InkTag: Secure Applications on an Untrusted Operating System

Owen S. Hofmann Sangman Kim Alan M. Dunn Michael Z. Lee Emmett Witchel

The University of Texas at Austin

{osh, sangmank, adunn, mzlee, witchel}@cs.utexas.edu

Abstract

InkTag is a virtualization-based architecture that gives strong safety guarantees to high-assurance processes even in the presence of a malicious operating system. InkTag advances the state of the art in untrusted operating systems in both the design of its hypervisor and in the ability to run useful applications without trusting the operating system. We introduce *paraverification*, a technique that simplifies the InkTag hypervisor by forcing the untrusted operating system to participate in its own verification. *Attribute-based access control* allows trusted applications to create decentralized access control policies. InkTag is also the first system of its kind to ensure consistency between secure data and metadata, ensuring recoverability in the face of system crashes.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Access controls, Invasive software

General Terms Security, Verification

Keywords Application protection, Virtualization-based security, Paraverification

1. Introduction

Operating systems are a vexing Achilles heel in the security architecture of modern computing systems. The OS is the root of trust, so compromising the OS compromises *every* program on the system. On discretionary access control operating systems like Linux and Windows, controlling any process running as root (administrator) is a kernel compromise because the root user can load code and data into the kernel's address space. If an application could remain safe even if the operating system were compromised, then operating system exploits would no longer have the security emergency status that they have today.

This paper introduces InkTag, a system in which secure, trustworthy programs can **efficiently verify an untrusted, commodity operating system's behavior**, with a small degree of assistance from a small, trusted hypervisor. OS implementations are complex. However, verifying OS behavior is possible without reimplementing OS subsystems in the hypervisor, because OS services often have simple specifications. OS complexity comes from supporting these simple services simultaneously for many different processes. Global behavior and resource management is much more complicated than the specification for an individual process. For instance, swapping and copy-on-write heuristics in Linux require many thousands of lines of code, but auditing an application's page tables and checksumming the page contents requires only a few hundred. Verifying that the OS provides system services correctly allows InkTag to avoid having to reason about the OS's implementation of these services.

Though feasible, efficiently and safely verifying OS behavior remains a significant challenge. The InkTag hypervisor must implement deep introspection into architecture-level primitives, such as page tables, to isolate trusted applications from an untrusted operating system. The range of "normal" operating system behavior is large, making recognition of malicious behavior a challenge. While verifying OS behavior is hard, doing it efficiently is even harder. Modern virtualization hardware improves performance by relieving hypervisor software from having to process many common operations. Unfortunately, it is often those exact operations, e.g., page table updates, that are crucial for verifying OS behavior.

InkTag introduces *paraverification*, which enables verification of OS behavior with limited hypervisor complexity. Most previous systems have attempted to verify unmodified operating systems. InkTag requires the untrusted OS to provide information and resources to both the hypervisor and application that allow them to efficiently verify the operating system's actions. Using paraverification to force the OS to make verification easier and more efficient is similar to the way paravirtualization forces an OS to make virtualization more efficient.

Prior work on untrusted operating systems [11] has focused on simply isolating trusted code and data from the OS, with minimal support for securely using OS features. InkTag addresses important issues in the completeness and usability of untrusted operating systems, such as providing users of an untrusted OS with flexible access control and crash consistency for hypervisor and OS data structures. InkTag advances the design and implementation of OS verification in the following ways:

1. InkTag introduces paraverification, where an untrusted operating system is required to perform extra computation to make verifying its own behavior easier.
2. InkTag is the first system to provide users of an untrusted OS with flexible access control, that allows applications to define access control policies for their own secure files (files with privacy and integrity managed by InkTag). Access control is vital for sharing data between processes with different levels of privilege. Our prototype applies flexible access control to a multi-user wiki application, providing hypervisor-enforced privacy, integrity, and access control for wiki code and data.
3. InkTag is the first system to provide crash consistency between security-critical metadata managed by the hypervisor and data managed by the untrusted OS.
4. InkTag directly addresses *Iago attacks* [8], a new class of attacks against systems providing trusted applications in untrusted operating systems that manipulates the return values

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

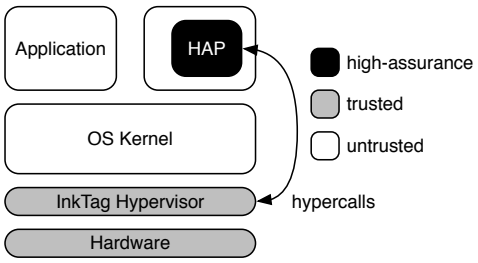

Figure 1. InkTag design overview. In InkTag high-assurance processes (HAPs) make hypercalls to the virtual machine hypervisor to verify the runtime behavior of the operating system. The hypervisor is trusted.

of system calls (e.g., `mmap`) to cause a trusted application to harm itself.

Section 2 gives an overview of InkTag, while Section 3 explains InkTag's high-level design. Section 4 introduces paraverification. Section 5 describes access control in InkTag, followed by our design for storage (§6), implementation (§7) and evaluation (§8). Section 9 covers related work and Section 10 concludes.

2. Overview

InkTag is a hypervisor-based system that protects trusted applications from an untrusted OS, allowing trusted applications to securely use untrusted OS services. The hypervisor protects application code, data, and control flow from the OS, allowing applications to execute in isolation. Mutually trusting secure applications can securely and privately share data without interference from the OS or other applications. Each secure application coordinates directly with the InkTag hypervisor via hypercalls to detect OS misbehavior.

Figure 1 shows an overview of the InkTag architecture. Trusted application code executes in a **high-assurance process, or HAP**, which is isolated from the OS. Nearly all application-level changes are contained in a small, 2000-line library (libinktag) the use of which is largely encapsulated in the standard C library. InkTag extends a standard hypervisor to monitor the untrusted OS using paravirtualized device drivers and virtualization hardware. InkTag defines new hypercalls for HAPs to verify OS behavior.

InkTag shares its basic threat model and security guarantees with previous work where a trusted hypervisor verifies an untrusted operating system's actions; such as SP^3 [47], and especially Overshadow [11]. We first discuss issues generic to all approaches, then starting with Section 2.4 discuss issues specific to InkTag.

2.1 Threat Model

InkTag assumes that the OS is completely untrusted and can behave in arbitrarily malicious ways. Applications running on the OS have both an untrusted context maintained by the OS used for requesting OS services, and a trusted context used for executing code that must be isolated from the OS. The developer trusts the InkTag hypervisor and the trusted execution context of the application. The trusted application context is isolated from the OS by the hypervisor.

InkTag does not address application-level bugs, and will not stop an application that deliberately divulges secret data (e.g., by putting it in the arguments of a system call that the OS handles).

InkTag cannot guarantee untrusted OS availability, but can detect this class of misbehavior. Trivially, a malicious OS could simply shut down every time it was started, though the InkTag hypervisor will detect such misbehavior. More subtle availability attacks are possible, such as deleting volatile data, which will be detected in a timely manner, but may result in the loss of data between misbehavior and detection.

2.2 Size of trusted computing base (TCB)

The operating system consists of millions of lines of code, so its elimination from the trusted computing base seems, *prima facie*, to increase security. However, the KVM hypervisor includes an entire OS (sometimes called a type 2 hypervisor). Eliminating trust in the guest instance of Linux is of little security value if the hypervisor contains its own instance.

However, simpler hypervisors (type 1) exist, and contain fewer lines of code than a typical operating system. Additionally, the hypervisor interface is a hardware interface, which is far simpler and easier to make secure than the hundreds of semantically complex system calls exported by a general-purpose operating system. For example, from 2010 to 2012, a search of the National Vulnerability Database [31] returns 12 exploits for Xen and 16 exploits for KVM that have an impact worse than denial of service. By contrast, there are 53 such vulnerabilities published for the Linux kernel in 2012, of which only 7 are driver vulnerabilities. These vulnerabilities spanned many different core kernel services such as memory management, file systems, network protocol implementations, and syscalls. The situation is not much better for Windows 7: in 2012, there were 9 privilege escalation and 31 remote code execution bulletins listed [1].

2.3 Security guarantees

Here we summarize the security guarantees provided to an InkTag application. The InkTag hypervisor ensures that a HAP's process context (registers) and address space are isolated from the the operating system. Then, InkTag ensures that a HAP can use a subset of services provided by the untrusted OS to interact with secure files (files with privacy and integrity managed by InkTag through encryption and hashing), and verify that those services were provided correctly. InkTag shares these basic security guarantees, as well as implementation techniques, with previous work such as Overshadow.

In addition to the majority of HAP code that executes in a trusted context, each HAP also contains a small amount of *untrusted trampoline* code that interacts with the operating system (this is similar to Overshadow's *uncloaked shim*). The InkTag hypervisor switches control between secure HAP code and the untrusted trampoline, while the untrusted operating system schedules among the untrusted trampoline and other contexts. This allows the InkTag hypervisor to control switches into and out of a secure context and ensure control flow integrity. Then, the InkTag hypervisor encrypts and hashes HAP pages to ensure privacy and integrity for the HAP's address space: this is analogous to Overshadow's *multi-shadowing* technique.

Control flow integrity As with traditional applications, a HAP running in InkTag may be interrupted at any time by the operating system. InkTag must not allow the operating system to read or modify the application's processor registers. Doing so could leak private data, or allow the operating system to modify the application's control flow or data by changing the instruction pointer, condition flags, or a register value. The InkTag hypervisor interposes on every context switch between a secure HAP and the operating system. On context switches, the hypervisor saves processor registers, and overwrites their values before switching to the OS.

