed address every time when the context switches from OS to protected applications. This fixed address could cause an SPT fault and in the SPT fault handler AppSec switches the SPT to run the protected application. The detail is presented in Section 4.2.2.

The global variables of DSOes would cause the whole system unusable if we use the aforementioned page tracking directly. For example, when sensitive application read its DSOes global variable for the first time and thus causes an SPT fault in the context of sensitive applications, AppSec masks the corresponding page not-present in the kernel SPT. This prevents other applications access the same DSOes variable.

We refine the aforementioned page tracking to tackle this problem. Figure 4 shows the page states transition graph. When the first access to a virtual page ("S") is a read operation, AppSec masks the corresponding page read-only and present on the SPTs of both kernel and sensitive application ("SR"). This is safe for the security sensitive application, because a memory page does not leak any privacy until it is written. While the first access to either "S" or "SR" state is a write operation, AppSec masks the corresponding page writable in the sensitive application SPT and not-present

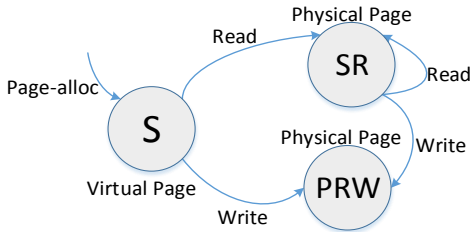


**Figure 4.** The state transition diagram of SPT page. "SR" means shared and readable, "P" means private and "W" means writable.

in the kernel SPT. Because of the COW (Copy On Write), OS will allocate a exclusive page to an application when a shared page is written. So masking this exclusive page not presented in kernel SPT would not cause whole system unusable.

Unlike the memory allocation, page free operation does not cause SPT fault. AppSec learns the memory page free operations by intercepting the corresponding system calls (i.e., *munmap* and brk). AppSec uses these system calls' parameters to infer which pages are freed and cleans their contents before masking them present in the OS kernel SPT.

### 4.2.2 Context Switch

AppSec need to intercept the context switch precisely to chose different SPTs and to protect the control flow integrity. Because security sensitive applications and OS kernel use different SPTs, when the context switch from user space to kernel space, a SPT fault will be raised. In the SPT fault handler, AppSec records the context switch site (i.e., system call return address and interrupt location) and changes the context switch back site to a fixed address directly. Besides, AppSec also records all system and their parameters to serve the later memory access verification. The fixed address can trigger a SPT fault proactively. So every context switch from kernel space to user space can be also intercepted.

When context switch from kernel space to user space, AppSec switches the SPT and drops the CPU privilege to level3. And then, AppSec pushes the original return address into application's stack and uses a ret instruction to take the control flow to application at the context switch site. AppSec ensures that every context switch from kernel space to user space can only transfer control flow to the aforementioned fixed address. OS can modify its stack to return at any address definitely. However, returning to other address makes CPU running with kernel NPT. When applications memory is touched (access, execute), AppSec can detect this malicious action in NPT fault handler. In order to tame the signal mechanism, AppSec records every signal handler address in a table by analyzing the corresponding system calls. Every time when OS does not transfer control flow to aforementioned fiexed address, AppSec check if it transfers to a signal handler. If not, AppSec provides a warning information through a trusted dialog box to user.

As we mentioned in Section 4.2.1, malicious OS kernel may inject malicious codes into protected applications and transfer control flow to the malicious codes when a system call returns. AppSec ensures control flow can only be transferred from OS kernel to user space at some known sites (i.e., ether the fixed address or a signal handler). So malicious OS kernel has no chance to execute the injected malicious codes. For the virtual page remapping attacks, an NPT fault will be raised when CPU execute in the faked physical page. However, an NPT fault should not occur at text segment because the safe loader has already loaded all text segments into memory. So AppSec can detect this attack in NPT handler. To prevent the fake child process attack, AppSec verifies whether a corresponding "fork" system call has been invoked by sensitive applications. Because all child process share the same code segment with their parent, even if the compromised OS kernel create a faked child process during the sensitive application invokes a "fork" system call, AppSec still ensures that OS cannot inject malicious code in this fake child process.

