Virtual Ghost: Protecting Applications from Hostile Operating Systems

John Criswell
University of Illinois at Urbana-Champaign
criswell@illinois.edu

Nathan Dautenhahn
University of Illinois at Urbana-Champaign
dautenh1@illinois.edu

Vikram Adve
University of Illinois at Urbana-Champaign
vadve@illinois.edu

Abstract

Applications that process sensitive data can be carefully designed and validated to be difficult to attack, but they are usually run on monolithic, commodity operating systems, which may be less secure. An OS compromise gives the attacker complete access to all of an application's data, regardless of how well the application is built. We propose a new system, *Virtual Ghost*, that protects applications from a compromised or even hostile OS. Virtual Ghost is the first system to do so by combining compiler instrumentation and run-time checks on operating system code, which it uses to create *ghost memory* that the operating system cannot read or write. Virtual Ghost interposes a thin hardware abstraction layer between the kernel and the hardware that provides a set of operations that the kernel must use to manipulate hardware, and provides a few trusted services for secure applications such as ghost memory management, encryption and signing services, and key management. Unlike previous solutions, Virtual Ghost does *not* use a higher privilege level than the kernel.

Virtual Ghost performs well compared to previous approaches; it outperforms InkTag on five out of seven of the LMBench microbenchmarks with improvements between 1.3x and 14.3x. For network downloads, Virtual Ghost experiences a 45% reduction in bandwidth at most for small files and nearly no reduction in bandwidth for large files and web traffic. An application we modified to use ghost memory shows a maximum additional overhead of 5% due to the Virtual Ghost protections. We also demonstrate Virtual Ghost's efficacy by showing how it defeats sophisticated rootkit attacks.

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541986

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

General Terms Security

Keywords software security; inlined reference monitors; software fault isolation; control-flow integrity; malicious operating systems

1. Introduction

Applications that process sensitive data on modern commodity systems are vulnerable to compromises of the underlying system software. The applications themselves can be carefully designed and validated to be impervious to attack. However, all major commodity operating systems use large monolithic kernels that have complete access to and control over all system resources [9, 25, 32, 34]. These operating systems are prone to attack [21] with large attack surfaces, including large numbers of trusted device drivers developed by numerous hardware vendors and numerous privileged applications that are as dangerous as the OS itself. A compromise of any of these components or of the kernel gives the attacker complete access to all data belonging to the application, whether in memory or offline. Developers of secure applications running on such a system generally have no control over any of these components and must trust them implicitly for their own security.

Several previous projects [11, 16, 22, 37] have created systems that protect applications from a compromised, or even a hostile, operating system. These systems have all used hardware page protection through a trusted hypervisor to achieve control over the operating system capabilities. They rely on a technique called *shadowing* or *cloaking* that automatically encrypts (i.e., shadows or cloaks) and hashes any application page that is accessed by the operating system, and then decrypts it and checks the hash when it is next accessed by the application. System call arguments must be copied between secure memory and memory the OS is allowed to examine. While these solutions are effective, they have several drawbacks. First, they rely upon encrypting any of an application's memory that is accessed by the OS; an

application cannot improve performance by protecting only a selected subset of data, or requesting only integrity checks on data but not confidentiality (i.e., only using hashing and not encryption). Second, they assume the OS runs as a guest on a standard hypervisor, which may not be attractive in certain settings, such as energy-constrained mobile devices. Third, they require that all system call arguments must always be copied, even if the data being transferred is not security-sensitive, which is the common case in many applications. Fourth, these solutions do not provide any security benefits to the kernel itself; for example, control flow integrity or memory safety for the operating system kernel cannot reuse the mechanisms developed for shadowing.

We propose a new approach we call *ghosting* that addresses all these limitations. Our system, *Virtual Ghost*, is the first to enforce application security from a hostile OS using compiler instrumentation of operating system code; this is used to create secure memory called *ghost memory*, which cannot be read or modified *at all* by the operating system (in contrast, previous systems like Overshadow [11] and InkTag [16] do not prevent such writes and only guarantee that the tampering will be detected before use by the application). Virtual Ghost introduces a thin hardware abstraction layer that provides a set of operations the kernel must use to manipulate hardware, and the secure application can use to obtain essential trusted services for ghost memory management, encryption, signing, and key management. Although the positioning of, and some of the mechanisms in, this layer are similar to a hypervisor, Virtual Ghost is unique because (a) unlike a traditional hypervisor, there is *no* software that runs at a higher privilege level than the kernel – in particular, the hardware abstraction layer runs at the same privilege level; (b) Virtual Ghost uses (simple, reliable) compiler techniques rather than hardware page protection to secure its own code and data; and (c) Virtual Ghost completely denies OS accesses to secure memory pages, not just encrypting and signing the pages to detect OS tampering.

Moreover, the compiler instrumentation in Virtual Ghost inherently provides strong protection against external exploits of the OS. First, traditional exploits, such as those that inject binary code, are not even expressible: all OS code *must* first go through LLVM bitcode form and be translated to native code by the Virtual Ghost compiler. Second, attacks that leverage existing native code, like return-oriented programming (ROP) [31], require control-flow hijacking, which Virtual Ghost explicitly prevents as well. In particular, Virtual Ghost enforces Control Flow Integrity (CFI) [3] on kernel code in order to ensure that the compiler instrumentation of kernel code is not bypassed. CFI automatically defeats control-flow hijacking attacks, including the latter class of external exploits. Together, these protections provide an additional layer of defense for secure applications on potentially buggy (but non-hostile) operating systems.

Another set of differences from previous work is in the programming model. First, applications can use ghost memory selectively for all, some, or none of their data. When using it for all their data, the secure features can be obtained transparently via a modified language library (e.g., libc for C applications), just as in previous work. Second, applications can pass non-ghost memory to system calls without the performance overheads of data copying. Third, when sending sensitive data through the operating system (e.g., for I/O), ghost applications can choose which encryption and/or cryptographic signing algorithms to use to obtain desired performance/security tradeoffs, whereas previous systems generally baked this choice into the system design. Finally, a useful point of similarity with previous work is that the usability features provided by previous systems (e.g., secure file system services in InkTag) are orthogonal to the design choices in Virtual Ghost and can be directly incorporated.

We have developed a prototype of Virtual Ghost and ported the FreeBSD 9.0 kernel to it. To evaluate its effectiveness, we ported three important applications from the OpenSSH application suite to Virtual Ghost, using ghost memory for the heap: `ssh`, `ssh-keygen`, and `ssh-agent`. These three can exchange data securely by sharing a common application key, which they use to encrypt the private authentication keys used by the OpenSSH protocols.

Since exploiting a kernel that runs on Virtual Ghost via an external attack is difficult, we evaluated the effectiveness of Virtual Ghost by adding a malicious module to the FreeBSD kernel, replacing the `read` system call handler. This module attempted to perform two different attacks on `ssh-agent`, including a sophisticated one that tries to alter application control flow via signal handler dispatch. When running without Virtual Ghost, both exploits successfully steal the desired data from `ssh-agent`. Under Virtual Ghost, both exploits fail and `ssh-agent` continues execution unaffected.

Our performance results show that Virtual Ghost outperforms InkTag [16] on five out of seven of the LMBench microbenchmarks, with improvements between 1.3x and 14.3x. The overheads for applications that perform moderate amounts of I/O (thttpd and sshd) is negligible, but the overhead for a completely I/O-dominated application (postmark) was high. We are investigating ways to reduce the overhead of postmark.

The rest of the paper is organized as follows. Section 2 describes the attack model that we assume in this work. Section 3 gives an overview of our system, the programmer's view when using it, including the security guarantees the system provides. Section 4 describes our design in detail, and Section 5 describes the implementation of our prototype. Section 6 describes our modifications to secure OpenSSH applications with our system. Sections 7 and 8 evaluate the security and performance of our system. Section 9 describes related work, Section 10 describes future work, and Section 11 concludes the paper.

2. System Software Attacks

In this section, we briefly describe our threat model and then describe the attack vectors that a malicious operating system might pursue within this threat model.