Address space integrity In addition to application registers, the InkTag hypervisor must ensure privacy and integrity for code and data in a HAP's address space. When an untrusted operating system attempts to read application memory, InkTag hashes memory and encrypts it, ensuring that the untrusted operating system cannot read application secrets. When the HAP accesses the memory again, the InkTag hypervisor decrypts it and verifies the hash, to ensure that the memory was not modified by the operating system.

The position and order of pages in an application's virtual address space is also an important integrity property. InkTag ensures that every page of memory is mapped at the virtual address requested by the application, either via information about the HAP's initial state contained in the ELF binary, or via a request to a memory mapping function such as `brk()` or `mmap()`. The problem of synchronizing the mapping information between application and hypervisor motivates our primary contribution, paraverification, described in detail in Section 4.

File I/O To perform useful work, a HAP isolated from an untrusted operating system must still be able to use a subset of OS services. The InkTag hypervisor must ensure that the application can still rely on those services even when running on a malicious OS.

Most importantly, InkTag provides HAPs with the ability to securely interact with files despite the fact that they are read from and written to disk by the untrusted operating system, by guaranteeing integrity for file memory mappings. InkTag applications primarily identify files through a 64-bit *object identifier*, or OID. Most file operations are expressed as operations on OIDs, even though real applications generally expect to use string filenames: mapping string filenames to OIDs is a contribution of InkTag, as discussed later in this section. When an application maps an OID into memory (through a call to `mmap`) and receives an address for the new mapping, InkTag ensures that later references to that address will access the desired file. Privacy and integrity for file data are ensured via InkTag's guarantee of address space privacy and integrity, by hashing and encrypting in-memory file data in response to accesses by the untrusted operating system. To handle file I/O via `read()` and `write()` system calls, our application-level library translates these calls into operations on memory-mapped files.

Process control HAPs may also create new processes through calls to `fork()` and execute binaries in these processes with `exec()`. The InkTag hypervisor ensures that the untrusted operating system executes these operations correctly. In the case of `fork()`, InkTag ensures that the new HAP is a clone of its parent. For `exec()`, InkTag ensures that the new HAP, specified by the identifier of the binary file passed to `exec()` is loaded into memory correctly (i.e., each section in the binary is loaded unmodified into the correct virtual address), based on the information specified by the ELF format binary.

Other OS services Because InkTag guarantees control flow and data integrity for HAPs, a HAP may safely invoke system calls not explicitly secured by InkTag. However, it must consider the results of those system calls as it considers any data provided by an untrusted source. For example, InkTag does not manage network I/O, however it is possible for applications to safely communicate over the network via mechanisms such as transport layer security (TLS [13]), that enable secure communication over an untrusted channel.

2.4 InkTag contributions

InkTag advances work on untrusted operating systems along two axes: the underlying architecture for isolating processes from the operating system, and the set of core OS services that applications may use securely.

Paraverification Isolating an application's address space from an untrusted operating system is a daunting task. Whereas previous work has used unmodified OS kernels, InkTag employs *paraverification*, a technique similar to paravirtualization, in which the untrusted kernel is required to send to the hypervisor information about updates to process state (that the hypervisor then checks for correctness). Paraverification simplifies the design of the InkTag hypervisor by allowing it to directly use high-level information from the kernel, rather than having to deduce that information from low-level updates such as changes to bits on process page tables.

Hardware virtualization The utility of virtualization has prompted the rapid introduction of hardware support for virtualizing processor state, as well as hardware support for virtualizing memory management. Eliminating software from these performance-critical processing paths is a clear advantage, but systems like InkTag require validation of OS updates to HAP page tables. InkTag minimizes the performance impact of validation via a combination of the efficiency afforded by paraverification and a two-level approach to protection. InkTag uses hardware MMU virtualization for coarse-grained separation between secure and insecure data. Then it uses software only when needed, to manage the userspace portions of HAP page tables.

Access control and naming The InkTag hypervisor allows HAPs to specify access control policies on secure files, with privacy and integrity managed by InkTag through encryption and hashing. InkTag's access control mechanism is described in detail in Section 5. Although InkTag applications identify files via an integer OID, most applications and users expect to reference files through a string name. InkTag allows applications to map from string names to OIDs, while maintaining important integrity properties (such as the trusted nature of the `/etc` directory).

Consistency To protect the integrity of file contents, InkTag, like similar systems, must maintain additional metadata in the form of hashes of file data pages. InkTag is the first such system to provide crash consistency between file metadata and data. Consistency is vital in this setting: without consistent data and metadata, the InkTag hypervisor cannot protect file integrity. An application must either discard the inconsistent data, or accept the possibility of tampering by an untrusted OS.

2.5 API

HAPs communicate with the InkTag hypervisor primarily by making hypercalls. InkTag maintains the simplicity of the hypervisor interface by adding only 14 hypervisor calls. Table 1 summarizes InkTag's hypercall interface. It refers to several concepts that will be introduced shortly, but what is clear is that the number of calls is limited and their function is mostly intuitive.

In addition to invoking operations through hypercalls, the InkTag hypervisor shares two data structures with the guest kernel and HAPs First, an InkTag application must describe the layout of its virtual address space. A HAP enters each of its memory mappings into an array of descriptors in its virtual address space, specifying the base address of the array as part of the `INIT` hypercall. Second, the untrusted kernel sends information about updates to process state to the InkTag hypervisor. These updates are communicated through a shared queue, similar to existing paravirtual interfaces.

3. Address space management

Address space management is the foundation for InkTag's security guarantees. We discuss $\mathbb{S}$-pages, InkTag's abstraction for secure address spaces, and how InkTag uses hardware memory management virtualization features that are part of modern virtualization hardware.

3.1 Objects and secure pages

InkTag's basic file abstraction is an *object*. All files, including binary executables, are represented by an InkTag object. Objects are identified by a 64-bit *object identifier* (OID). Section 5.4 discusses translating between human-readable names and OIDs, however OIDs are the main abstraction used by the InkTag hypervisor.

Hypercall	Arguments	Description
Process control		
INIT	*control addr*	Starts a new HAP. The HAP passes to the hypervisor the address of a control structure, which specifies which binary the HAP was loaded from, and the base address of the HAP's list of virtual memory mappings.
EXEC	*new hap OID*	Start a HAP, ensuring that it is loaded from the binary identified by *OID*.
CLONE		Create a new HAP that is a duplicate of the current state of the calling HAP.
SWITCH_TO_HAP	*hap id*	Invoked by untrusted trampoline code to switch context back into a HAP.
SYSCALL	*new PC*	On any hypercall, a HAP may request a switch out of secure execution to invoke a system call by specifying a program counter value for a service routine in untrusted code. SYSCALL is used when invoking a system call is the only desired behavior.
Memory management		
UNMAP	*memory range*	Ensure that $\mathbb{S}$-pages within the specified virtual address range are unmapped.
REMAP	*old_range, new_range*	Move any $\mathbb{S}$-page mappings from *old_range* to *new_range*.
Files and access control		
ACCESS	*OID*	Check if the current HAP has access to *OID*.
CREATE	*OID, namespace*	Create a new file within the given namespace.
SET_LENGTH	*OID*	Set the length of a file for which the HAP has write permission.
OID_ACL	*OID, acl*	Set the ACL on a file.
ADD_DROP	*add_attr, drop_attr*	Add and/or drop attributes from a HAP's list of attributes.
Paraverification		
MMU_REGISTER	*queue addr*	Invoked by the untrusted kernel to specify a location in memory that contains a queue of updates to HAP address spaces (such as page table updates).
MMU_FLUSH		Invoked by the untrusted kernel to notify the InkTag hypervisor that the queue is full, and must be processed.

Table 1. The hypercall interface to the InkTag hypervisor

Throughout the rest of the paper, we use the term OID interchangeably with object.

Objects are comprised of *secure pages (*$\mathbb{S}$*-pages)*, which are the basic mechanism by which InkTag enforces address space privacy, address space integrity, and access control policy for files. $\mathbb{S}$-pages consist of a block of data (4 KB for most pages on x86 processors), in memory or on disk, with additional metadata. $\mathbb{S}$-pages include a hash of the data contained with the page, as well as information about which resource the page describes, in the form of a $\langle OID, offset \rangle$ pair. An object identifies a set of pages that share a single OID, and may refer to a file on disk or a private memory region created dynamically by an application (e.g., an anonymous mmap).

The InkTag hypervisor encrypts $\mathbb{S}$-pages to ensure privacy, and hashes them to ensure integrity. When a HAP accesses an $\mathbb{S}$-page for which it has read permission, the InkTag hypervisor transparently decrypts the page, allowing the HAP access to cleartext. If an $\mathbb{S}$-page is accessed by the operating system, a regular application, or a HAP without read permission, the InkTag hypervisor detects the access and re-encrypts the page. Even if a malicious operating system can read the data within an $\mathbb{S}$-page, InkTag guarantees privacy because the OS can read only encrypted data.

Similarly, only the hypervisor can update the hash associated with an $\mathbb{S}$-page. When a HAP updates an $\mathbb{S}$-page for which it has write permission, the InkTag hypervisor updates the hash. If the OS modifies the data in the $\mathbb{S}$-page, the InkTag hypervisor will detect the modification, because hashing the modified data will not match the recorded hash.

The untrusted OS views $\mathbb{S}$-pages as standard data pages, and remains responsible for placing $\mathbb{S}$-pages in memory and on disk. The additional metadata attached to $\mathbb{S}$-pages is transparent to the guest OS: the InkTag hypervisor updates and tracks $\mathbb{S}$-page metadata as the operating system or application moves or transforms the data such as by mapping a file in memory or by writing a page to the virtual disk.

