Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions

Xi Xiong
The Pennsylvania State University
xixiong@cse.psu.edu

Donghai Tian
The Pennsylvania State University
Beijing Institute of Technology
donghai@psu.edu

Peng Liu
The Pennsylvania State University
pliu@ist.psu.edu

Abstract

Kernel extensions are widely used by attackers to compromise the operating system kernel. With the presence of various untrusted extensions, it remains a challenging problem to comprehensively preserve the integrity of OS kernels in a practical and generic way. In this paper, we present HUKO, a hypervisor-based integrity protection system designed to protect commodity OS kernels from untrusted extensions. In HUKO system, untrusted kernel extensions can safely run to provide desired functionalities. The behaviors of untrusted extensions, however, are confined by mandatory access control policies, which significantly limit the attacker's ability to compromise the integrity of the kernel. To guarantee multi-aspect protection and enforcement, HUKO leverages hardware assisted paging to transparently isolate *untrusted extensions from the OS kernel. Moreover, HUKO overcomes the challenge of mediation overhead by introducing a novel design named subject-aware protection state transition to eliminate unnecessary privilege transitions caused by mediating allowed accesses. Our approach is practical because it requires little change for either OS kernel or extensions, and it can inherently support multiple commodity operating systems and legacy extensions. We have implemented a prototype of HUKO based on the open source Xen hypervisor. The evaluation results show that HUKO can comprehensively protect the integrity for both Linux and Windows kernel from various kinds of malicious extensions with an acceptable performance cost.*

1 Introduction

Kernel-level extensions are widely supported in commodity operating systems to extend the kernel's functionality. However, the extension interface could also be leveraged by attackers to tamper the integrity of the OS kernel. For example, attackers can install malicious extensions such as kernel rootkits to hide their activities in the system. On the other hand, the existence of buggy third-party device drivers exposes many vulnerabilities which can be exploited by attackers to inject their malicious code into the kernel space. These untrusted extensions threaten the kernel integrity greatly, yet unfortunately in many cases users have to let them run in order to provide the desired functionalities and availability. Therefore, preserving the OS kernel integrity from the presence of untrusted extensions remains a challenging problem.

Previous research efforts on protecting the OS kernel primarily target at one aspect of kernel integrity protection, such as code integrity [27, 25], data integrity [10, 31] and control flow/data integrity [33, 23, 35]. While these approaches are effective against certain categories of attacks, the lack of multi-aspect protection renders the system's incapability to deal with multiple types of malicious activities. For example, systems that only guarantee the integrity of kernel code and hooks are vulnerable to DKOM (Direct Kernel Object Manipulation) attacks. Similarly, protecting kernel code and data is not enough for defeating new control flow attacks such as return-oriented rootkits [28, 24]. Moreover, current approaches are also limited in countering advanced attacks such as direct kernel stack manipulation in commodity systems, in which the attacker manipulates control and/or non-control data in the kernel stack shared by all code entities in the OS kernel.

Another difficulty is about making the protection scheme practical and generic. Several proposals [27, 25, 22] preserve kernel code integrity by preventing untrusted code from executing in the kernel space to defeat code injection and malwares. However, they also eliminate all the benign functionalities and availability provided by untrusted extensions. Quite a few security approaches [23, 10, 26, 11, 14] utilize the knowledge of kernel data structures to achieve fine-grained auditing and intrusion detection. However, these approaches are dependent upon data structure semantics of a specific kernel, making them difficult to adapt different OS kernels with another version or from other venders. Moreover, the performance overhead induced by dynamically reconstructing and tracking fine-grained kernel objects makes these approaches not that suitable for an online protection system.

To achieve tamperproof and transparency in a system that protects the OS kernel, a common approach is to leverage the virtual machine monitor (VMM), which provides another layer of indirection. In such systems, to protect a security sensitive-kernel object, the VMM intercepts all the events that access this object and validates each event based on the protection policy. This approach is effective for protecting a small number of crucial objects in the kernel. However, severe performance problem arises once the quantity of protected objects becomes large, say, the entire kernel code and data area. The reason is that, no matter how VMMs are trapping these events (e.g., via instruction instrumentation or page protection), performing mediation for each event will always cause control transfers between the VMM and the guest, which will need multiple time-consuming privilege transitions (e.g., ring faults or VMEXITs). Researchers have proposed techniques such as *hook indirection* [33] to mitigate the performance problems for hook protection. However, this approach is only useful for protecting objects that are scattered across page boundaries, yet still cannot be applied to the entire kernel code and data.

This paper presents HUKO, a hypervisor-based integrity protection system designed to protect commodity operating system kernels from untrusted extensions. HUKO allows users to execute untrusted extensions in the kernel space to provide desired functionalities. The behaviors of untrusted extensions, however, are confined by mandatory access control policies, which significantly limit the attacker's ability to compromise the integrity of the kernel. In order to achieve multi-aspect protection, HUKO leverages hardware assisted paging to *transparently isolate* untrusted extensions from the OS kernel so that it could mediate all interactions (including memory modification, control transfers and DMA) between extensions and the kernel. Regarding kernel stack integrity, HUKO's approach includes a VMM-level *private stack* with lazy synchronization to offer a transparent and efficient stack separation and permission management for unmodified OS kernels. To address the challenge of mediation performance, HUKO introduces a design named *subject-aware protection state transition* to eliminate unnecessary privilege transitions caused by mediating benign accesses. HUKO is a *practical* approach because it requires little change for either OS kernel or extensions. Also it does not depend on semantic knowledge of kernel data structures so that it can inherently support multiple commodity operating systems and legacy extensions.

We have implemented HUKO prototype based on the open source Xen hypervisor. To facilitate HUKO's design, we leverage contemporary hardware virtualization techniques such as Intel's EPT, VPID and VT-d[1] [4, 5]. We evaluated HUKO's protection effectiveness by running malicious kernel extensions in both Linux and Windwos. Our experiments show that HUKO can protect the kernel integrity in the presence of various kinds of malicious extensions, including DKOM and return-oriented rootkits. In terms of mediation performance, the evaluation results show that the average performance overhead in application level benchmarks is ranged from less than 1% to 21%. Even for extreme cases when HUKO isolates the entire `ext3` file system (the largest module in our Linux OS) from the kernel, the mediation overhead for extracting a Linux kernel tarball is about 21%, with the protection state transfer rate at 390,000 per second.

We believe that HUKO provides a generic and transparent framework for running untrusted code in OS kernel with enhanced integrity protection for commodity systems. Also, this framework could be used to enforce mandatory access control policies inside commodity OS kernels with an acceptable impact on performance.

The remainder of this paper is organized as follows. We first describe the threat model, the integrity properties that HUKO enforces and our assumptions in Section 2. Section 3 provides an overview of the design of HUKO. Section 4 details the design and implementation of the entire architecture. Our evaluation experiments for both the protection effectiveness and performance of HUKO are shown in Section 5. We discuss limitations and future work of our system in Section 6. Finally, Section 7 introduces related work and Section 8 is the conclusion.

2 Kernel Integrity Threat Model

In this paper, we focus on attacks that the adversary utilizes the kernel extension interface to compromise the kernel integrity, which is the most common method to attack a commodity OS kernel. To specifically illustrate the threats, we present three different attack scenarios as follows: (1) The attacker gains the root privilege of the entire system, then he loads malicious extensions such as kernel-level rootkits into the OS kernel. (2) The attacker exploits a vulnerability existed in a benign kernel extension (e.g., a buggy device driver) to inject malicious code and therefore changes the extension's behavior. (3) A careless normal user loads an unverified kernel extension (e.g., a third-party device driver), which contains malicious code. There are various ways in which these malicious code could damage the control flow integrity and data integrity of the ker-

[1]AMD also has similar techniques with different names.

nel, for example, direct modification of kernel code, modifying control data (e.g., system call table, IDT and function pointers), modifying non-control data (e.g., process descriptors and file system metadata), writing to the kernel space via malicious DMA requests, and stack manipulation (e.g., return-oriented attacks).

We classify subjects in an operating system kernel into three categories. The first category is the OS kernel, which HUKO aims to protect. The second category consists of trusted kernel extensions, which are kernel extensions trusted by the system administrator. Generally their code need to be attested and verified to guarantee security. The third category is untrusted extensions, which are extensions that may be compromised or inherently malicious. Rootkits and unverified device drivers belong to this category.

HUKO protects the integrity of the OS kernel by enforcing the following properties in a mandatory protection system:

- ***Kernel code/data integrity***: code, static data and dynamic data of the OS kernel are protected from being modified by untrusted extensions via direct memory access or DMA access.

- ***Architectural state integrity***: architectural environment describing the execution state of the OS kernel such as segment registers, control registers and certain flag registers cannot be altered by untrusted extensions.