2.1 Threat Model

We assume that a user-space (i.e., unprivileged) application wishes to execute securely and perform standard I/O operations, but without trusting the underlying operating system kernel or storage and networking devices. Our goal is to preserve the application's integrity and confidentiality. Availability is outside the scope of the current work; we discuss the consequences of this assumption further, below. We also do not protect against side-channel attacks or keyboard and display attacks such as stealing data via keyboard loggers or from graphics memory; previous software solutions such as Overshadow [11] and InkTag [16] do not protect against these attacks either, whereas hardware solutions such as ARM's TrustZone [6] and Intel's SGX [24] do.

We assume that the OS, including the kernel and all device drivers, is malicious, i.e., may execute arbitrary hostile code with maximum privilege on the underlying hardware. We *do not assume that a software layer exists that has higher privilege than the OS*. Instead, we assume that the OS source code is *ported* to a trusted run-time library of low-level functions that serve as the interface to hardware (acting as a hardware abstraction layer) and supervise all kernel-hardware interactions. The OS is then compiled using a modified compiler that instruments the code to enforce desired security properties, described in later sections.[1] We assume that the OS can load and unload arbitrary (untrusted) OS modules dynamically, but these modules must also be compiled by the instrumenting compiler. Moreover, we assume that the OS has full read and write access to persistent storage, e.g., hard disk or flash memory.

We do *not* prevent attacks against the application itself. In practice, we expect that a secure application will be carefully designed and tested to achieve high confidence in its own security. Moreover, in practice, we expect that a secure application (or the secure subsets of it) will be much smaller than the size of a commodity OS, together with its drivers and associated services, which typically run into many millions of lines of code. For all these reasons, the developers and users of a secure application will not have the same level of confidence in a commodity OS as they would in the application itself.

Application availability is outside the scope of the current work. The consequence, however, is only that an attacker could deny a secure application from making forward progress (which would likely be detected quickly by users or system administrators); she could not steal or corrupt data produced by the application, even by subverting the OS in arbitrary ways.

[1] It is reasonable to expect the OS to be ported and recompiled because, in all the usage scenarios described in Section 1, we expect OS developers to make it an explicit design goal to take the OS out of the trusted computing base for secure applications.

2.2 Attack Vectors

Within the threat model described in Section 2.1, there are several attack vectors that malicious system software can take to violate an application's confidentiality or integrity. We describe the general idea behind each attack vector and provide concrete example attacks.

2.2.1 Data Access in Memory

The system software can attempt to access data residing in application memory. Examples include:

- The system software can attempt to read and/or write application memory directly via load and store instructions to application virtual addresses.
- Alternatively, the OS may attempt to use the MMU to either map the physical memory containing the data into virtual addresses which it can access (reading), or it may map physical pages that it has already modified into the virtual addresses that it cannot read or write directly (writing).
- The system software can direct an I/O device to use DMA to copy data to or from memory that the system software cannot read or write directly and memory that the system software can access directly.

2.2.2 Data Access through I/O

A malicious OS can attempt to access data residing on I/O devices or being transferred during I/O operations. Examples include:

- The OS can read or tamper with data directly from any file system used by the application.
- The OS can read or tamper with data being transferred via system calls to or from external devices, including persistent storage or networks.
- The OS can map unexpected blocks of data from I/O devices into memory on an `mmap` system call, effectively substituting arbitrary data for data expected by the application.

2.2.3 Code Modification Attacks

A malicious OS may attempt to change the application code that operates upon application data, in multiple ways, so that the malicious code would execute with the full memory access and I/O privileges of the application. Some examples:

- The OS can attempt to modify the application's native code in memory directly.
- The OS could load a malicious program file when starting the application.

- The OS could transfer control to a malicious signal handler when delivering a signal to the application.
- The OS could link in a malicious version of a dynamically loaded library (such as libc) used by the application.

2.2.4 Interrupted Program State Attacks

A malicious OS can attempt to modify or steal architectural state of an application while the application is not executing on the CPU. Examples include:

- Malicious system software could attempt to read interrupted program state to glean sensitive information from program register values saved to memory.
- Alternatively, it could modify interrupted program state (e.g., the PC) and put it back on to the processor on a return-from-interrupt or return-from-system call to redirect application execution to malicious code.

2.2.5 Attacks through System Services

A more subtle and complex class of attacks is possible through the higher-level (semantic) services an OS provides to applications [10, 28]. While these attacks are still not well understood, our solution addresses an important subset of them, namely, memory-based attacks via the `mmap` system call (the same subset also addressed by InkTag). Examples include the following:

- The OS is the source of randomness used by pseudorandom number generators to create random seeds, e.g., via the device /dev/random. The OS can compromise the degree of randomness and even give back the same random value on different requests, violating fundamental assumptions used for encrypting application data.
- The OS could return a pointer into the stack via `mmap` [10], thereby tricking the application into corrupting its stack to perform arbitrary code execution via return-to-libc [35] or return-oriented programming [31].
- The OS could grant the same lock to two different threads at the same time, introducing data races with unpredictable (and potentially harmful) results.

It is important to note that the *five categories of attack vectors* listed above are intended to be comprehensive but the specific examples within each category are not: there may be several other ways for the OS to attack an application within each category. Nevertheless, our solution enables applications to protect themselves against *all attacks* within the first four categories and a subset of attacks from the fifth.

3. Secure Computation Programming Model

The key feature we provide to a secure application is the ability to compute securely using secure memory, which we refer to as *ghost memory*, and to exchange data securely with external files and network interfaces. Applications do *not* have to be compiled with the SVA-OS compiler or instrumented in any particular way; those requirements only apply to the OS. In this section, we discuss the programmer's interface to secure computation. In Section 4, we show how our system, which we call *Virtual Ghost*, prevents the operating system from violating the integrity or confidentiality of an application that uses secure computation.

3.1 Virtual Ghost Memory Organization

Virtual Ghost divides the address space of each process into three partitions. The first partition holds traditional, user-space application memory while a second partition, located at the high end of the virtual address space, holds traditional kernel-space memory. The kernel space memory is mapped persistently and is common across all processes, while the user-space memory mappings change as processes are switched on and off the CPU. Operating systems such as Linux and BSD Unix already provide this kind of user-space/kernel-space partitioning [9, 25].

A new third partition, the ghost memory partition, is application-specific and is accessible only to the application and to Virtual Ghost. Physical page frames mapped to this partition hold application code, thread stacks, and any data used by secure computation. These page frames logically belong to the process and, like anonymous `mmap()` memory mappings, are unmapped from/mapped back into the virtual address space as the process is context switched on and off the processor.

Some applications may choose not to protect any of their memory, in which case the Ghost memory partition would go unused (their code and thread stacks would be mapped into ordinary application memory). Others may choose to protect *all* of their memory except a small portion used to pass data to or from the operating system. The latter configuration is essentially similar to what Overshadow provides, and Virtual Ghost provides this capability in the same way – by interposing wrappers on system calls. This is described briefly in Section 6.

3.2 Ghost Memory Allocation and Deallocation

An application wishing to execute securely without trusting the OS would obtain one or more chunks of ghost memory from Virtual Ghost; this memory is allocated and deallocated using two new "system calls" shown in Table 1. The instruction `allocgm()` asks Virtual Ghost to map one or more physical pages into the ghost partition starting at a specific virtual address. Virtual Ghost requests physical page frames from the operating system, verifies that the OS has removed all virtual to physical mappings for the frames, maps the frames starting at the specified address within the application's ghost partition, and zeroes out the frames' contents. The instruction `freegm()` tells Virtual Ghost that the block of memory at a specified virtual address is no longer needed and can be returned to the OS. Virtual Ghost unmaps the

Name	Description
`allocgm(void * va, uintptr_t num)`	Map `num` page frames at the virtual address `va` (which must be within the ghost memory region).
`freegm(void * va, uintptr_t num)`	Free `num` page frames of ghost memory starting at `va` (which must be previously allocated via `allocgm`)

Table 1. Ghost Memory Management Instructions

frames, zeroes their contents, and returns them to the operating system.