Each HAP must provide a description of its address space to the InkTag hypervisor, in the form of a list of memory mappings $[\langle address\_range, OID, offset \rangle, \ldots]$, each of which defines a sequence of $\mathbb{S}$-pages. With a description of the address space, the hypervisor may then validate individual page table updates requested by the untrusted OS. For example, suppose the OS attempts to map virtual address V to physical frame P in a HAP's page tables. The hypervisor examines the HAP's memory map for a range that includes V. If one exists, the hypervisor then verifies that the HAP has access to the specified OID. Finally, the hypervisor checks that frame P actually contains the correct $\mathbb{S}$-page, by checking P's hash against the stored metadata. If all of these checks succeed, the hypervisor now considers P to be a physical frame containing an $\mathbb{S}$-page, and the HAP has a valid mapping of address V to P. Thus, when the HAP attempts to access the $\mathbb{S}$-page, InkTag will decrypt its contents and provide the HAP with cleartext access.

3.2 Nested paging

The InkTag hypervisor is designed to run on modern processors that support hardware assistance for virtualization. Such processors are designed to simplify the task of writing hypervisor software by automatically creating a self-contained environment for the guest OS, without requiring manual intervention by hypervisor software.

One of the primary tasks of any hypervisor (including InkTag), is virtualizing memory management. The hypervisor must ensure that the guest operating system has access only to those pages of memory that represent its virtualized physical address space. For early x86 hypervisors, this required intercepting page table updates made by the guest OS, transforming *guest-physical addresses* (physical addresses from the point of the virtualized guest) into *host-physical addresses* (actual physical addresses of the frames of memory that constitute the guest's virtualized physical memory),

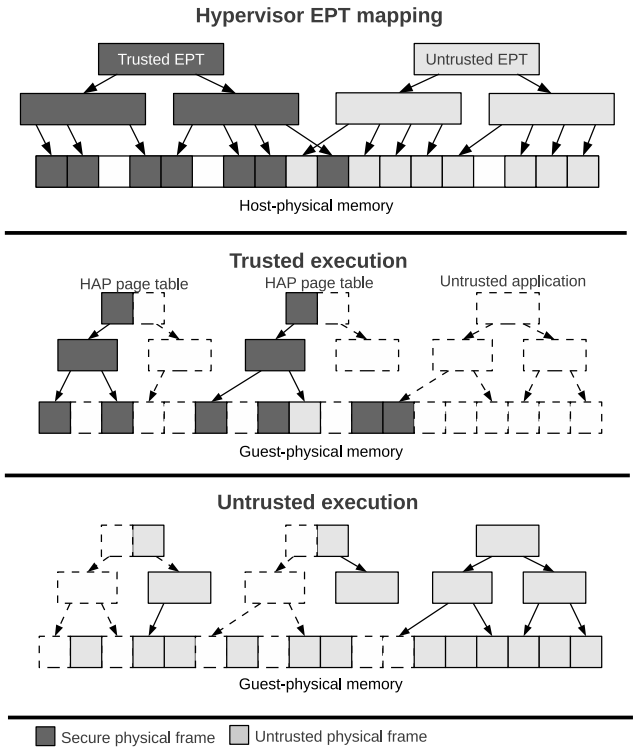

Figure 2. Address space protection using both EPT and management of HAP page tables. The InkTag hypervisor uses two EPTs to divide access between physical frames containing cleartext $\mathbb{S}$-pages, and those containing untrusted data or encrypted $\mathbb{S}$-pages. Then, it manages HAP page tables to restrict access within the set of secure frames.

and installing modified page tables for the guest OS that contain the transformed mappings.

Hardware MMU virtualization As might be expected, virtualizing memory in this manner adversely affects both hypervisor complexity and hypervisor performance. In response, more recent x86 processors have supported *nested paging*. With nested paging, guest memory accesses are translated through two separate page tables. First, guest-virtual addresses are translated into guest-physical addresses by traditional page tables managed entirely by the guest OS. Then, guest-physical addresses are translated into host-physical addresses by the *extended page table* (EPT)[1]. The EPT is managed by the hypervisor, but does not need to be updated in response to changes to guest page tables, as it only maps between guest-physical and host-physical address spaces. Meanwhile, the guest OS is free to perform arbitrary modifications to its own page tables, as all accesses will be restricted by EPTs to memory explicitly approved by the hypervisor.

Nested paging is a significant step forward for hypervisors. However, as discussed in section 3.1, the InkTag hypervisor's primary means of enforcing privacy, integrity, and access control is through detailed management of OS page table updates. A key goal for designing the InkTag hypervisor is to retain the necessary control over guest OS page table updates, while still being able to harness the performance benefits of modern virtualization hardware.

Nested isolation InkTag takes advantage of hardware MMU virtualization by using a combination of hardware EPT support and management of individual OS page table updates. Rather than use a single EPT for all of guest execution, the InkTag hypervisor uses two separate EPT trees. The *trusted EPT* is installed during isolated HAP execution, while the *untrusted EPT* is used during execution of the operating system and other applications (Figure 2). The contents of both EPTs are entirely managed by the hypervisor, and are therefore trustworthy. The trusted/untrusted label refers to the contents of the physical frames they map. The trusted EPT primarily maps physical frames that contain cleartext $\mathbb{S}$-pages, while the untrusted EPT maps all other frames, including encrypted $\mathbb{S}$-pages, data belonging to the OS, and untrusted applications.

Using separate EPTs for trusted and untrusted contexts allows InkTag to coarsely control access to secure pages. Physical frames holding cleartext $\mathbb{S}$-pages are not mapped in the untrusted EPT If the OS or an untrusted application accesses a cleartext $\mathbb{S}$-page frame, the access causes a fault that is handled by the hypervisor. InkTag hashes the contents of the frame, encrypts the frame's contents, maps it in the untrusted EPT, and unmaps it from the trusted EPT. If the trusted HAP accesses the frame again, the hypervisor decrypts the frame, verifies the contents against the hash, maps it in the trusted EPT, and unmaps it from the untrusted EPT.

In addition to the coarse access control provided by EPTs, InkTag must subdivide access to physical frames among executing HAPs. When executing in trusted mode, every HAP can potentially access any physical frame holding a cleartext $\mathbb{S}$-page. However, not every HAP should have access to all $\mathbb{S}$-pages. The InkTag hypervisor restricts access for an individual HAP to a subset of physical frames by managing OS page table updates for the HAP address space. Importantly, the InkTag hypervisor is only required to manage HAP page tables, and only for the part of the address space accessible in user mode (the lower half of the address space in the x86-64 architecture). All other page tables (including the kernel address space for HAPs) can be managed by the OS without hypervisor intervention.

By combining the access control for physical frames provided by EPT with management of guest page tables only when necessary, InkTag isolates HAPs from an untrusted operating system while still taking advantage of modern virtualization hardware.

4. Paraverification

The task of managing $\mathbb{S}$-pages requires that the InkTag hypervisor have deep visibility into low-level OS operations, such as updating page tables. This kind of detailed introspection introduces complexity into the hypervisor that can impede efforts to reason about its correctness. In addition, interposing on low-level operations harms performance with needless traps into the hypervisor.

Previous systems have attempted to remove trust from the operating system in a way that is largely transparent to *both the application and operating system*. This section highlights the significant challenges to application security and system performance presented by this approach.

4.1 Verification challenges

InkTag creates a secure address space for HAPs by managing only the user mode portions of HAP page tables, as described in Section 3.2. Here we explain all the steps necessary for the InkTag hypervisor to detect and interpret a page table update. InkTag must intercept low-level page table updates (*"Set page table entry at address A to x."*), determine their high-level effects (*"Map physical frame P at virtual address V."*), and compare those effects against the address space specified by an application (*"The application wants to map the $\mathbb{S}$-page S at address V, do the contents of the physical frame have the same hash as S?"*).

[1] EPT is the terminology used for nested paging on Intel processors. Although InkTag was designed for Intel processors, we believe the design to be equally applicable to AMD processors.

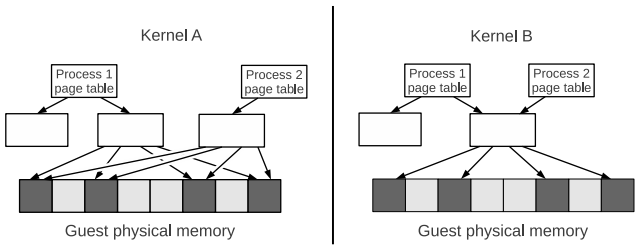

Figure 3. Two kernels mapping the address space of process that share a file. Both are non-malicious, but kernel B confounds efficient verification efforts by sharing page tables between processes.

Interpreting low-level updates Consider the task of placing a memory mapping into an application's page tables. The InkTag hypervisor can protect page table memory so the OS faults into the hypervisor when it attempts to write the page table. However, in order to determine the OS's intent, the hypervisor must then unprotect page table memory, wait for the OS to update the page table, and then retroactively determine what state changed by examining a significant amount of context information, such as saved backups of previous page table state and the role of the modified page in the application's page tables.

To efficiently interpret low-level page table updates, InkTag maintains state that allows it to determine high-level effects without reconstructing the entire address space on each update. For example, InkTag remembers which physical frames are used in application page tables, their position in the page table hierarchy, and the range of virtual addresses that they might map. Then, when the kernel sets a single entry at the leaf of the page table tree, InkTag can determine which virtual address is affected without tracing the page table from its root.