- ***Control flow integrity***: (1) control transfers from untrusted extensions to the OS kernel, including function calls, jumps and preemptions, are restricted to a set of kernel service functions named *trusted entry points* (TEPs) specified by the OS provider or the administrator; (2) function call consistencies such as call-return consistency are strictly enforced.

- ***Stack integrity***: (1) malicious code cannot be injected into stack frames belonging to the OS kernel; (2) For an untrusted extension, manipulating control data (i.e., function pointers, return addresses) in its own stack frames cannot subvert control flow integrity stated above; (3) non-control data (i.e., saved registers, parameters and variables) and control data in stack frames owned by OS kernel or other extensions cannot be corrupted by an untrusted extension.

For practical and usability reasons, the default mandatory access control policy of HUKO does not prohibit the OS kernel from reading information from untrusted extensions, which is different from classic integrity models such as Biba. However, if there is a need to satisfy this strict integrity requirement, the flexible mediation and enforcement mechanism in HUKO can still support system administrators to write policies with appropriate exceptions to enforce the "*no read down*" property.

HUKO is designed to be an added-on layer which provides an enhanced integrity protection for various operating system kernels with an affordable performance cost. As a design principle, HUKO relies on as little semantics of any specific kernel as possible. On the other side, HUKO is not the elixir for every kernel security threats. For example, HUKO is limited in verifying the correctness of function parameters and general data passed between the OS kernel and extensions, which could open certain avenues that impact kernel integrity in indirect ways. Also our system does not prevent the untrusted extension from abusing the privilege granted by the OS kernel in current stage. We discuss these limitations and possible solutions in Section 6.

This paper is focused on protecting the integrity of OS kernels. Other security issues, such as attacks on secrecy (e.g., information leakage) and availability (e.g., interrupt flooding, abuse of resource) of OS kernels are not in the scope of this paper. Also, this work concentrates on dealing with threats from the kernel extension interface, and we assume that the hardware is trusted for the OS kernel. Regarding attacks to the kernel directly from the userspace, HUKO prevents untrusted kernel extensions from executing user-level content and prohibits user programs to write kernel memory. Previous work such as Secvisor [27] provides in-depth research on protecting the OS kernel from userspace intrusions using a hypervisor, and we believe that its method can be effectively integrated with HUKO to achieve a more comprehensive protection. At last, in HUKO system, the hypervisor is the trusted computing base which we assume its integrity is preserved.

3 HUKO Overview

3.1 Design Principles

The following paragraphs describe three major principles which motivated our research and guided our design process of the HUKO system.

- ***Multi-aspect Protection.*** The architecture must guarantee that the kernel integrity properties stated in Section 2 are enforced with mandatory protection. Security-sensitive operations that involve interactions between untrusted extensions and the OS kernel, including memory reference, DMA, control transfers and stack modification, must be mediated and validated upon mandatory integrity policies.

- ***Performance.*** The architecture must not have high performance impact due to mediation, object reconstruction/tracking or enforcing protection.

- ***Ease-of-Adoption.*** The architecture should support multiple commodity operating systems and any unmodified legacy kernel extension. The architecture should not change the semantics of either OS or the extensions. Also, the architecture should be a layered approach which requires little deployment efforts.

3.2 Design Overview

HUKO provides a transparent protection environment for commodity OS kernels in which untrusted kernel extensions can run with an enhanced protection. In HUKO system, we name all the kernel objects that are supposed to be protected by our mechanism *security-sensitive objects*. These objects are labeled and tracked by the labeling component in HUKO's hypervisor. Depending on the various purposes of deploying HUKO integrity protection, security-sensitive objects can be labeled as 1) the entire kernel code and data region, or 2) a given set of kernel objects that may be tampered by attackers to achieve specific goals, for example, hiding a malicious process by manipulating hooks and process descriptors. To guarantee multi-aspect protection and generality, in our design, by default we label and track the entire kernel code and data region as security-sensitive objects.

The following paragraphs abstractly explain various challenges we faced in designing the system as well as key features of HUKO.

Mediation Overhead. Regarding how to achieve the mandatory access control mechanism, an intuitive way is to intercept every access to security-sensitive objects, then to validate whether the access is permitted by the policy or not. This approach is straightforward and convenient for out-of-boxed monitoring, however, it is not practical because the mediation overhead is considerable even if the number of objects to be monitored is relatively small. We observed that many security-sensitive objects in the kernel are highly frequently accessed by operating system kernel itself. For example, in Linux, `task_struct` is a typical security-sensitive data object because it can be manipulated by rootkits to perform process hiding and privilege escalation. On the other hand, `task_struct` is also a crucial accounting and scheduling data structure which would be modified several times by the scheduler during each context switch. Posing mediation on these legal accesses through an external reference monitor (i.e., VMM) causes enormous amount of unnecessary privilege transitions (e.g., page faults, ring faults and `VMEXIT`), which result in serious impact on performance.

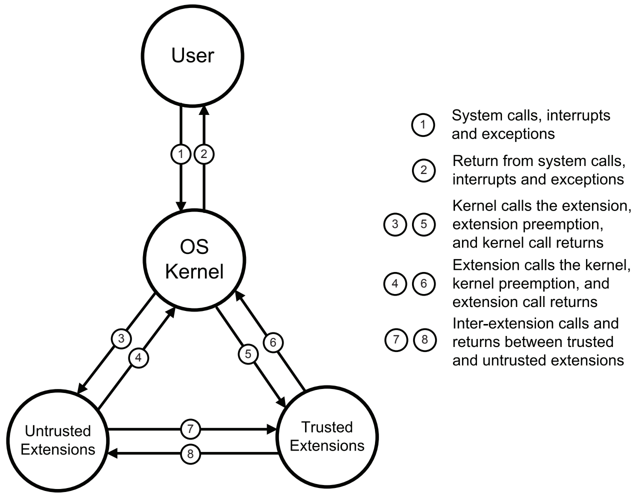
User
OS Kernel
Untrusted Extensions
Trusted Extensions
1 System calls, interrupts and exceptions
2 Return from system calls, interrupts and exceptions
3 5 Kernel calls the extension, extension preemption, and kernel call returns
4 6 Extension calls the kernel, kernel preemption, and extension call returns
7 8 Inter-extension calls and returns between trusted and untrusted extensions

Figure 1. The protection state transition diagram.

To overcome this limitation, HUKO adopts a design named *subject-aware state transition* which divides the system workflow into multiple protection states. The behavior of the protection mechanism is determined by the current protection state, which is further determined by precisely distinguishing the type of current subject in the guest system context. Specifically, if the current subject is an untrusted extension, HUKO does complete mediation on all accesses to security-sensitive objects in order to protect the kernel integrity. By contrast, in the case when the OS kernel is executing, HUKO poses minimal interposition on object accesses. It only needs to audit control transfer events that cause a protection state transition. In this way, the total number of privilege transitions caused by mediation is significantly reduced, which grants HUKO much better mediation performance. Table 1 illustrates an example of different protection behaviors that are associated with different protection states. From it we could see that the number of events that lead to privilege transitions (presented in grey cells) is minimized due to the subject-aware state transition mechanism in HUKO.

Figure 1 is the state diagram which shows the various protection states of HUKO system as well as the state transition events. Currently HUKO has four protection states, which correspond to the OS kernel, trusted extensions, untrusted extensions, and the user space, respectively. The state transition events include inter-subject function calls, various types of jump, interrupt handling, preemptions, system calls and associated returns from these routines. Mediating these events is essential to guarantee comprehensive control flow integrity, which we further discuss in Section 4.5. Tracking the state transition is mainly achieved by the

Object Label	Subject Category / Protection State								
	OS Kernel			Trusted Extensions			Untrusted Extensions		
	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
Trusted Entry Points	allow	allow	allow	allow	allow	audit allow	allow	deny	audit allow
Other OS Code	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
OS Data	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
Trusted Extension	allow	allow	audit allow	allow	allow	allow	allow	deny	deny
Untrusted Extension	allow	allow	audit allow	allow	allow	audit allow	allow	allow	allow
Private Stack Frames	allow	allow	deny	allow	allow	deny	allow	allow	deny
Other Stack Frames	allow	allow	deny	allow	allow	deny	allow	deny	deny
Trusted DMA	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
Shared DMA	allow	allow	allow	allow	allow	allow	allow	allow	allow
User Space Content	allow	allow	audit allow	allow	allow	audit allow	allow	allow	deny

Table 1. A sample MAC policy for preventing extensions from writing to kernel or executing unauthorized kernel code. The shaded cells indicate the corresponding events are mediated by the VMM and involve privilege transitions. Other events do not cause privilege transitions in HUKO. The write operation includes both normal write and DMA write. The not-listed “user" protection state is simply configured to deny any write to the kernel space.

isolation mechanism in HUKO, which we describe in Section 4.3.