These instructions are not designed to be called directly by application-level code (although they could). A more convenient way to use these instructions is via a language's run-time library (e.g., the C standard library), which would use them to create language allocation functions that allocate ghost memory, e.g., using a modified version of `malloc`.

Note that ghost memory pages cannot be initialized using demand-paging from persistent storage into physical memory. Ghost Memory is like anonymous `mmap` memory, which Virtual Ghost can provide to an application at startup and when the application allocates it via `allocgm()`. To get data from the network or file system into ghost memory, the application must first read the data into traditional memory (which is OS accessible) and then copy it (or decrypt it) into ghost memory. Other systems (e.g., InkTag [16]) perform the decrypt/copy operation transparently via the demand-paging mechanism. By requiring the application to decrypt data explicitly, Virtual Ghost avoids the complications of recovering encrypted application data after a crash (because the encryption keys are visible to the application and not hidden away within the Virtual Ghost VM). It also gives the application more flexibility in choosing different encryption algorithms and key lengths. Furthermore, this approach simplifies the design and reduces the size of Virtual Ghost, thus keeping Virtual Ghost's Trusted Computing Base (TCB) small.

3.3 I/O, Encryption, and Key Management

Applications that use ghost memory require secure mechanisms for communicating data with the external world, including local disk or across a network. Duplicating such I/O services in Virtual Ghost would significantly increase the size and complexity of the system's TCB. Therefore, like Overshadow [11] and InkTag [16], we let the untrusted OS perform all I/O operations. Applications running on Virtual Ghost must use cryptographic techniques (i.e., encryption, decryption, and digital signatures) to protect data confidentiality and detect potential corruption when writing data to or reading data from the OS during I/O.

Upon startup, applications need encryption keys to access persistently stored data (such as files on a hard drive) stored during previous executions of the application. Virtual Ghost provides each application with an application-specific public-private key pair that is kept secure from the OS and all other applications on the system. The application is responsible for encrypting (decrypting) secret data using this key pair before passing the data to (after receiving the data from) explicit I/O operations. Wrappers for widely used I/O operations such as `read()` and `write()` can make it largely transparent for applications to do the necessary encryption and decryption.

Using encryption for local IPC, network communication, and file system data storage allows applications to protect data confidentiality and to detect corruption. For example, to protect data confidentiality, an application can encrypt data with its public key before asking the OS to write the data to disk. To detect file corruption, an application can compute a file's checksum and encrypt and store the checksum in the file system along with the contents of the file. When reading the file back, it can recompute the checksum and validate it against the stored value. The OS, without the private key, cannot modify the file's contents and update the encrypted checksum with the appropriate value.

Unlike programmed I/O, swapping of ghost memory is the responsibility of Virtual Ghost. Virtual Ghost maintains its own public/private key pair for each system on which it is installed. If the OS indicates to Virtual Ghost that it wishes to swap out a ghost page, Virtual Ghost will encrypt and checksum the page with its keys before providing the OS with access. To swap a page in, the OS provides Virtual Ghost with the encrypted page contents; Virtual Ghost will verify that the page has not been modified and place it back into the ghost memory partition in the correct location. This design not only provides secure swapping but allows the OS to optimize swapping by first swapping out traditional memory pages.

3.4 Security Guarantees

An application that follows the guidelines above on a Virtual Ghost system obtains a number of strong guarantees, even in the presence of a hostile or compromised operating system or administrator account. All these guarantees apply to application data allocated in ghost memory. By "attacker" below, we mean an entity that controls either the OS or any process other than the application process itself (or for a multi-process or distributed application, the set of processes with which the application explicitly shares ghost memory or explicitly transfers the contents of that memory).

1. An attacker cannot read or write application data in memory or in CPU registers.
2. An attacker cannot read or write application code and cannot subvert application control flow at any point during the application execution, including application startup, signal delivery, system calls, or shutdown.

3. An attacker cannot read data that the application has stored unencrypted in ghost memory while the data is swapped out to external storage, nor can it modify such data without the corruption being detected before the data is swapped back into memory.
4. An attacker cannot read the application's encryption key, nor can it modify the application's encryption key or executable image undetected. Such modifications will be detected when setting the application up for execution and will prevent application startup.
5. By virtue of the application's encryption key being protected, data encrypted by the application cannot be read while stored in external storage or in transit via I/O (e.g., across a network). Likewise, any corruption of signed data will be detected.

4. Enforcing Secure Computation

In this section, we show how Virtual Ghost is designed to prevent the operating system from violating the integrity or confidentiality of secure applications. Section 5 describes implementation details of this design.

4.1 Overview of Virtual Ghost

Our approach for protecting ghost memory is conceptually simple: we instrument (or "sandbox") memory access instructions in the kernel to ensure that they cannot access this memory. Virtual Ghost also instruments kernel code with control-flow integrity (CFI) checks to ensure that our sandboxing instrumentation is not bypassed [3, 38].

While sandboxing prevents attacks that attempt to access ghost memory directly, it does not prevent the other attacks described in Section 2.2. Our design additionally needs a way to restrict the interactions between the system software and both the hardware and applications. For example, our system must be able to verify (either statically or at runtime) that MMU operations will not alter the mapping of physical page frames used for ghost memory, and it must limit the types of changes that system software can make to interrupted program state.

To achieve these goals, our system needs a framework that provides both compiler analysis and instrumentation of operating system kernel code as well as a way of controlling interactions between the system software, the hardware, and applications. The Secure Virtual Architecture (SVA) framework [12, 13] meets these requirements, and we modify and extend it to support Virtual Ghost. A high-level overview of our approach, built on SVA, is shown in Figure 1.

4.2 Background: Secure Virtual Architecture

As Figure 1 shows, SVA interposes a compiler-based virtual machine (VM)[2] between the hardware and the operating system. All operating system software, including the kernel and device drivers, is compiled to the virtual instruction set implemented by SVA. The SVA VM translates code from the virtual instruction set to the native instruction set of the hardware, typically ahead-of-time, and caches and signs the translations (although it could also be done just-in-time while the kernel is running) [14].

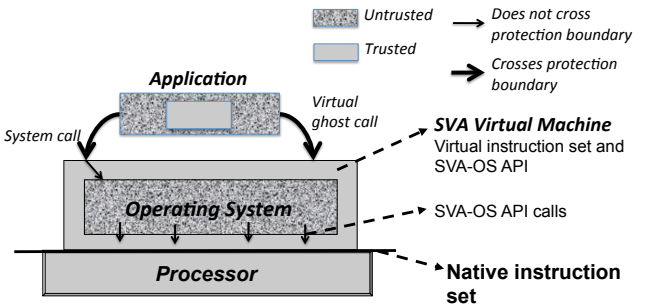

Figure 1. System Organization with Virtual Ghost

The core SVA virtual instruction set is based on the LLVM compiler intermediate representation (IR) [19], which is used in many popular systems (e.g., MacOS X, iOS, and Android) for static and/or just-in-time compilation. The LLVM IR enables efficient and precise compiler analyses at compile time, link time, install time, and run time for arbitrary languages [19].

SVA adds a set of instructions to LLVM called SVA-OS; these instructions replace the hardware-specific operations used by an OS to communicate with the hardware and to do low-level state manipulation [12–14]. SVA-OS handles operations such as saving and restoring processor state, signal handler dispatch, MMU configuration, and primitive reads and writes to I/O devices. System calls from an application are first fielded by SVA-OS, which saves processor state and performs minor bookkeeping, before passing the system call on to the OS as an ordinary function call. Modifying an OS to use the SVA-OS instructions is essentially like porting the OS to a new architecture, albeit with a clean, higher-level function interface that simplifies the port.