Maintaining state to efficiently verify page table updates requires InkTag to make basic assumptions about the structure of page tables. Recording the address mapped by each page of a page table requires that page tables are arranged in trees, and that separate page tables do not share any pages. A significant challenge for InkTag is that it is possible for an operating system to violate these assumptions, while still correctly managing an application's address space.

Figure 3 shows two applications that both map the same 2MB region of the same file, with both mappings aligned on a 2MB boundary (2MB is the range of virtual addresses mapped by a single leaf of a page table in the x86-64 architecture, which maps 512 4KB pages). An operating system could share a page between the page tables of both applications, while still correctly mapping each application's address space.

Even with an operating system that is both non-malicious and respects assumptions about page table structure, it is possible for the order in which the hypervisor receives updates to create the appearance of malicious or non-standard behavior. The technique of trapping writes to page tables, allowing the OS to make updates, then later examining the (potentially multiple) modified entries means that the hypervisor may not perceive updates in the same order as they were performed by the guest OS. Suppose the OS deallocates a page of file data, and then reuses that page as a page table for a different process. In this order, these updates may be benign. In the other order, it appears that the OS is allowing one application access to another's page tables, a likely violation of address space integrity.

Determining application intent Once a low-level page table update has been interpreted as an operation on a HAP's virtual address space, InkTag must determine if the mapping installed by the operating system is consistent with the application's operations on its address space. The application itself records its address space operations by making a hypercalls for all such operations, such as the `mmap()` system call. The hypervisor must communicate with the application to synchronize this information, and should do so with low overhead. Page faults can be a performance-critical operation, and mechanisms exist in the Linux kernel to quickly query the contents of the address space on a page fault, including balanced trees and caches of recently faulted areas. InkTag should handle faults without significant additional overhead, and also without simply duplicating these performance-oriented structures in both the hypervisor and application.

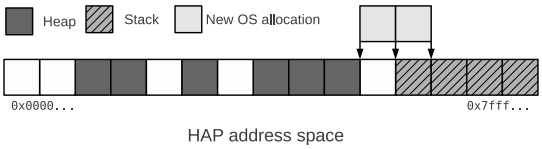

Figure 4. An Iago attack. An application relying on the OS to allocate its address space may be subverted by a malicious OS, if the OS allocates memory regions that are not disjoint.

Protecting applications from OS duplicity Although InkTag can isolate a HAP from the operating system, the application must still interact with the OS in order to use essential services, such as opening files and mapping its address space. Traditionally, through system calls such as `mmap()`, applications allow the operating system to determine where in their address space to map resources. Although `mmap()` allows an application to specify a fixed address for a mapping, this feature is seldom used by application code. Applications' reliance on OS allocation of the address space opens the door to *Iago attacks* [8], a class of attacks against systems with untrusted operating systems.

Iago attacks exploit the fact that existing applications and libraries, most importantly the standard C library, do not expect a malicious operating system. They do not verify that a virtual address returned by the OS in response to `mmap()` corresponds to an existing mapping in the application address space. For example, an application expects to run with its heap and stack in disjoint regions of its virtual address space. If the application requests a new memory mapping, the operating system could return an address that overlaps the application's stack. Writes to the new mapping will overwrite portions of the stack, introducing a vector to a traditional return-to-libc or return-oriented programming attack [43].

4.2 Paraverification

We introduce a new technique called **paraverification** to simplify the hypervisor by requiring the untrusted operating system to participate in verifying its own behavior. Paraverification helps InkTag efficiently address the challenges of verifying address space integrity, drawing inspiration from commonly-used *paravirtualization* techniques [5], which improve performance when an OS is run in a virtual machine. Both paraverification and paravirtualization work by having the OS communicate a high-level description of its intent directly to the hypervisor. Indeed, our paraverification implementation uses the Linux kernel's paravirtualization interface. Before modifying a process's page tables in the example above, the OS must first make a hypercall to correlate the page table update with a high-level application request. The kernel's paravirtualization interface includes a natural hook for this hypercall.

Although the guest operating system participates in verification, it safely remains untrusted because the hypervisor protects resources that it does not trust the OS to modify. Rather than protecting application page tables, detecting faults from the untrusted

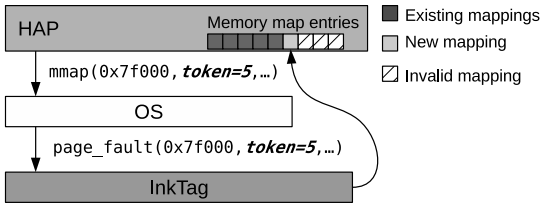

Figure 5. Paraverified isolation. A HAP maintains a list of memory mappings in its secure address space, providing the untrusted OS indices into the list. The untrusted OS must pass the same index to the InkTag hypervisor in order to handle page faults.

OS, and trying to re-verify address space integrity, the InkTag hypervisor protects application page tables and then considers any access to be malicious. The OS cannot update the tables directly, it must use the paravirtual interface, and the hypervisor will respond to unexpected accesses by taking corrective action (such as killing the OS).

4.3 Paraverified isolation

InkTag isolates a HAP's address space using paraverified operations on secure pages. As described in section 4.1, InkTag must validate OS page table updates to ensure that HAP virtual addresses map the correct, unmodified $\mathbb{S}$-page. To do so, the untrusted operating system must pair each page table update with data proving that the update reflects the application's intent.

When a HAP maps a region of memory to a file, it provides the untrusted OS kernel a secure *token* that describes the mapping. The token is an unforgeable statement from the application to the hypervisor that fully describes the requested mapping. One possibility for a token would be an HMAC on a description of the desired mapping, using a secret key shared between the HAP and the hypervisor. InkTag does not use an HMAC, but a simple integer, which we now explain.

Because InkTag isolates a HAP from the operating system and must manage HAP page tables, it can optimize the communication of tokens from application to the untrusted OS to the hypervisor. All InkTag HAPs maintain a list of the mappings that make up their address space, in the form of a list with nodes allocated from a single array at a known virtual address in the application's address space. The untrusted OS cannot forge or modify entries in the array, as it does not have access to the HAP's address space. Because the hypervisor intercepts all page table updates for the HAP, it can trivially keep a small translation lookaside buffer for just the virtual addresses that map the array of nodes. In InkTag, a token to describe a memory mapping consists of a simple integer index into its list of maps.

On initialization, a HAP invokes a hypercall to inform the hypervisor about the base and limit of its mapping list. When the HAP creates a new memory mapping, it allocates a new entry from its array of nodes, initializes it with information about the mapping: the address range, OID and offset the HAP intends to map, as well as a marker to indicate that this entry is now valid. The HAP then sends the index of the entry to the untrusted OS as a token. When the OS incurs a page fault, it uses its existing structures for indexing memory mappings (already in service to handle the page fault) to locate the token and sends it to the hypervisor along with the remaining information describing the page table update: the address of the page table entry, the updated page table value, and the affected virtual address.

Upon receiving the page table update and token, the hypervisor ensures the token describes a valid index in the HAP's array. If so, it uses its lookaside buffer to translate the virtual address and retrieves the mapping information. If the described address range matches the fault, the hypervisor uses the provided object and offset information to verify the contents of the newly mapped physical frame. If the address range does not match the fault, the index is not contained within the HAP's array, or the index does not specify a valid entry, the hypervisor will not install the new mapping. In the rare event that the virtual address corresponding to the entry is not mapped, the hypervisor does not install the mapping and injects a page fault into the application when it is next scheduled. The page fault, if correctly handled by the untrusted OS, will cause the hypervisor to refill its lookaside buffer, the application retries the original access and faults again, and the hypervisor may now access the entry for the token.

Paraverification for HAP address spaces significantly reduces the complexity of the InkTag hypervisor. Significant OS code is dedicated to efficiently looking up memory ranges during memory management. Without paraverification, InkTag must duplicate this code so that it may efficiently respond to changes in the HAP's page tables. Instead, InkTag leverages the existing OS index structures by requiring that the OS look up the relevant token for a new mapping.

4.4 Verification of address space invariants

An Iago attack subverts application security by violating invariants that the application assumes are true about its address space: that mappings returned by the operating system do not overlap. However, an untrusted OS may violate this invariant at any time. In response, a HAP could take on the responsibility of allocating regions of its address space, only requesting new mappings at fixed addresses, and not accepting any variation from the untrusted OS. However, we wish to avoid importing significant OS functionality into either the hypervisor or application. Alternately, a HAP may verify that each mapping allocated on its behalf by the OS respects necessary invariants. We take this approach with InkTag, while shifting the burden of proving that mappings respect invariants from the HAP to the untrusted operating system.

InkTag HAPs use an array of descriptors to enumerate the contents of their address space. They maintain a linked list of entries, sorted in address order, with integer indices serving as previous and next pointers. When a HAP requests a new mapping from the OS, in addition to returning the newly allocated address, the OS also must return a token to the application: the index of the application's entry in its list of maps that is immediately previous to the new address allocated by the untrusted OS. As a result, the HAP can trivially both validate that the new map does not overlap any existing maps, and insert it into the list in the proper location, without needing to maintain its own sorting structures.

As with page table updates, paraverification for address space invariants allows applications to defend against a duplicitous operating system, while relying on existing indexing structures within the untrusted OS to perform most verification tasks.

5. Access control

Isolation and address space integrity provide the building blocks for secure HAP execution under an untrusted operating system. However, real systems require usable mechanisms for securely sharing data. InkTag is the first system to provide access control under an untrusted OS.

