MiniBox: A Two-Way Sandbox for x86 Native Code

Yanlin Li, Adrian Perrig, Jonathan McCune, James Newsome, Brandon Baker, Will Drewry

February 21, 2014

CMU-CyLab-14-001

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

MiniBox: A Two-Way Sandbox for x86 Native Code

Yanlin Li
CyLab/CMU
yanlli@cmu.edu

Adrian Perrig
CyLab/CMU
adrian.perrig@inf.ethz.ch

Jonathan McCune
Google, Inc.
jonmccune@google.com

James Newsome
Google, Inc.
jnewsome@google.com

Brandon Baker
Google, Inc.
bsb@google.com

Will Drewry
Google, Inc.
drewry@google.com

Abstract

This paper presents MiniBox, the first two-way sandbox for x86 native code. MiniBox not only isolates the memory space between OS protection modules and an application, but also provides a minimized and secure communication interface between OS protection modules and the application. MiniBox is cross-platform and can be applied in Platform-as-a-Service (PaaS) cloud computing to provide two-way protection between a customer's application and the cloud platform OS. We implement a prototype of MiniBox on both Intel and AMD multi-core systems and port several applications to MiniBox. Evaluation results show that MiniBox is efficient and practical.

1 Introduction

Platform-as-a-Service (PaaS) is one of the most widely commercialized forms of cloud computing. In 2012, there were already 1 million active applications on Google App Engine [16]. On PaaS cloud computing, it is critical to protect the cloud platform from the large number of *untrusted* applications sent by customers. Thus, virtualized infrastructure (e.g., Xen [8]) is deployed to isolate customers' applications. However, the virtualized infrastructure has a large Trusted Computing Base (TCB) and may have vulnerabilities [48, 31]. Once a guest OS is compromised, the malicious guest OS may in turn compromise the hypervisor or other guest OSes. To better protect the cloud platform, a sandbox (e.g., Java sandbox [21]) is deployed on cloud platforms to protect the guest OS from being compromised by *untrusted* applications. However, security on PaaS is not only a concern for cloud providers but also a concern for cloud customers. As shown in the sandbox architecture in Figure 1, current sandbox technology provides only one-way protection, which protects the OS from an *untrusted* application. The security-sensitive portion of an application, which is *untrusted* for cloud providers is completely exposed to malicious code on the OS. Also, current sandboxes expose a large interface to the *untrusted* applications, and may have vulnerabilities that malicious applications can exploit. A malicious application that breaks out of the sandbox may compromise the OS.

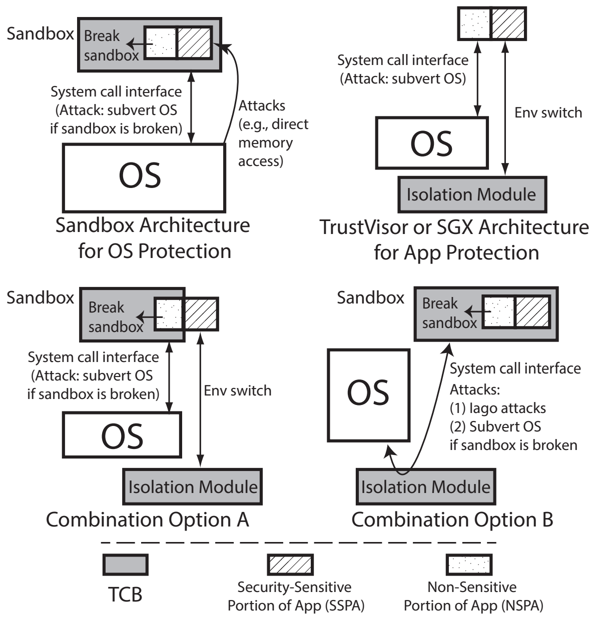
Sandbox
Break sandbox
System call interface (Attack: subvert OS if sandbox is broken)
Attacks (e.g., direct memory access)
OS
Sandbox Architecture for OS Protection
System call interface (Attack: subvert OS)
Env switch
OS
Isolation Module
TrustVisor or SGX Architecture for App Protection
Sandbox
Break sandbox
System call interface (Attack: subvert OS if sandbox is broken)
Env switch
OS
Isolation Module
Combination Option A
Sandbox
Break sandbox
OS
System call interface
Attacks:
(1) Iago attacks
(2) Subvert OS if sandbox is broken
Isolation Module
Combination Option B
TCB
Security-Sensitive Portion of App (SSPA)
Non-Sensitive Portion of App (NSPA)

Figure 1: Sandbox architecture, TrustVisor or Intel SGX architecture, and combination options.

In this paper, we rethink the security model of PaaS cloud computing and argue that a two-way sandbox is desired. The two-way sandbox not only protects a benign OS from a misbehaving application (*OS protection*) but also protects an application from a malicious OS (*application protection*). Researchers have explored several approaches for either protecting the OS from an *untrusted* application [55, 18, 27, 29] or protecting security-sensitive applications (or the security-sensitive portion of an application) from a malicious OS [25, 12, 13, 54, 17, 20, 45, 42, 36, 35, 7, 11, 28, 44, 57, 14]. Unfortunately, none of these schemes provides two-way protection, and it remains quite challenging to design a two-way sandbox.

TrustVisor [35] and Intel Software Guard Extensions (Intel SGX) [19, 24, 4] provide efficient memory space isolation mechanisms to protect the Security-Sensitive Portion of an Application (SSPA) from a malicious OS (Figure 1, TrustVisor or Intel SGX architecture). On TrustVisor or Intel SGX, memory access from the OS to the SSPA or from the SSPA to the OS is disabled by an isolation module, which is a hypervisor (on TrustVisor) or CPU hardware extensions (on Intel SGX). However, the Non-Sensitive Portion of an Application (NSPA) is not isolated from the OS, and the NSPA may contain malware that can compromise the OS.

Google Native Client (NaCl) [55] and Microsoft Drawbridge [18, 40] are application-layer one-way sandboxes for native code. It seems that combining an application-layer sandbox and an efficient memory space isolation mechanism is promising for the two-way sandbox design because such design supports cross platform and does not require any modifications to the OS. Figure 1 shows two combination options. In option A, the SSPA runs in an isolated memory space while a sandbox confines the NSPA. However, in this design application developers need to split the application into security-sensitive and non-sensitive portions, requiring large porting effort. In option B, the sandbox is also included inside the isolated memory space to avoid porting effort. The isolation module forwards system calls (from the sandbox) to the OS. However, there are several issues with this option. First, because the sandbox is complex and exposes a large interface to the application, a malicious application may exploit vulnerabilities in the sandbox and in turn subvert the OS. Second, a malicious OS may be able to compromise the application through Iago attack [10]. In Iago attack, a malicious OS can subvert a protected process by returning a carefully chosen sequence of return values to security sensitive system calls. For instance, if a malicious OS returns a memory address that is in the application's stack memory for an *mmap* system call, the sensitive data (e.g., return address) in the stack may be overwritten by the mapped data, which can result in a buffer overflow attack or a return-to-libc attack. Finally, because the OS is isolated from the sandbox and the application, it is challenging to support the application execution in an isolated memory space.

Challenges. In summary, the challenges we need address in the two-way sandbox design include:

1. Best combining a memory isolation mechanism and a sandbox to establish two-way protection.
2. Minimizing and securing the communication interface between OS protection modules and the application to reduce attack surfaces.
3. Preventing Iago attack for application protection.
4. Supporting application execution in an isolated memory space.

In this paper, we present MiniBox, the first two-way sandbox for native x86 applications. Leveraging the hypervisor-based memory isolation mechanism (proposed by TrustVisor) and a mature one-way sandbox (i.e., NaCl), MiniBox provides efficient two-way protection. MiniBox significantly reduces the interface between OS protection modules (sandboxing modules) and the application, exposes a minimized and secure communication interface between OS protection modules and the application. MiniBox protects the application against Iago attack and provides an efficient execution environment with secure file I/O for the application. The secure file I/O provides not only integrity and confidentiality protection, but also rollback prevention, secure key management, and access control. Using a special toolchain, application developers can concentrate on application development without bothering with the low-level protection mechanisms or splitting an application.

We implemented a prototype of MiniBox on both Intel and AMD multi-core systems, based on the Google Native Client (NaCl) [55] open source project and the public implementation of the TrustVisor hypervisor [35, 47, 46]. We ported several applications to MiniBox. Evaluation results show that MiniBox is practical and provides an efficient execution environment for isolated applications.

Contributions.

1. In this paper, we rethink the security model of PaaS cloud computing and make the first attempt toward a practical two-way sandbox for native applications.
2. MiniBox provides a minimized and secure communication interface between OS protection modules and the application to protect against each other.
3. MiniBox protects the application from Iago attack, and provides an efficient execution environment with secure file I/O for the application.
4. MiniBox makes the complex two-way protection mechanism transparent to application developers, making porting effort minimal.
5. We implement MiniBox and evaluate MiniBox using microbenchmarks and macrobenchmarks. Evaluation shows that MiniBox is practical and efficient.

Organization. The reminder of this paper is organized as follows: Section 2 briefly introduces the background knowledge. Section 3 describes our assumptions, adversary model. Section 4 presents the design of MiniBox in details. MiniBox implementation is described in Section 5 and evaluation results are described in Section 6. Limitations and future work are discussed in Section 7. Finally, we treat related work in Section 8 and offer conclusions in Section 9.

2 Background

2.1 TrustVisor

TrustVisor [35] is a minimized hypervisor that isolates the Security-Sensitive Portion of an Application (SSPA) from the rest of the system and offers efficient trustworthy computing abstractions (via a μTPM API) to the isolated SSPA with a small TCB.

Memory Protection. TrustVisor isolates the memory pages containing itself and any registered SSPA from the guest OS and DMA-capable devices by configuring nested page tables and the IO Memory Management Unit (IOMMU). TrustVisor exposes hypercall interfaces for applications in the guest OS to register and unregister a SSPA. When the SSPA is *registered*, information including the memory pages of the SSPA is passed to TrustVisor. TrustVisor configures nested page tables to isolate the memory pages of the SSPA from the guest OS. Any access from the guest OS to the SSPA or from the SSPA to the guest OS causes a nested page fault that will be intercepted by the hypervisor. When a SSPA is unregistered, TrustVisor zeroes the data memory in the SSPA, and removes the memory protections on the SSPA's address space.

Integrity Measurement. TrustVisor employs a two-level integrity measurement mechanism for measuring the integrity of the hypervisor and registered SSPA. TrustVisor is booted using the Dynamic Root of Trust and Measurement (DRTM) mechanism [5, 22] available on commodity x86 processors. The chipset computes an integrity measurement (cryptographic hash) of the hypervisor and extends the resulting hash into a Platform Configuration Register (PCR) in the Trusted Platform Module (TPM). TrustVisor computes an integrity measurement for each registered SSPA, and extends that measurement result into the PCR of the SSPA's μTPM instance. The TPM Quote from the hardware TPM and the μTPM Quote from the SSPA's μTPM instance comprise the complete chain of trust for remote attestation.

2.2 Google Native Client

Google Native Client (NaCl) [55] is a sandbox for x86 native code (called Native Module) using Software Fault Isolation (SFI) [34, 43, 49].

Validator. To guarantee the absence of privileged x86 instructions that can break out of the SFI sandbox in a Native Module, a validator in NaCl reliably disassembles the Native Module and validates the disassembled instructions as being safe to executes.

Runtime Framework. NaCl provides a simple runtime framework including a context switch function and a system call dispatcher to support the execution of a Native Module. On 32-bit x86, the runtime framework and the Native Module are isolated by segmentations. NaCl simulates system calls for a Native Module using a *Trampoline Table and Springboard*. There is a Trampoline Table in each Native Module, and a 32-byte entry in the Trampoline Table for each supported system call. The Google NaCl toolchain ensures that control transits to one of the entries in the Trampoline Table, instead of to a traditional system call. The Trampoline Table entries switch the active data and code segments, and jump to the context switch function in NaCl. The context switch function saves the thread context of the Native Module and transfers control to the system call dispatcher in NaCl. The system call dispatcher exposes only a closed set of system call interface to the Native Module, sanitizes the system calls parameters, conducts access control to constrain the file access of the Native Module, and finally calls the corresponding handler in the OS. After the handler execution completes, the context switch function restores the execution context of the Native Module and calls the *Springboard*, which performs the inverse of the control transitions in Trampoline Table entries.

3 Assumptions and Attacker Model

Assumptions. We assume that the attacker cannot have physical attacks against the hardware units (e.g., CPU and TPM). We assume that the attacker cannot break standard cryptographic primitives and that the small TCB of MiniBox is free of vulnerabilities.

Attacker Model For Application Protection. We assume that the attacker can execute arbitrary code on the OS. For example, the attacker can compromise and control the OS. The attacker that controls the OS may attempt to tamper with the protected application by accessing the application memory contents or handling the system calls of the application in malicious ways (i.e., Iago attack). The attacker can also inject malicious code into the application binary or into the service runtime binary before the application runs in an isolated memory space. The attacker may also subvert DMA-capable devices (e.g., the network adapter) on the platform in an attempt to modify memory contents through DMA. The attacker may also attempt to access security-sensitive files (e.g., private keys or credentials) of the application. However, we do not prevent denial of service attacks. For example, the attacker may prevent the OS from handling system calls for the application. We do not prevent the attacker from compromising the application through memory safety bugs (e.g., buffer overflows) or insecure design in the application. One example of the insecure design is that an application seeds a pseudo-random number generator by the return value of a system call handled by the untrusted OS. It is the developer's responsibility to take measures to eliminate memory safety bugs or such inse-

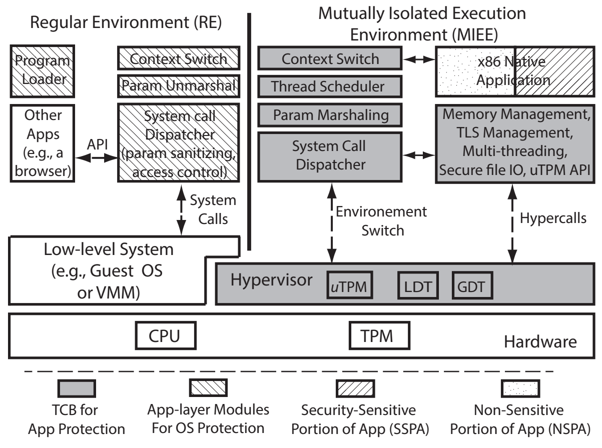
Regular Environment (RE)
Mutually Isolated Execution Environment (MIEE)
Program Loader
Context Switch
Param Unmarshal
Other Apps (e.g., a browser)
API
System call Dispatcher (param sanitizing, access control)
System Calls
Context Switch
Thread Scheduler
Param Marshaling
System Call Dispatcher
x86 Native Application
Memory Management, TLS Management, Multi-threading, Secure file IO, uTPM API
Environement Switch
Hypercalls
Low-level System (e.g., Guest OS or VMM)
Hypervisor
μTPM
LDT
GDT
CPU
TPM
Hardware
TCB for App Protection
App-layer Modules For OS Protection
Security-Sensitive Portion of App (SSPA)
Non-Sensitive Portion of App (NSPA)

Figure 2: MiniBox System Architecture.

cure design, that may leak data or break the control flow integrity [1, 3, 2] of the application. However, on MiniBox, the application can measure the integrity of critical inputs (i.e., known inputs) and extend the results into the μTPM PCR for remote attestation. Finally we do not prevent side-channel attacks [59].

Attacker Model For OS Protection. We assume that the untrusted application may contain malicious code. The malicious code may attempt to subvert the OS or access sensitive files on the OS.

4 System Design

4.1 MiniBox Architecture

We combine TrustVisor hypervisor and NaCl sandbox as the starting point for MiniBox design. MiniBox splits NaCl sandbox into OS protection modules and service runtime modules, and exposes a minimized and secure interface between OS protection modules and the application to protect against each other. Figure 2 shows the MiniBox architecture. As shown in this figure, a minimized hypervisor underpins the system. The hypervisor sets up the two-way memory space isolation between the Mutually Isolated Execution Environment (MIEE) and the Regular Environment (RE), creates a GDT and LDT instance, and a μTPM instance for the MIEE. In the MIEE, beyond the x86 native application itself, a minimized service runtime is included, containing: a context switch module that stores and switches thread contexts between the application and the service runtime; a system call dispatcher that distinguishes between regular and sensitive system calls, calls handlers in the MIEE for sensitive calls, or invokes the parameter marshaling module for non-sensitive calls; a parameter marshaling module that prepares parameter information for non-sensitive calls (for the hypervisor); system call handlers for handling security-sensitive system calls; and a thread scheduler that schedules the execution of multiple threads comprising an application. In sensitive call handlers, the service runtime supports dynamic memory management, thread local storage management, multi-threading management, secure file I/O, and μTPM API. The hypervisor and the minimized service runtime in the MIEE comprise the TCB for application protection.

In the RE, a user-level program loader sets up the MIEE and loads the application into the MIEE; a context switch module stores and restores the thread context of the RE during environment switches between the RE and MIEE; a parameter unmarshaling model unmarshals system call parameters; and a system call dispatcher confines the system call interface exposed to the application (allowing only a closed set of system calls), sanitizes the system call parameters, conducts access control to constrain the file access of the application, and forwards the non-sensitive system calls to corresponding handlers in the RE. On MiniBox, the program loader, the context switch module, the parameter unmarshaling module, the system calls dispatcher in the RE and the hypervisor comprise the TCB for OS protection.

Finally, MiniBox adopts the TrustVisor integrity measurement (recall Section 2.1) to enable a remote verifier to verify the integrity of the hypervisor, the minimized service runtime, and the isolated application. In this way, MiniBox prevents adversaries from injecting malicious code into the hypervisor, the service runtime or the application before the memory isolation is established without being detected. For example, *the integrity measurement and remote attestation mechanisms can prevent a malicious program loader from injecting malicious code into the application binary without being detected. This is also the reason that the program loader is not in the TCB for application protection.*

4.2 Minimized & Secure Communication Interface

Except passing system call information between the MIEE and the RE, the hypervisor exposes a minimal interface (i.e., several hypercalls) to the rest of the system. It is reasonable to trust that the minimal hypercall interface is free of vulnerabilities and malicious code in the RE or the MIEE cannot subvert the hypervisor over the hypercall interface. However, memory space isolation itself is insufficient to establish two-way protection because the communication interface between the RE (OS protection modules) and the MIEE (the application) may have vulnerabilities. MiniBox provides a minimized and secure communication interface between OS protection modules and the isolated application to protect against each other. This section first describes the communication interface between OS protection modules and an isolated application on NaCl, then describes how MiniBox minimizes and protects the communication interface.

The communication interface that NaCl exposes to an *untrusted* application includes:

1. In program loading time, the NaCl program loader exposes an interface to the application binary for obtaining section information.
2. The NaCl validator disassembles application binary and validates all application instructions. Because x86 instruction set is large and complex, the validator exposes a large interface to the application. If NaCl validator is broken, a malicious application will be able to break out of NaCl sandbox.
3. During runtime, the interface that NaCl exposes to an isolated application is a closed set of system call interface.

Minimizing Communication Interface. On MiniBox, the communication interface between OS protection modules and the application consists of only the program loader and the system call interface. Because privileged instructions cannot break out of the nested paged-based memory isolation, the NaCl validator is not included in MiniBox, *which significantly reduces the interface exposed to the application.* Without the validator, privileged instructions in the application can break out of the segmentation-based isolation and compromise the service runtime in the MIEE. However, a malicious service runtime in the MIEE cannot break out of the hypervisor-based memory isolation or compromise the OS through the thin interface between the MIEE and the RE.

Secure Communication. On MiniBox, the hypervisor is the only communication channel between the RE and the MIEE. Each non-sensitive call causes environment switches between the MIEE and the RE. For each environment switch from the MIEE out to the RE, the parameter marshaling module in the MIEE updates the parameter information of the system call that will be used by the hypervisor for copying parameters between the two environments. However, the parameter marshaling module in the MIEE cannot specify where the parameters will be stored in the RE. The hypervisor copies the system call parameters to a parameter buffer in the RE, and constrains the total data size of system call parameters that will be copied (to prevent buffer overflow attack). In this way, malicious code in the MIEE cannot overwrite critical data (e.g., stack contents) in the RE. To prevent a misbehaving application from sending arbitrary system call parameters to the RE, the system call dispatcher in the RE checks the system call parameters before sending them to the OS. For example, the system call dispatcher checks every pointer parameter that it is safe to access the memory space the parameter points to.

After the system call is handled, the system call dispatcher copies return values to the parameter buffer in the RE and triggers the environment switch back to the MIEE. When MiniBox switches from the RE *back to* the MIEE, the hypervisor uses the same parameter information specified by the MIEE to copy parameters from the parameter buffer in RE to the MIEE. This prevents malware in the RE from attempting to compromise MIEEs by manipulating parameter information.

In addition, because all sensitive system calls are handled inside the MIEE, a malicious OS cannot compromise the application by Iago attack. Also, with the secure I/O provided in the MIEE, sensitive files of the application are protected from a malicious OS.

4.3 Service Runtime

4.3.1 Dynamic Memory Management

Typical x86 programs request or remove data memory through system calls. MiniBox supports three system calls (i.e., `sysbrk`, `mmap`, and `munmap`) to provide dynamic memory management for the application running inside the MIEE. To prevent the OS from returning arbitrary memory addresses for `sysbrk` or `mmap` system call (i.e., Iago attacks) or removing arbitrary data memory pages from the MIEE, memory management system calls are handled in the MIEE.

Design. One naive design is pre-allocating and registering a large amount of data memory in the MIEE as data memory available for the application. Such a design has low execution time overhead, but it wastes memory and is not flexible. Another design is allowing the hypervisor to allocate memory pages as the application's data memory. However, the MiniBox hypervisor does not support swapping of memory pages to disk, and cannot be sure that pages marked as unused by the guest OS are actually present in memory. To resolve this issue, we design the system call handlers that request more data memory (i.e., `sysbrk` and `mmap`) in two modules: one in each of the isolated and regular environments. When the application requests more data memory (through `sysbrk` or `mmap`) but the requested data memory is not in the MIEE, the system call handler in the MIEE calls the module in the RE that allocates the memory page(s) and touches (i.e., writes zero to) them to ensure that the new memory page(s) are loaded into physical memory, and then returns to the handler inside the MIEE (note that the module in the RE cannot determine the memory address, so Iago attack are prevented). The system call handler inside the MIEE then makes a hypercall to the hypervisor to add the new memory page(s) to the MIEE. The x86 program invokes the `sysbrk` system call to set the program's *break address* and request more data memory for use as heap. When the new break address resides on a memory page that is already included in the MIEE, the system call handler within the MIEE simply updates the memory break address and returns, without a context switch to the RE. The `munmap` handler inside the MIEE makes a hypercall to unregister memory from the MIEE.

Hypercall Interface. The hypervisor exposes two hypercall interfaces to an MIEE for data page registration and unregistration. *Hypercall_Reg_Data* (used by `sysbrk` and `mmap`) registers additional data pages to an MIEE while *Hypercall_UnReg_Data* (used by `munmap`) performs the corresponding unregistration of data pages from an MIEE.

Security Protection. To prevent Iago attack caused by `mmap` or `sysbrk`, the hypervisor checks that the newly registered pages are not already registered to the MIEE (so that the malicious OS cannot overwrite stack contents of the application in the MIEE). To prevent leakage of sensitive data in either direction, the MiniBox hypervisor *zeroes* memory pages during registration and unregistration. To prevent a misbehaving or malicious application from adding privileged data pages (e.g., kernel pages) into MIEE, the hypervisor checks that the newly registered pages are user-level memory pages that are in ring 3, and correspond to the same OS process that originally registered the MIEE (i.e., through the value of CR3). Presently MiniBox does not allow additional memory to be mapped as executable, as this represents a challenge for integrity measurement and a significant increase in attack surface. Thus, the hypervisor checks that the requested memory pages are data pages that are not executable. In *data* memory page unregistration, the hypervisor checks that the unregistered memory pages are data pages that are already registered to the MIEE.

4.3.2 Thread Local Storage Management

Background. On 32-bit Linux, the native code built with the NaCl Newlib toolchain stores the memory address of its Thread Local Storage (TLS) as the base address of a segment descriptor in the LDT. During program initialization or when a new thread is created, `tls_init` system call initializes the TLS base address and updates the appropriate LDT entry. During execution, the `tls_get` system call is frequently called to get the TLS base address.

Design. Because the TLS and LDT represent critical configuration data, MiniBox handles the `tls_init` and `tls_get` entirely within the MIEE. The MiniBox hypervisor creates an LDT instance for each MIEE and supports a hypercall interface to the MIEE to handle `tls_init` system call. MiniBox supports caching the latest TLS address inside the MIEE, so that the `tls_get` handler can quickly return the latest TLS base address to the application without calling the hypervisor.

4.3.3 Multi-threading

Background. NaCl applies a 1:1 thread model (i.e., creating a kernel thread for each Native Module user-level thread) and uses the OS to handle thread-related system calls (e.g., thread synchronization system calls) and schedule the execution of Native Module threads.

Design. If MiniBox applies the same multi-threading mechanism, the OS controls the thread context of the application threads. A malicious OS could break the Control Flow Integrity (CFI) [1, 3, 2] of the isolated application by changing the thread context. Also, when the OS handles all thread synchronization system calls, a malicious OS could break the application CFI by arbitrarily changing application thread states. To protect the application thread context from being accessed by the OS, MiniBox can store the thread context in the MIEE and never leak it out of the MIEE. Also, the service runtime in the MIEE can verify the thread synchronization results by duplicating all supported thread synchronization system call handlers. In this design, all thread context and the application CFI are protected from a malicious OS. However, the complexity of this design is comparable to implementing the multi-threading operations within the MIEE. Also, if thread-related system calls are handled by the OS, the environment switches caused by thread-related system calls will increase the overhead of application execution in the MIEE. Thus, *to reduce execution overhead and avoid duplicated operations, MiniBox supports multi-threaded application execution via a user-level multi-threading mechanism entirely within the MIEE. System calls to create, exit and synchronize threads are handled in the MIEE. MiniBox also provide a thread scheduler to schedule the thread execution of an isolated application in the MIEE.*

Thread Scheduler. The thread scheduler is invoked each time there is a system call. After a system call is handled, control returns to the *thread scheduler* inside the MIEE before the context switch module is invoked (r-s1 in Figure 3). Before scheduling the execution of application threads, the thread scheduler first obtains the thread ID of the thread that made the system call, and saves the thread context in the corresponding thread context data structure. The scheduler checks the state of each thread, and schedules the execution of runnable threads using a round-robin algorithm. The thread scheduler finally calls the context switch module (r-s2 in Figure 3), which resumes the execution of the scheduled thread by restoring the thread context of the scheduled thread and jumping to the Springboard within the application (r-s3 in Figure 3).