SVA-OS appears to the kernel as an ordinary run-time library; in particular, the OS can use these operations as direct function calls without crossing a protection boundary. The SVA compiler translates software from the virtual instruction set to the native instruction set. Moreover, the SVA-OS instructions are designed so that the SVA VM can perform run-time checks on all kernel-hardware interactions. *This combination of two capabilities in SVA is unique: it combines the compiler capabilities of a high-level language with the supervisory control of a hypervisor.*

Together, this combination allows SVA to enforce security policies on both application and kernel code [12, 13]. For

[2] We use the term "virtual machine" to mean a virtual architecture implemented via a translator and run-time system, similar to a language virtual machine, and not a guest operating system instance.

example, SVA has previously been used to provide memory safety to the Linux kernel, including the assurance that, even with dangling pointer and MMU configuration errors, SVA would enforce a strong set of safety guarantees and sound points-to analysis [12, 13]. In the present work, we do *not* require either memory safety or sound points-to analysis from SVA. Instead, we use the combination of SVA capabilities to provide secure computation and I/O with ghost memory.

4.3 Preventing Data Accesses in Memory

In this and the next few subsections, we discuss how Virtual Ghost addresses each of the five attack vectors described in Section 2.2.

4.3.1 Controlling Direct Memory Accesses

The SVM VM must perform two kinds of instrumentation when generating native code for the kernel (both the core kernel and dynamically loaded kernel modules). First, it must instrument loads and stores to prevent them from accessing ghost memory and the SVA VM internal memory. Recall that application pages are usually left mapped in the kernel's address space when entering the kernel on a system call, interrupt or trap. By instrumenting loads and stores, we avoid the need to unmap the ghost memory pages or (as Overshadow and Inktag do) to encrypt them before an OS access. Second, it must ensure that the instrumentation cannot be skipped via control-flow hijacking attacks [3].

By strategically aligning and sizing the SVA VM and ghost memory partitions, the load/store instrumentation can use simple bit-masking to ensure that a memory address is outside these partitions. The control-flow integrity instrumentation places checks on all returns and indirect calls to ensure that the computed address is a valid control-flow target. To support native code applications, our control-flow integrity checks also ensure that the address is within the kernel address space. A side benefit of our design is that the operating system kernel gets strong protection against control flow hijacking attacks.

4.3.2 MMU Protections

To prevent attacks that use illegal MMU page table entries, Virtual Ghost extends the MMU configuration instructions in SVA-OS to perform additional checks at run-time, which ensure that a new MMU configuration does not leave ghost memory accessible to the kernel. Specifically, Virtual Ghost does not permit the operating system to map physical page frames used by ghost memory into any virtual address. Virtual Ghost also prevents the operating system from modifying any virtual to physical page mapping for virtual addresses that belong to the ghost memory partition.

Virtual Ghost also protects other types of memory in addition to ghost memory. The SVA VM internal memory has the same restrictions as ghost memory. Native code is similarly protected; Virtual Ghost prevents native code pages from being remapped or made writable. This prevents the OS from bypassing the instrumentation or inserting arbitrary instructions into application code. Native code, unlike ghost memory or SVA internal memory, is made executable.

4.3.3 DMA Protections

Ghost Memory should never be the target of a legitimate DMA request. If Virtual Ghost can prevent DMA transfers to or from ghost physical page frames, then it can prevent DMA-based attacks.

SVA requires an IOMMU [4] and configures it to prevent I/O devices from writing into the SVA VM memory [12, 13]. It further provides special I/O instructions for accessing processor I/O ports and memory-mapped I/O devices. Both SVA and Virtual Ghost must prevent the OS from reconfiguring the IOMMU to expose ghost memory to DMA transfers. How this is done depends on whether the hardware uses I/O ports or memory-mapped I/O to configure the IOMMU. If the hardware uses I/O ports, then SVA uses run-time checks within the I/O port read and write instructions [12]. If the hardware uses memory-mapped I/O, then SVA and Virtual Ghost simply use the MMU checks described above to prevent the memory-mapped physical pages of the IOMMU device from being mapped into the kernel or user-space virtual memory; instead, it is only mapped into the SVA VM memory, and the system software needs to use the I/O instructions to access it.

4.4 Preventing Data Accesses During I/O

Virtual Ghost relies on application-controlled encryption and hashing to prevent data theft or tampering during I/O operations, and uses automatic encryption and hashing done by the Virtual Ghost VM for page swapping, as described in Section 3.3.

The main design challenge is to find a way to start the application with encryption keys that cannot be compromised by a hostile OS. As noted earlier, Virtual Ghost maintains a public/private key pair for each system on which it is installed. We assume that a Trusted Platform Module (TPM) coprocessor is available; the storage key held in the TPM is used to encrypt and decrypt the private key used by Virtual Ghost. The application's object code file format is extended to contain a section for the application encryption keys, which are encrypted with the Virtual Ghost public key. Each time the application is started up, Virtual Ghost decrypts the encryption key section with the Virtual Ghost private key and places it into its internal SVA VM memory before transferring control to the application. An application can use the `sva.getKey()` instruction to retrieve the key from the Virtual Ghost VM and store a copy in its ghost memory. If an application requires multiple private keys, e.g., for communicating with different clients, it can use its initial private key to encrypt and save these additional keys in persistent storage. Thus, Virtual Ghost and the underlying hardware together enable a *chain of trust* that cannot be compromised by the OS or other untrusted applications:

TPM storage key ⇒ Virtual Ghost private key ⇒
Application private key ⇒ Additional application keys.

The use of a separate section in the object code format allows easy modification of the keys by trusted tools. For example, a software distributor can place unique keys in each copy of the software before sending the software to a user. Similarly, a system administrator could update the keys in an application when the system is in single-user mode booted from trusted media.

4.5 Preventing Code Modification Attacks

Virtual Ghost prevents the operating system from loading incorrect code for an application, modifying native code after it is loaded, or repurposing existing native code instruction sequences for unintentional execution (e.g., via return-oriented programming).

To prevent the system software from loading the wrong code for an application, Virtual Ghost assumes that the application is installed by a trusted system administrator (who may be local or remote). The application's executable, including the embedded, encrypted application key described in Section 3.3, is signed by Virtual Ghost's public key when the application binary is installed. If the system software attempts to load different application code with the application's key, Virtual Ghost refuses to prepare the native code for execution. The same mechanism is used to authenticate dynamically loaded native code libraries before they are linked in to the application.

To prevent native code modification in memory, Virtual Ghost ensures that the MMU maps all native code into non-writable virtual addresses. It also ensures that the OS does not map new physical pages into virtual page frames that are in use for OS, SVA-OS, or application code segments.

Virtual Ghost prevents repurposing existing instruction sequences or functions simply because the Control Flow Integrity (CFI) enforcement (for ensuring that sandboxing instructions are not bypassed) prevents all transfers of control not predicted by the compiler. For example, a buffer overflow in the kernel could overwrite a function pointer, but if an indirect function call using that function pointer attempted to go to any location other than one of the predicted callees of the function, the CFI instrumentation would detect that and terminate the execution of the kernel thread.

Finally, Virtual Ghost also prevents the OS from subverting signal dispatch to the application, e.g., by executing arbitrary code instead of the signal handler in the application context. Although this is essentially a code modification attack, the mechanisms used to defend against this are based on protecting interrupted program state, described next.

4.6 Protecting Interrupted Program State

An attacker may attempt to read interrupted program state (the program state that is saved on a system call, interrupt, or trap) to glean confidential information. Alternatively, she may modify the interrupted program state to mount a control-hijack or non-control data attack on the application. Such an attack could trick the application into copying data from ghost memory into traditional memory where the operating system can read or write it. Note that such an attack works even on a completely memory-safe program. We have implemented such an attack as described in Section 7.

The SVA framework (and hence Virtual Ghost) calls this interrupted program state the *Interrupt Context*. The creation and maintenance of the Interrupt Context is performed by the SVA virtual machine. While most systems save the Interrupt Context on the kernel stack, Virtual Ghost saves the Interrupt Context within the SVA VM internal memory. Virtual Ghost also zeros out registers (except registers passing system call arguments for system calls) after saving the Interrupt Context but before handing control over to the OS. With these two features, the OS is unable to read or modify the Interrupt Context directly or glean its contents from examining current processor state.