Transparent Isolation. As we stated above, HUKO should have the ability to (1) distinguish the current subject in the guest context, (2) track all state transition events, (3) support different access control policies for different subject categories, and (4) mediate data modification flows and control flows between subject categories. Achieving these is non-trivial for commodity monolithic-kernel operating systems (e.g., Linux and Windows) since the OS kernel and its extensions reside within the same address space, and it is even more challenging especially considering our two design principles: external approach and good performance.

To tackle this challenge, we design an *isolation component* in HUKO's VMM to transparently isolate the extensions from the OS kernel. The isolation mechanism leverages hardware-assisted paging (HAP), which is a hardware-based virtualization technique supported by many modern processors. In our scheme, the enhanced memory virtualization component in HUKO's VMM maintains separate sets of HAP tables for each protection state in the system. These sets of HAP tables are synchronized with each other so that their corresponding entries are mapped to the same machine frame. Moreover, regarding security-sensitive objects, different HAP tables are reflecting different access rights according to the subject category and mandatory access control policies. Switching between these HAP tables is swift because it only involves a change to the HAP base pointer. In addition, HUKO significantly reduces the number of TLB flushes involved in each HAP table switch by utilizing Intel's Virtual-Processor Identifiers (VPIDs) technology. The multiple HAP table design renders efficient and practical isolation between the OS kernel and extensions, and it enforces separate access control policies for each type of subject accessing various kernel objects such as dynamic data structures, I/O buffers and kernel functions. Regarding kernel stack integrity, HUKO leverages the multiple HAP tables to achieve a VMM-level *private stack* with lazy synchronization mechanism to offer a transparent and efficient stack separation, which we discuss in Section 4.4.

Object Labeling. In mandatory protection systems, objects are labeled indicating their security properties to facilitate mediation. HUKO does *object labeling* in order to let the VMM identify security sensitive objects in the kernel. The labeling procedure is at the page granularity in the way that the labeling component assigns labels to the specific physical pages that contain security sensitive objects. There are two reasons for this. First, according to our design principles, HUKO is intended to rely on as little semantic knowledge of operating system as possible. Second, for a hypervisor-based approach, fine-grained dynamic object tracking in kernel often introduces too much reconstruction and tracking overhead, which is not practical for an online protection system. On the other hand, to ameliorate problems caused by the protection granularity gap, HUKO has mixed page labeling mechanism for handling pages that contain mixed code and data, as well as pages that are shared by both kernel and extensions.

Another issue is about how to track dynamic data for both kernel and extensions. To address this, HUKO inserts a trusted driver (labeled as a trusted extension) into the operating system to notify the hypervisor about the allocation and reclamation of the kernel memory. The driver is also aware of the owner subject of each page and reports updates to the hypervisor during runtime. We further discuss mixed page handling and dynamic content tracking in Section 4.2.

Protection Workflow. Table 1 shows a sample protection policy that regulates the data accesses as well as code executions of untrusted extensions. In this policy, the policy maker needs to specify a set of kernel functions as the trusted entry points. In practice, trusted entry points can be exported functions in the kernel symbol table or picked specifically by the system administrator. To preserve control flow integrity, besides kernel function calls, kernel preemption and return instructions should also be considered, which we will discuss in Section 4.5. In addition, this policy also prevents untrusted extensions from directly writing to the OS kernel or any trusted extensions, no matter the write is performed via memory instructions or DMA transfers.

HUKO enforces mandatory access control over the entire life period of any untrusted extension. To achieve this, HUKO tracks the lifetime of an extension by hooking the extension allocation, loading and unlinking routine of the kernel. These events will be trapped to the hypervisor and the labeling component will manipulate the corresponding page labels to perform dynamic tracking. Unless specified by the administrator, HUKO labels all newly loaded extensions as untrusted. During the protection process, if any event that violates the access control policy happens, HUKO will trigger a protection alarm from the hypervisor and provide essential information (e.g., type of policy violation and the execution context) to the system administrator for making proper security decisions.

4 Architecture Design and Implementation

Figure 2 provides the overview of the HUKO Architecture. There are four major components corresponding to principle functionalities in HUKO's design: object labeling, transparent isolation, stack integrity protection, as well as mediation and enforcement. In the following subsections we first provide a brief background on Hardware-Assisted Paging (HAP) technology used in our prototype. Then we discuss each major component in detail. In Section 4.6, we briefly describe the implementation of HUKO prototype on the Xen hypervisor.

4.1 Hardware-Assisted Paging Overview

To achieve memory virtualization, a common design for VMMs is to load shadow page tables (SPT) into the hardware MMU, which translate from guest linear addresses (GLA) to machine-physical addresses (MPA). However, to maintain this indirect mapping, the hypervisor must intercept and do SPT synchronization upon guest CR3 switches and each update of the guest page table (GPT). The hardware-assisted paging (HAP) technology is introduced to avoid the software overhead incurred under shadow paging. One implementation of HAP is Intel's Extended page tables (EPT) technology [4]. When this feature is turned on, the ordinary IA-32 page tables (referenced by control register CR3) translate from GLA to guest-physical addresses (GPA). In addition, the hardware MMU maintains a separate set of page tables (the EPT tables) which translate from guest-physical addresses (GPA) to the machine-physical addresses (MPA) that are used to access machine memory. As a result, guest OS can be allowed to modify its own IA-32 page tables and directly handle page faults. This allows a VMM to avoid the VMEXITs associated with shadow paging, which are a major source of virtualization overhead.

The reason why HUKO is built atop hardware assisted paging rather than the software-based shadow paging mechanism is two fold. The first reason is for better performance, which we just stated. Secondly, in SPT, access rights in SPT entries are synchronized with the corresponding GPT entries. Hence, changing the access rights in SPT entries for our protection purpose may potentially affect the correctness of guest OS for handling its own access rights. By contrast, in HAP, access rights in HAP entries and access rights in GPT entries are two completely different sets. Moreover, the HAP violation handling is transparently separated from the page fault handling mechanism of the guest OS, which makes it more flexible and easier to guarantee correctness.

4.2 Object Labeling

As shown in Table 1, in order to enforce the MAC policy, HUKO assigns various kinds of security labels to different kernel objects. The object labeling component is responsible for identifying kernel objects from the physical memory and managing security properties of these objects. As stated in Section 3.2, based on our design principles, HUKO directly associates object labels to the corresponding HAP entries. In specific, the labeling component makes use of a set of reserved bits in EPT entries. These reserved bits are never utilized by default so that changing these bits does not affect the hypervisor's functionalities. By encoding labels using these bits, HUKO currently can support 32 different potential object labels, providing flexibility and extendability to the protection scheme. This mechanism also reduces the time and memory space involved in every mediation and authorization action.

Handling Mixed Pages. In a commodity operating system kernel such as Linux, memory regions for kernel code,

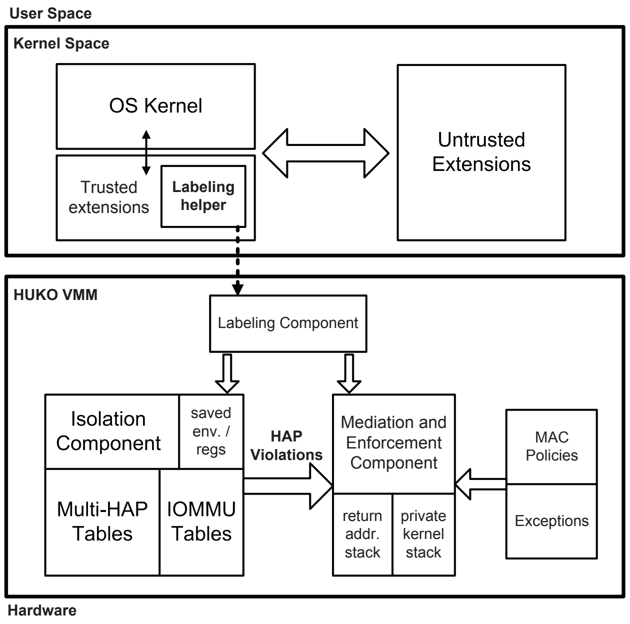

Figure 2. Overview of the HUKO Architecture.

kernel data and extensions are usually page aligned, which facilitates the labeling procedure in HUKO. However, there are still existences of mixed pages in which different objects co-exist together. To ensure comprehensiveness and correctness of the protection, the labeling component must be able to track objects within two categories of mixed pages: (1) pages containing both kernel code and kernel data, and (2) pages containing both the kernel and extensions.