Note that, before calling the thread scheduler, the hypervisor and the parameter marshaling module in the MIEE have already unmarshaled the system call param-

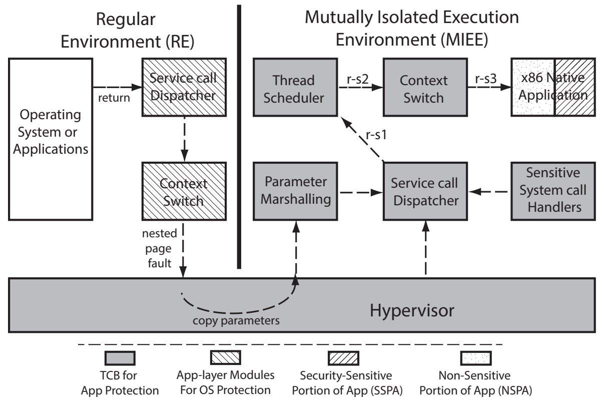

Figure 3: application system call return flow.

eters, and copied the returned parameters to the application's stack or data memory.

Limitation. MiniBox does not make the scheduler work preemptively, so application developers must notice this and always use supported system calls for thread synchronization (e.g., avoid a situation where a thread performs busy waiting by watching a global variable in a loop instead of calling a blocking system call). Also, the application-layer thread scheduler does not support multi-thread parallel computation to improve the performance of a threaded application on multi-core systems. One design is allowing the hypervisor to conduct thread scheduling and manage the parallel computation on multiple cores, which will significantly increases the hypervisor complexity. This is also a tradeoff between keeping a small TCB and supporting more functionalities. As future work we will investigate how to support parallel computation for a threaded application running inside the MIEE on multi-core systems.

4.4 MIEE Preemption and Scheduling

As described in Section 4.3.3, MiniBox does not preempt an application thread running in the MIEE. However, if an application thread is in an endless loop, the thread will not freeze the entire system because the MIEE is preemptive on MiniBox. When system switches into a MIEE, the hypervisor starts a timer for the MIEE and preempts the code execution in the MIEE when the timer expires. After preempting the MIEE, the hypervisor stores the MIEE context and transfers control to the regular environment by simulating a special system call (i.e., *MIEE_sleep*). The *MIEE_sleep* handler sleeps for a while and then calls the hypervisor to resume the code execution in the MIEE. In this way, the hypervisor transfers the control to the OS, which can schedule the execution of other processes. When multiple MIEEs are registered (one MIEE in one process), the OS can implicitly schedule the execution of multiple MIEEs by scheduling process execution. However, the question is how much CPU time (time quantum) should be assigned to each MIEE by the hypervisor. One design is that the hypervisor exposes a hypercall interface to the regular environment and the MIEE to enable the OS and the isolated application in the MIEE to configure the MIEE process priority. The hypervisor assigns CPU time to each MIEE based on the MIEE process priority.

4.5 Secure File I/O

On MiniBox, the application running in the MIEE still needs to access the file system in the regular environment, so the file system calls are forwarded to the OS. However, to protect the file contents and metadata of an isolated application, MiniBox supports secure file I/O for applications running in the MIEE through six system calls: `secure_write`, `secure_read`, `secure_open`, `secure_close`, `create_siokey`, `read_siokey`. The six system calls are handled in the MIEE. The secure file I/O not only protects the integrity and confidentiality of file contents and file metadata, but also provides rollback prevention, secure key management, and access control.

Confidentiality and Integrity. `secure_write` encrypts the data written by the application (with a symmetric secret key) and sends the encrypted data to the general file I/O, while `secure_read` decrypts the data and returns the decrypted data to the application in the MIEE. In `secure_write` and `secure_read`, the data is written or read by a chain of blocks of a constant size. To protect the integrity of file contents and file metadata, including file name and path, a hash tree is constructed and computed over the blocks of file contents and file metadata in the MIEE (this approach has been demonstrated in the Trusted Database System [33], VPFS [51] and jVPFS [52]). A HMAC of the master hash is computed in the MIEE and stored at the end of the file (as file contents). When a file created by secure file I/O is opened, `secure_open` reads the HMAC and verifies the integrity of the file contents and metadata by reconstructing the hash tree. `secure_open` stores the hash tree in the MIEE. When a data block is read, `secure_read` verifies the integrity of the data block based on the stored hash tree. When file contents are modified, `secure_write` updates the hash tree stored in the MIEE. When a file is closed, `secure_close` recomputes the master hash and the HMAC, and stores the updated HMAC at end of the file. This allows the integrity of file contents and file metadata to be verified. The attacker cannot remove, add, or replace data blocks in the file because any changes will invalidate the HMAC. The attacker cannot replace the file with other files that are created by the same application running in the MIEE either because file metadata is also verified.

Rollback Prevention (Freshness). MiniBox adds a counter in each HMAC computation to guarantee freshness of files stored through the secure file I/O. The counter is sealed by the μTPM. Because the μTPM cannot provide freshness for sealed contents, the integrity of the counter is measured every time the same application runs in the MIEE (the measurement result is extended into μPCR for remote attestation). This allows a verifier to verify the freshness during remote attestation.

Key Management. Before using secure file I/O, the application running in the MIEE must call `create_siokey` to create the secret keys used in secure file I/O (i.e., a symmetric encryption key and a HMAC key). The application specifies the file name and file path for storing the keys when calling `create_siokey`. `Create_siokey` first checks if the file already exists. If not, `create_siokey` creates new secret keys, seals the secret keys with the current μPCR values. Then it stores the sealed secret keys in the file, and returns the key ID to application. If the file already exists (i.e., keys are already created), `create_siokey` reads the sealed keys from the untrusted file system, unseals the keys and returns the key ID to the application. `read_siokey` reads the secret keys saved in the MIEE. Thus the application must call `create_siokey`, which saves the secret keys in the MIEE, before calling `read_siokey`.

Access Control and Migration. Because the secret keys are sealed with the current μPCR (i.e., the integrity measurement of the application), the sealed keys can only be unsealed by the μTPM when the same application runs in the MIEE. Thus, any data encrypted through secure File I/O can only be decrypted and verified when the same application runs in the MIEE. To share the sensitive files with other applications running in the MIEE (e.g., an updated version of the application), the application can seal the secret keys with the integrity measurement result of other applications, and share the sealed keys to other applications. Then, other applications running in the MIEE can unseal the secret keys (using `create_siokey`) and access the secret files.

Cache Buffer. On MiniBox, environment switches between the MIEE and the RE cause high overhead in file I/O (Section 6). To reduce the number of environment switches, MiniBox creates a cache buffer in the MIEE for each opened file descriptor. Both general file I/O and secure file I/O benefit from the cache buffer because the number of environment switches is reduced. However, the cached data in this cache buffer is still encrypted. Another cache buffer can be created in secure file I/O layer to store decrypted data and improve the secure file I/O performance.

Limitations. Compared with complex virtual private file systems (e.g., VPFS [51] and jVPFS [52]), MiniBox does not provide recoverability. A complete private file system is beyond the scope of this paper.

4.6 Flexible Parameter Marshaling

On MiniBox, the hypervisor copies system call parameters between the MIEE and the regular environment, and different system calls have different parameters (number, type, address, and size of parameters). Some system calls may have complex parameters (e.g., multi-level pointers or structures). Thus, MiniBox employs a flexible parameter marshaling mechanism for environment switching.

We divide parameters into four types: Integer, Pointer Out, Pointer In and Pointer OutIn. Integer denotes a native machine-width integer passed between the two environments. The hypervisor copies the value of the integer parameter between the environments during environment switches. A Pointer Out parameter is a pointer that points to a data buffer that is passed from the MIEE to the regular environment. The buffer size and address are included in the parameter and passed to the hypervisor. The hypervisor copies the data indicated by this parameter from the MIEE to the regular environment only when the system switches from the MIEE to the regular environment. Pointer In parameters are the inverse of Pointer Out parameters: data passes from the regular environment to the MIEE. The hypervisor copies the data indicated by this parameter from the regular environment to the MIEE only when the system switches from the regular environment to the MIEE. A Pointer OutIn parameter is a pointer that points to a data buffer that needs to be passed in both directions. The hypervisor copies the data indicated by this parameter in both directions.

Marshaling Complex Parameters. For system calls that accept complex parameters (e.g., multi-level pointers or structures), the parameter marshaling module in the MIEE first serializes parameters (in user-space) into a format that can be decoded based on included parameter type information. These serialized parameters are then used by the hypervisor. The user-level parameter marshaling is transparent to the hypervisor, and thus avoids adding complexity to the hypervisor.

4.7 Exceptions, Interrupts, and Debug

Exceptions and Interrupts. When system runs in a MIEE, the processor cannot access exception and interrupt handlers in the OS. Thus, the hypervisor is configured to intercept exceptions (e.g., *segment fault*, *invalid opcode*) and Non-Maskable Interrupts (NMIs) when system runs in a MIEE. Maskable interrupts are disabled when system runs in a MIEE. When NMIs happen, the hypervisor handles NMIs and resumes the code execution in the MIEE. When an exception happens, the hypervisor first checks whether the exception is because the application in the MIEE needs more stack pages. If so, the hypervisor calls a module in the regular environment to allocate more data pages as stack pages, adds the stack pages into the MIEE, and resumes the code execution in the MIEE. If not, the hypervisor terminates the code execution in the MIEE by simulating an *Exit* system call. The *Exit* call is forwarded to the program loader, which unregisters the MIEE from the hypervisor via hypercall.

Debug. Though MiniBox provides compatible execution environment with NaCl, the NaCl debug tool for application cannot be directly used on MiniBox because on MiniBox the OS cannot access the memory contents in the MIEE. However, MiniBox can be configured in a debug mode, in which the hypervisor functionalities are disabled, and an application layer module passes parameters between the two environments. In debug model, memory management and TLS management calls are handled by the OS. In this way, the memory isolation is disable and application developers can use NaCl debug tool for MiniBox application development.

4.8 Program Loader

In MiniBox, a user-level program loader prepares the service runtime for the application, loads the application binary to page-aligned memory, registers the whole thing as a MIEE through hypercalls, and finally launches the execution of the application.

Initialization. After loading the application into page-aligned memory, the program loader initializes the relevant LDT for segments of the application (code, data and stack), initializes the system call parameter information for environment switch, and populates the initial thread context of the application. The program loader allocates a 32 MB stack section for each application at the high end of the application's address space. The application accepts *arguments* upon its initial invocation like a typical process. The program loader copies the arguments into the application's stack memory.

MIEE Registration. Before launching the execution of the application, the program loader registers the MIEE comprising the service runtime, and the application's code, data and stack sections. During registration, the hypervisor sets up memory protection for the MIEE, instantiates a fresh μTPM instance, instantiates a GDT and LDT for the MIEE, measures the memory contents of the MIEE, and extends the measurement results into a PCR in the μTPM instance for remote attestation. Before registration, the program loader has full access permissions to the application and the service runtime. Thus, it could potentially maliciously modify the contents of the application or service runtime. *However, any such modifications or injected malicious contents will cause the μTPM's PCR to take on a different value than expected. As a result, the MIEE will be unable to generate correct μTPM Quotes in remote attestation or unseal the secret data that are sealed with the expected μTPM PCR value.*

Launching Application. After registration, the program loader launches application execution by triggering the environment switch into the MIEE. Inside the MIEE, the context switch module initializes the application thread context, switches segment selector registers, and starts application execution.

MIEE Unregistration. After the application completes its execution it invokes an *Exit* system call that is forwarded to the program loader. After receiving this system call, the program loader unregisters the MIEE from the hypervisor via hypercall. The MIEE data memory is zeroed and the memory protections on the MIEE's memory space are removed by the hypervisor.

5 Implementation

We implemented a prototype of MiniBox on both AMD and Intel multi-core systems, with 32-bit Ubuntu 10.04 LTS as the guest OS. This section describes the MiniBox implementation in details.

5.1 Hypervisor

The implementation of the MiniBox hypervisor is based on the public implementation of TrustVisor hypervisor [35, 46] with support for multi-core and both AMD and Intel processors. We changed the parameter marshaling implementation and added a hypercall interface for handling sensitive system calls. TrustVisor does not instantiate an LDT for a registered piece of application logic. Thus, we also added code to create new GDT entries and instantiate an LDT for every MIEE, and added code to handle GDT- and LDT-related operations. The original implementation of TrustVisor hypervisor has 14414 source lines of code (SLoC), computed using the sloccount tool.[1] Our implementation adds an additional 691 SLoC. Figure 4 shows the code size of each module that we added or modified in TrustVisor hypervisor. Note that the parameter marshaling module and the extended hypercall interface are independent of the CPU manufacturer, The GDT- and LDT-related module support both AMD and Intel processors. Thus, as with TrustVisor hypevisor, the MiniBox hypervisor also works on both AMD and Intel processors.

MiniBox hypervisor module	**SLOC**
Parameter Marshaling	201
Extended Hypercall Interface and Handlers	230
GDT and LDT-related	260
Total	691

Figure 4: Source Lines of Code (SLOC) added to TrustVisor hypervisor.

[1]`http://www.dwheeler.com/sloccount/`

5.2 Program Loader and Service Runtime

We implement the user-level program loader, the minimized service runtime in the MIEE, the context module and the system call dispatcher in the regular environment based on the Google Native Client (NaCl) open source project (SVN revision 7110). We have focused our work on the 32-bit x86 architecture, though there are no fundamental barriers to expanding to 64-bit. In the NaCl source code, we implement code to conduct MIEE registration and unregistration (which entails preparing an MIEE's sections, and `vmcalls` to register/unregister the MIEE) by 299 SLoC. We also implement the minimized service runtime in the MIEE within the NaCl source code, adding 3550 SLoC. The secure file I/O module has a large code base (1065 SLoC) because it contains cryptographic primitives for AES and HMAC. The implemented service runtime can be configured in debugging mode for application development using the NaCl debugging tool (recall Section 4.7). The parameter marshaling module in the MIEE has a large codebase because it needs to be capable of preparing parameter information for the 42 different system calls that are handled in the RE (23 SLOC for each system call on average). The sensitive system call handlers we implemented are mainly for handling thread synchronization (e.g., mutexes, semaphores, and condition variables), which results in a larger codebase than other modules. We use a custom linker script when building the NaCl ELF loader to link the service runtime framework in page-aligned memory pages. Figure 5 summarizes the SLOC added to Google's NaCl source code.

Module	**SLOC**
MIEE registration and unregistration	299
Context Switch in regular environment.	29
Total in regular environment.	**328**
System Call Dispatcher in MIEE	379
Parameter Marshaling in MIEE	970
Thread Synchronization in MIEE	711
Secure File I/O in MIEE	1065
Other Sensitive Call Handlers in MIEE	373
Context Switch in MIEE.	52
Total in MIEE	**3550**

Figure 5: Source Lines of Code (SLOC) of modules added to the NaCl source code.

5.3 System Calls

MiniBox adopts NaCl system call interface to expose a closed set of system call interface to the isolated application. MiniBox does not support dynamic code for the application, so NaCl dynamic code system calls are removed on MiniBox. However, MiniBox extends the NaCl system call interface with μTPM API, network I/O system calls, and secure file I/O calls, supporting a total of 75 system calls for the application. Figure 6 shows the system calls supported by MiniBox. The implementation entails adding header files and statically linked libraries into the NaCl Newlib toolchain, and modifying the NaCl source code to (1) add extended system call entries to the application's Trampoline Table and add corresponding parameter marshaling functions. and (2) add corresponding hypercalls to the MiniBox service runtime. MiniBox supports 17 socket system calls. The network I/O system calls are forwarded to the regular environment, because they are treated as part of the untrusted communication channel. Secure communication (e.g., SSL) can be implemented in the application layer to protect the data in network I/O.

In the MIEE, the supported thread synchronization system calls include semaphores, mutexes, and condition variables, which have the same functionality as the corresponding POSIX APIs. The thread synchronization implementation passes the internal thread synchronization test suite included in the NaCl source code. The secure file I/O calls encrypt/decrypt the data using AES with a 128-bit key in CBC mode and computes HMAC-SHA-1 using a 160-bit key.

The μTPM API is exposed to applications through system calls. The implementation entails adding μTPM header files and a statically linked μTPM library into the NaCl Newlib toolchain, and modifying the NaCl source code to (1) add μTPM API entries to the application's Trampoline Table, and (2) add corresponding hypercalls to the MiniBox service runtime framework.

6 Evaluation

In this section, we present the evaluations including system call overhead, file I/O overhead, network I/O, and application performance in the MIEE on MiniBox. Experiments were conducted on a Dell PowerEdge T105 server with a Quad-Core AMD Opteron Processor running at 2.3 GHz with 4 GB memory. The operating system is Ubuntu 10.04 with 32-bit kernel Linux 2.6.32.27. To obtain accurate timing results, the hypervisor does not preempt the MIEE.

Performance Impact. Vasudevan et al. [46] already presented the performance impact on a guest OS running on XMHF (e.g., TrustVisor). Since MiniBox hypervisor extends TrustVisor with hypercall interface, and modified parameter marshaling, neither of which affect the guest OS performance, we do not evaluate the performance impact on a guest OS in this paper. Yee et al. [55, 56] evaluated the code size and performance impact on the application developed using the NaCl toolchain. MiniBox uses the NaCl Newlib toolchain with

Operations	**System Calls**
μTPM$*$	μTPM_PCR_Read, μTPM_PCR_Extend μTPM_Random, μTPM_Seal, μTPM_unSeal, μTPM_PCR_Quote
Memory$*$	sysbrk, mmap, munmap
TLS$*$	tls_init, tls_get
Thread$*$	thread_create, thread_exit, thread_nice, sched_yield
Mutex$*$	mutex_create, mutex_lock, mutex_trylock, mutex_unlock
Condition$*$	cond_create, cond_wait, cond_signal, cond_broad
Semaphore$*$	sem_create, sem_wait, sem_post, sem_getvalue
Secure File$*$	secure_read, secure_write, secure_open secure_close, create_siokey, read_siokey
File	dup, dup2, open, close, read, write, lseek, ioctl, stat, fstat
Time	time_of_day, clock, nanosleep
Inter-Module Communication (IMC) [55]	imc_bound, imc_accept, imc_connect, imc_send, imc_recv, imc_objcreate, imc_socket
Socket	accept, bind, connect, send, recv, listen getpeername, getsockname, getsockopt recvmsg, recvfrom, sendmsg, sendto setsockopt, shutdown, socket, socketpair
Others	nameservice, getdents, exit, getpid sysconf

Figure 6: System calls supported by MiniBox. Starred calls ($*$) are handled inside the MIEE or hypervisor; the remaining calls are forwarded to the regular environment and handled by the OS.

extended API. We do not repeat those evaluations in this paper.

Porting Effort. The NaCl toolchain is mature. A number of web applications and open source libs have been ported to run on NaCl using the NaCl toolchain. MiniBox use the NaCl toolchain with extended API for application development. It is excepted that the porting effort on MiniBox is similar to the porting effort on NaCl. Thus, we do not evaluate the porting effort for legacy code using the NaCl toolchain in this paper.

6.1 MiniBox Microbenchmarks

Environment Switch Overhead. For every non-sensitive system call, the hypervisor is invoked to conduct environment switches between the MIEE and the RE, in which procedure the hypervisor (1) synchronizes all CPU cores through a operation called *CPU-quiescing* [46], (2) configures the nested page table for memory protection, and (3) copies system call parameters between the MIEE and the RE. The environment

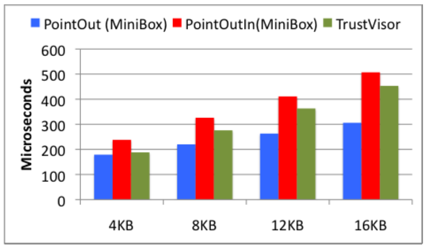

Figure 7: Environment switch benchmarks in *us*. Average of 100 runs and standard deviation is less than 5%.

switch causes overhead on non-sensitive system calls and may affect application performance in the MIEE. We evaluate the overhead of environment switches with different parameter types (e.g., Pointer Out, Pointer OutIn) and sizes (amount of data to pass into/out of the MIEE). TrustVisor conducts similar operations with MiniBox during environment switches. However, TrustVisor supports a single pointer type that describes a data buffer, which is copied in both directions. Similar experiments are also conducted on vanilla TrustVisor v0.1 with identical parameter sizes, and similar parameter types.

The evaluation results (Figure 7) show that on MiniBox the environment switch overhead increases by about 40 microseconds per 4KB parameter size for Pointer Out and 80 microseconds per 4KB parameter for Pointer OutIn. The evaluation results also show that the parameter marshaling mechanism on MiniBox is more efficient than the marshaling mechanism on TrustVisor for data that needs to be copied in only one direction (Pointer Out in Figure 7). However, when the parameter type is Pointer OutIn, the parameter marshaling mechanism on MiniBox is slower than the mechanism on TrustVisor. For Pointer OutIn, MiniBox also copies parameters in both directions. However, the MiniBox hypervisor obtains parameter information from the MIEE for every switch (Section 4.2), whereas TrustVisor always uses the same parameter information for the MIEE. Thus, MiniBox takes more time to conduct environment switch than TrustVisor when the parameter type is Pointer OutIn.

System Call Overhead. In the MIEE, non-sensitive system calls are handled in the OS with environment switches while sensitive system calls are handled either in the application layer inside the MIEE or by the hypervisor through hypercalls. It is important to know the system call overhead in the MIEE to understand the performance impact on the application running in the MIEE. The system call overhead in the MIEE is measured, and compared with the corresponding system calls on vanilla NaCl, and MiniBox in debug model (recall Section 4.7).

The evaluation results (Figure 8) show that the non-sensitive system calls (e.g., file operation calls) that in-

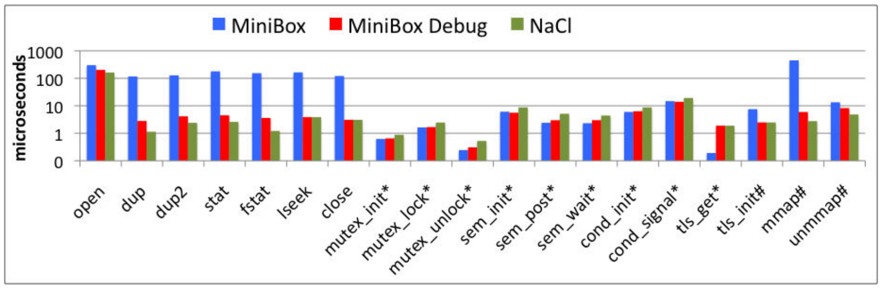

Figure 8: System call benchmarks in *us*. Average of 100 runs and standard deviation is less than 5%. Calls with $*$ are sensitive calls handled inside the MIEE without environment switches. Calls with # are sensitive calls that involve hypercall or environment switches.

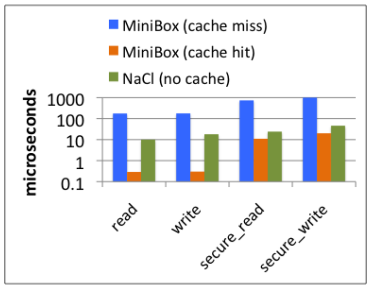

Figure 9: File I/O benchmarks in *us*. Average of 100 runs and standard deviation is less than 2%.

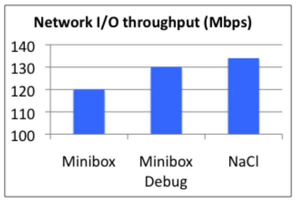

Figure 10: Network I/O benchmarks in *Mbps*. Average of 100 runs and standard deviation is less than 2%.

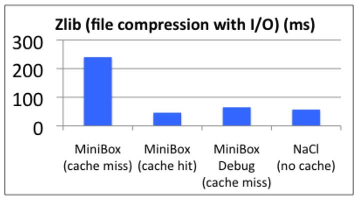

Figure 11: zlib file compression with file I/O benchmarks in *ms*. Average of 10 runs and standard deviation is less than 2%.

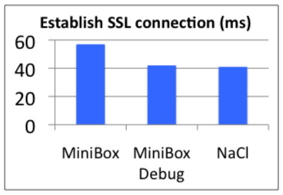

Figure 12: SSL connection benchmarks in *ms*. Average of 10 runs and standard deviation is less than 3%.

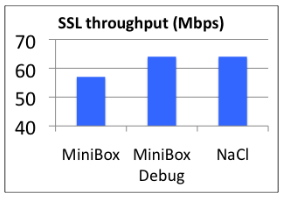

Figure 13: SSL throughput benchmarks in *Mbps*. Average of 10 runs and standard deviation is less than 1%.

volve environment switches on MiniBox are slower than on vanilla NaCl. However, the corresponding system calls on MiniBox in debug mode have similar performance to those on vanilla NaCl. Thus the overhead of these system calls on MiniBox is mainly caused by environment switches. The sensitive system calls that are handled within the MIEE without any environment switch (e.g., thread synchronization calls) have similar performance to those on vanilla NaCl. The sensitive system calls that involve hypercall and environment switches (e.g., memory management system calls) on MiniBox are slower than on vanilla NaCl. The overhead is expected because of the environment switch overhead and the operations conducted by the hypervisor.

File I/O. We evaluate the file I/O overhead on MiniBox and compare it to the file I/O on vanilla NaCl and MiniBox in debug mode. We measure reads & writes of 32B for both general file I/O and secure file I/O. The measurement results (Figure 9) show that when the data is cached in the MIEE (cache-hit), the cache buffer significantly reduces the file I/O overhead for both general file I/O and secure file I/O.

Network I/O. We evaluate the network I/O throughput on MiniBox and compare it to the network I/O throughput on MiniBox in debug mode and vanilla NaCl (we add the socket interface on the NaCl). The server runs in the MIEE using MiniBox on the Dell T105 while the client runs on a plain Linux machine on a Dell Optiplex 755 desktop with two Intel Core2 Duo processors running at 2.0 GHz with 2 GB memory. The operating system on the Dell Optiplex machine is Ubuntu 8.04.4 LTS with a 32-bit Linux kernel 2.6.24.30. Both the server and the client connect to a Netgear Gigabit Ethernet Switch using a Gigabit Ethernet Adapter. During each connection, the client sends 16 KB data to the server and we measure the network I/O throughput. The results (Figure 10) show that network I/O on MiniBox is about 10% slower than on vanilla NaCl. *Thus, although the environment switches impose a small overhead on MiniBox, the network throughput remains high.*

6.2 Application Benchmarks

CPU-bound application (AES key search and BitCoin). We measure the performance of CPU-bound applications on MiniBox and compare it to the performance of equivalent applications on vanilla NaCl and MiniBox in debug mode. We first evaluate *AES key search*, which encrypts a 128-Byte plaintext using a 128-bit key in CBC mode 200,000 times, simulating a AES key search operation. We ported CBitCoin [37]), an open source BitCoin implementation to run on MiniBox. We measure the time to construct a BitCoin block, requiring 200,000 SHA-1-256 computations. The results (Figure 14) show that *MiniBox does not add any noticeable overhead for CPU-bound applications over NaCl.*

I/O-bound application (Zlib). We evaluate the per-

	MiniBox	MiniBox Debug	NaCl
AES Key Search	168	168	168
BitCoin	236	236	236

Figure 14: AES key search and BitCoin block generation benchmarks in *ms*. Average of 10 runs and standard deviation is less than 2%.

formance of I/O-intensive applications on MiniBox by testing Zlib [32], an open source library used for data compression. Zlib is already ported to run on NaCl as part of the naclports project, and did not require additional porting efforts to run on MiniBox. We measure the time elapsed to read 1 MB of file data from the file system over the general file I/O, and then compress the read data. The file data always misses the cache buffer, so every *read* operation involves an environment switch. The evaluation results (Figure 11) show that because of environment switches, the zlib application on MiniBox is slower than vanilla NaCl. The slowdown is mainly caused by the environment switches since MiniBox in debug mode has the same performance as vanilla NaCl. We repeated the measurement on MiniBox while storing the file data in the cache buffer in the MIEE. The zlib application reads file data with cache-hit without environment switches. The measurement result shows that the overhead is significantly reduced. *Thus, while file I/O in MiniBox can be expensive in the worst case, we expect that the cache buffer will significantly improve the application performance in practice.*

SSL Server. We ported the entirety of OpenSSL [39] (version 1.0.0.e) to run on MiniBox. We also run the SSL server on the NaCl by adding socket system call interface on the NaCl. In this experiment, the Dell Optiplex machine serves as the SSL client, and the Dell T105 acts as the SSL server. The SSL client runs on a plain Linux while the SSL server runs inside the MIEE on MiniBox. We recorded both the time required to create an SSL connection and the overall SSL throughput. The SSL client sends 16KB data to the SSL server during each connection. As in previous experiments, both machines connected to a Netgear Gigabit Ethernet Switch via a Gigabit Ethernet Adapter. The results show that MiniBox impose about a 15% overhead to SSL connections (Figure 12) and that SSL throughput on MiniBox has about a 10% slowdown (Figure 13). The overhead is mainly caused by network and file I/O, since MiniBox in debug mode has the same performance as NaCl.

7 Discussion

Improving Performance and Alternative Implementations. The hypervisor-based isolation mechanism causes overhear in environment switches. It is expected that the hardware-based isolation mechanism provided by Intel SGX will decrease the environment switch overhead (Intel SGX has not been released on commercial processors). The *VMFUNC* instruction [26] released on the latest Intel 4th Generation Processor enables software in a guest Virtual Machine (VM) to switch EPTs without a VM exit. It is expected that the *VMFUNC* instruction will decrease the environment switch overhead. However, the *VMFUNC* instruction does not switch other critical system configurations (e.g., the GDT or IDT). As future work we will investigate the approach to perform environment switch using the *VMFUNC* instruction in *a secure way*. Microsoft Drawbridge provides efficient sandboxing mechanism to confine native applications and requires minimal porting efforts for legacy POSIX applications. An alternative MiniBox implementation is to integrate the Drawbridge service runtime with TrustVisor or Intel SGX.

Supporting Multi-tenant Cloud Platform. The MiniBox hypervisor prototype supports only a single guest OS. There is no fundamental barrier to port MiniBox with a virtual machine monitor like Xen [8] that supports multiple tenants, though doing so increases the TCB size. CloudVisor [58] demonstrates the approach to minimize the TCB on multi-tenant cloud platforms by leveraging nested virtualization technology. Nested virtualization can be added in MiniBox to support multi-tenant cloud platforms with a small TCB. On multi-tenant cloud platforms, the virtual machine (VM) may be constructed, destructed, saved, restored, or migrated. It is critical to protect the MIEE including the application during VM management, as discussed and demonstrated in CloudVisor. Similar actions can be employed by MiniBox to protect the integrity and confidentiality of MIEEs (including applications) in VM management on multi-tenant cloud platforms. The MiniBox hypervisor can encrypt or decrypt the memory contents of MIEEs including applications in VM management, and verify the integrity of the MiniBox hypervisor on other machines to guarantee that MIEEs are only migrated to machines with a verified hypervisor. Also, the MiniBox hypervisor needs to encrypt or decrypt the μTPM instance (including the PCRs and μTPM secret keys) together with an MIEE in VM management, to make the trustworthy computing abstractions provided to the MIEE transparent to the VM management. The above features can be added in MiniBox with sufficient engineering effort.

Control Flow Integrity (CFI). Since the application that runs on MiniBox is isolated using nested page tables at the hypervisor level, and always runs in ring 3, MiniBox does not share NaCl's CFI requirement to be able to reliably disassemble and validate all instructions. Therefore, the CFI mechanisms implemented in NaCl are not necessary in MiniBox. The NaCl CFI mechanism depends upon its toolchain inserting many `nop` instructions into the compiled program, which decreases performance. The benefit of keeping the CFI mechanism, however, is that programs compiled by the same toolchain will be compatible with both NaCl and MiniBox. Also, the NaCl CFI mechanism does raise the bar for an adversary who wants to attack a specific application running in the MIEE.

Supporting Dynamic Code. MiniBox does not support dynamic code for applications because marking data memory as executable to support dynamic code adds challenges for integrity measurement and significantly increases the attack surface. The NaCl Newlib toolchain already provides dynamic code APIs [6]. As future work, we will investigate a mechanism to support dynamic code for applications on MiniBox while measuring and protecting the code integrity of applications.

8 Related Work

Protecting Applications. Systems aspiring to protect entire applications from a potentially compromised OS have been proposed (e.g., InkTag [25], Overshadow [13], SP3 [17], Proxos [45], PrivExec [38], and others [54, 12, 14]). Most of these schemes mainly focus on protecting application data from malicious code on an operating system and expose sensitive system calls to the untrusted OS, thus making the protected application vulnerable to Iago attack. InkTag [25] secures applications running on an untrusted OS by verifying that the untrusted OS behaves correctly using a trustworthy hypervisor. It prevents `mmap`-based Iago attack by verifying memory address invariants. However, in InkTag some other security-sensitive system calls (e.g., thread synchronization and TLS-related calls) are still performed by the untrusted OS without being verified. Proxos [45] splits system calls and forwards sensitive system calls to a trusted private OS to protect applications from an untrusted OS. However, Proxos needs application developers to specify the splitting rule and has a large TCB (encompassing the entire private OS).

Protecting Security-Sensitive Code. Researchers have also explored many systems for isolating sensitive code using virtualization, microkernels, and other low-level mechanisms [20, 45, 42, 36, 35, 7], or by running the code inside trusted hardware [11, 28, 44]. The virtualization-based schemes contain a large TCB. Other schemes either do not enjoy compatibility with a large set of commodity systems or require significant porting effort. TrustVisor [35] and Flicker [36] isolate the Security Sensitive Portion of an Application (SSPA) from an untrusted OS with a small TCB. Because constrained interfaces are exposed between the isolated SSPA and the untrusted OS in TrustVisor and Flicker, the isolated SSPA in both is not vulnerable to Iago attack. However, TrustVisor and Flicker do not provide a minimized service runtime to support the execution of the isolated SSPA. Thus, porting security-sensitive applications on TrustVisor or Flicker requires significant efforts. Nizza [42] also requires developers to perform similar operations to port sensitive applications to Nizza. MiniBox exposes trustworthy computing abstractions to the application running on MiniBox. These facilities are implemented at the hypervisor layer, extending the chain of trust from hardware support for dynamic root of trust. In this sense they are similar to TrustVisor [35], or SICE [7].

Sandbox for x86 Native Code. Google Native Client [55] confines untrusted native code using SFI [34, 43, 49] and enables developers to port native code as web applications. However, the NaCl sandboxing mechanism depends on a validator that validates all instructions in the application. If the validator has vulnerabilities, privileged instructions may break out of NaCl sandbox. drawbridge [18, 40] isolates an application in a picoprocess and provides a library OS to the isolated application. In this way, a misbehaving application cannot compromise other application or the OS on Drawbridge. The isolation property is similar to MiniBox. However, Drawbridge provides only one-way protection. A malicious OS can compromise the application running in a picoprocess. TxBox [27] confines an untrusted application by executing the application in a system transaction and conducting security check. MBox [29] protects the host file system from an untrusted application by exposing a virtual file system on top of the host file system for the application. Both TxBox and MBox only protect the OS from an untrusted application. Capsicum [50] supports capability-sandbox for applications on UNIX-like OS (e.g., FreeBSD). It focuses on application compartmentalization and fine-grained access control. The protection mechanisms provided by Capsicum can be applied on MiniBox as part of the OS protection modules.

Cloud Security. We describe previous work that secures cloud computation against malicious cloud administrators or malware on cloud platforms. CloudVisor [58] protects the guest virtual machine instance (VMI) from unauthorized access by isolating the guest VMI from other VMIs (including the control VMI) and from the Xen hypervisor by leveraging nested page-based isolation and nested virtualization technology. However, CloudVisor does not prevent attacks from inside the guest VMI. For example, the adversary (e.g., a malicious cloud administrator) could inject malware into the guest OS image before the guest OS is launched on the cloud platform without being detected. Also the large code size of the guest OS makes it challenging to guarantee the runtime security of the guest OS though the guest VMI is protected by CloudVisor. Credo [41] measures the integrity of the guest OS during bootstrap with a static root of trust. However, the large code size of the guest OS makes it challenging to guarantee the runtime security on that OS. A certified operating system for cloud computing [30, 23, 53] may protect the customer's application from malware on the cloud. However, without attestation, malicious cloud administrators could inject malicious code into the customer's application or the certified OS on the cloud without being detected. SSC [9] exposes control VMs to cloud customers, enabling cloud customers to protect and control their own VMs. However, SSC still has a large TCB and does not focus on application-level protection. Recent work to break up popular hypervisors into systems more closely resembling micro-kernel architectures also show promise for TCB reduction [15].

9 Conclusion

MiniBox makes the first attempt toward a two-way sandbox with a minimized and secure communication interface between OS protection modules and an application. MiniBox provides efficient sandboxing mechanism with a small TCB, protects applications against Iago attack, and provides an efficient execution environment for isolated applications. Though the environment switch on MiniBox causes overhead in file I/O, the overhead can be reduced by cache buffers. MiniBox does not require any modifications to the OS and offers an easy way to port native code. We expect that MiniBox can be used in PaaS cloud computing or web browsers to offer two-way protections.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. CFI: Principles, implementations, and applications. In *Proc. ACM Conference and Computer and Communications Security (CCS)* (2005).

[2] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow Integrity Principles, Implementation, and Applications. *ACM Transaction on Information and System Security (TISSEC) 13* (2009), 1 – 40.

[3] Abadi, M., Budiu, M., Úlfar Erlingsson, and Ligatti, J. A theory of secure control flow. In *Proc. Conference on Formal Engineering Methods* (2005).

[4] Abatu, U., Gueron, S., Johnson, S. P., and Scarlata, V. R. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM.

[5] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. AMD Publication no. 24593 rev. 3.14, Sept. 2007.

[6] Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D. L., Sehr, D., Biffle, C. L., and Yee, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. *SIGPLAN Not. 47*, 6 (June 2011), 355–366.

[7] Azab, A. M., Ning, P., and Zhang, X. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proc. ACM Conference on Computer and Communications Security* (2011).

[8] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. Symposium on Operating Systems Principles* (2003).

[9] Butt, S., Srivastava, H. A. L.-C. A., and Ganapathy, V. SelfService Cloud Computing. In *Proc. ACM Conference and Computer and Communications Security (CCS)* (2012).

[10] Checkoway, S., and Shacham, H. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Mar. 2013).

[11] Chen, B., and Morris, R. Certifying program execution with secure processors. In *Proceedings of HotOS* (2003).

[12] Chen, H., Zhang, F., Chen, C., Yang, Z., Chen, R., Zang, B., Yew, P., and Mao, W. Tamper-resistant execution in an untrusted operating system using a VMM. Tech. Rep. FDUPPITR-2007-0801, Fudan University, 2007.

[13] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS* (2008).

[14] Cheng, Y., Ding, X., and Deng, R. AppShield: Protecting applications against untrusted operating system. In *Singaport Management University Technical Report, SMU-SIS-13-101* (2013).

[15] Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., and Warfield, A. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proc. ACM Symposium on Operating Systems Principles* (2011).

[16] Darrow, B. Google App Engine by the numbers. `http://gigaom.com/2012/06/28/google-app-engine-by-the-numbers/`.

[17] Dewan, P., Durham, D., Khosravi, H., Long, M., and Nagabhushan, G. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proc. Spring Simulation Multiconference* (2008).

[18] Douceur, J. R., Elson, J., Howell, J., and Lorch, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 339–354.

[19] Frank, M., Ilya, A., Alex, B., V, R. C., Hisham, S., Vedvyas, S., and R, S. U. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM, pp. 10:1–10:1.

[20] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *Proc. ACM Symposium on Operating System Principles (SOSP)* (2003).

[21] Gong, L. Java 2 Platform Security Architecture. `http://docs.oracle.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html`.

[22] Grawrock, D. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[23] Gu, L., Vaynberg, A., Ford, B., Shao, Z., and Costanzo, D. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Proceedings of the 3rd ACM SIGOPS Asia-Pacific Workshop on Systems* (2011).

[24] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and Del Cuvillo, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM, pp. 11:1–11:1.

[25] Hofmann, O., Dunn, A., Kim, S., Lee, M., and Witchel, E. Inktag: Secure applications on an untrusted operating system. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Mar. 2013).

[26] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 3b: System programming guide, part 2. Order Number: 325384-048US, Sept. 2013.

[27] Jana, S., Porter, D. E., and Shmatikov, V. Txbox: Building secure, efficient sandboxes with system transactions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 329–344.

[28] Jiang, S., Smith, S., and Minami, K. Securing web servers against insider attack. In *Proc. Computer Security Applications Conference* (2001).

[29] Kim, T., and Zeldovich, N. Practical and effective sandboxing for non-root users. In *Proceedings of the 2013 USENIX conference on USENIX annual technical conference* (2013), USENIXATC'13.

[30] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an OS kernel. In *ACM SOSP* (2009).

[31] Kortchinsky, K. CloudBurst: A VMware Guest to Host Escape Story. In *Black Hat'09*.

[32] loup Gailly, J., and Adler, M. zlib open source library. `http://www.zlib.net`.

[33] Maheshwari, U., Vingralek, R., and Shapiro, W. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4* (Berkeley, CA, USA, 2000), OSDI'00, USENIX Association, pp. 10–10.

[34] McCamant, S., and Morrisett, G. Evaluating sfi for a cisc architecture. In *Proc. USENIX Security* (2006).

[35] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[36] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Apr. 2008).

[37] Mitchell, M., Sterling, A., and Miller, A. Cbitcoin open source project. `http://code.google.com/p/naclports/`.

[38] Onarlioglu, K., Mulliner, C., Robertson, W., and Kirda, E. Privexec: Private execution as an operating system service. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 206–220.

[39] OpenSSL Project team. OpenSSL. `http://www.openssl.org/`, May 2005.

[40] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. Rethinking the library os from the top down. *SIGPLAN Not. 46*, 3 (Mar. 2011), 291–304.

[41] Raj, H., Robinson, D., Tariq, T. B., England, P., Saroiu, S., and Wolman, A. Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor. In *Microsoft Technical Report MSR-TR-2011-130* (2011).

[42] Singaravelu, L., Pu, C., Härtig, H., and Helmuth, C. Reducing TCB complexity for security-sensitive applications. In *EuroSys* (2006).

[43] Small, C., and Seltzer, M. I. Misfit: Constructing safe extensible systems. *IEEE Concurrency 6*, 3 (1998), 34–41.

[44] Smith, S. W., and Weingart, S. Building a high-performance, programmable secure coprocessor. *Computer Networks 31*, 8 (Apr. 1999).

[45] Ta-Min, R., Litty, L., and Lie, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *ACM SOSP* (2006).

[46] Vasudevan, A., Chaki, S., Jia, L., McCune, J., Newsome, J., and Datta, A. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (May 2013).

[47] Vasudevan, A., McCune, J. M., and Newsome, J. Design and implementation of an extensible and modular hypervisor framework. Tech. Rep. CMU-CyLab-12-014, CMU CyLab, June 2012.

[48] Vulnerability, C., and Exposure. Xen: security vulnerability. `http://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html`.

[49] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *ACM SOSP* (1993).

[50] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 3–3.

[51] Weinhold, C., and Härtig, H. Vpfs: building a virtual private file system with a small trusted computing base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 81–93.

[52] Weinhold, C., and Härtig, H. jvpfs: adding robustness to a secure stacked file system with untrusted local storage components. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 32–32.

[53] Yang, J., and Hawblitzel, C. Safe to the last instruction: automated verification of a type-safe operating system. *SIGPLAN Not. 45*, 6 (June 2010), 99–110.

[54] Yang, J., and Shin, K. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proc. ACM Conference on Virtual Execution Environments (VEE)* (2008).

[55] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Orm, T., Okasaka, S., Narula, N., Fullagar, N., and Google Inc. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy* (2009).

[56] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM 53*, 1 (2010), 91–99.

[57] Yee, B. S. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[58] Zhang, F., Chen, J., Chen, H., and Zang, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (2011).

[59] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.