The OS does need to make controlled changes to the Interrupt Context. For example, process creation needs to initialize a new Interrupt Context to return from the `fork()` system call [9], and signal handler dispatch needs to modify the application program counter and stack so that a return-from-interrupt instruction will cause the application to start executing its signal handler [9, 25].

SVA provides instructions for manipulating the Interrupt Context, and Virtual Ghost enhances the checks on them to ensure that they do not modify the Interrupt Context in an unsafe manner, as explained below.

4.6.1 Secure Signal Handler Dispatch

Virtual Ghost provides instructions for implementing secure signal handler dispatch. Signal handler dispatch requires saving the Interrupt Context, modifying the interrupted program state so that the signal handler is invoked when the interrupted application is resumed, and then reloading the saved Interrupt Context back into the interrupted program state buffer when the `sigreturn()` system call is called.

Virtual Ghost pushes and pops a copy of the Interrupt Context on and off a per-thread stack within the SVA VM internal memory, whereas the original SVA let this live on the kernel stack for unprivileged applications [12, 13] (because SVA did not aim to protect application control flow from the OS, whereas Virtual Ghost does). This enhancement to the original SVA design ensures that the OS cannot modify the saved state *and* ensures that the OS restores the correct state within the correct thread context.

SVA provides an operation, `sva.ipush.function()`, which the operating system can use to modify an Interrupt Context so that the interrupted program starts execution in a signal handler. The OS passes in a pointer to the application function to call and the arguments to pass to this function, and Virtual Ghost modifies the Interrupt Context on the operating system's behalf. For efficiency, we allow

`sva.ipush.function()` to modify the application stack even though the stack may be within ghost memory. Since Virtual Ghost only adds a function frame to the stack, it cannot read or overwrite data that the application is using.

To ensure that the specified function is permissible, Virtual Ghost provides an operation, `sva.permitFunction()`, which the *application* must use to register a list of functions that can be "pushed"; `sva.ipush.function()` refuses to push a function that is not in this list. To simplify application development, we provide wrappers for the `signal` and `sigaction` system calls, which register the signal handlers transparently, without needing to modify the application.

4.6.2 Secure Process Creation

A thread is composed of two pieces of state: an Interrupt Context, which is the state of an interrupted user-space program, and a kernel-level processor state that Virtual Ghost calls *Thread State* that represents the state of the thread before it was taken off the CPU.[3] Creating a new thread requires creating a new Interrupt Context and Thread State.

While commodity operating systems create these structures manually, Virtual Ghost provides a single function, `sva.newstate()`, to create these two pieces of state. This function creates these new state structures within the SVA VM internal memory to prevent tampering by the system software. The newly created Interrupt Context is a clone of the Interrupt Context of the current thread. The new Thread State is initialized so that, on the next context switch, it begins executing in a function specified by the operating system. In order to maintain kernel control-flow integrity, Virtual Ghost verifies that the specified function is the entry point of a kernel function.

Any ghost memory belonging to the current thread will also belong to the new thread; this transparently makes it appear that ghost memory is mapped as shared memory among all threads and processes within an application.

Executing a new program (e.g., via the `execve()` system call) also requires reinitializing the Interrupt Context [9]: the program counter and stack pointer must be changed to execute the newly loaded program image, and, in the case of the first user-space process on the system (e.g., `init`), the processor privilege level must be changed from privileged to unprivileged. Virtual Ghost also ensures that the program counter points to the entry of a program that has previously been copied into SVA VM memory. (Section 4.5 describes how Virtual Ghost ensures that this program is not tampered with by the OS before or during this copy operation.)

Finally, any ghost memory associated with the interrupted program is unmapped when the Interrupt Context is reinitialized. This ensures that newly loaded program code does not have access to the ghost memory belonging to the previously executing program.

[3] SVA divided Thread State into Integer State and Floating Point State, as an optimization. Virtual Ghost does not do that.

4.7 Mitigating System Service Attacks

System service attacks like those described in Section 2.2.5 are not yet well understood [10]. However, Virtual Ghost provides some protection against such attacks that are known.

First, Virtual Ghost employs an enhanced C/C++ compiler that instruments system calls in ghosting applications to ensure that pointers passed into or returned by the operating system are not pointing into ghost memory. This instrumentation prevents accidental overwrites of ghost memory. This protects private data and prevents stack-smashing attacks (because the stack will be located in ghost memory).

Second, the Virtual Ghost VM provides an instruction for generating random numbers. The random number generator is built into the Virtual Ghost VM and can be trusted by applications for generating random numbers. This defeats Iago attacks that feed non-random numbers to applications [10].

While these protections are far from comprehensive, they offer protection against existing system service attacks.

5. Implementation

To create Virtual Ghost, we implemented a new version of the SVA-OS instructions and run-time for 64-bit x86 processors, reusing code from the original 32-bit, single-core implementation [12, 13]. Our implementation runs 64-bit code only. The Virtual Ghost instructions are implemented as a run-time library that is linked into the kernel.

We ported FreeBSD 9.0 to the Virtual Ghost/SVA-OS instruction set. We used FreeBSD instead of Linux (which we had used before with SVA-OS) because FreeBSD compiles with the LLVM compiler out of the box.

Virtual Ghost builds on, modifies, and adds a number of SVA-OS operations to implement the design described. We briefly summarize them here, omitting SVA-OS features that are needed for hosting a kernel and protecting SVA itself, but not otherwise used by Virtual Ghost for application security. We first briefly describe the compiler instrumentation and end with a brief summary of features that are not yet implemented.

Compiler Instrumentation: The load/store and CFI instrumentation is implemented as two new passes in the LLVM 3.1 compiler infrastructure. The load/store instrumentation pass transforms code at the LLVM IR level; it instruments mid-level IR loads, stores, atomic operations, and calls to `memcpy()`. The CFI instrumentation pass is an updated version of the pass written by Zeng et. al. [38] that works on x86_64 machine code. It analyzes the machine code that LLVM 3.1 generates and adds in the necessary CFI labels and checks. It also masks the target address to ensure that it is not a user-space address. We modified the Clang/LLVM 3.1 compiler to use these instrumentation passes when compiling kernel code. To avoid link-time interprocedural analysis, which would be needed for precise call graph construction, our CFI instrumentation uses a very conservative call graph: we use one label both for call sites (i.e., the targets

of returns) and for the first address of every function. While conservative, this call graph allows us to measure the performance overheads and should suffice for stopping advanced control-data attacks.

We placed the ghost memory partition into an unused 512 GB portion of the address space (`0xffffff0000000000` – `0xffffff8000000000`). The load/store instrumentation determines whether the address is greater than or equal to `0xffffff0000000000` and, if so, ORs it with 2^{39} to ensure that the address will not access ghost memory. While our design would normally use some of this 512 GB partition for SVA internal memory, we opted to leave the SVA internal memory within the kernel's data segment; we added new instrumentation to kernel code, which changes an address to zero before a load, store, or `memcpy()` if it is within the SVA internal memory. This additional instrumentation adds some overhead but simplifies development.

To defend against Iago attacks through the `mmap` system call, a separate mid-level LLVM IR instrumentation pass performs identical bit-masking instrumentation to the return values of `mmap()` system calls for user-space application code.[4] This instrumentation moves any pointer returned by the kernel that points into ghost memory out of ghost memory. In this way, Iago attacks using `mmap()` [10] cannot trick an application into writing data into its own ghost memory. If an application stores its stack and function pointers in ghost memory, then our defense should prevent Iago attacks from subverting application control flow integrity.

Memory Management: SVA-OS provides operations that the kernel uses to insert and remove page table entries at each level of a multi-level page table and ensures that they do not violate internal consistency of the page tables or the integrity of SVA code and data. Virtual Ghost augments these operations to also enforce the MMU mapping constraints described in Section 4.3.2.

Launching Execution: Virtual Ghost provides the operation, `sva.reinit.icontext()`, to *reinitialize* an Interrupt Context with new application state. This reinitialization will modify the program counter and stack pointer in the Interrupt Context so that, when resumed, the program will begin executing new code.