A major type of mixed pages in the kernel is large sized page (e.g., 2MB superpage). In most cases, different objects reside in the same superpage, yet their boundaries are still aligned to the 4KB address regions. Based on this observation, given a large mixed page, HUKO *splits* the corresponding EPT superpage entry into multiple subpage entries (e.g., 2MB page entry to 512 4KB sub-entries) and assigns individual object labels to each subpage. Splitting EPT superpage entries improves the granularity of labeling and eliminates a majority of mixed page problems without changing the guest page table (GPT) entries. On the other hand, regarding mixed pages of 4KB size, HUKO assigns each of them with a mixed label. For example, considering a mixed page that has a mixed label of both kernel data and extension code, the hypervisor would trap all events that modify this page regardless of the current protection state. Then HUKO examines the physical address to see if it is in the range of extension text area and finally determines the object identity.

Tracking Dynamic Contents. Associating kernel objects to HAP page frames requires dynamically tracking of these objects. For static objects such as kernel code, static kernel data (including global variables), and trusted entry points, HUKO tracks them by leveraging the kernel symbol table (e.g., `Systemmap` file in Linux). On the other hand, for dynamic contents such as dynamic kernel data, stack and heap region, and loadable extensions, it is difficult and time consuming to track them at the hypervisor layer because of the semantic gap. HUKO tackles this problem by loading a trusted extension named *labeling helper* into the guest kernel. The labeling helper is responsible for letting the hypervisor be aware of the allocation and deallocation of kernel dynamic pages as well as the owner subject of each kernel page. This component is the only OS-dependent part in our system and we implemented a prototype in Linux. Specifically, dynamic data owned by an extension come from two major sources in Linux: (1) the page frame allocator for allocating bulk of pages, and (2) the SLAB allocator for allocating fixed sized of registered cache objects. For both cases, the labeling helper hooks the allocation and deallocation events and gathers information from the SLAB allocator (i.e., `kmem_cache_alloc`), the free page allocator, and the `load_module` routine. This information includes owner subject of the page (e.g., OS kernel or extension), the content type (e.g., kernel data or extension code), the guest page frame number, the virtual address range (for handling mixed pages), and the timestamp of each event. Then the labeling helper passes these information to HUKO via the hypercall interface, and the labeling component labels the corresponding EPT entries accordingly. To guarantee tamperproof, the labeling helper itself is labeled as a trusted extension at the load time so that it is protected by HUKO. Furthermore, HUKO prohibits read accesses to the labeling helper to prevent the leakage of protection information.

4.3 Isolation Component

The isolation component in HUKO is responsible for achieving complete mediation by establishing separate address spaces for different categories of subjects (i.e., the OS kernel, trusted extensions and untrusted extensions) to reside in. Subjects can freely access code and data in their own address spaces without interposition from the hypervisor. However, inter-address-space activities such as data writing and control transfer must be mediated and controlled by the VMM.

Multi-HAP Construction. The isolation component is built upon our enhanced memory virtualization mechanism named *multi-HAP*. Multi-HAP enables extensions and the kernel to share the same virtual-to-physical mapping of the entire kernel space, while it also enables the hypervisor to set different object access rights for different subject categories. In this scheme, the hypervisor maintains separate sets of HAP tables for each protection state (refer to Figure 1) in the system.[2] Figure 3 illustrates the architecture of the multi-HAP mechanism. For simplicity, only two sets of HAP tables are shown here, corresponding to the OS kernel state and the untrusted extension state, respectively. There is a HAP base pointer which points to the root level of a HAP table. During a protection state switch, HUKO changes the value of the HAP base pointer to another HAP table root, which represents another set of access rights. The access rights in HAP table entries are determined by the object label of the entry as well as the access control policy, and are updated when any object label changes.

To intercept control transfer events between different subject categories, for each protection state, HUKO manipulates the execution bit of its HAP table entries so that all the pages that do not belong to the subject category (corresponding to the protection state) are not executable. Attempts to execute content on these pages would cause HAP violations and are handled by the hypervisor. Section 4.5 describes this procedure in detail.

Synchronization. An important difference between multi-HAP and user-level page tables managed by the kernel is that, each HAP table in multi-HAP must maintain the entire mapping of the whole kernel space, rather than the address space associated with the protection state. This is because HUKO should allow the OS kernel and extensions to read each other's address space freely without any interposition. Therefore, the isolation component should always synchronize the entire kernel address mappings among HAP tables. We modify the hypervisor code so that changes to one HAP table (including allocating a new entry, changing an entry and removing an entry) always propagate to other HAP tables.

Optimize TLB Flushes. Considering the enormous function calls and returns between the OS kernel and extensions, the protection state transition rate in HUKO is very high (see Section 5.3). If the hypervisor flushes TLB on every page table switch during a state transition, the performance degradation due to the TLB misses caused by flushing is substantial. To mitigate this problem, HUKO takes advantage of Intel's Virtual-processor identifiers (VPIDs) technology, which enables a logical processor of the hypervisor to manage cache information for multiple linear-address spaces. In HUKO's VMM, we associate each protection state with a 16-bit VPID so that mappings and access rights are tagged according to the VPID in the address translating cache. During the state transition time, the EPT table switch

[2]It can be extended to support separate HAP tables for each subject, if needed.

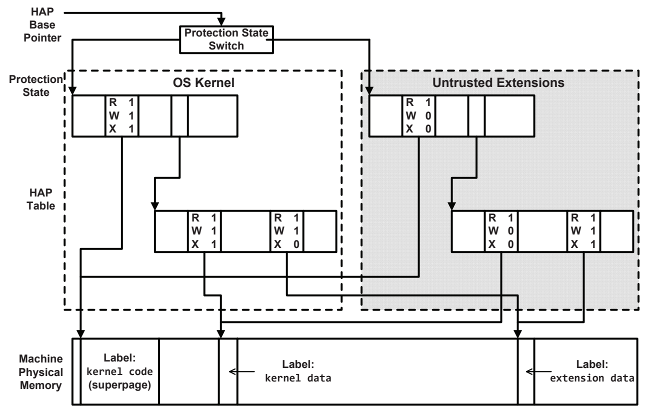

Figure 3. The multiple HAP tables for achieving isolation and mediation.

does not cause flush of the entire translating cache - it only flushes entries with specific VPIDs, which significantly reduces the TLB misses and improves the performance.

Preserving Architectural State. Sometimes malicious or compromised extensions could subvert certain invariants of the architectural state to fulfil their attacks. For example, a malicious extension could change the `GS` segment selector to point to its own version of processor data area (`pda`), which provides the kernel with incorrect information about the kernel stack, MMU state and IRQ processing. Therefore, HUKO must enforcing the integrity of system environment by preserving these invariants of architectural state.

Our approach takes advantage of the fact that, during a privilege transition, the architectural state is saved in the virtual machine descriptor (i.e., `VMCS` for Intel VT) and a virtual CPU struct (i.e., `vcpu` for Xen) of the VMM for future reloading. Hence we could straightforwardly integrate the architectural state protection with our subject-aware protection state design. In specific, at the time when the kernel enters untrusted extension protection state, HUKO saves the architectural state from the `VMCS` and `vcpu` to its own memory space. When the kernel is switching from untrusted extension state back to the OS kernel state, HUKO restores all the architectural state invariants by writing the saved values to the virtual machine descriptor and the virtual CPU struct.

4.4 Kernel Stack Integrity

Besides code, static and heap data, there is another important avenue which malicious extensions could exploit to subvert OS kernel integrity: the kernel stack. In specific, adversaries could perform the following actions to compromise the property of stack integrity stated in Section 2: (1)

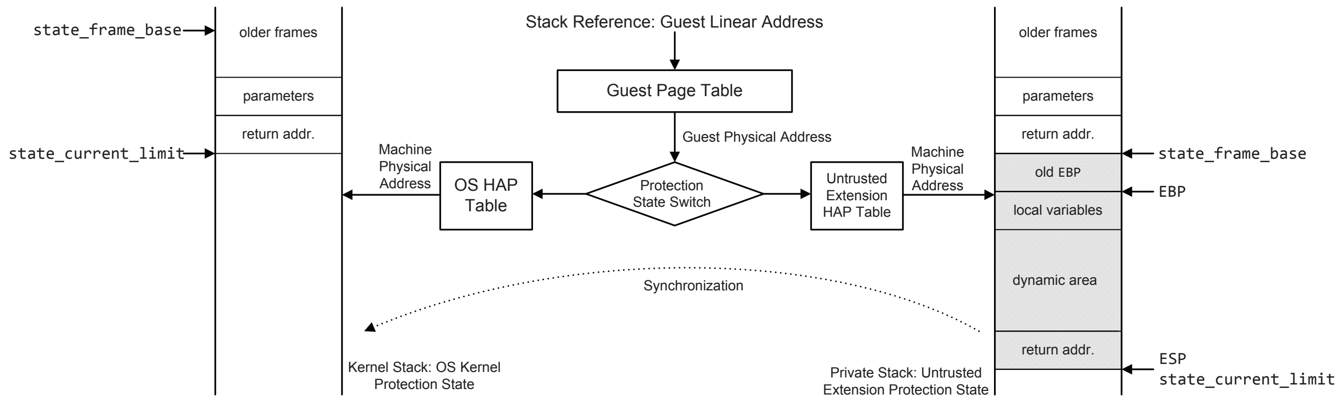

Figure 4. The transparent separated stack design supported by multi-HAP. The figure illustrates the two stacks at the time of protection state transfer in case an untrusted extension is making a call to the OS kernel. The shaded indicates the active stack frames (owned by the untrusted extension) which are going to propagate to the OS kernel stack.

injecting malicious code into the stack; (2) manipulating control data (i.e., function pointers, return addresses) in its own stack frames to subvert control flow integrity of the OS kernel. For instance, return-oriented and jump-oriented attacks belong to this category; (3) corrupting non-control and control data (i.e., saved registers, parameters and variables) in stack frames owned by OS kernel or other extensions. For example, a malicious extension could change the local variables and function parameters on the stack frame to let a certain kernel function return a false data value, or it may manipulate kernel IRQ and exception stack frames to change the behavior that OS kernel handles interrupts and exceptions.