In implementation, changing a context switch site to a fixed address may crash sensitive applications. That is because OS checks the switch site address in some cases. Fortunately, these cases are rare and we can deal with them respectively. For example, the " *time*" function uses "*vsyscall*" to get the time. OS checks if the context switch site locates between *0xffffffffff600000* and *0xffffffffff601000* which is the *vsyscall* mapping address range. If not, OS terminates the application. In this example, OS kernel returns back to the application according to a value stored in the user space stack instead of the context switch site. So AppSec modifies this value instead of the context switch site to pass the OS checking.

#### 4.2.3 Memory Access Verification

During run-time, in order to get system service, a sensitive application must allow OS to access its memory. It is necessary for AppSec to distinguish the legitimate operations from the illegal operations according to application's intention. We use a straightforward mechanism to solve this problem. AppSec records every system call parameters when the system call is invoked. During runtime, when OS kernel accesses protected application memory, a SPT fault is raised. In the SPT fault handler, AppSec verifies if the kernel memory access is as security sensitive application wanted according to the system call parameters. If the memory access range is not the application wanted, the memory access will be denied or only ciphertext will be copied into kernel buffer depending on user's configuration. If the memory access range is as the application specified, AppSec copies the protected application memory content to OS kernel's buffer. So OS cannot get applications private data even if the private data locates in the same page with system call buffer.

AppSec uses IOMMU to prevent malicious OS kernel issuing illegal DMA operations to get a protected application memory data. IOMMU uses another page table to translate all DMA operation addresses. We use the same mechanism as memory isolation to isolate security sensitive application's memory data from malicious DMA operations. All devices use the kernel SPT as their I/O page table in default. When a protected application needs to do DMA operation, AppSec constructs a temporary I/O page table to translate DMA address dynamically.

#### 4.2.4 ret2user and ret2dir Attacks Defense

As the aforementioned mechanism used by AppSec, we have an extra benefit. That is AppSec can prevent the return-to-user (ret2user) [32] and ret2dir (return-to-direct-mapped memory) [33] attacks effectively.

ret2user attacks exploit the OS vulnerabilities to redirect corrupted kernel pointers to malicious code residing in user space and then run the malicious codes with kernel privilege. AppSec intercepts every context switch from kernel space to user space and drop the CPU privilege to level 3 compulsorily. So even if attackers transfer control flow to malicious codes, it still cannot be executed with kernel privilege.

ret2dir attacks are a little bit more complicated than ret2user attacks. They map the malicious codes which reside in user space memory page into OS kernel virtual address space by leveraging a kernel region that directly maps part or all of a system's physical memory. It can bypass all existing ret2user defenses, like SMEP, SMAP and kGuard [8, 30, 32]. However, with AppSec which uses different SPTs for applications and OS kernel, application's physical pages are masked not present in kernel's SPT. Codes with kernel privilege still cannot access data residing in user space even if the corresponding physical pages are mapped into kernel's virtual address space.

## 5. Trusted Human-Machine Interaction Path

Previous systems mainly focus on the security of memory data, and the security of I/O path is usually neglected. However, a practical computing environment facing end users involves a lot of human-machine interactions which are easily intercepted by a compromised kernel. It is necessary for AppSec to build such a trusted I/O path from end-user to the sensitive application against a compromised OS kernel.

It it challenging to protect the human-machine interaction path. The popular operating system integrates all device drivers into kernel space, like Linux and Windows. If OS kernel is compromised, all these device drivers are infected. To isolate the trusted path device drivers from OS, one method is to modify the commodity device driver to eliminate any dependencies on the OS like [50]. However, the close-couple design architecture makes it difficult to decouple device drivers from OS.

AppSec is inspired by exokernel [24], VirtuOS [38], Xen frontend backend driver model [26] and X system [7]. As depicted in Figure 1, AppSec uses a dedicated OS, *I/O do-*

*main*, to host all human-machine interaction device drivers. The computing domain can communicate with the I/O domain only through the X system interfaces. So, even if the OS kernel of computing domain is compromised, the I/O domain and user's I/O operation are still safe.

In this section, we first state how to isolate all human-machine interaction devices from the computing domain. On the top of human-machine interaction devices isolation, we show how to retrofit the traditional X system to enforce the communication between X client and X server and protect application's X resource.

### 5.1 Human-Machine Interaction Devices Isolation

The devices isolation comprise two aspects: device access isolation and device interrupt isolation.

#### 5.1.1 Device Access Isolation

AppSec isolates human-machine interaction devices from the computing domain by intercepting the PCI/PCIe configuration space access. During booting, OS kernel traverses the PCI/PCIe configuration space to enumerate all presented devices. For the X86 architecture, the PCI/PCIe configuration space is accessed via two special I/O ports (i.e., *0xCF8* and *0xCFC* in default) or MMIO regions [5, 14]. AppSec intercepts the configuration space access and masks all human-machine interaction devices not-present when the computing domain booting. Similarly, AppSec masks other devices excluding the trusted path devices not-present when the I/O domain booting. For some devices whose absence may lead the OS to stall, AppSec creates these necessary virtual devices and provides the corresponding fake PCI/PCIe configuration information. All operations to these virtual devices will be discarded and AppSec just returns operation success to upper OS.

During runtime, compromised OS can capture all data passing through the human-machine interaction devices by configuring other devices to overlap these devices' I/O port space or physical memory space. In order to defend against these overlap attacks, AppSec verifies every device configuration operation. If computing domain tries to overlap I/O domain devices configuration range, AppSec denies the device configuration operation and generates an interrupt to the I/O domain. The I/O domain will show a corresponding attack information on the screen immediately.

#### 5.1.2 Interrupt Isolation

All device accesses are interrupt-driven. Without interrupt isolation, computing domain can send out any spoofed interrupt to I/O domain which may cause I/O domain wrong reaction. To make matters worse, the computing domain can sniff the human-machine interaction interrupt events to infer user's privacy. For example, by analyzing the time series of keyboard interrupt with pattern recognition, attacker can get what an user inputs easily.

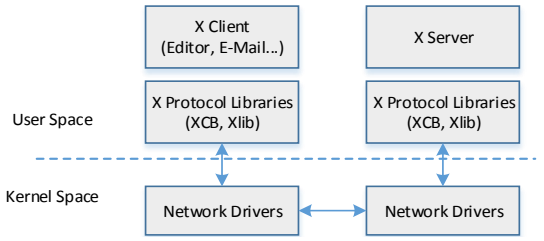


**Figure 5.** Data transmission path between X client and X server.

AppSec uses three different methods to isolate interrupts between computing domain and I/O domain. To prevent the computing domain sending spoofed interrupt to the I/O domain, AppSec intercepts all inter processor interrupts (IPI) and makes sure that all the IPI destination are confined in their own domain. To prevent the computing domain sniffing devices interrupt events and guarantee all human-machine interaction devices interrupts are delivered to the I/O domain directly, AppSec uses the other two methods for different interrupt source. For traditional interrupts which are signaled by asserting an interrupt line (pin), AppSec uses IOAPIC to route them to the I/O domain. For message signaled interrupts (MSI) which are signaled by writing a particular address, AppSec leverages the interrupt remapping features of IOMMU to route them to I/O domain.

### 5.2 X System Retrofit

X system is the native display technology on UNIX and Linux systems. It uses a client-server model to provide the basic framework for a GUI environment. X server renders screen contents on the display devices. It forwards the user inputs from input devices like keyboard, mouse and touchscreen to applications which are called X clients. The innate design of X system makes it inadequate for a safe computing environment. AppSec retrofits X system in two aspects to protect user's privacy.

#### 5.2.1 Encrypted Channels

X client communicates with X server through a network connection. As shown in Figure 5, data is encapsulated by X protocol libraries and transmitted by OS. Most of the data transmission is not encrypted [3]. When the OS kernel is compromised, attacker can get all these transmitted data by sniffing the network connection.

AppSec modifies the X protocol libraries and adds encryption functions to build an secure channel between X client and X server. When an X client tries to get connection to X server, AppSec retrofits the original connection functions to make sure every connection is encrypted. Because AppSec guarantees the code integrity of DSOes, this encryption is un-bypassed and all data transmitted is safe even though the OS kernel is compromised.

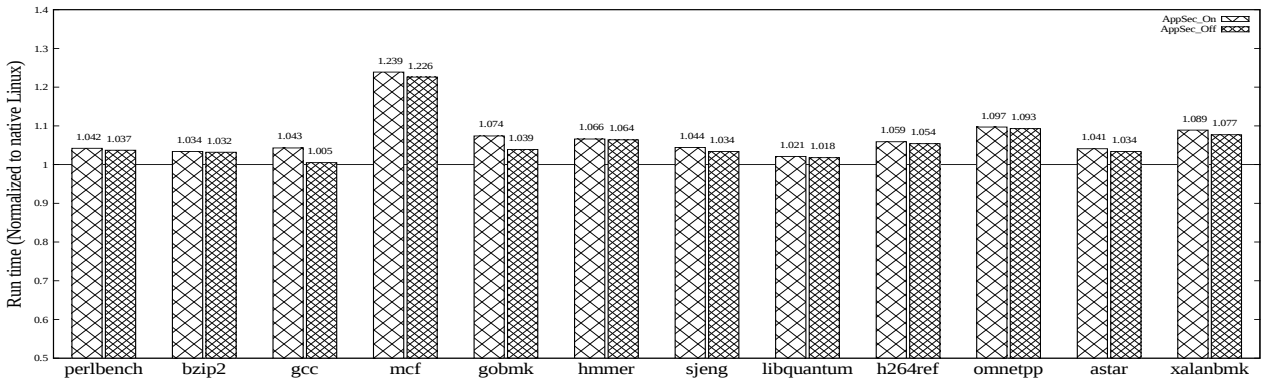


**Figure 6.** SPEC CPU2006 performance comparison between AppSec ON and OFF, which is relative to native Linux.

#### 5.2.2 Privilege-based Window Access Control

One design philosophy of X system is that all applications are good and non-malicious. This leads to the lack of window-level isolation and allows any client to have full access other X client resource like clipboard and screen display. This is disastrous for a security sensitive computing environment.

AppSec introduces privilege-base window resource access control policy in X system. All windows are divided into different privilege groups, and low group cannot access high group's X resource. This is similar to the CPU ring. An application can be added into the high privilege group only if it is designated by user explicitly. We use two practical cases to show how this privilege-based access control work and what modification should be done to the X system.

***Case 1. Clipboard:*** User uses a document editor to edit a top-secret document. At the same time, he/she may want to search on the Internet with a browser to enrich this document. Traditionally, the browser can access the document editor's clipboard by sending *XConvertSelection* request to the X server. X server forwards this request to the editor as a *SelectionRequest* request without any verification. After receiving *SelectionRequest*, the editor uses *XChangeProperty* to pass its clipboard contents, which may contain private data, to the web browser. With AppSec, user can add the text editor into the high privilege group manually and leaves the web browser in the low privilege group. In the *ProcChangeProperty* function of X server which serves client's *XChangeProperty* request, we check if the browser privilege is lower than the editor. If that, AppSec encrypts all clipboard contents and transmits the ciphertext to the browser.

***Case 2. Screenshot:*** Similar to case 1, a text editor is used to edit a top-secret document. Attacker may get the screenshot by invoking the *XGetImage* function stealthily. In current implementation, AppSec modifies the *ProcGetImage* function, which serves the client getImage request, to check whether there are higher privileged windows than the invoker. If there are, AppSec draws a full-screen rectangle to overlap these high privileged windows before X server getting the display contents. At last, AppSec sends a *GraphicsExposure* event to all windows to recover their display contents.