Virtual Ghost uses the x86_64 Interrupt Stack Table (IST) feature [2] to protect the Interrupt Context. This feature instructs the processor to always change the stack pointer to a known location on traps or interrupts regardless of whether a change in privilege mode occurs. This allows the SVA VM to direct the processor to save interrupted program state (such as the program counter) within the SVA VM internal memory so that it is never accessible to the operating system.

Signal Delivery: Virtual Ghost implements the `sva.icontext.save()` and `sva.icontext.load()` instructions to save and restore the Interrupt Context before and after signal handler dispatch. These instructions save the Interrupt Context within SVA memory to ensure that the OS cannot read or write it directly.

What Is Not Yet Implemented: Our implementation so far does not include a few features described previously. (1) While explicit I/O is supported, the key management functions are only partially implemented: Virtual Ghost does not provide a public/private key pair; does not use a TPM; and application-specific keys are not embedded in application binaries (instead, a 128-bit AES application key is hard-coded into SVA-OS for our experiments). (2) Swapping of ghost memory is not implemented. (3) The system does not yet include the DMA protections. We believe that IOMMU configuration is rare, and therefore, the extra protections for DMA should not add noticeable overhead. (4) Finally, some operations in the FreeBSD kernel are still handled by inline assembly code (e.g., memory-mapped I/O loads and stores), but we do not believe that these unported operations will significantly affect performance once they are ported properly.

Trusted Computing Base: Since the Virtual Ghost implementation is missing a few features, measuring the size of its Trusted Computing Base (TCB) is premature. However, the vast majority of the functionality has been implemented, and the current code size is indicative of the approximate size of the TCB. Virtual Ghost currently includes only 5,344 source lines of code (ignoring comments, whitespace, etc.). This count includes the SVA VM run-time system and the passes that we added to the compiler to enforce our security guarantees. Overall, we believe that the complexity and attack surface of Virtual Ghost are far smaller than modern production hypervisors like XenServer but approximately comparable to a minimal hypervisor.

6. Securing OpenSSH

To demonstrate that our system can secure real applications, we modified three programs from the OpenSSH 6.2p1 application suite to use ghost memory: `ssh`, `ssh-keygen`, and `ssh-agent`. The `ssh-keygen` program generates public/private key pairs which `ssh` can use for password-less authentication to remote systems. The `ssh-agent` server stores private encryption keys which the `ssh` client may use for public/private key authentication. We used a single private "application key" for all three programs so that they could share encrypted files.

We modified `ssh-keygen` to encrypt all the private authentication key files it generates with the application key; our `ssh` client decrypts the authentication keys with the application private key upon startup and places them, along with all other heap objects, into ghost memory. Since the OS cannot gain access to the application key, it cannot decrypt the authentication keys that are stored on disk, and it cannot read the cleartext versions out of `ssh`'s or `ssh-keygen`'s ghost memory.

[4] An alternative implementation method would be to implement a C library wrapper function for `mmap()`.

We modified the FreeBSD C library so that the heap allocator functions (`malloc()`, `calloc()`, `realloc()`) allocate heap objects in ghost memory instead of in traditional memory; the changes generate a 216 line patch. To ease porting, we wrote a 667-line system call wrapper library that copies data between ghost memory and traditional memory as necessary. This wrapper library also provides wrappers for `signal()` and `sigaction()` that register the signal handler functions with Virtual Ghost before calling the kernel's `signal()` and `sigaction()` system calls. The compiler and linker did not always resolve system calls to our wrapper functions properly, so we made some manual modifications to the programs. We also modified the programs to use traditional memory (allocated via `mmap()`) to store the results of data to be sent to `stdout` and `stderr` to reduce copying overhead. In total, our changes to OpenSSH can be applied with a patch that adds 812 and deletes 68 lines of code (OpenSSH contains 9,230 source lines of code).

We tested our applications on the system used for our experiments (see Section 8). We used `ssh-keygen` to generate a new private and public key for DSA authentication; the generated private key was encrypted while the public key was not encrypted. We then installed the public key on another system and used `ssh` to log into that system using DSA authentication.

For `ssh-agent`, we added code to place a secret string within a heap-allocated memory buffer. The rootkit attacks described in Section 7 attempt to read this secret string. The goal of adding this secret string is that `ssh-agent` treats it identically to an encryption key (it can use the string internally but never outputs it to another program). We used a secret string as it is easier to find in memory and easier to identify as the data for which our attack searches.

Our enhanced OpenSSH application suite demonstrates that Virtual Ghost can provide security-critical applications with in-memory secrets (e.g., the keys held by `ssh-agent`) and secure, long-term storage (e.g., the authentication keys created by `ssh-keygen` and read by `ssh`). It also demonstrates that a suite of cooperating applications can securely share data on a hostile operating system via a shared application key.

7. Security Experiments

To evaluate the security of our system, we built a malicious kernel module that attempts to steal sensitive information from a victim process. This module, based on the code from Joseph Kong's book [18], can be configured by a non-privileged user to mount one of two possible attacks – direct memory access or code injection – on a given victim process. The malicious module replaces the function that handles the `read()` system call and executes the attack as the victim process reads data from a file descriptor.

In the first attack, the malicious module attempts to directly read the data from the victim memory and print it to the system log.

In the second attack, the malicious module attempts to make the victim process write the confidential data out to a file. The attack first opens the file to which the data should be written, allocates memory in the process's address space via `mmap()`, and copies exploit code into the memory buffer. The attack then sets up a signal handler for the victim process that calls the exploit code. The malicious module then sends a signal to the victim process, triggering the exploit code to run in the signal handler. The exploit code copies the data into the `mmap`'ed memory and executes a `write()` system call to write the secret data out to the file opened by the malicious module.

We used both attacks on our ssh-agent program, described in Section 6. When we install the malicious module without instrumenting its code and run ssh-agent with `malloc()` configured to allocate memory objects in traditional memory, both attacks succeed.

We then recompiled our malicious module using our modified Clang compiler to insert the instrumentation required for Virtual Ghost. We reran both attacks on ssh-agent with `malloc()` configured to allocate memory objects in ghost memory. The first attack fails because the load/store instrumentation changes the pointer in the malicious module to point outside of ghost memory; the kernel simply reads unknown data out of its own address space. The second attack is thwarted because `sva.ipush.function()` recognizes that the exploit code isn't one of the functions registered as a valid target of `sva.ipush.function()`.

Note that a number of other possible attacks from within kernel code *are simply not expressible in our system*, e.g., anything that requires using assembly code in the kernel (such as to access CPU registers), or manipulating the application stack frames, or modifying interrupted application state when saved in memory. The second attack above illustrates that much more sophisticated multi-step exploits are needed to get past the basic protections against using assembly code or accessing values saved in SVA VM memory or directly reading or writing application code and data (as in the first attack). Virtual Ghost is successfully able to thwart even this sophisticated attack.

8. Performance Experiments

We ran our performance experiments on a Dell Precision T1650 workstation with an Intel® Core™ i7-3770 processor at 3.4 GHz with 8 MB of cache, 16 GB of RAM, an integrated PCIE Gigabit Ethernet card, a 7200 RPM 6 Gb/s 500 GB SATA hard drive, and a 256 GB Solid State Drive (SSD). Files in `/usr` were stored on the SSD. For network experiments, we used a dedicated Gigabit Ethernet network. The client machine was an iMac with a 4-core hyper-threaded Intel® Core™ i7 processor at 2.6 GHz with 8 GB of RAM.

We evaluated our system's performance on microbenchmarks as well as a few applications. We used microbenchmarks to see how Virtual Ghost affects primitive OS operations, `thttpd` and `sshd` for network applications, and Postmark [30] for a file system intensive program.

We conducted our experiments by booting the machine into single-user mode to minimize noise from other processes running on the system. Our baseline, unless otherwise noted, is a native FreeBSD kernel compiled with the LLVM 3.1 compiler and configured identically to our Virtual Ghost FreeBSD kernel.

8.1 Microbenchmarks

In order to understand how our system affects basic OS performance, we measured the latency of various system calls using LMBench [26] from the FreeBSD 9.0 ports tree. For those benchmarks that can be configured to run the test for a specified number of iterations, we used 1,000 iterations. Additionally, we ran each benchmark 10 times.