For case (1), by setting the NX bit of corresponding HAP entries of kernel stack frames, HUKO ensures that code on kernel stack frames could never be executed. Regarding case (2), HUKO mediates the protection state transfers and maintains a dedicated return address stack to guarantee the control flow integrity, which we will describe in Section 4.5. To defend against attacks in category (3), HUKO grants untrusted extensions read permission to the entire kernel stack, but only gives them write permission to its own stack frames.

To efficiently manage kernel stack permissions in an unmodified commodity OS (e.g, Linux) is a non-trivial job, because of the following reasons: first, in such system, there is only one kernel stack for all kernel control paths associated with each user thread. Moreover, the stack frames are not page-aligned, making it difficult to set permissions for individual stack frames using current architecture. On the other hand, in terms of performance, it is not affordable to validate each stack modification made by untrusted extensions because stack modifications are too frequent.

The stack protection design of HUKO overcomes the above limitations. In order to preserves single kernel stack semantic and support unmodified commodity OSes, during the protection state of untrusted extensions, HUKO creates and maintains a private copy of the current kernel stack at the VMM layer, which is transparent and not observable from the guest OS. By manipulating GPA to MPA mappings in the Multi-HAP table, HUKO casts the same linear address range of the kernel stack to different machine frames for OS kernel and untrusted extensions. In this way, an untrusted extension is given a "faked" view that it shares the same kernel stack with other code entities in the kernel, however, its stack operations are automatically redirected to the private kernel stack copy placed on shadow machine frames reserved by HUKO. On the other hand, to protect stack integrity in an efficient manner, HUKO adopts a "lazy synchronization" design: instead of checking permissions each time the stack is accessed, HUKO only performs stack synchronization when current protection state is switching between untrusted extensions and the OS kernel. During synchronization, HUKO propagates stack modifications from the private stack to the real kernel stack with the following rule enforced: only changes made to its own stack frames are propagated to the real kernel stack, while updates outside its own stack frames are discarded.

In the following we use Linux as an example to illustrate the private stack design achieved by multi-HAP tables, which is shown in Figure 4. In Linux, each user process is associated with a two-page sized kernel stack. The scope of the current kernel stack can be determined by the `ESP` register and the per-CPU data structure pointed by the `GS` segment selector. HUKO maintains two data values for each protection state: `state_frame_base` and `state_current_limit`, respectively. These two values designate the active stack frames associated with each protection state, and only in these stack frames modifications are propagated to the other stack. During each protection state transfer, HUKO updates `state_frame_base` and `state_current_limit` based on the values of `EBP` and `ESP` registers at that time point.

4.5 Mediation and Enforcement

The goal of the mediation and enforcement component is to audit all the write flow and control transfer events between untrusted extensions and the kernel. Also it is responsible for validating these events to enforce integrity protection according to mandatory access control policies.

EPT Violation Handling. HUKO relies on the EPT violation mechanism to achieve mediation and protection enforcement. Figure 5 depicts the work flow of how HUKO handles various kinds of EPT violations. When an EPT violation occurs, HUKO first checks if the physical frame is labeled as a valid kernel object. If yes, then it checks if the violation is caused by our protection mechanism or by emulated MMIO and log-dirty events. An EPT violation caused by HUKO's protection mechanism indicates a sensitive control transfer event or a sensitive data access. To properly handle it, HUKO first examines the following information: (1) the qualification bits which reveal the actual type of the violation, (2) the current state, and (3) the label of the faulting frame. Then it determines whether to allow the operation or to trigger a protection alarm based on information collected and the access control policies.

As we stated in Section 3.2, subjects in HUKO can freely read and write their own code and data. Also, inter-subject read accesses are always allowed in our default policy. These allowed events do not cause any EPT violation so that they cannot be logged by the hypervisor. However, for forensics purposes, the system administrator may want to audit some types of crucial events yet still allow these events to happen. Hence, HUKO adds another action named *audit allow* to enable logging of these specific data accesses. To implement the audit allow mechanism, HUKO sets the access rights in the corresponding EPT entries so that audit-allowed events would cause EPT violations and be audited by the hypervisor. Then HUKO *emulates* the offending instructions without changing the previously set access rights. In this way, the audit allow operation is completed and the EPT entries can still be used to trap further events of the same kind.

Protecting Control Flow Integrity. As previously stated, HUKO sets the execution bits of multi-HAP entries so that only untrusted extension code can be executed in the untrusted extension protection state. When an execution violation indicating a control transfer from an untrusted extension to the OS kernel occurs, HUKO enforces the control flow integrity rules under the following conditions: (1) the untrusted extension is calling the kernel via `call` and `jmp` instructions. In this case, HUKO allows the operation only when the violating address belongs to a trusted entry point. This prevents untrusted extensions from accessing unauthorized kernel functions or jumping to arbitrary positions in the kernel. (2) The kernel preempts the untrusted extension for higher priority interrupts. In this case, HUKO ensures that the violating address belongs to an interrupt handler routine in the IDT table. (3) The extension returns to the kernel from a previous call. This could be leveraged by return-oriented rootkits to divert the control flow to a sequence of return-oriented instructions in the kernel. To tackle this problem, HUKO maintains a separate return address stack to keep track of the call/return sequences between the OS kernel and untrusted extensions. In this way, we guarantee the return address to the kernel must correspond to the address of the kernel code that made the call. Also, the sequence of return addresses must satisfy the last-in-first-out property. Considering the fact that most return-oriented attacks need an initial return to the first return-oriented instruction sequence, our approach provides an effective counter method.

Handling DMA writes. Besides memory writes performed by CPU instructions, DMA is another way for extensions to write data into the kernel memory. Previous proposals [32] have limited capability of handling DMA because the data transfer is not controlled by the processor or memory controller. Fortunately, the introduction of hardware IOMMUs (Intel's VT-d and AMD's IOMMU) brings the possibility to efficiently mediate and control DMA memory access. When used in virtualization, the IOMMU can enable pass-through device models which support independent address translations using IOMMU page tables for DMA activities.

In HUKO prototype, we leverage the DMA remapping mechanism provided by Intel's VT-d technology [5] to protect the kernel integrity from DMA writes. Currently we explicitly set the IOMMU page tables so that pages labeled as OS kernel and trusted extensions cannot be used in DMA. On the other hand, HUKO allows DMA activities on the pages that are labeled as untrusted extensions. Our ongoing work employs multiple IOMMU page tables and switch facilities for different protection states, which is very similar to the multi-HAP mechanism. This scheme introduces new DMA object labels shown in Table 1 and allows the kernel and all extensions to do DMA in a protected manner. Another more flexible optimization is to integrate the

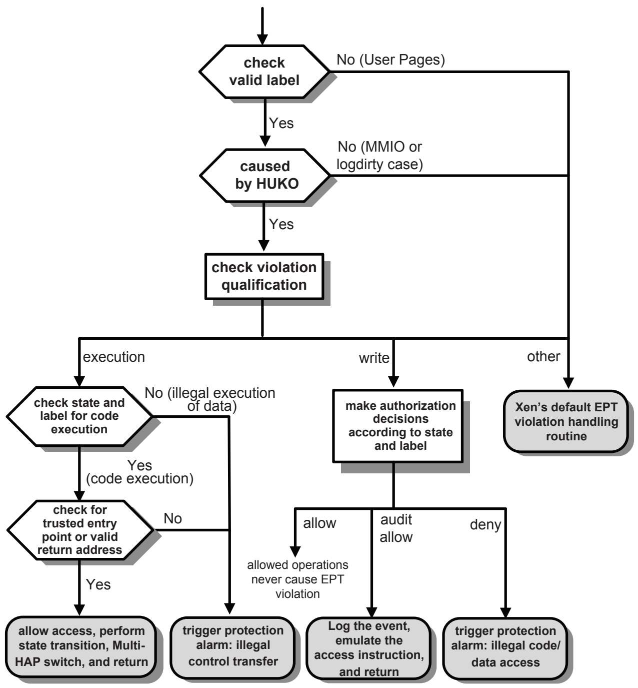
check valid label
No (User Pages)
Yes
caused by HUKO
No (MMIO or logdirty case)
Yes
check violation qualification
execution
write
other
check state and label for code execution
No (illegal execution of data)
make authorization decisions according to state and label
Xen's default EPT violation handling routine
Yes (code execution)
check for trusted entry point or valid return address
No
allow
audit allow
deny
allowed operations never cause EPT violation
Yes
allow access, perform state transition, Multi-HAP switch, and return
trigger protection alarm: illegal control transfer
Log the event, emulate the access instruction, and return
trigger protection alarm: illegal code/ data access

Figure 5. The EPT violation handling diagram of HUKO.