**Table 1.** NPT performance for MCF (in seconds).

| NPT Enabled | NPT Disabled |
|---|---|
| 681.693 | 570.615 |

## 6. Evaluation

In this section, we evaluate the performance overhead imposed by AppSec. We ran our evaluations on a Sugon A620r-G server with two AMD Opteron(tm) 6320 processors at 2.8 GHz, 16GB of DRAM, two integrated PCIe Gigabit Ethernet cards. The Debian wheezy were installed with the Linux version 3.10 in our experiments. We used the performance of native Linux as a base line in all experiments. *AppSec On* denotes we ran the test in a safe environment provided by AppSec and *AppSec Off* denotes we ran the test on the AppSec platform but not in the safe environment.

We evaluated AppSec performance overhead on application benchmarks as well as a few microbenchmarks. We used SPEC CPU2006, Apache Benchmark and Google V8 benchmark to obtain the effect of AppSec to application in practice. When AppSec provided a safe environment for a sensitive application, all processors NPT feature were enabled. In order to get the performance influence to other concurrent applications, we enabled all processors' NPT in all application benchmarks, no matter if AppSec was on or off. Microbenchmarks were used to see how AppSec affects primitive OS operations and we just disabled all processors' NTP when AppSec was off.

### 6.1 Application Benchmarks

Figure 6 shows the performance overhead incurred to SPEC CPU2006 benchmark suit. All results are normalized to native Linux. AppSec incurs at most 10% performance overhead to all tests excluding the mcf test. This is because these tests are CPU-bound and there are little OS interactions. The worst result is the mcf test. That is because there are a lot of

**Table 2.** Apache web server performance (requests per second).

| Concurrent Transactions | Native Linux | AppSec On | AppSec Off |
|---|---|---|---|
| 5 | 13655.58 | 12808.75 | 13600.53 |
| 7 | 14170.44 | 13592.32 | 14000.78 |

TLB misses during its execution. In the native Linux kernel, it only needs 4 times of memory access to handle one TLB miss at most. While when NPT is enabled, every TLB miss in the guest OS would incur 4*4 times of memory access to finish the TLB mapping at most. We executed mcf test with NPT enabled and disabled respectively. The result shown in table 1 confirms our analysis. The overhead is mainly caused by the hardware virtualization technology and could be reduced effectively if the hardware virtualization technology is improved.

Table 2 shows the performance overhead incurred to Apache web server. We used Apache Benchmark (ab) to issue 50,000 transactions with the specified number of concurrent transactions to the server. In each transaction, a 45-byte index page was transferred from the server to the Apache Benchmark client. Compared to native Linux, AppSec only incurs about 6% performance overhead. This is because we use extra pages to cache the frequent access memory content to reduce the number of interaction between AppSec and OS kernel. AppSec copies application buffer contents of the most frequently accessed page to a new page. The new page is mapped read-only in both NPTs of kernel and applications. Once the new is modified, the cache is invalid.

**Table 3.** Firefox web browser performance.

| Native Linux | AppSec OFF | AppSec ON |
|---|---|---|
| 3384 | 3306 | 3183 |
| - | 97.7% | 94.1% |

In order to obtain the effect of AppSec to user during the actual use, we chosen the Google V8 benchmark [2] running on mozilla firefox to evaluate the AppSec overhead. Table 3 shows the results. Higher score means higher performance. There is no more than 3% overhead AppSec incurred to the firefox when turning AppSec off. When turning AppSec on, the performance overhead is no more than 6%.

### 6.2 Microbenchmarks

We conducted some more micro experiments to obtain how much the overhead of AppSec interacting with OS kernel and user space contributes to the overall overhead. Table 4 shows the results of our experiments. We used *null system call* to evaluate the overhead of space switching between an application and OS. *mmap* and *page fault* were used to measure the performance of the proposed page tracking technology. *fork* expressed the overhead of creating a sensitive application process. File operations shown the overhead of AppSec incurring to some normal OS interaction operations.

**Table 4.** Latency microbenchmark results (in microsecond).

| | Native | AppSec On | AppSec Off |
|---|---|---|---|
| null syscall | 0.023 | 0.14 | 0.031 |
| open/close | 0.294 | 1.75 | 0.307 |
| mmap | 3841.8 | 42629.3 | 3862.4 |
| page fault | 2.21 | 8.85 | 2.33 |
| file create | 11.5 | 29.7 | 11.6 |
| file delete | 12.6 | 31.2 | 12.7 |
| fork | 65.22 | 3685.12 | 68.3 |

When we disable AppSec, there is no interaction between VMM and guest OS. So, compared to native Linux, AppSec incurs almost no performance overhead. However, when it is enabled, every space switch and OS interaction is intercepted and it incurs a little high performance overhead.

Compared with other tests, *mmap* and *fork* incur a little high performance overhead. That is because, when application mapping a file, AppSec marks the corresponding virtual address range and updates the NPT table when the application access this address. Similarly, when an application process being created, AppSec traverses its whole virtual address space, finds all physical pages it used to ensure no malicious pages are injected. However, the *fork* operation is unfrequent in the desktop environment and the user is oblivious to this performance overhead.

## 7. Discussion

### 7.1 Attestation Chain

In order to build an attestation chain to run a security sensitive application, AppSec leverages the trusted platform modules (TPM) [6]. The TMP chip ensures that the power-on boot process starts from a trusted condition and *OSV* VMM is not modified. *OSV* uses hardware virtualization technology to isolate its memory pages from guest OS which guarantees its integrity during runtime. Every time when a protected application is launched, AppSec checks the integrity of safe loader to provide a trust environment automatically. AppSec outputs all the integrity measurement results on screen directly or sends them to other computer for remote attestation. Any integrity check fault would be recorded and then AppSec terminates the guest OS.

### 7.2 Limitation and Future Work

AppSec ensures that OS kernel can only access protected application data according to application's intention. However, the correctness of OS cannot be verified which maybe used by attacker to exploit the Iago attacks. Although we can check some system call results during context switch, it is unlikely to be tractable for arbitrary applications given the complexity of OS interfaces (e.g., Linux includes more than 300 system calls and Windows well over 1000). We will leverage Drawbridge [12] to mitigate this problem. What's more, the security of human-machine interaction relies heav-

ily on the security of X server. AppSec can leverage some control flow integrity tools [9, 49] to enforce the security of X server.

Now, every system call causes a lot of context or world switches (e.g, from guest user to guest kernel, from guest to host). This incurs a high performance overhead. In the future, we would use the FlexSC architecture [43] to reduce the performance overhead. Besides, we also plan to port our prototype to Intel platform with the help of Intel's Trust Execution Technology (TXT).

## 8. Related Work

In this section, we compare AppSec with the existing techniques of protecting user privacy in two different aspects.

### 8.1 Memory Data Protection

XOM [35], AEGIS [45] and Cryptopage [22] are proposed to protect sensitive data from being leaked or tampered. They do not trust the physical resources, like main memory and encrypt chips. In contract, we mainly focus on the software level attacks. Besides, these protections need to modify the CPU architecture, which makes them difficult to deploy.

PrivExe [39] provides an operating system service for private execution. Virtual Ghost [19] leverages LLVM compiler [34] to instrument OS and checks OS code in run-time. They all need to modify or re-compile OS kernel. In contract, AppSec protects full application from hostile OS transparently and excludes OS kernel from our TCB.

TrustVisor [36], Intel SGX instruction set [37], SICE [11] and Fides [44] partition an application into "*secret*" part and "*public*" part. They ensure that the secret part can only be accessed by the code in public part and the public part can only be invoked through the specified APIs. One significant drawback of these work is that the protected applications need to be modified or their architectures need to be redesigned. That need enormous efforts and is not available for some legacy commodity software. In contract, AppSec provides a whole application protection. AppSec does not need to re-design legacy software or re-compile them which is very important in practice.

Similar to AppSec, Overshadow [16], SP$^3$ [48], InkTag [31] and Chaos [15] use a virtual machine monitor to isolate the security sensitive application from OS. The main idea of these work to protect the memory data is using the hardware virtualization to intercept kernel memory access and encrypt all data flowing to OS kernel. As we stated in Section 1, this arbitrary encryption would make applications unusable. Besides, some of them needs to instrument OS kernel or leverage the existing hypervisor paravirtualization interfaces [15, 31]. What's more, all of them compile applications statically to avoid the complexity of DSOes. In contract, AppSec introduces a safe loader to protect DSOes which avoids compiling applications statically. Kernel memory access is verified according to application's intention which avoids the un-usability caused by encrypting all data arbitrarily. AppSec provides a secure human-machine interaction channel to enforce the security of input/output data. Besides, all of these work is based on sophisticated VMMs like VMware, Xen which have a very large performance overhead and are prone to be attacked [1, 4]. AppSec uses a lightweight hypervisor whose interfaces have been verified.

Haven takes a further step by using Intel SGX instruction to protect a whole application without any modification to protected application binary [13]. AppSec can leverage its LibOS mechanism to prevent against malicious behaviours like Iago attacks. However all human-machine interaction data is still unsafe and we have not found how Haven deal with the problems caused by DSOes. Another major advantage of AppSec is that AppSec introduces a trusted human-machine interaction channel to enforce the security of human-machine interactions. This is absence in previous related work.

### 8.2 Trusted Human-Machine Interaction

Zhou [50] and Dunn [23] use hypervisor to enforce human-machine interaction. However, they either need to modify device drivers to eliminate any dependencies on the commodity OS or hinder applications communicating with others.

DriverGuard [17] provides a trusted I/O flow between commodity peripheral devices and some privileged code blocks in device driver. UTP [25] uses the TPM chip to build a uni-directional trusted path. It ensures that user's transactions is indeed submitted by a human operating the computer. However, they all rely on the security of OS and the innate problem of X window system still expose X resource to risk.

EROS [42] Window System is a trusted window system for the EROS capability-based operating system. It is built on the primitive mechanisms of EROS operating system and is capable of enforcing mandatory access control. However, a compromised OS can still access all human-machine interaction devices to steal user privacy and the special operating system requirement makes it difficult to use for most traditional security sensitive application.

Compared to these work, AppSec first isolates all human-machine interaction devices from compromised OS kernel. On top of that, AppSec further proposes a privilege-based window system. It ensures that normal application cannot access the security sensitive application's X resource.

## 9. Conclusion

AppSec represents a significant step forward in protecting security sensitive applications' private data on an untrusted operating system. By protecting the code integrity of DSOes and tracking pages skillfully, we enable sensitive applications to use AppSec transparently. AppSec protects the security of memory private data by verifying kernel mem-

ory access according to application's intention. With human-machine interaction devices isolation and a privilege-based windows system, AppSec addresses the human-machine interaction issues such as keyboard interception and screen capture. The major advantages of AppSec are that AppSec secures both the memory data and human-machine interaction data and all protections provided by AppSec do not need to re-design, modify or recompile applications and OS. The prototype shows that AppSec only incurs 6%∼10% performance overhead.

## Acknowledgments

This research was supported in part by NSFC under Grant No.(60933003, 61272460), Ph.D. Programs Foundation of Ministry of Education of China under Grant No. (201202011 10010) and 863 Program (2012AA010904).

## References

[1] Xen Arbitrary Code Execution. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3124`.

[2] Google V8 Benchmark Suite. URL `http://v8.googlecode.com/svn/data/benchmarks/v7/run.html`.

[3] The connection methods to the X server. URL `https://www.debian.org/doc/manuals/debian-reference/ch07.en.html#_the_connection_methods_to_the_x_server`.

[4] VMWare Arbitrary Code Execution. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1209`.