Access control mechanisms in InkTag should meet the following criteria:

- **Efficiency.** Ultimately, the hypervisor will be responsible for enforcing access control, and must do so on performance-critical events, such as updating page tables. In addition, we wish to avoid bloating the trusted computing base by requiring the hypervisor to evaluate complex policy decisions. A good

access control mechanism will have a simple and efficient hypervisor implementation.

- **Familiarity.** A wide spectrum of access control mechanisms exist, both in the literature and in practice. However, most systems still rely on, and are well served by, users and groups. InkTag's access control mechanism should map easily onto these familiar primitives.
- **Flexibility.** Although users and groups will ease adoption, we also believe that the security-critical applications that InkTag is designed to support will benefit from the ability to create custom, descriptive access control policies. A shared, omnipotent user such as the Unix "root" creates similar security problems to a shared operating system, so users of InkTag should be able to create policies for their data without the blessing of a system administrator.

Unlike traditional access control systems that define principals (such as users in Unix), InkTag allows new principals, or even new types of principals, to be defined in a decentralized way. For example, an InkTag system might implement decentralized groups, in which any set of users can create a new group and agree to a group administrator. Similarly, an InkTag system can implement decentralized user login: each user must only trust her own personal login program, not a special system binary (§5.3).

5.1 Attributes for access control

Access control in InkTag is based on *attributes*. An attribute is a string, such as $.user.alice$ or $.group.prof$. Each HAP in InkTag carries a list of attributes that is inherited across events such as `fork()` and `exec()`, similar to the way in which Unix processes have an effective user or group id. Each OID has an access control list that specifies the attributes that must be carried by a HAP for that HAP to access the object. OID access control lists are divided into three access modes: read, write, and modify. Each access mode specifies an *attribute formula*, a logical formula that must evaluate to true for a HAP to access the object.

The read and write access modes of an OID specify formulas that must evaluate to true for a principal to read or write the object. For example, $W = .user.alice$ is a simple formula that evaluates to true (allowing write access) if a HAP has the $.user.alice$ attribute. $R = (.user.alice) \vee (.group.prof)$ is a formula that evaluates to true (allowing read access) if a HAP has either the $.user.alice$ or the $.group.prof$ attribute. The modify access mode allows a HAP to modify the attribute formula for any of the access modes in an OID's access control list.

In addition to access control lists on OIDs, attributes themselves also specify access control lists, albeit with only two modes: add and modify. The add access mode allows a HAP that satisfies the associated attribute formula to add the attribute to its own list. Access control lists on attributes are the mechanism by which HAPs in InkTag can take on the role of different principals, analogous to a `setuid` binary in Unix that allows a user to access resources owned by another principal. For example, the $.user.alice$ attribute might have the access mode $A = .apps.login$, which means that any HAP that has the $.apps.login$ attribute can add the $.user.alice$ attribute. A HAP can drop an attribute it owns at any time via a hypercall. As with files, the modify access mode for an attribute allows a HAP to change an attribute's access control list. Modifying attributes is how InkTag HAPs manage their access control credentials.

All attribute formulas are expressed in disjunctive normal form (DNF) without any negations. This makes attribute formulas simple to evaluate in the hypervisor, and also easy for people to understand. Checking attribute formulas requires only 207 lines of code in the InkTag hypervisor.

5.2 Decentralized access control

An important goal for InkTag's attribute system is *decentralization*. An InkTag user should be able to define new principals and policies to control access to her files. Decentralized access control allows high-assurance, multi-user services to define their own access control policies, enforced by the hypervisor, without relying on a system administrator.

InkTag decentralizes attributes with *hierarchically named attributes*. Attributes are named hierarchically, as a list of components separated with a '.' character. If a HAP has attribute X, then the HAP may create new attributes named $X.Y$ for any Y. For example, if user `alice` is represented using the attribute $.user.alice$, she might create the attribute $.user.alice.photos$ for her photo-sharing program. This attribute could be used both to restrict photo access to specific authorized programs, as well as to ensure that a photo-sharing program does not access unrelated files, by running it with only the attribute $.user.alice.photos$, and not the parent attribute $.user.alice$.

5.3 Login and decentralized login

User login provides an instructive case study for access control. First, consider a system that wants to have a single trusted login program. The system must provide a trusted path from that binary to a shell that runs with a user's attribute (e.g., $.user.alice$).

The InkTag hypervisor starts every HAP with a special attribute $.bin.\langle \mathsf{oid} \rangle$, where $\langle \mathsf{oid} \rangle$ is the string representation of the file's OID. The system administrator defines the $.apps.login$ attribute with an add access mode $A = .bin.\langle \mathsf{login\ oid} \rangle$ where $\langle \mathsf{login\ oid} \rangle$ identifies the login binary. When the login binary starts, it makes a hypercall to obtain the $.apps.login$ attribute. The $.user.alice$ attribute has an add access mode $A = .apps.login$, allowing the login program to add the $.user.alice$ attribute when presented with the proper credential (such as a password). Users trust the login program to drop the $.apps.login$ attribute, so once Alice has control, the HAP can no longer change users. The login program then `exec`s Alice's shell, which runs with the $.user.alice$ attribute.

With centralized login, all users trust the login binary (and its administrator). With decentralized login, a user need only trust her own login binary, though she still must obtain permission for her user name from an administrator (after all, it is the login process being decentralized, not the add user operation). The user administrator (whoever can run programs that have the $.user$ attribute) establishes the $.user.alice$ attribute with an add access mode $A = .apps.login.alice$. Alice compiles her login program, informs the login administrator (who can run programs that have the $.apps.login$ attribute), to create the $.apps.login.alice$ attribute with add access mode $A = .bin.\langle \mathsf{alice\ login\ oid} \rangle$.

To log in, Alice executes her login program, which is given the attribute $.bin.\langle \mathsf{alice\ login\ oid} \rangle$ by the hypervisor. The login program obtains the $.apps.login.alice$ attribute, dropping the attribute $.bin.\langle \mathsf{alice\ login\ oid} \rangle$. It then checks Alice's credential (e.g., password), and if valid, obtains the $.user.alice$ attribute, dropping the attribute $.apps.login.alice$, and starting a shell.

With decentralized login, a user can completely control how they log in, using whatever credentials and process they desire. There is no single login binary that serves as a target for malicious attacks. A compromised decentralized login binary gives the attacker credentials only for the compromised user. In practice, systems would likely provide a default login binary for users who do not want to create their own.

As part of future work, we plan on investigating the security implications of InkTag's attribute-based access control.

5.4 Naming and integrity

InkTag does not currently support sets of OIDS (i.e., directories). However, the hierarchical layout of traditional file systems does convey an important property that is essential for application security: file integrity. Consider the standard Unix `/etc` directory. Applications rely on the property that only trusted system administrators can create or modify files in `/etc`, because those configuration files can dramatically change application behavior. InkTag must provide some mechanism to convey similar security-essential information.

InkTag provides integrity guarantees for files with specialized attributes called *namespaces*. Namespaces are strings created hierarchically, as attributes, and have access control lists that allow a HAP to add the namespace to its list of attributes. Although we consider attributes for access control and namespaces to be conceptually distinct, they are functionally identical.

Namespaces convey integrity information by acting as gatekeepers to file creation. When an application creates a new file, the InkTag hypervisor must assign the file an OID. Each OID is generated from two components: a namespace and an arbitrary string, similarly to the way in which a file is created within a directory, with a given file name. To generate an OID, the application must carry the desired namespace in its list of attributes. The hypervisor hashes the two components, and uses the result as the new OID. Any HAP that later accesses the file knows that it was created by a HAP that carried the associated namespace.

Note that namespaces do not restrict file access—a HAP may open a file created within a namespace regardless of whether it carries that namespace in its list. Consider a configuration directory similar to `/etc`. In InkTag, there would exist a $.ns.etc$ namespace, with the add access mode $A = .group.sysadmin$. A HAP run by a system administrator (member of the sysadmin group, thus carrying the $.group.sysadmin$ attribute) may create a file by adding the $.ns.etc$ attribute, specifying a name (e.g. "passwd"), and passing both components to the hypervisor, which permits the OID's creation.

A HAP opening the file generates the OID by hashing $.ns.etc$ and "passwd," though it is not required to carry the $.ns.etc$ namespace, only to know of its existence. The HAP can trust that the contents of the file were generated by a system administrator, because only a member of of the sysadmin group could create the file initially (as checked by the hypervisor), and it trusts such principals to correctly manage access control for files they create.

6. Storage and consistency

InkTag stores secure page metadata in memory for any secure pages whose data segments also reside in memory. The untrusted operating system is responsible for placing the data segments of secure pages on the virtual disk. For secure pages to be durable, InkTag must also store secure metadata: the OID and offset corresponding to each block of data, its hash, and the encryption initialization vector (IV) necessary to decrypt the data. This section addresses a number of practical challenges in persistently and transparently storing $\mathbb{S}$-page metadata, including addressing consistency between OS and InkTag storage.

Data layout InkTag must synchronize updates to $\mathbb{S}$-pages and metadata, and should store secure metadata efficiently. When the OS issues a read request for secure page data, the hypervisor should not require significant additional lookup work in order to also read in secure page metadata. Also, storing secure metadata should not confound OS disk scheduling by adding disk seeks to store or retrieve secure metadata.

InkTag addresses these goals by interspersing storage of secure page data and metadata on the physical disk, then presenting the data storage to the untrusted OS as a contiguous virtual disk without the sectors employed to store secure metadata. The size of the media, as seen by the untrusted OS, is smaller than the size of the physical drive or backing file. When reading or writing a secure page, the secure InkTag metadata will always reside in the closest metadata storage block, causing limited performance overheads.

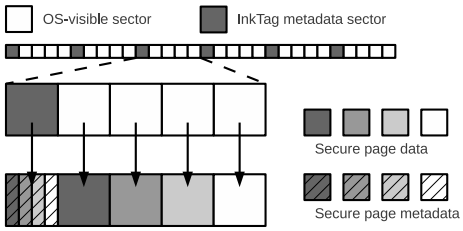

Figure 6. InkTag disk layout. Data and metadata are interleaved to optimize disk scheduling.

Synchronizing storage of secure data and metadata InkTag employs paraverification techniques to properly synchronize storage of secure data and metadata. Before the untrusted OS may write out a page via the virtual disk, it must notify the InkTag hypervisor. If the physical frame being written contains data for an $\mathbb{S}$-page, InkTag ensures that the page is encrypted, and passes the relevant metadata to the backend driver for the virtual disk. The backend consumes this metadata while writing pages to the physical disk, placing each piece of secure metadata in the metadata block closest to disk block containing the data for the $\mathbb{S}$-page.

Providing guarantees on data availability Although InkTag is unable to provide availability guarantees in many cases, the hypervisor can enforce OS deadlines for writing out dirty data. For example, if a reasonable upper bound for dirty data residing in the OS is 30 seconds, InkTag may suspect OS malfeasance if it has not detected a write of a particular secure page after 45 seconds. Although the hypervisor may not be able to retrieve $\mathbb{S}$-pages in memory that the OS has simply erased, it can prevent applications from proceeding under the incorrect assumption that $\mathbb{S}$-pages are safely on disk. Similarly, a HAP may notify the InkTag hypervisor when it explicitly requests that dirty pages be written out (such as when invoking the `msync` system call), and receive confirmation when the writeback actually occurs, or a warning that the OS has not complied.

Preventing deletion or loss of high-assurance data Because filesystem indexing structures vary widely between file systems, it is difficult to verify their correctness at hypervisor level. As a result, a malicious OS could appear to comply with InkTag policy by writing out file data blocks but not updating filesystem metadata, leaving file data blocks inaccessible.

InkTag provides a secure `fsck` mechanism for *conservation of high-assurance data* in the face of this threat. Secure page metadata includes file and offset information, allowing InkTag to reconstruct secure files independently of OS indexing structures. In addition, InkTag may prevent the OS from overwriting a secure page, unless the OS is replacing the page with a newer version of the same secure page, or if a newer version of the page has previously been written elsewhere on disk.

Consistency for secure pages in the face of crashes InkTag is the first system to address consistency requirements for an untrusted operating system. Without proper filesystem consistency, file data may become unavailable, file updates may be lost, and access control changes may not be honored.

When the OS writes an $\mathbb{S}$-page to disk on behalf of a HAP, both the new $\mathbb{S}$-page and its hash must be stored. If the system crashes

after only writing the $\mathbb{S}$-page contents or the hash, valid data could become unavailable, because InkTag would be unable to verify its authenticity. InkTag keeps two versions of each $\mathbb{S}$-page's hash: the version for the page on disk before the update, and after. Before a data write, InkTag will store the updated hash on disk. Because disk drives write blocks atomically, a hash matching the data will always be on disk, and high-assurance data on the backing store will always be available.

In our current prototype, per-file metadata (such as access control information and the file's length) is stored separately from the guest filesystem, in storage private to the hypervisor. We leave to future work enforcing consistency between per-file and per-page metadata.

7. Implementation

This section describes our prototype implementation of InkTag. InkTag consists of three major components: the InkTag hypervisor, extensions to the untrusted guest OS to support paraverification, and tools to compile user-level applications to run as HAPs.

7.1 InkTag hypervisor

The InkTag hypervisor is built as an extension to the KVM (Kernel Virtual Machine) hypervisor, a standard module included in the Linux kernel. We extend the Linux 2.6.36 kernel and KVM implementation. InkTag is built to support Intel's VMX hardware virtualization support, although we believe its design to be equally applicable to hardware virtualization support in AMD processors.

During execution of the untrusted guest operating system and non-InkTag applications, the hypervisor behaves almost identically to a hypervisor without InkTag support, with most virtualization tasks handled by hardware virtualization extensions present in the processor. The untrusted EPT (§3.2) is used to translate guest-physical addresses during untrusted execution, and most page table operations (except for those on a HAP's address space) occur without hypervisor intervention.

Scheduling Upon scheduling a HAP, InkTag must ensure that the untrusted OS cannot execute any code in a high-assurance context. To protect the HAP from the OS, it disallows automatic vectoring of interrupts and exceptions by the virtualization hardware. If InkTag allowed the hardware to vector interrupts automatically, then the operating system would gain control while still executing in a high-assurance context, with cleartext access to secure pages. Intel processors allow for fine-grained control over enabled virtualization features via bits in the virtual machine control structure (VMCS), the hardware descriptor used to control virtualization. To schedule a HAP, InkTag clears many of the feature bits in the VMCS, installs the trusted EPT, and then transfers control to the HAP.

When an interrupt or exception occurs during HAP execution, InkTag saves and then clears the HAP's register file. The untrusted OS must be prevented from reading or writing HAP registers. InkTag directs the instruction pointer to a small *untrusted trampoline* in a part of the HAP's address space unprotected by InkTag. This trampoline is responsible for interactions between the HAP and the guest OS (such as invoking system calls), and also for rescheduling the HAP when the HAP's process context is rescheduled by the operating system. The HAP receives system call results from the trampoline, but does not trust it — all information is validated as if it came from the untrusted OS.

Page tables Each HAP has two page tables, and shares the system-wide trusted and untrusted EPTs. One page table is written directly by the untrusted kernel, which we refer to as the *OS page table*. It is written directly by the OS (even when using the paravirtualized interface, Linux also updates the page table directly), and its contents are not trusted. The other page table is written only by the InkTag hypervisor in response to calls by the untrusted OS that are successfully verified, which we refer to as the *hypervisor page table*. The hypervisor page table is trusted. For a non-malicious OS, its page table should be a superset of the hypervisor page table.

When a HAP runs in untrusted mode (e.g., just before making a system call), `cr3` points to the OS page table, and when a HAP runs in trusted mode, the hypervisor points `cr3` to the hypervisor page table. In trusted mode, there is a single untrusted mapping, for the pages used to marshal and unmarshal system call arguments. InkTag must guarantee that the OS has not overlapped the mapping of this area with any trusted mapping. In untrusted mode, the process can map any $\mathbb{S}$-page, but access to that page will be detected by the permission bits in the untrusted EPT, and the hypervisor will encrypt and hash the page before giving the untrusted code access to the page.

Although the untrusted OS cannot access the hypervisor page tables, it is responsible for their allocation. On each page table allocation on behalf of a HAP, the guest OS must also allocate a hypervisor page table page and send both to the InkTag hypervisor in order to successfully add to the page table tree. The hypervisor protects the hypervisor page table page from the guest OS until that part of the page table tree is deallocated, at which point the guest regains access.

7.2 Untrusted OS extensions

We extend the untrusted guest kernel to support paraverification. Information about updates to HAP page tables form the majority of paraverification events, and take advantage of existing paravirtualization callbacks present in the Linux kernel. We install an untrusted module into the guest kernel to handle these callbacks. The kernel module is responsible for receiving tokens from HAPs requesting memory mappings, and passing these tokens to the InkTag hypervisor on page faults.

Although untrusted, the guest kernel module is cooperative when uncompromised, and attempts to minimize the amount of communication with the InkTag hypervisor. Any hypercall causes a VM-exit (a context switch from guest to hypervisor), thus unnecessary communication should be avoided to maximize performance. The untrusted kernel module tracks which processes contain HAPs, and communicates page table information only for those processes. In addition, the kernel module performs extensive batching of page table updates. While the Linux paravirtualization interface supports some batching of address space updates (such as a series of page table updates occurring before a TLB flush), our kernel module extends this buffering significantly, because the InkTag hypervisor can safely wait to process most updates until directly before scheduling a HAP. This allows communication of paraverification information without any additional VM-exits, as scheduling a HAP already requires passing control to the InkTag hypervisor.

7.3 Building HAPs

In our current prototype, InkTag HAPs must be compiled from C source code. C applications are built as HAPs primarily by replacing the standard C library and startup files with InkTag-specific versions. We modify standard C startup files to initialize the current process as a HAP before passing control to the C library. On initialization, the HAP sends the OID of its executable to the InkTag hypervisor, which initializes a HAP context, and adds the $.bin.\langle\mathsf{oid}\rangle$ attribute. Note that any process may invoke initialization claiming to be any executable OID. However, after initialization the InkTag hypervisor will ensure integrity for the process address space. Thus to continue executing as a HAP, the process must construct an address space that is byte-for-byte identical, and thus functionally equivalent, to the originally claimed binary.

	Linux	InkTag	Overhead
null	0.04	2.23	55.80×
open/close	0.87	6.90	7.95×
ctxsw 2p/0k	0.71	1.01	1.41×
File create	5.46	12.92	2.36×
File delete	3.40	7.56	2.23×
mmap	4059.20	40360.00	9.94×
pagefault	0.89	6.68	7.50×
fork	99.00	567.80	5.74×
fork+exec	290.60	882.60	3.04×

Table 2. LMbench latency microbenchmark results (in microseconds.)

We automate HAP interaction with the InkTag hypervisor by interposing on system calls in the standard C library. For example, when an application calls `mmap()`, the system call is intercepted by our trusted InkTag library. The library performs the system call, validates the result to ensure that the untrusted OS does not violate invariants for the address space (§4.4), and passes a token to the untrusted OS for handling page faults in the newly mapped region (§4.3).