IOMMU page tables with the multi-HAP page tables so that IOMMU can utilize the guest-to-machine physical address translation as well as access control enforcement provided by the multi-HAP mechanism.

Supporting Exceptions. Given the complexity of commodity operating system kernels and the variety of enormous extensions in the wild, it is necessary for HUKO to support exceptions for access control enforcement. There are three types of exceptions in HUKO. The first type offers an untrusted extension the privilege to write into specific objects in the kernel. The second type allows an untrusted extension to make certain calls to the kernel, but not through trusted entry points. The third type of exception is about exporting write permissions in kernel stack frames. These exceptions are provided by the administrator to achieve specific needs on flexibility and performance, and they are stored and protected in the VMM memory space. Section 5.1 provides a further discussion in Linux OS.

In our current prototype implementation HUKO uses mixed page labels to handle exceptions. Pages that contain exception objects are labeled as "*mixed exception*", and the hypervisor will check the virtual address upon each violation to determine whether the event is an exception. This approach has bad performance in case the number of exceptions is large or exceptions occur frequently. We have an optimized design for handling exceptions and mixed pages. In that scheme, HUKO copies all the exception objects onto a set of allocated exception pages. By dynamic patching of instructions, HUKO redirects all the operations accessing exception objects to the corresponding copy on the exception pages at the run time. This method reduces the total number of EPT violations on exception pages and mixed pages. We plan to implement this optimization in our future work.

4.6 Modifications to Xen

We implemented HUKO by modifying the Xen hypervisor (version 3.4.2 x86-64 HVM Guest), which is a full-fledged open source hypervisor commonly used in various enterprise systems. The HAP mechanism used in the isolation and labeling component is based on Intel's EPT, yet it does not require much effort to adapt AMD's NPT. The total amount of code added to the Xen hypervisor is approximately 3,300 lines. And the Linux implementation of the labeling helper trusted extension consists of about 450 lines of code.

A major effort of our prototype implementation is to extend the memory virtualization sub-system of Xen to support the multi-HAP mechanism. In HUKO prototype, each HAP table is essentially a four-level EPT paging structure. The root-level index of each paging structure is stored in an array named `huko_phys_table_index`, which is placed in the architecture-specific per-domain structure `arch_domain`. To construct multi-HAP tables, HUKO first traverses all the existing physical-to-machine (p2m) mappings from the domain's `page_list`. Then it allocates EPT entries using free pages maintained by `p2m_freelist`, which are Xen's reserved pages for storing p2m mappings. The security label of each GFN is stored in bits 61:57 of the corresponding EPT entry and managed by the labeling component. HUKO then decides the access rights of an EPT entry from its security label, the MAC policy as well as the protection state which it belongs to. HUKO keeps this allocation process until all HAP tables are established. During each state transition, HUKO switches among multiple EPT paging structures by changing the EPTP pointer and associated VPID in the VMCS fields.

For each protection state, we introduced a security control block (SCB) which is linked to the `domain` structure. The SCB stores essential information for tracking a protection state, such as the identity of the current subject, the virtual address range of the subject's code and data, the previous protection state, the address of the last entry point, a copy of stack pointers, and a link to its return address stack. To achieve mediation and policy enforcement, we added additional routines to the paging violation handler of EPT and the Vt-d pass-through (IOMMU) driver, which are `ept_handle_violation()` and `iommu_page_fault()`, respectively. We exported two new hypercalls to the labeling helper for delivering run-time information to the labeling component.

5 Evaluation

In this section, we describe the deployment and experimental evaluation of the HUKO prototype. There are two goals of our evaluation. The first is to evaluate HUKO's effectiveness for defending against various real-world malicious extensions that damage the OS kernel integrity in different ways. The second goal is to measure the performance cost introduced by HUKO using both application-level and micro benchmarks.

All experiments were conducted on a Dell PowerEdge T310 Server with a 2.4GHz Intel Xeon X3430 and 4GB memory. The Xen hypervisor version is 3.4.2. The dom0 system is fedora 12 with kernel version 2.6. We used a 64bit Ubuntu Linux (8.04.4) with kernel version 2.6.24 as our guest OS. All Linux partitions were configured to use the `ext3` file system. For Windows experiment, we chose Windows XP SP2 as our guest system.

5.1 Deploying HUKO

As stated in Section 3.1, HUKO is intended to minimize the required effort for deploying the protection system. Instead of establishing protection domains at the OS layer [32] or at the hardware architecture layer [37], the implementation of almost all the functionalities (i.e., memory protection and access control) in HUKO is at the virtualization layer, which makes the protection mechanism guest-independent, adaptive, and easy-to-undeploy. Moreover, HUKO does not enforce access control for specific kernel objects, and it only has several generic types for object labeling. While this approach sacrifices the benefits of semantic-rich access control at finer granularity, it does offer a much easier configuration compared to rich-typed protection system such as SELinux [7]. In the following paragraphs we use the Linux OS as an example to briefly describe the deployment of HUKO.

The first step is to set up the basic information about kernel layout, objects and TEPs. In Linux, most of these information could be acquired from the kernel symbol table associated with the specific kernel. For example, the address range of Linux kernel code is determined by kernel symbol `_text` and `_etext`. Similarly, the boundaries of initialized and uninitialized kernel static data can be identified by symbol `_edata` and `_end`. At runtime, the labeling helper is responsible for collecting dynamic information for object labeling. For instance, the code and data range for an extension could be retrieved from the accounting data structure `module` when the extension is being loaded into the kernel.

In Linux, most kernel APIs and global data are exported to the kernel symbol tables using the `EXPORT_SYMBOL` macro. The address of kernel symbols can also be retrieved in the `System.map` file. In this way we could collect all the entry addresses for exported kernel functions. In our current prototype, we treat all the exported kernel APIs as the Trusted Entry Points (TEPs). In our future work, we are expecting to extend HUKO to achieve the least privilege property, by which we infer and enforce the set of kernel APIs that a specific extension can call. We do a further discussion on this issue in Section 6.

Besides common settings, administrators sometimes also need to provide extension-specific exceptions to make an extension run correctly. There are mainly three types of exceptions in a HUKO system. The first type of exceptions consists of non-exported functions. In Linux, certain kernel functions are not explicitly exported, instead, they are accessed by direct address reference or address assigning to function pointers. Fortunately, these cases are not recommended nowadays and getting rare in recent Linux kernels. To deal with them, the administrator should manually specific the entry address of these kernel APIs as TEPs. The second category of exceptions consists of OS kernel data of which the kernel intentionally grants write permission to extensions. In many cases, the shared data are used as various kinds of buffers and caches in the kernel, and they are usually still page-aligned. The labeling helper notifies the hypervisor when these data are allocated, and HUKO assigns `Shared_Data` type to these pages in the multi-HAP table to allow write access for both OS kernel and untrusted extension protection states. Shared data that are not page-aligned with non-shared kernel data are required to set up exceptions using mixed pages. Regarding write-sharing for kernel global variables, the administrator could specify their address in the exceptions according to the kernel symbol table. The third category of exceptions belongs to stack permission which OS kernel needs to grant extensions write permission to its local variables on the stack. For example, OS kernel could pass the address of a local variable to an extension in parameters during a function call. To address these situations, the administrator should specify the addresses of functions that require stack exceptions and how many previous frames need to be modified by each function. Then at the time that control returns to these functions, instead of synchronizing only its own stack frames of the extension, HUKO synchronizes all the necessary previous stack frames specified by the given exception.

Untrusted Extension	Behavior	Violation Triggered	Violating Object Label
EnyeLKM	add binary code to kernel	Illegal code access	OTHER OS CODE
all-root	DKOM (modify task_struct) modify control data (sys_call_table)	Illegal data access	OS DATA
adore-ng	modify function pointers	Illegal data access	OS DATA
hp	DKOM (modify task_struct linked list)	Illegal data access	OS DATA
lvtes	call unauthorized function (module_free)	Invalid code execution	OTHER OS CODE
return-oriented extension	modify return addr. on the stack	Invalid return address	Return addr. stack
FUTo (Windows)	DKOM (modify PspCidTable)	Illegal data access	OS DATA
TCPIRP (Windows)	modify function pointers	Illegal data access	OS DATA
basic_int (Windows)	add binary code to kernel	Illegal code access	OTHER OS CODE

Table 2. Protection effectiveness of HUKO against a collection of malicious extensions.

5.2 Protection Effectiveness

We evaluated the effectiveness of HUKO for kernel integrity protection with a collection of malicious extensions on both Windows and Linux. These extensions include 8 real-world rootkits and one self-implemented malicious extension for return-oriented attacks, which are shown in Table 2. As a result, all of these malicious extensions triggered protection alarms once they attempted to damage the kernel integrity. In the following paragraphs we describe three representative experiments in detail.

Code Integrity. EnyeLKM [3] is a Linux kernel rootkit which modifies the kernel text by putting "salts" inside `system_call` and `sysenter_entry` handlers. With HUKO protection, an illegal code modification alarm was triggered when either `set_idt_handler` or `set_sysenter_handler` was called. Both functions were trying to add binary text to kernel object labeled as `OTHER_OS_CODE`.

Data Integrity. The `all-root` [1] rootkit is a simple DKOM Linux kernel rootkit that modifies both control and non-control data to achieve privilege escalation. In its initialization routine `init_module`, this rootkit replaces the `sys_getuid` entry of the `sys_call_table` with its own function `give_root`, which changes the uid, gid, euid and egid field of the current `task_struct` to 0 (root). In this attack, the first modified data belongs to static control data while the latter belongs to dynamic non-control data. When we launched this attack in a system protected by HUKO, it immediately triggered a protection alarm indicating an illegal data access (caused by the first modification) from untrusted extensions to an object labeled as `OS_DATA`. In order to test the second data modification, we deliberately made decisions to allow the first modification and let the system continue to run. Then we executed a `getuid` system call from the user space to trigger the malicious replacement function. Again, HUKO triggered an illegal data access alarm, which was also caused by directly modifying dynamic non-control kernel data (labeled as `OS_DATA`) at the "untrusted extension" protection state.

Control Flow Integrity. Besides malicious extensions that modify control-data (e.g., function pointers) or make illegal call/jump to the kernel, the return-oriented attack is another way of tampering control flows in the kernel. To evaluate HUKO's effectiveness in countering such attacks, we implemented a return-oriented malicious extension in our experiment. Upon called, this extension modifies its return address on the stack to an arbitrary point in the kernel text area, which is recognized as a return-instruction gadget. We loaded this extension to a Linux system protected by HUKO. As a result, HUKO successfully prevented the control flow diversion caused by the modified return address, since the LIFO property of the return address stack was no longer kept.

5.3 Performance Overhead

To measure the performance cost introduced by HUKO, we ran a set of benchmarks to compare the performance of a guest system protected by HUKO with one that does not. For each benchmark, we labeled one or several relevant kernel extensions as untrusted so that they were isolated from the kernel. For all workloads we enforced the sample policy showed in Table 1. To fully test HUKO's performance overhead under stressed conditions, we chose two largest and most active kernel extensions in our Linux system: `8139too` and `ext3`. The `8139too` is the network interface card driver and the `ext3` extension is the file system module. These extensions are invoked multiple times for each network I/O requests or file system operations so that they have the highest control transfer rates with the OS kernel. Hence, marking them as untrusted generally represents the worst-case performance of HUKO when the

Benchmark	**Untrusted Extensions**	Number of Protection State Transitions	Native Performance	HUKO Performance	**Relative Performance**
Dhrystone 2	`8139too + ext3`	N/A	$10,855,484$ lps	$10,176,782$ lps	0.94
Whetstone	`8139too + ext3`	N/A	$2,270$ MWIPS	$2,265$ MWIPS	1.00
Lmbench (pipe bandwidth)	`8139too + ext3`	N/A	$2,535$ MB/s	$2,213$ MB/s	0.87
Apache Bench (throughput)	`8139too`	$56,037$	$2,261$ KB/s	$1,955$ KB/s	0.86
Kernel Decompression	`ext3`	$17,471,989$	$35,271$ ms	$44,803$ ms	0.79
Kernel Build	`ext3`	$148,823,045$	$2,804$ s	$3,106$ s	0.90

Table 3. Performance results of application-level benchmarks.

system is performing I/O intensive tasks.

The application benchmarks and their configuration are presented as follows: (1) Dhrystone 2 of the Unix Bench suite [8] using register variables. (2) Double-Precision Whetstone of the Unix Bench. (3) LmBench [6] pipe bandwidth measuring the performance of IPC interface provided by the kernel. (4) Kernel Decompression by extracting a Linux 2.6.24 kernel gzipped tarball using `tar -xzf` command. (5) Building a 2.6.24 Linux kernel using default configurations. (6) Apache Bench configured to have 5 concurrent clients issuing 20 http requests (16KB HTML) per client.

Table 3 presents the results of these application level benchmarks. The second column indicates which extension is labeled as untrusted, while the third column shows the total number of protection state transitions in each workload. Some numbers are not available because the corresponding workload is part of a continuous benchmark suite. From the results, we can see that the performance of HUKO system is from 0.79 to 1.00 of the baseline. We also found that the performance overhead added-on by HUKO largely depends on the frequency of control transfers between untrusted extensions and the kernel. Hence, if the workload is CPU-bound, the performance cost is minimal. The overhead gets higher only when an untrusted extension is responsible for highly frequent operations such as disk I/O. In the kernel decompression experiment, the protection state transfer rate reaches about 39,0000 per second, which renders HUKO the worst case of performance: 0.79 of the baseline.

Besides application level benchmarks, we also performed several micro-benchmark tests on process creation with Lmbench. We labeled `ext3` and `8139too` as untrusted extensions in our system protected by HUKO. Regarding the test item `process fork + exit`, it took HUKO system 100.31 μs to complete the operation while the native system took 92.87 μs. For `process fork + execve`, HUKO system spent 377.47 μs compared to the native time of 296.47 μs. For `process fork + /bin/sh -c`, it took HUKO system 884.57 μs compared to the native time of 697.38 μs.

6 Limitations and Future Work

We believe that HUKO provides a transparent security layer which greatly enhances the integrity protection for commodity operating system kernels. Nonetheless, it also has limitations in defending against certain security threats. In the following, we discuss these limitations and possible solutions as our future direction.

Kernel APIs. In HUKO system, controls from untrusted extensions to the OS kernel are restricted to a set of trusted entry points, which are essentially legitimate kernel APIs that exported to kernel extensions. However, in commodity operating systems, the kernel is usually not designed to tolerate or defend against malicious extensions, which may results in the lack of robustness and security of kernel APIs. Moreover, programming languages used to build commodity OS kernels security do not support features like type enforcement. For these reason, it is possible that attackers can exploit the "legitimate" kernel interface to subvert the integrity of kernel. Examples of such attacks include: (1) calling legitimate kernel APIs with undesired object reference to compromise kernel objects, (2) abuse of privileges, (e.g., video cam driver accesses kernel APIs for the networking stack), and (3) exploiting memory and type bugs of the kernel API functions. Comprehensively addressing these issues would require major design improvements on specific kernel (e.g., [29, 16, 20]), such as kernel object model, access control model, type enforcement, verification and privilege separation. In addition, these approaches can be layered atop HUKO, which serves as a VMM-level reference monitor for mediating kernel object access, checking API calls and their parameters.

To obtain a better mandatory security policy, we are looking for a deeper understanding of the behavior of the OS kernel. In specific, we are interested in figuring out security-sensitive kernel data along the execution path of each TEP. This could be achieved by static program analysis with security annotations. Based on the properties such as privilege, availability level and resource category of these kernel data, we could achieve a good classification of TEPs in terms of resource manipulation and privilege. In this way, the security and resource semantics of TEPs are further revealed, which could help improve the security of TEPs whose privileges are originally unified in commodity OSes.

Information flow. Another category of possible attacks is through explicit and implicit information flow. For instance, OS kernel may explicitly grant write access to extensions on its own data objects (e.g., via shared memory, API or messages), on the other hand, extensions may write low integrity data to some places where kernel may read afterwards. Both situations violate the traditional integrity model. It is known that there is no existing information flow control inside commodity OS kernels since tracking fine-grained information flow is costly in regard to current programming language and architecture. Alternatively, we plan to investigate applying end points such as input filters and verifiers between OS kernel and extensions to regulate the function parameters and information passed to the OS kernel.

7 Related Work

The idea and design of HUKO draw inspiration from a variety of topics of past research work, which include kernel integrity protection, kernel malware analysis, device driver isolation and mandatory access control models.

Kernel integrity protection. There are a number of previous research efforts aiming at protecting the integrity of the operating system kernel, such as code integrity protection [27, 25, 22], data integrity protection [10, 31] and control data/flow integrity protection [33, 23, 35]. Secvisor [27] is a hypervisor based protection system which guarantees the life-time code integrity of the kernel. It leverages advanced features from AMD processors, which are analogous to those used in HUKO. HUKO differs from Secvisor in the following aspects: Firstly, Secvisor is intended to prohibit any untrusted code executing in the kernel space, while HUKO does allow untrusted kernel extensions running securely to provide functionality and availability. Thus HUKO needs to enforce additional protection such as data integrity and control flow integrity to restrict the behavior of untrusted extensions. Secondly, Secvisor's tiny hypervisor design renders the system a very small TCB, which grants the system a more secure foundation which is easier to be verified. In comparison, HUKO is based on Xen hypervisor with a larger TCB, yet it saves deployment and configuration effort for existing Xen virtual machines.

Kernel malware analysis. Several recent projects such as Panorama [39], K-Tracer [9], HookFinder [18], HookMap [34], and Poker [26] focus on analyzing the behavior of kernel-level malwares. These research work are complementary to HUKO protection system because they provide extensive knowledge of how malwares damage the integrity of the kernel. These knowledge would further help HUKO to enforce more effective access control policies on various kinds of kernel objects to offer comprehensive protection.

Device driver isolation. Another major category of related research work is on isolating buggy device drivers to improve the reliability of operating systems. Examples of these systems include Nooks [32], MINIX 3 [19], and SafeDrive [41]. Such systems are mainly targeted for fault resistance and dependability, and they could effectively prevent system crashes caused by design defects and programming mistakes of device drivers. These approaches are complementary to HUKO in enhancing the robustness and availability of OS kernels. Our system resembles Nooks since both approaches establish hardware-enforced protection domains to isolate kernel components. However, by the time Nooks was designed, there was no supporting hardware features such like NX bits, EPT, VPID, IOMMU, etc. By leveraging these advanced features, HUKO significantly reduces the amount of OS modifications and has a better performance. Also, HUKO offers more protection from malicious extensions, e.g., it preserves architectural state from being modified by untrusted extensions. As a VMM-based approach, HUKO has a smaller TCB and attack surface compared with OS-based approaches. Language-based approaches such like SafeDrive provide type enforcement and prevent memory errors, though they often require the source code of extension for recompilation, which limits their applicability for binary drivers. In contrast, HUKO can support unmodified legacy extensions.

Mandatory access control. HUKO enforces mandatory access control policies over subjects and objects in the OS kernel. There are many systems that are designed for improving operating system security by adding mandatory access control, e.g., LOMAC [17], SELinux [7], AppArmor [2], UMIP [21] and Loki [40]. These systems provide flexible, powerful and fine-grained protection to preserve system-level integrity. However, they are all enforcing MAC at the OS abstraction level and cannot be applied to mediate the activities of kernel-level objects.

Address space separation. As part of our design, HUKO isolates untrusted extensions from the OS kernel using the memory virtualization mechanism provided by VMMs. There are also a number of systems achieving different research goals using various techniques that isolate two entities which previously belong to the same address space. MMP [36, 37] achieves address space isolation and fine-grained permission mapping by extending the hardware architecture. XFI [15] provides permission management within system address spaces using binary rewriting. NativeClient [38] offers sandboxing and isolation to native x86 modules by leveraging x86 segmentation and code validation. SIM [30] proposes a secure In-VM monitoring approach which places the kernel-level monitor in a protected address space using shadow paging. Overshadow [13] and Bastion [12] leverages multiple shadow tables to protect application data from the rest of the system. In comparison, HUKO focuses on protecting the integrity of the OS kernel. Also HUKO is based on hardware-assisted paging rather than software-based shadow paging mechanism to reduce the number of VMEXITs and improve the TLB performance.

8 Conclusion

We have presented the design, implementation and evaluation of HUKO, a hypervisor-based layered system that comprehensively protects the integrity of commodity OS kernels from untrusted extensions. HUKO leverages several contemporary hardware virtualization techniques as well as its novel software design to achieve its design principles: multi-aspect protection, acceptable performance and ease-of-adoption. Our experiments show that HUKO can effectively protect the kernel integrity from various kinds of malicious extensions with an acceptable performance overhead. We believe that HUKO provides a practical framework for running untrusted extensions in OS kernel with enhanced integrity protection for commodity systems.

Acknowledgements

We would like to thank our paper shepherd Mihai Christodorescu and the anonymous reviewers, for their helpful comments on earlier versions of this paper.

This work was supported by AFOSR FA9550-07-1-0527 (MURI), NSF CNS-0905131, ARO W911NF-09-1-0525 (MURI), and AFRL FA8750-08-C-0137.

References

[1] All-root. http://packetstormsecurity.org/UNIX/penetration /rootkits/all-root.c.

[2] Apparmor. http://www.novell.com/linux/security/apparmor/.

[3] Enyelkm. http://www.packetstormsecurity.com/UNIX/penetration /rootkits/enyelkm-1.3-no-objs.tar.gz.

[4] Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide. http://www.intel.com/Assets/PDF/manual/253669.pdf.

[5] Intel virtualization technology for directed i/o. ftp://download.intel.com/technology/computing /vptech/Intel(r)_VT_for_Direct_IO.pdf.

[6] Lmbench. http://www.bitmover.com/lmbench/.

[7] Nsa. security enhanced linux. http://www.nsa.gov/selinux/.

[8] Unixbench. http://ftp.tux.org/pub/benchmarks/System/unixbench/.

[9] M. S. Andrea Lanzi and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *Network and Distributed System Security Symposium*, 2009.

[10] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society.

[11] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 555–565, New York, NY, USA, 2009. ACM.

[12] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Bangalore, India, Jan 9-14 2010.

[13] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, New York, NY, USA, 2008. ACM.

[14] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 566–577, New York, NY, USA, 2009. ACM.

[15] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: software guards for system address spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[16] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 177–190, New York, NY, USA, 2006. ACM.

[17] T. Fraser. Lomac: Low water-mark integrity protection for cots environments. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 230, Washington, DC, USA, 2000. IEEE Computer Society.

[18] Z. L. Heng Yin and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[19] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum. Fault isolation for device drivers. In *IEEE/IFIP International Conference on Dependable Systems and Networks, 2009. DSN '09.*, 2009.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220, New York, NY, USA, 2009. ACM.

[21] N. Li, Z. Mao, and H. Chen. Usable mandatory integrity protection for operating systems. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 164–178, Washington, DC, USA, 2007. IEEE Computer Society.

[22] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *SS'08: Proceedings of the 17th USENIX Security Symposium*, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association.

[23] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 103–115, New York, NY, USA, 2007. ACM.

[24] T. H. R. Hund and F. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Security'09: Proceedings of the 18th USENIX Security Symposium*, 2009.

[25] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID '08: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] R. Riley, X. Jiang, and D. Xu. Multi-aspect profiling of kernel rootkit behavior. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer systems*, pages 47–60, New York, NY, USA, 2009. ACM.

[27] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 335–350, New York, NY, USA, 2007. ACM.

[28] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561, New York, NY, USA, 2007. ACM.

[29] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[30] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 477–487, New York, NY, USA, 2009. ACM.

[31] A. Srivastava, I. Erete, and J. Giffin. Kernel data integrity protection via memory access control. Technical Report GT-CS-09-04, Georgia Institute of Technology, 2009.

[32] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 207–222, New York, NY, USA, 2003. ACM.

[33] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 545–554, New York, NY, USA, 2009. ACM.

[34] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *RAID '08: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, pages 21–38, Berlin, Heidelberg, 2008. Springer-Verlag.

[35] J. Wei, B. D. Payne, J. Giffin, and C. Pu. Soft-timer driven transient kernel control flow attacks and defense. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 97–107, Washington, DC, USA, 2008. IEEE Computer Society.

[36] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 304–316, New York, NY, USA, 2002. ACM.

[37] E. Witchel, J. Rhee, and K. Asanović. Mondrix: memory isolation for linux using mondriaan memory protection. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2005. ACM.

[38] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Security and Privacy, IEEE Symposium on*, 0:79–93, 2009.

[39] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 116–127, New York, NY, USA, 2007. ACM.

[40] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI 2008*, pages 225–240. USENIX Association, 2008.

[41] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.