[5] PCI Local Bus Specification. URL `http://www.math.uni.wroc.pl/~p-wyk4/so/pci23.pdf`.

[6] Trusted Platform Module (TPM) Summary. URL `http://www.trustedcomputinggroup.org/resources/trusted_platform_module_tpm_summary`.

[7] X Window System. URL `http://en.wikipedia.org/wiki/X_Window_System`.

[8] INTEL® 64 AND IA-32 ARCHITECTURES SOFTWARE DEVELOPER'S MANUAL. Instruction Set Extensions Programming Reference. *Intel Corporation*, January 2013.

[9] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[10] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *6th Conference on Innovative Data Systems Research*, Jan. 2013.

[11] A. Azab, P. Ning, and X. Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.

[12] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 239–252. ACM, 2013.

[13] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 267–283. USENIX Association, 2014.

[14] A. D. Central. BIOS and Kernel Developer's Guide for AMD Family 15h Models 00h-0Fh Processors.

[15] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. 2007.

[16] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGPLAN Notices*, volume 43, pages 2–13. ACM, 2008.

[17] Y. Cheng, X. Ding, and R. H. Deng. Driverguard: A fine-grained protection on i/o flows. In *Proceedings of European Symposium on Research in Computer Security*, pages 227–244. Springer, 2011.

[18] I. Corporation. Lagrande technology preliminary architecture specification. *Intel Publication*, (D52212), 2006.

[19] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the nineteenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014.

[20] Y. Dai, Y. Shi, Y. Qi, J. Ren, and P. Wang. Design and verification of a lightweight reliable virtual machine monitor for a many-core architecture. *Frontiers of Computer Science*, pages 1–10.

[21] Y. Dai, Y. Qi, J. Ren, Y. Shi, X. Wang, and X. Yu. A lightweight VMM on many core for high performance computing. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, pages 111–120. ACM, 2013.

[22] G. Duc and R. Keryell. Cryptopage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 483–492. IEEE, 2006.

[23] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[24] D. R. Engler, M. F. Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[25] A. Filyanov, J. M. McCuney, A.-R. Sadeghiz, and M. Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 1–12. IEEE, 2011.

[26] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual

machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, pages 1–1, 2004.

[27] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.

[28] C. Gebtry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *32nd International Cryptology Conference*, 2012.

[29] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[30] V. George, T. Piazza, and H. Jiang. Technology Insight: Intel© Next Generation Microarchitecture Codename Ivy Bridge, 2011. URL `www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf`.

[31] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. 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)*, pages 265–278. ACM, 2013.

[32] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, Berkeley, CA, USA, 2012. USENIX Association.

[33] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. Ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, 2014.

[34] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[35] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35 (11):168–177, 2000.

[36] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy (SP)*, pages 143–158. IEEE, 2010.

[37] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM, 2013.

[38] R. Nikolaev and G. Back. Virtuos: an operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 2013)*, pages 116–132. ACM, 2013.

[39] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda. PRIVEXEC: Private Execution as an Operating System Service. In *IEEE Symposium on Security and Privacy*. IEEE, 2013.

[40] R. A. Popa, C. M. Redfield, N. Xeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *23rd ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[41] M. Seaborn. Plash: tools for practical least privilege, 2008. URL `http://plash.beasts.org/index.html`.

[42] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the eros trusted window system. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, pages 12–12. USENIX Association, 2004.

[43] L. Soares and M. Stumm. Flexsc: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI*. ACM, 2010.

[44] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, 2012.

[45] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171, 2003.

[46] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein. Mrcrypt: static analysis for secure cloud computations. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 271–286. ACM, 2013.

[47] A. Virtualization. Secure Virtual Machine Architecture Reference Manual. *AMD Publication*, (33047), 2005.

[48] J. Yang and K. 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*, pages 71–80. ACM, 2008.

[49] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Usenix Security*, pages 337–352, 2013.

[50] Z. Zhou, V. Gligor, J. Newsome, and J. McCune. Building verifiable trusted path on commodity x86 computers. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 616–630. IEEE, 2012.