InkTag does not expose information about hashes and encryption keys for $\mathbb{S}$-pages to applications. Thus HAPs must interact with secure files by mapping them into their address space. We implement `mmap()`-based versions of standard `read()` and `write()` system calls to support applications that rely on those calls for file I/O.

7.4 Block Driver

To implement transparent loading of $\mathbb{S}$-page metadata, we add a new block driver implementation to the QEMU (the userspace portion of KVM) block driver interface. The new block driver transparently translates read and write requests from the hardware emulation layer. Doing the translation at this level puts us at the lowest layer before the actual hardware allowing us to better adjust and handle the block requests.

Our secure metadata consists of two 32-byte hashes, an OID, and an offset. We place secure page metadata once every 32 pages of normal data. We track which disk sectors contain valid $\mathbb{S}$-pages using a bitmap. This bitmap is mapped and updated as HAPs execute, and is written to disk only upon shutdown. This policy is safe even in the event of a crash: there will be enough data on disk to be able to recover the bitmap.

8. Evaluation

In this section we evaluate the performance overhead imposed by our InkTag prototype built in the KVM hypervisor. We evaluated InkTag's performance using two different machines: we run latency and SPEC CPU benchmarks on an i7 860 running at 2.80GHz, and InkTag block storage and application benchmarks on an Intel i7 870 at 2.93GHz. Both machines have quad-core processors, 8GB of memory, and run Ubuntu 10.04.4.

We modify the 2.6.36 Linux kernel and QEMU 0.12.5 for InkTag, and run unmodified versions for the baseline. VM guests run with a single virtual CPU, 2GB of memory, and the same kernel as the host. In the InkTag guest, all benchmark binaries run as HAPs.

8.1 Microbenchmarks

Table 2 shows results from the LMBench [30] suite of OS microbenchmarks. LMBench is a series of portable microbenchmarks focused on measuring individual OS operations in isolation. We restrict our evaluation to focus on file operations, memory manipulation, and process creation, as these are the areas that will be affected by running as an InkTag HAP. We modify LMBench only enough to turn its components into HAPs: 68 lines of modifications to the build system and 5 lines of configuration changes.

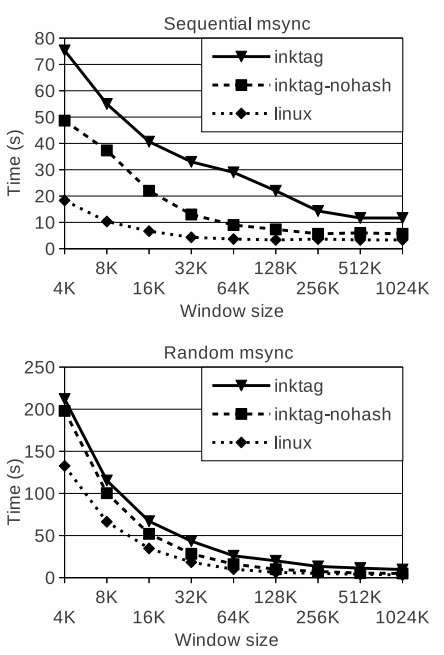

Figure 7. InkTag storage backend performance as measured by sequential or random msync()s on a memory-resident file.

The null syscall benchmark primarily measures the latency of switching between an application and the OS, and represents the worst case for InkTag. A HAP must context switch from user context, to the virtual machine, then into the operating system, and then return along the opposite path. The high latency for switching between application and OS directly impacts the performance of nearly all of the LMBench microbenchmarks, as they measure interactions between an application and the operating system. Additionally, operations that involve any kind of page table update, such as `mmap`, `fork`, and `fork+exec`, are also affected due to the InkTag hypervisor validating each page table update. These overheads appear large in isolation; however, most applications are significantly less sensitive to system call latency than microbenchmarks. Most of the LMBench benchmarks show a difference in latency that is 10s of microseconds or less.

8.2 Storage

We evaluate InkTag's storage backend with a benchmark that synchronizes regions varying in size from a 256MB secure file cached in memory to the virtual disk. We disable host OS caching for our virtual disk, to best simulate the effect of actual disk scheduling on I/O throughput. Figure 7 shows the performance of syncing varying window sizes, from 4KB to 1MB, either sequentially through the file or in random order. In addition, we show numbers for a version of InkTag in which we have disabled encryption and hashing, in order to isolate the effect of disk scheduling on performance. The encryption and hashing occur when the OS touches $\mathbb{S}$-pages to sync them to disk. Our InkTag prototype interleaves $\mathbb{S}$-page data and metadata at an interval of 32 pages (128KB). For window sizes above 128KB, InkTag approaches the performance of a standard block device, as the InkTag block driver can combine a page of metadata with 32 data pages in a single write to the backing device. Beneath that threshold, InkTag's performance suffers, especially for sequential writes. This is due to InkTag's metadata lay-

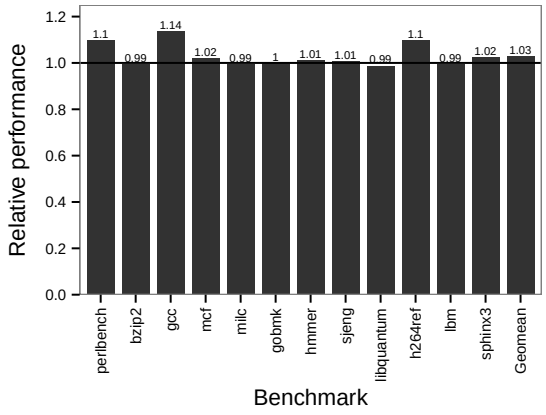

Figure 8. SPEC CPU2006 benchmark performance. "Geomean" indicates the geometric mean of relative performance.

	Linux	InkTag
Apache latency	195 ms	220 ms (1.13×)
Apache throughput	462.42 req/s	453.93 req/s (1.02×)
Dokuwiki throughput	13.6 req/s	8.83 req/s (1.54×)

Table 3. InkTag performance for large applications.

	Apache		DokuWiki	
	Linux	InkTag	Linux	InkTag
Check hash	-	209	-	2,911,649
Check zero hash	-	57	-	2,893,517
Update hash	-	82	-	1,029
EPT fault	689	1,131	10,668	78,055
VM-exit	171,145	1,217,042	138,801	11,216,363

Table 4. Counts of performance-critical events during benchmark execution. We count the number of times InkTag must hash a data page ("Check hash"), hash a data page that should be zero-initialized ("Check zero hash"), encrypt a page and update its hash ("Update hash"), fault on a nested page table ("EPT fault"), and context-switch out of the guest ("VM-exit").

out. For example, sequential writes to each of the 4KB pages in a single cluster of data pages represents a good case for disk scheduling. InkTag, however, must write the metadata page followed by the data page for each of these writes, causing the disk to seek back and forth instead of writing sectors in sequential order.

8.3 Application benchmarks

We measure the overhead imposed by InkTag with three different types of applications: CPU-bound SPEC benchmarks, the Apache web server, and DokuWiki, a complete wiki application converted to use InkTag attributes for authentication.

SPEC With little OS interaction, CPU-bound applications exhibit little performance overhead when running as HAPs. Figure 8 shows results for selected benchmarks from the SPEC 2006 [19] suite (InkTag does not support Fortran). Out of twelve benchmarks, nine benchmarks run within a 3% performance overhead of unmodified KVM, and `gcc` benchmark has the largest overhead of 14%.

Apache Table 3 shows results for our evaluation of the performance of the Apache webserver when compiled as a HAP. We run the standard `ab` benchmarking tool included with Apache on the machine hosting the virtualized guest, providing nearly unlimited bandwidth from the web server to client. We execute 10,000 requests from client to server, at a concurrency level of 100. The Apache web server serves requests with a 13% overhead in latency, and a 2% overhead in throughput relative to normal virtualized execution. Apache represents a relatively good case for InkTag: with several long-lived processes, Apache rarely has to pay the increased costs imposed by InkTag for application initialization and teardown.

DokuWiki In order to demonstrate the ability of InkTag to provide security for realistic workloads, we modified the DokuWiki[2] wiki server to take advantage of InkTag secure files and access control. DokuWiki is a wiki written in PHP that stores wiki pages as files in the server filesystem. We recompiled the PHP CGI binary to work with InkTag and ran DokuWiki as a CGI script. We added an InkTag authentication module to DokuWiki to allow a user to log in with their system credentials (similar to the decentralized login process described in §5.3) and to restrict access to wiki content via InkTag access control.

To test the effect of InkTag on a representative set of modifications to a representative DokuWiki installation, we downloaded a set of 6,430 revisions of 765 pages from the DokuWiki website (which is itself run using DokuWiki) to simulate wiki activity. We evaluate DokuWiki with a 90% read workload, which we believe a reasonable characterization of a wiki workload. Each write replaces a page with the subsequent revision of a page in the downloaded DokuWiki corpus. We measured the total wallclock time for 10 clients to perform a collective 1,000 requests on the wiki. Our wiki client makes use of an XML RPC interface that DokuWiki provides to avoid the need for programmatically interfacing with DokuWiki forms.

As a HAP with InkTag authentication, DokuWiki runs with a 1.54× overhead over a baseline virtualized execution. As a PHP application, DokuWiki maps a large number of scripts (with integrity assured by the InkTag hypervisor) into memory and exercises a significant amount of anonymous temporary memory. As with OS users, InkTag's authentication aligns along process boundaries. Thus, we must run DokuWiki as an inefficient CGI application, not as an Apache module. CGI is a performance worst case for InkTag: each request initializes and destroys an entire application address space.

Virtualization metrics Table 4 shows counts for a number of performance-critical events during the execution of our large application benchmarks. Specific to InkTag execution are the number of times physical frames are hashed, as well the number of times the hash of the associated $\mathbb{S}$-page is updated (this event also counts the number of times an $\mathbb{S}$-page must be encrypted). With a few long-lived processes, most of the address space for the Apache web server remains mapped in the trusted EPT, requiring relatively few hash updates. DokuWiki, which constructs and destroys an address space for each request, has a large number of hash operations.

Of particular note are the number of times InkTag is requested to verify the hash for a page consisting entirely of zeroes. In fact, the vast majority of hash operations are invoked to determine if a page is initialized to zero (2.8 out of 2.9 million hash operations for the DokuWiki benchmark). InkTag optimizes this case: when asked to verify the hash of a physical frame, InkTag compares the hash value with the hash of a zero page. If the page should contain only zero, InkTag simply verifies that property, rather than computing a full digest. As a result, computation of hashes is not a significant factor in InkTag's performance overhead. Similarly, while encryption is necessary for privacy, it does not significantly affect running time: the majority of pages that would otherwise be encrypted due to access by the operating system are in fact anonymous memory regions that have been unmapped by an application. The entire

[2] `http://www.dokuwiki.org`

memory region is being destroyed, so it needs only be erased, not encrypted for privacy and hashed for integrity.

A major factor for InkTag performance is the number of times the processor must switch context between the virtual machine and the host. In the DokuWiki benchmark, for example, InkTag must exit the virtual machine nearly two orders of magnitude more often than a standard execution. We hope to investigate ways to reduce the cost of such context switches as part of future work.

9. Related work

Untrusted operating systems InkTag, XOMOS [26], SP^3 [47], and Overshadow [11, 35] share the goal of minimizing the ability of an untrustworthy system component to tamper with a sensitive application. Previous work focused on isolating high-assurance applications from the system, while InkTag focuses on allowing the application to *use* untrusted system services, providing access control and crash recovery for persistent storage. For example, while Overshadow guarantees that user processes are isolated from the operating system, it does not implement access control for secure data: once an application that has created a secure file terminates, there is no meaningful way for processes to share access to that file.

Just as InkTag allows a trusted process and hypervisor to stop trusting the OS, CloudVisor [49] allows a trusted process, OS, and nested hypervisor to stop trusting the hypervisor and other cloud management software. Another approach, exemplified in Proxos [44], essentially reimplements portions of the OS in the application; this approach does not address shared abstractions between mutually untrusting programs.

Virtual machines The use of virtual machine monitors to help protect operating system and application execution is not a new concept; there have been systems ranging from providing dedicated virtual hardware per secure application [18] to enforcing kernel entry and exit points to provide system integrity [42]. By allowing applications to make use of extensive operating system facilities for sharing data, while still verifying its behavior, InkTag provides a much more flexible solution than heavyweight per-application virtualization. At the other end of the spectrum, simple code integrity is not sufficient to ensure operating system safety [21].

Trusted computing Recent systems, including Flicker [29], TrustVisor [28], and Memoir [33] leverage the TPM to completely isolate sensitive computations, such as encryption and random number generation, from the OS. These systems can protect an application's random number generator's internals from being leaked to the OS, but they cannot protect larger code that requires OS functionality, such as file access.

Benign OS integrity Another related branch of research attempts to prevent malicious inputs from compromising a non-malicious OS, including HookSafe [45], KernelGuard [39], and several others [2, 10, 27, 40]. These systems often prevent specific classes of security problems and are not designed for the strong adversarial model InkTag can defend against.

VM introspection Several systems attempt to enforce security properties on a non-malicious guest OS by interpreting low-level events based on expert information [24, 34, 38, 46], automatic source code extraction [17, 20, 23], or inferred from observing program executions [4, 14]. This interpretation is fragile (called the semantic gap [9]) and, broadly speaking, work in this area assumes a weak adversarial model. InkTag's paraverification avoids the semantic gap and provides fundamentally stronger guarantees.

Sandboxing This work is concerned with assuring the integrity of necessary OS functionality; yet this is easily conflated with the goal of isolating untrusted applications, or sandboxing. Several recent sandboxing architectures have explored techniques that limit OS access [15, 36, 48], or monitor system calls [3, 6, 16, 22, 37] to protect the OS's sharing abstractions from a malicious application.

Hierarchically Named Access Control In recent years, work in the area of hierarchical naming and attribute based access control has landed mostly in the area of distributed systems and grid computing [7, 12]. ABAC [41] and XACML [32] are both projects aimed at bringing attribute based access control to the enterprise world. InkTag differs in scope, whereas these projects try to define attributes on employees, InkTag tries to implement flexible access control at the level of individual processes.

UserFS [25] allows principals to hierarchically manage creation and deletion of sub-principals. The system uses traditional UIDs as a namespace for managing access control of files and resources. UserFS achieves the goals of efficiency and familiarity but has difficultly easily expressing group semantics and even more flexible access control policy since processes can have only one UID or effective UID at a time. InkTag does not depend on the complicated code in the guest operating system, and can achieve more flexible policies through attributes.

10. Conclusion

InkTag represents a significant step forward in verifying the behavior of untrusted operating systems. By removing the burden of attempting to verify a completely unmodified operating system, paraverification enables a simple, high-performance hypervisor implementation. InkTag is the first such system to enable access control for secure data, as well as address essential system issues such as crash consistency between OS-managed data and secure metadata.

Acknowledgments

We thank the anonymous reviewers and Carl Waldspurger for their comments on earlier versions of this paper. This research is supported by NIH R01 LM011028-01, NSF CNS-1228843, CNS-0905602, and NSF CAREER CNS-0644205.

References

[1] Microsoft security bulletin search, 2012. `http://technet.microsoft.com/security/bulletin`.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, 2005.

[3] Anurag Acharya and Mandar Raje. MAPbox: Using parameterized behavior classes to confine applications. In *USENIX Security*, 2000.

[4] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC*, 2008.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[6] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. REMUS: A security-enhanced operating system. *TISSEC*, 5(1), 2002.

[7] Rakesh Bobba, Omid Fatemieh, Fariba Khan, Carl A. Gunter, and Himanshu Khurana. Using attribute-based access control to enable attribute-based messaging. In *ACSAC*, 2006.

[8] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *ASPLOS*, March 2013.

[9] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *HotOS*, pages 133–, 2001.

[10] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security*, 2005.

[11] Xioaxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffery Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, May 2008.

[12] Lorenzo Cirio, Isabel F. Cruz, and Roberto Tamassia. A role and attribute based access control system using semantic web technologies. In *OTM*, 2007.

[13] Tim Dierks and Eric Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol: Version 1.2. `http://tools.ietf.org/html/rfc5246`, 2008.

[14] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Oakland*, May 2011.

[15] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, 2008.

[16] Timothy Fraser, Lee Badger, and Marc Feldman. Hardening COTS software with generic software wrappers. In *Oakland*, 1999.

[17] Timothy Fraser, Matthew R. Evenson, and William A. Arbaugh. VICI—virtual machine introspection for cognitive immunity. In *ACSAC*, pages 87–96, 2008.

[18] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, October 2003.

[19] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[20] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, March 2011.

[21] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *USENIX Security*, 2009.

[22] Kapil Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*, 2000.

[23] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *CCS*, pages 128–138, 2007.

[24] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *USENIX*, 2006.

[25] Taesoo Kim and Nickolai Zeldovich. Making linux protection mechanisms egalitarian with UserFS. In *USENIX Security*. USENIX Association, 2010.

[26] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, pages 178–192. ACM Press, 2003.

[27] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux kernel integrity measurement using contextual inspection. In *STC*, 2007.

[28] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Oakland*, May 2010.

[29] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, April 2008.

[30] Larry McVoy and Carl Staelin. LMbench: portable tools for performance analysis. In *USENIX*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[31] NIST. National vulnerability database. `http://nvd.nist.gov/`, 2012.

[32] OASIS. eXtensible access control markup language. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`, 2012.

[33] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Oakland*, 2011.

[34] Bryan D. Payne, Martim D. P. de A. Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *ACSAC*, 2007.

[35] Dan R. K. Ports and Tal Garfinkel. Towards application security on untrusted operating systems. In *HotSec*, San Jose, CA, USA, 2008. USENIX.

[36] Shaya Potter and Jason Nieh. Apiary: Easy-to-use desktop application fault containment on commodity operating systems. In *USENIX*, 2010.

[37] Neils Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

[38] Nguyen Anh Quynh and Yoshiyasu Takefuji. Towards a tamper-resistant kernel rootkit detector. In *SAC*, 2007.

[39] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating dynamic data kernel rootkit attacks via VMM-based guest-transparent monitoring. In *ARES*, Fukuoka, Japan, March 2009.

[40] Junghwan Rhee and Dongyan Xu. LiveDM: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. Technical report, Purdue University, West Lafayette, IN, February 2010.

[41] Mike Ryan, Ted Faber, Mei-Hui Su, John Wroclawski, and Steve Schwab. A.B←A.C. `http://abac.deterlab.net/`, 2012.

[42] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, pages 335–350, 2007.

[43] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, pages 552–61. ACM Press, October 2007.

[44] Richard Ta-min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, pages 279–292, 2006.

[45] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *CCS*, 2009.

[46] Min Xu, Xuxian Jiang, Ravi Sandhu, and Xinwen Zhang. Towards a VMM-based usage control framework for OS kernel integrity protection. In *SACMAT*, 2007.

[47] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, pages 71–80, 2008.

[48] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Oakland*, pages 79–93, 2009.

[49] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.