Test	Native	Virtual Ghost	Overhead	InkTag
null syscall	0.091	0.355	3.90x	55.8x
open/close	2.01	9.70	4.83x	7.95x
mmap	7.06	33.2	4.70x	9.94x
page fault	31.8	36.7	1.15x	7.50x
signal handler install	0.168	0.545	3.24x	-
signal handler delivery	1.27	2.05	1.61x	-
fork + exit	63.7	283	4.40x	5.74x
fork + exec	101	422	4.20x	3.04x
select	3.05	10.3	3.40x	-

Table 2. LMBench Results. Time in Microseconds.

File Size	Native	Virtual Ghost	Overhead
0 KB	166,846	36,164	4.61x
1 KB	116,668	25,817	4.52x
4 KB	116,657	25,806	4.52x
10 KB	110,842	25,042	4.43x

Table 3. LMBench: Files Deleted Per Second.

File Size	Native	Virtual Ghost	Overhead
0 KB	156,276	33,777	4.63x
1 KB	97,839	18,796	5.21x
4 KB	97,102	18,725	5.19x
10 KB	85,319	18,095	4.71x

Table 4. LMBench: Files Created Per Second.

As Table 2 shows, our system can add considerable overhead to individual operations. System call entry increases by 3.9 times. We compare these relative slowdowns with InkTag, which has reported LMBench results as well [16]. Our slowdowns are nearly identical to or better than InkTag on 5/7 microbenchmarks: all except `exec()` and file deletion/creation. Our file deletion/creation overheads (shown in Tables 3 and 4) average 4.52x and 4.94x, respectively, across all file sizes, which is slower than InkTag. System calls and page faults, two of the most performance critical OS operations, are both considerably faster on Virtual Ghost than on InkTag.

While Virtual Ghost adds overhead, it provides double benefits for the cost: in addition to ghost memory, it provides protection to the OS itself via control flow integrity.

8.2 Web Server Performance

We used a statically linked, non-ghosting version of the `thttpd` web server [29] to measure the impact our system had on a web server. We used ApacheBench [1] to measure the bandwidth of transferring files between 1 KB and 1 MB in size. Each file was generated by collecting random data from the `/dev/random` device and stored on the SSD. We configured ApacheBench to make 100 simultaneous connections and to perform 10,000 requests for the file for each run of the experiment. We ran each experiment 20 times.

Figure 2 shows the mean performance of transferring a file of each size and displays the standard deviation as error bars. The data show that the impact of Virtual Ghost on the Web transfer bandwidth is negligible.

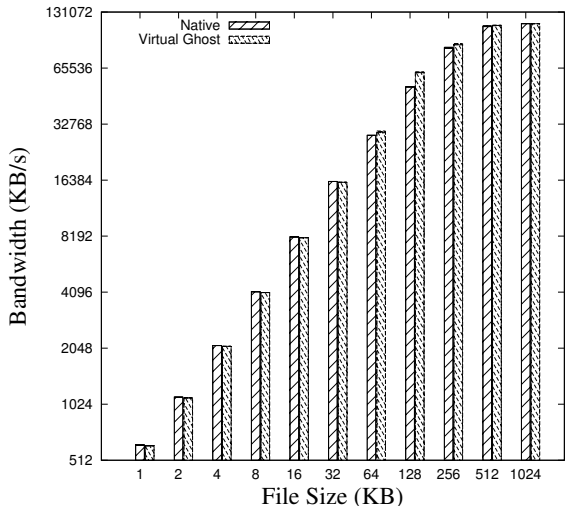

Figure 2. Average Bandwidth of thttpd

8.3 OpenSSH Performance

To study secure bulk data transfer performance, we measured the bandwidth achieved when transferring files of various sizes using the OpenSSH server and client [36]. We measured the performance of the `sshd` server without ghosting and our ghosting `ssh` client described in Section 6. Files were created using the same means described in Section 8.2.

8.3.1 Server Performance Without Ghosting

We ran the pre-installed OpenSSH server on our test machine and used the standard Mac OS X OpenSSH `scp` client to measure the bandwidth achieved when transferring files.

We repeated each experiment 20 times and report standard deviation bars. The baseline system is the original FreeBSD 9.0 kernel compiled with Clang and configured identically to our Virtual Ghost FreeBSD kernel.

Figure 3 shows the mean bandwidth for the baseline system and Virtual Ghost. We observe bandwidth reductions of 23% on average, with a worst case of 45%, and negligible slowdowns for large file sizes.

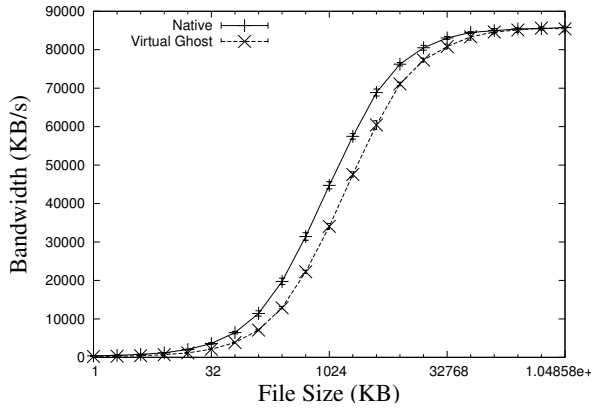

Figure 3. SSH Server Average Transfer Rate

8.4 Client Performance With Ghosting

To measure the effect of using ghost memory, we measured the average bandwidth of transferring file sizes of 1 KB to 1 MB using both the unmodified OpenSSH `ssh` client and our ghosting `ssh` client described in Section 6. We transferred files by having `ssh` run the `cat` command on the file on the server. We ran both on the Virtual Ghost FreeBSD kernel to isolate the performance differences in using ghost memory. We transferred each file 20 times. Figure 4 reports the average of the 20 runs for each file size as well as the standard deviation using error bars. (The numbers differ from those in Figure 3 because this experiment ran the ssh client on Virtual Ghost, instead of the sshd server.)

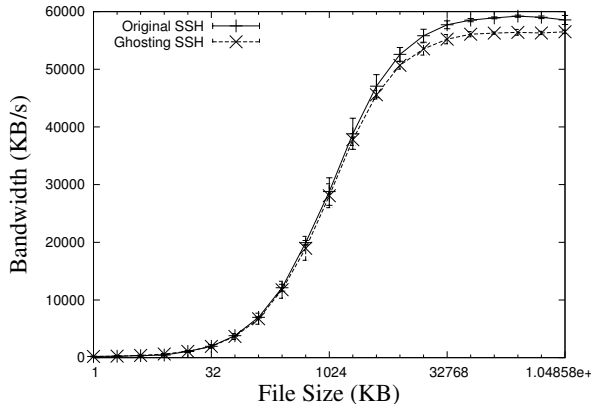

Figure 4. Ghosting SSH Client Average Transfer Rate

As Figure 4 shows, the performance difference between using traditional memory, which is accessible to the OS, and ghost memory is small. The maximum reduction in bandwidth by the ghosting `ssh` client is 5%.

8.5 Postmark Performance

In order to test a file system intensive benchmark, we ran Postmark [30]. Postmark mimics the behavior of a mail server and exercises the file system significantly.

Native (s)	Std. Dev.	Virtual Ghost (s)	Std. Dev.	Overhead
14.30	0.46	67.50	0.50	4.72x

Table 5. Postmark Results

We configured Postmark to use 500 base files with sizes ranging from 500 bytes to 9.77 KB with 512 byte block sizes. The read/append and create/delete biases were set to 5, and we configured Postmark to use buffered file I/O. All files were stored on the SSD. Each run of the experiment performed 500,000 transactions.

We ran the experiment 20 times on both the native FreeBSD kernel and the Virtual Ghost system. Table 5 shows the results. As Postmark is dominated by file operations, the slowdown of 4.7x is similar to the LMBench open/close system call overhead of 4.8x. Due to its file system intensive behavior, Postmark represents the worst case for the application benchmarks we tried.

9. Related Work

Several previous systems attempt to protect an application's code and data from a malicious operating system. Systems such as Overshadow [11, 28], SP3 [37], and InkTag [16] build on a full-scale commercial hypervisor (e.g., VMWare Server or XenServer). The hypervisor presents an encrypted view of the application's memory to the OS and uses digital signing to detect corruption of the physical pages caused by the OS. These systems do *not* prevent the OS from reading or modifying the encrypted pages. To simplify porting legacy applications, such systems include a shim library between the application and the operating system that encrypts and decrypts data for system call communication.

Hypervisor-based approaches offer a high level of compatibility with existing applications and operating systems, but suffer high performance overhead. Additionally, they add overhead to the common case (when the kernel reads/writes application memory correctly via the system call API). They also do not provide additional security to the operating system and do not compose as cleanly with other kernel security solutions that use hypervisor-based approaches.

Virtual Ghost presents a different point in the design space for protecting applications from an untrusted OS. First, it uses compiler instrumentation ("sandboxing" and control-flow integrity) instead of page protections to protect both application ghost memory pages as well as its own code and metadata from the OS (similar to systems such as SPIN [8], JavaOS [33], and Singularity [15, 17]).

Second, it completely prevents the OS from reading and writing ghost memory pages rather than allowing access to the pages in encrypted form. Third, although Virtual Ghost introduces a hardware abstraction layer (SVA-OS) that is somewhat similar to a (minimal) hypervisor, SVA-OS does not have higher privilege than the OS; instead, it appears as a library of functions that the OS kernel code can call directly. Fourth, system calls from the secure application to the OS need not incur encryption and hashing overhead for non-secure data. Fifth, Virtual Ghost gives the application considerably more control over the choices of encryption and hashing keys, and over what subset of data is protected. Finally, the compiler approach hardens the OS against external exploits because it prevents both injected code and (by blocking control-flow hijacking) exploits that use existing code, such as return-oriented programming or jump-oriented programming. Moreover, the compiler approach can be directly extended to provide other compiler-enforced security policies such as comprehensive memory safety [12, 13].

In addition to isolation, InkTag [16] also provides some valuable usability improvements for secure applications, including services for configuring access control to files. These features could be provided by SVA-OS in very similar ways, at the cost of a non-trivial relative increase in the TCB size.

Other recent efforts, namely TrustVisor [22], Flicker [23], and Memoir [27], provide special purpose hypervisors that enable secure execution and data secrecy for pieces of application logic. They obtain isolated execution of code and data secrecy via hardware virtualization. These approaches prevent the protected code from interacting with the system, which limits the size of code regions protected, and data must be sealed via trusted computing modules between successive invocations of the secure code. In contrast, Virtual Ghost provides continuous isolation of code and data from the OS without the need for secure storage or monopolizing system wide execution of code.

Several systems provide hardware support to isolate applications from the environment, including the OS. The XOM processor used by XOMOS [20] encrypts all instructions and data transferred to or from memory and can detect tampering of the code and data, which enables a secure application on XOMOS to trust nothing but the processor (not even the OS or the memory system). HyperSentry [7] uses server-class hardware for system management (IPMI and SMM) to perform stealthy measurements of a hypervisor without using software at a higher privilege level, while protecting the measurements from the hypervisor. These mechanisms are designed for infrequent operations, not for more extensive computation and I/O.

ARM Security Extensions (aka TrustZone) [6] create two virtual cores, operating as two isolated "worlds" called the Secure World and the Normal World, on a single physical core. Applications running in the Secure World are completely isolated from an OS and applications running in the Normal World. Secure World applications can use peripheral devices such as a keyboard or display securely. Virtual Ghost and any other pure-software scheme would need complex additional software to protect keyboard and display I/O. Intel Software Guard eEtensions (SGX) provide isolated execution zones called enclaves that are protected from privileged software access including VMMs, OS, and BIOS [24]. The enclave is protected by ISA extensions and hardware access control mechanisms and is similar to Virtual Ghost in that it protects memory regions from the OS. Additionally, SGX provides both trusted computing measurement, sealing, and attestation mechanisms [5]. Unlike TrustZones and SGX, Virtual Ghost requires no architectural modifications and could provide a secure execution environment on systems that lack such hardware support.

10. Future Work

We have several directions for future work. First, we plan to investigate how applications can use Virtual Ghost's features along with cryptographic protocols to protect themselves from sophisticated attacks. For example, how should applications ensure that the OS does not perform replay attacks by providing older versions of previously encrypted files? How should applications share keys and authenticate themselves to each other? Second, while Virtual Ghost provides applications with great flexibility in protecting their data, taking advantage of this flexibility can be tedious. We plan to investigate library support that will make writing secure applications with Virtual Ghost more convenient.

11. Conclusion

In this paper, we present Virtual Ghost, which provides application security in the face of untrusted operating systems. Virtual Ghost does not require a higher-privilege layer, as hypervisors do; instead, it applies compiler instrumentation combined with runtime checks on operating system code to provide ghost memory to applications. We ported a suite of real-world applications to use ghost memory and found that Virtual Ghost protected the applications from advanced kernel-level malware. We observe comparable performance, and much better in some instances, than the state of the art in secure execution environments, while also providing an additional layer of protection against external attacks on the operating system kernel itself.

Acknowledgments

The authors would like to thank Bin Zeng, Gang Tan, and Greg Morrisett for sharing their x86 CFI code with us. The authors also thank the anonymous reviewers for their comments and feedback.

This material is based on work supported by the AFOSR under MURI award FA9550-09-1-0539, and by NSF Grant CNS 07-09122.

References

[1] Apachebench: A complete benchmarking and regression testing suite. http://freshmeat.net/projects/ apachebench/, July 2003.

[2] *Intel 64 and IA-32 architectures software developer's manual*, volume 3. Intel, 2012.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13:4:1–4:40, November 2009.

[4] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: System programming, September 2006.

[5] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM, 2013.

[6] ARM Limited. ARM security technology: Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[7] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[8] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, pages 267–284, Copper Mountain, CO, USA, 1995.

[9] D. P. Bovet and M. Cesati. *Understanding the LINUX Kernel*. O'Reilly, Sebastopol, CA, 2^{nd} edition, 2003.

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

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

[12] J. Criswell, N. Geoffray, and V. Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.

[13] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, pages 351–366, Stevenson, WA, USA, October 2007.

[14] J. Criswell, B. Monroe, and V. Adve. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, Boston, MA, USA, June 2006.

[15] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys*, 2006.

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

[17] G. C. Hunt and J. R. Larus. Singularity Design Motivation (Singularity Technical Report 1). Technical Report MSR-TR-2004-105, Microsoft Research, Dec 2004.

[18] J. Kong. *Designing BSD Rootkits*. No Starch Press, San Francisco, CA, USA, 2007.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Conf. on Code Generation and Optimization*, pages 75–88, San Jose, CA, USA, Mar 2004.

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

[21] LMH. Month of kernel bugs (MoKB) archive, 2006. http://projects.info-pull.com/mokb/.

[22] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

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

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

[25] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, Inc., Redwood City, CA, 1996.

[26] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[27] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the 2011 IEEE Symposium on*

Security and Privacy, SP '11, pages 379–394, Washington, DC, USA, 2011. IEEE Computer Society.

[28] D. R. K. Ports and T. Garfinkel. Towards application security on untrusted operating systems. In *Proceedings of the 3rd conference on Hot topics in security*, HOTSEC'08, pages 1:1–1:7, Berkeley, CA, USA, 2008. USENIX Association.

[29] J. Poskanze. thttpd - tiny/turbo/throttling http server, 2000. http://www.acme.com/software/thttpd.

[30] Postmark. Email delivery for web apps, July 2013.

[31] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[32] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[33] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System.* Addison-Wesley, Reading, MA, USA, 1999.

[34] A. Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[35] Solar Designer. return-to-libc attack, August 1997. http://www.securityfocus.com/archive/1/7480.

[36] The OpenBSD Project. Openssh, 2006. http://www.openssh.com.

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

[38] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM.