# MiniBox: A Two-Way Sandbox for x86 Native Code

Yanlin Li
*CyLab/CMU*

Jonathan McCune
*CyLab/CMU, Google Inc.*

James Newsome
*CyLab/CMU, Google Inc.*

Adrian Perrig
*CyLab/CMU*

Brandon Baker
*Google Inc.*

Will Drewry
*Google Inc.*

## Abstract

This paper presents MiniBox, the first two-way sandbox for x86 native code, that not only protects a benign OS from a misbehaving application, but also protects an application from a malicious OS. MiniBox can be applied in Platform-as-a-Service cloud computing to provide two-way protection between a customer's application and the cloud platform OS. We implement a MiniBox prototype running on recent x86 multi-core systems from Intel or AMD, and we 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, 1 million active applications were running on Google App Engine [14]. On PaaS cloud computing, it is critical to protect the cloud platform from the large number of *untrusted* applications sent by customers. Thus, a virtualized infrastructure (e.g., Xen [7]) and sandbox (e.g., Java sandbox [19]) are deployed to isolate customers' applications and protect the guest OS. However, security on PaaS is not only a concern for cloud providers but also a concern for cloud customers. As shown in Figure 1-A, current sandbox technology provides only one-way protection, which protects the OS from an *untrusted* application. The security-sensitive Piece of Application Logic (PAL) is completely exposed to malicious code on the OS. Also, current sandboxes expose a large interface to *untrusted* applications, and may have vulnerabilities that malicious applications can exploit.

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 [16, 23, 25, 48] or protecting security-sensitive applications (or security-sensitive PALs) from a malicious OS [6, 10, 11, 12, 13, 15, 18, 21, 24, 31, 32, 38, 39, 40, 47]. Unfortunately, none of these schemes provides two-way protection, and many challenges remain to design a two-way sandbox.

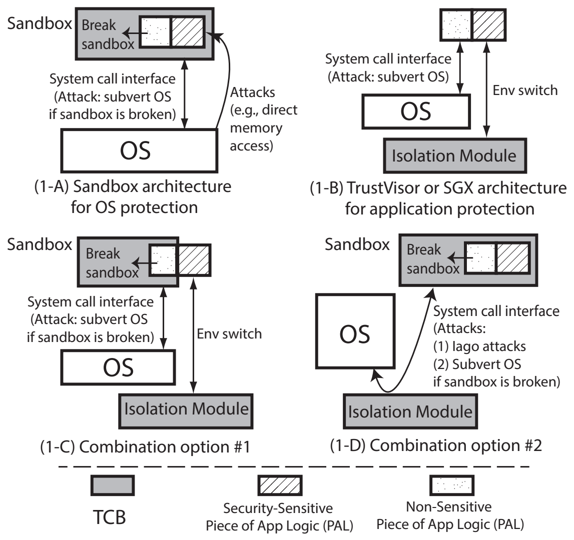

Sandbox
Break sandbox
System call interface (Attack: subvert OS if sandbox is broken)
Attacks (e.g., direct memory access)
OS
(1-A) Sandbox architecture for OS protection
System call interface (Attack: subvert OS)
Env switch
OS
Isolation Module
(1-B) TrustVisor or SGX architecture for application protection
Sandbox
Break sandbox
System call interface (Attack: subvert OS if sandbox is broken)
Env switch
OS
Isolation Module
(1-C) Combination option #1
Sandbox
Break sandbox
OS
System call interface (Attacks: (1) Iago attacks (2) Subvert OS if sandbox is broken)
Isolation Module
(1-D) Combination option #2
TCB
Security-Sensitive Piece of App Logic (PAL)
Non-Sensitive Piece of App Logic (PAL)

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

TrustVisor [31] and Intel Software Guard Extensions (Intel SGX) [4, 17, 20] are examples of systems that provide efficient memory space isolation mechanisms to protect a security-sensitive PAL from a malicious OS (Figure 1-B). On TrustVisor or Intel SGX, memory access from the OS to the security-sensitive PAL or from the security-sensitive PAL 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 PAL is not isolated from the OS, and the non-sensitive PAL may contain malware that can compromise the OS.

Google Native Client (NaCl) [48] and Microsoft Drawbridge [16, 36] are examples of application-layer one-way sandboxes for native code. We found that combining an application-layer sandbox and an efficient memory space isolation mechanism is promising for the two-way sandbox design. However, it it not straightforward. Figure 1-C and 1-D show two combination options. In option #1, the security-sensitive PAL runs in an isolated memory space while a sandbox confines the non-sensitive PAL. However, in this design application developers need to split the application into security-sensitive and non-sensitive PALs, requiring substantial porting effort. In option #2, the sandbox is included inside the isolated memory space to avoid porting. 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 attacks [9]. In Iago attacks, a malicious OS can subvert a protected process by returning a carefully chosen sequence of return values to 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, sensitive data (e.g., a return address) in the stack may subsequently be overwritten by the mapped data. 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. Thus, both options have obvious shortcomings and we shall not choose them for the two-way sandbox design.

In this paper, we present MiniBox, the first two-way sandbox for x86 native applications. Leveraging a hypervisor-based memory isolation mechanism (proposed by TrustVisor) and a mature one-way sandbox (NaCl), MiniBox offers efficient two-way protection. MiniBox splits the NaCl sandbox into OS protection modules (software modules performing OS protection) and service runtime (software modules supporting application execution), runs the service runtime and the application in an isolated memory space (Section 4.1), and exposes a minimized and secure communication interface between the OS protection modules and the application (Section 4.2). MiniBox also splits the system call interface available to the isolated application as sensitive calls (the calls that may cause Iago attacks) and non-sensitive calls (the calls that cannot cause Iago attacks), and protects the application against Iago attacks by handling sensitive calls inside the service runtime in the isolated memory space (Section 4.3). MiniBox also provides secure file I/O for the application (Section 4.3.4). Using a special toolchain, application developers can concentrate on application development with small porting effort (Section 6). We implement a MiniBox prototype based on the Google Native Client (NaCl) [48] open source project and the TrustVisor hypervisor [31, 41] (Section 5), and port several applications to MiniBox. Evaluation results show that MiniBox is practical and provides an efficient execution environment for isolated applications (Section 6).

**Contributions.**

1. We design, implement, and evaluate MiniBox, the first attempt toward a practical two-way sandbox for x86 native applications.
2. MiniBox demonstrates it is possible to provide a minimized and secure communication interface between OS protection modules and the application to protect against each other.
3. MiniBox demonstrates it is possible to protect against Iago attacks, and provide an efficient execution environment with secure file I/O for the application.

## 2 Background

### 2.1 TrustVisor

TrustVisor [31] is a minimized hypervisor that isolates a PAL from the rest of the system and offers efficient trustworthy computing abstractions (via a $\mu$TPM API) to the isolated PAL with a small TCB. TrustVisor isolates the memory pages containing itself and any registered PALs 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 PAL. When a PAL is *registered*, information including the memory pages of the PAL is passed to TrustVisor. TrustVisor configures nested page tables to isolate the memory pages of the PAL from the guest OS. TrustVisor is booted using the Dynamic Root of Trust for Measurement mechanism [5] 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 PAL, and extends that measurement result into the PAL's $\mu$TPM instance. The TPM Quote from the hardware TPM and the $\mu$TPM Quote from the PAL's $\mu$TPM instance comprise the complete chain of trust for remote attestation.

### 2.2 Google Native Client

Google Native Client (NaCl) [48] is a sandbox for x86 native code (called Native Module) using Software Fault Isolation (SFI) [30, 42]. 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 execute. NaCl provides a simple service runtime including a context switch function and a system call dispatcher to support the execution of a Native Module. On 32-bit x86, the service runtime and the Native Module are isolating using the CPU's segmentation mechanism [22]. 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. For each 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 transfers control to the system call dispatcher in NaCl. The system call dispatcher exposes only a subset of the OS system call interface to the Native Module, sanitizes the system call parameters, conducts access control to constrain the file access of the Native Module, and finally calls the corresponding handler in the OS. The *Springboard* performs the inverse of the control transitions in the Trampoline Table entries.

## 3 Assumptions and Attacker Model

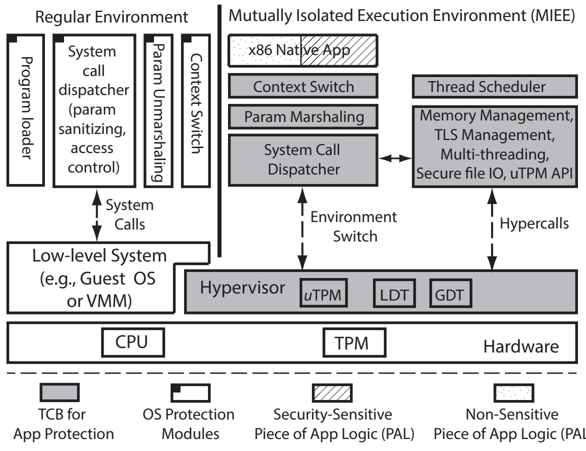
**Assumptions.** We assume that the attacker cannot conduct physical attacks against the hardware units (e.g., CPU and TPM). We assume that the attacker cannot break standard cryptographic primitives and that the TCB of MiniBox is free of vulnerabilities. For application protection, we also assume that the application does not have any memory safety bugs (e.g., buffer overflows) or insecure designs. One example of the insecure designs 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 insecure designs. For OS protection, we assume that the system call interface that the OS protection modules expose to the application (a subset of the OS system call call interface) is free of vulnerabilities, and that the OS does not have concurrency vulnerabilities [43] in system call wrappers.

**Attacker Model For Application Protection.** We assume that the attacker can execute arbitrary code on the OS. For example, the attacker may compromise and control the OS, and then attempt to tamper with the protected application by accessing the application memory contents or handling the system calls of the application in malicious ways (Iago attacks). The attacker may attempt to inject malicious code into the application binary or into the service runtime binary before the application runs in an isolated memory space *without being detected*. The attacker may subvert DMA-capable devices on the platform in an attempt to modify memory contents through DMA. The attacker may also attempt to access security-sensitive files of the application. However, we do not prevent denial of service attacks. Finally we do not prevent side-channel attacks [51].

**Attacker Model For OS Protection.** The untrusted application may attempt to subvert the hypervisor or break out of the hypervisor-based memory isolation. The application may also attempt to read or modify sensitive files that do not belong to the application on the system. The application may attempt to subvert the OS by making arbitrary system calls with carefully-chosen parameters.

## 4 System Design

### 4.1 MiniBox Architecture



Regular Environment
Mutually Isolated Execution Environment (MIEE)
Program loader
System call dispatcher (param sanitizing, access control)
Param Unmarshaling
Context Switch
x86 Native App
Context Switch
Thread Scheduler
Param Marshaling
Memory Management, TLS Management, Multi-threading, Secure file IO, uTPM API
System Call Dispatcher
System Calls
Environment Switch
Hypercalls
Low-level System (e.g., Guest OS or VMM)
Hypervisor
uTPM
LDT
GDT
CPU
TPM
Hardware
TCB for App Protection
OS Protection Modules
Security-Sensitive Piece of App Logic (PAL)
Non-Sensitive Piece of App Logic (PAL)

*Figure 2:* MiniBox System Architecture.

Figure 2 shows the MiniBox architecture. As shown in this figure, a 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, and creates a $\mu$TPM instance for the MIEE.

On MiniBox, the hypervisor and a service runtime in the MIEE comprise the runtime TCB for application protection. In the MIEE, beyond the x86 native application, a 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 non-sensitive and sensitive 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 sensitive 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 $\mu$TPM API.

On MiniBox, the OS protection modules include a user-level program loader, a context switch module, a parameter unmarshaling module, and a system calls dispatcher in the regular environment. In the regular environment, the user-level program loader sets up the MIEE and loads the application into the MIEE; the context switch module stores and restores the thread context of the regular environment during environment switches between the regular environment and MIEE; the parameter unmarshaling module unmarshals system call parameters; and the system call dispatcher confines the system call interface exposed to the application (allowing only a subset of the OS 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 regular environment.

Finally, MiniBox adopts TrustVisor's integrity measurement (recall Section 2.1) to enable a remote verifier to verify the integrity of the hypervisor, the 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. *This is also the reason that the program loader is not in the TCB for application protection.*

### 4.2 Communication Interfaces

The MiniBox hypervisor exposes a small interface to the rest of the system. MiniBox minimizes and secures the communication interface between OS protection modules and the application to protect against each other.

**Hypervisor Interface.** Other than passing system call information between the MIEE and the regular environment, the hypervisor exposes a small interface (i.e., only several hypercalls) to the rest of the system. Thus, assuming the small hypercall interface is free of vulnerabilities, malicious code in the rest of the system cannot compromise the hypervisor or break out of the hypervisor-based memory isolation.

**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 hypervisor-based memory isolation, the NaCl validator (that validates that the application binary does not contain privileged instructions) 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.

**Secure Communication.** On MiniBox, the hypervisor is the only communication channel between the regular environment and the MIEE. Each non-sensitive system call causes environment switches between the MIEE and the regular environment. For each environment switch from the MIEE out to the regular environment, 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 regular environment. The hypervisor copies the system call parameters to a parameter buffer in the regular environment, and constrains the total data size of system call parameters (to prevent buffer overflow attacks). In this way, malicious code in the MIEE cannot overwrite critical data (e.g., stack contents) in the regular environment. To prevent a misbehaving application from sending arbitrary system call parameters to the regular environment, the system call dispatcher in the regular environment checks the system call parameters before sending them to the OS. For example, the system call dispatcher checks the value of every pointer parameter and guarantees that it is safe to access the memory space the parameter points to. If a check fails, the system call dispatcher returns an error code without calling the corresponding system call handler.

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

### 4.3 Service Runtime

#### 4.3.1 Dynamic Memory Management

MiniBox supports three system calls (`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 the `sysbrk` or `mmap` system calls (Iago attacks) or removing arbitrary data memory pages from the MIEE, memory management system calls are handled inside the MIEE.

**Design.** One naive design is pre-allocating and registering a large amount of data memory in the MIEE as data memory for the application. This design has low execution time overhead, but it wastes memory and is inflexible. 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 but the requested data memory is not in the MIEE, the system call handler in the MIEE calls the module in the regular environment that allocates the memory page(s) and 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. 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 `munmap` handler inside the MIEE makes a hypercall to unregister memory from the MIEE.

**Security Protection.** To prevent Iago attacks 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. Presently MiniBox does not allow additional memory to be mapped as executable, as this represents 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 on vanilla NaCl stores the memory address of its Thread Local Storage (TLS) as the base address of a segment descriptor in the Local Descriptor Table (LDT) [22]. 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, 2, 3] 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.*

**Thread Scheduler.** MiniBox provides a thread scheduler to schedule the thread execution of the application in the MIEE. The thread scheduler is invoked each time there is a call from an entry of the *Trampoline Table* (recall Section 2). After the call is handled, control returns to the *thread scheduler* inside the MIEE before the context switch module is invoked. 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, which resumes the execution of the scheduled thread.

### 4.3.4 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 five system calls: `secure_write`, `secure_read`, `secure_open`, `secure_close`, `create_siokey`. The five system calls are handled in the MIEE.

**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 [28], VPFS [45] and jVPFS [46]). 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 $\mu$TPM. Because the $\mu$TPM 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 $\mu$PCR 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 $\mu$PCR 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.

**Access Control and Migration.** Because the secret keys are sealed with the current $\mu$PCR (i.e., the integrity measurement of the application), the sealed keys can only be unsealed by the $\mu$TPM 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 regular environment 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.

## 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 the 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 each process), the OS can implicitly schedule the execution of multiple MIEEs by scheduling process execution. However, the question is how much CPU time 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 Exceptions, Interrupts, and Debugging

**Exceptions and Interrupts.** While the code in a MIEE is running, the processor cannot access exception and interrupt handlers in the OS. Thus, the hypervisor is configured to intercept exceptions (e.g., *segmentation 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.

**Debugging.** Though the MiniBox execution environment is compatible with NaCl's, the NaCl debugging tool for application development 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 debugging mode, in which the hypervisor functionalities are disabled, and an application layer module passes parameters between the two environments. In debugging model, memory management and TLS management calls are handled by the OS. In this way, the memory isolation is disabled and application developers can use the NaCl debugging tool for MiniBox application development. An alternative way is including the NaCl debugging tool in the MIEE and supporting an interface to access the debugging tool from the regular environment. In this way, the developers can debug the application when the memory isolation is enabled.

## 5 Implementation

We implement a MiniBox prototype running on recent x86 multi-core systems from Intel or AMD, 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 (version 0.1.2) [31, 41] with support for multi-core and both AMD and Intel processors. We changed the parameter marshaling implementation [26] and added a hypercall interface for handling sensitive system calls. We added code to create new Global Descriptor Table (GDT) [22] 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.

### 5.2 Program Loader and Service Runtime

We implement the user-level program loader, the 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 in 299 SLoC. We implement the 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 (recall Section 4.5).

### 5.3 System Calls

MiniBox adopts NaCl system call interface to expose a subset of the OS 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. MiniBox extends the NaCl system call interface with $\mu$TPM API, network I/O system calls, and secure file I/O calls, supporting a total of 75 system calls for the application (a list of supported system calls is described in [26]). 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 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.

## 6 Evaluation

In this section, we present the evaluations including system call overhead, file I/O overhead, network I/O, and

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

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.** MiniBox hypervisor extends the TrustVisor with hypercall interface and modified parameter marshaling [26], neither of which affects the guest OS performance. Thus, MiniBox hypervisor imposes similar guest overhead to the TrustVisor [41]. Yee et al. [48, 49] presented that the NaCl toolchain can cause significant increase in code size (2% to 57% on SPEC2000 benchmarks), but non-significant impact on performance (on average less than 5% on SPEC2000 benchmarks).

**Porting Effort.** MiniBox uses the NaCl toolchain with extended API for application development and imposes similar porting efforts to the NaCl. Yee et al. [48, 49] presented that porting an internal implemented H.264 decoders (11K lines of C code) to NaCl requires adding about twenty lines of C code, and porting Bullet[2] to NaCl took only a few hours. Compared to NaCl, MiniBox requires additional porting effort for application protection. For instance, application developers must understand the MiniBox protection mechanisms and avoid insecure application designs (recall Section 3). Application developers must understand the trustworthy computing abstractions exposed to every MIEE, and correctly use them.

### 6.1 MiniBox Microbenchmarks

**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. The system call overhead in the MIEE was measured, and compared with the corresponding system calls on vanilla NaCl, and MiniBox in debugging model (recall Section 4.5). The evaluation results (Figure 3) show that the non-sensitive system calls (e.g., file operation calls) that involve environment switches on MiniBox are slower than on vanilla NaCl. However, the corresponding system calls on MiniBox in debugging 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.

[2]http://www.bulletphysics.com

**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 debugging mode. We measure reads & writes of 32B for both general file I/O and secure file I/O. The measurement results (Figure 4) 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 debugging mode and vanilla NaCl. The server runs in the MIEE using MiniBox on the Dell T105 while the client runs on plain Linux 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 5) 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 debugging mode. We first evaluate *AES key search*, which encrypts a 128-Byte plain-text using a 128-bit key in CBC mode 200,000 times, simulating a AES key search operation. We port CBitCoin [33]), an open source BitCoin implementation to run on MiniBox. We measure the time to construct a BitCoin block, requiring 200,000 SHA-256 computations. The results show that *MiniBox does not add any noticeable overhead (less than 1% [26]) for CPU-bound applications over NaCl.*

**I/O-bound application (Zlib).** We evaluate the performance of I/O-intensive applications on MiniBox by testing Zlib [27], an open source library used for data compression. Zlib is already ported to run on NaCl as part of the naclports project, and does 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 6) show that because of

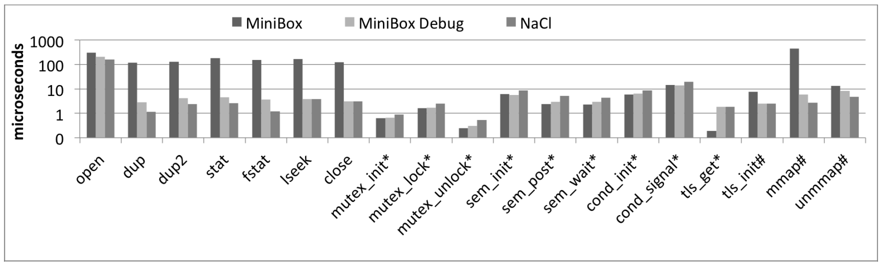


*Figure 3:* System call benchmarks in *us*. Average of 100 runs and standard deviation is less than 5%. Calls with $*$ are sensitive calls handled inside the MIEE without environment switches. Calls with # are sensitive calls that involve hypercall or environment switches.

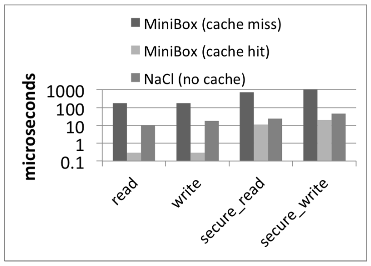


*Figure 4:* File I/O benchmarks in *us*. Average of 100 runs and standard deviation is less than 2%.

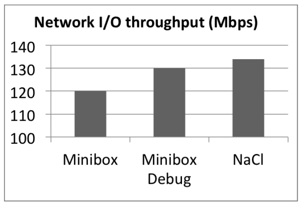


*Figure 5:* Network I/O benchmarks in *Mbps*. Average of 100 runs and standard deviation is less than 2%.

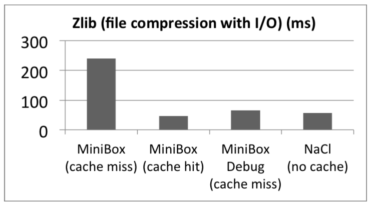


*Figure 6:* zlib file compression with file I/O benchmarks in *ms*. Average of 10 runs and standard deviation is less than 2%.

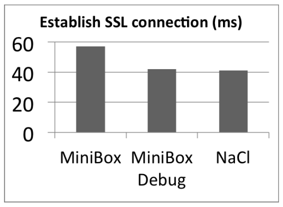


*Figure 7:* SSL connection benchmarks in *ms*. Average of 10 runs and standard deviation is less than 3%.

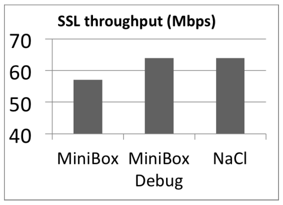


*Figure 8:* SSL throughput benchmarks in *Mbps*. Average of 10 runs and standard deviation is less than 1%.

environment switches, the zlib application on MiniBox is slower than on vanilla NaCl. The slowdown is mainly caused by the environment switches since MiniBox in debugging mode has the same performance as vanilla NaCl. We repeat the measurement on MiniBox while storing the file data in the cache buffer in the MIEE. The zlib application read 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 port the entirety of OpenSSL [35] (version 1.0.0.e) to run on MiniBox. We also run the SSL server on 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 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 of data to the SSL server during each connection. As in previous experiments, both machines connect 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 7) and that SSL throughput on MiniBox has about a 10% slowdown (Figure 8). The overhead is mainly caused by environment switches, since MiniBox in debugging mode has the same performance as NaCl.

## 7 Related Work

**Protecting Applications.** Systems aspiring to protect entire applications from a potentially compromised OS have been proposed (e.g., [8, 11, 12, 13, 15, 21, 29, 34, 40, 47]). 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 attacks. InkTag [21] secures applications running on an untrusted OS by verifying that the untrusted OS behaves correctly using a trustworthy hypervisor. It prevents `mmap`-based Iago attacks 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 [40] 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. Baumann et al. [8] proposed to run entire legacy applications in the isolated memory space provided by Intel SGX, and proposed to include a library OS in the isolated memory space to prevent Iago attacks. The proposed protection mechanisms (for application protection) are similar to the mechanisms on MiniBox. Mai et al. [29] proposed mechanisms to prove that the OS implements the application security invariants (e.g., secure storage and memory isolation) correctly. The proposed verification approach is promising for application isolation.

**Protecting Security-Sensitive Code.** Researchers have explored many systems for isolating sensitive code using virtualization, microkernels, and other low-level mechanisms [6, 18, 31, 32, 38, 40], or by running the code inside trusted hardware [10, 24, 39]. 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 [31] and Flicker [32] isolate a PAL from an untrusted OS with a small TCB. However, porting security-sensitive applications on TrustVisor or Flicker requires significant efforts. Nizza [38] also requires developers to perform similar operations to port sensitive applications to Nizza.

**Sandbox for x86 Native Code.** Google Native Client [48] confines untrusted native code using SFI [30, 42] and enables developers to port native code as web applications. Drawbridge [16, 36] isolates an application in a picoprocess and provides a library OS to the isolated application. However, Native Client and Drawbridge provide only one-way protection. TxBox [23] confines an untrusted application by executing the application in a system transaction and conducting security check. MBox [25] 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. Capsicum [44] supports capability-sandbox for applications on UNIX-like OS (e.g., FreeBSD). It focuses on application compartmentalization and fine-grained access control. Systrace [37] improves the host OS security by confining the program privilege using a configurable system call policy. The protection mechanisms provided by MBox, Capsicum and Systrace can be applied on MiniBox as part of the OS protection modules.

## 8 Limitations and Future Work

**Application Interface.** MiniBox includes the entire application (the security-sensitive and non-sensitive PALs) in the MIEE and does not prevent adversaries from compromising the application through malicious inputs. The application can measure the integrity of critical inputs (known inputs) and extend the results into the $\mu$TPM PCR for remote attestation. However, the isolated application may expose a large interface to unknown inputs. Schemes that focus on protecting a security-sensitive PAL [6, 18, 31, 32, 38, 40] can significantly reduce the attack surface by exposing a constrained interface between the security-sensitive PAL and the untrusted OS. On those schemes, the security-sensitive PAL remains secure when the application is compromised by the OS. Thus, for protecting the security-sensitive PAL, MiniBox may expose a larger attack surface to the untrusted OS than schemes that focus on protecting the security-sensitive PAL.

**Thread Scheduling.** Application developers must consider that MiniBox does not make the scheduler work preemptively (recall Section 4.3.3), and so must always use supported system calls for thread synchronization (e.g., avoid situations where a thread performs busy waiting by watching a global variable in a loop instead of calling a blocking system call). In addition, the application-layer thread scheduler does not support multi-thread parallel computation to improve the performance of threaded applications on multi-core systems. One design is to allow the hypervisor to conduct thread scheduling and to manage the parallel computation on multiple cores, which will significantly increase the hypervisor complexity. As future work, we will investigate how to support parallel computation for a threaded application running inside the MIEE on multi-core systems. However, security-sensitive applications more concerned with a small TCB than performance may prefer not to include code for such complex operations in the hypervisor. To solve this issue, MiniBox can allow the application to configure the hypervisor functionality (e.g., disable the support for multi-thread parallel computation) at registration time, and can boot the hypervisor with the application-preferred configurations.

**System Call Interface.** Exposing a large system call interface to the application increases the attack surface for OS protection; thus, MiniBox exposes a subset of the OS system call interface to the application to confine the application's operations. However, it will be interesting to investigate how to support the entire OS system call interface on MiniBox. If the entire OS system call interface is supported, statically linked legacy applications may be able to run on MiniBox. As future work, we will examine the OS system call interface, obtain a comprehensive list of sensitive calls, and investigate how to support the entire OS system call interface on MiniBox.

**Improving Performance.** The hypervisor-based isolation mechanism causes overhead in environment switches. It is expected that the hardware-based isolation mechanism provided by Intel SGX will decrease the environment switch overhead. The *VMFUNC* instruction [22] released on the latest Intel 4th Generation Processor enables software in a guest Virtual Machine to switch nested page tables without a Virtual Machine 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 how to perform secure environment switch using the *VMFUNC* instruction.

**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 [7] that supports multiple tenants, though doing so increases the TCB size. CloudVisor [50] 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. On multi-tenant cloud platforms, the virtual machine (VM) may be constructed, destructed, saved, restored, or migrated. It is critical to protect the MIEE during VM management. The MiniBox hypervisor can encrypt or decrypt the memory contents of MIEEs 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 $\mu$TPM instance together with a MIEE in VM management, to make the trustworthy computing abstractions provided to the MIEE transparent to the VM management.

## 9 Conclusion

MiniBox is a hypervisor-based sandbox that provides two-way protection between x86 native applications and the guest OS. MiniBox protects the guest OS through hypervisor-based memory isolation and OS protection modules. MiniBox significantly reduces the attack surface for both OS protection and application protection by minimizing and securing the interface between OS protection modules and the application, and protects against Iago attacks on the application. The MiniBox design and protection mechanisms are promising for establishing two-way protection on commodity computer systems. In addition, MiniBox significantly decreases the porting effort compared to previous systems for isolating security-sensitive PALs, making MiniBox practical for wide adoption. Thus, we anticipate that MiniBox will be widely adopted on systems where two-way protection is desired (e.g., the PaaS cloud computing platforms).

## Acknowledgements

Many thanks to J. Bradley Chen and Haibo Chen for helpful discussion about this work. We thank the anonymous reviewers for their time, attention, and valuable suggestions, and thank Bennet Yee, Kyle Orland, and Steve Matsumoto for improving the writing in the paper.

## References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. CFI: Principles, Implementations, and Applications. In *Proceedings of ACM Conference and Computer and Communications Security* (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* (2009), 1 – 40.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. A theory of secure control flow. In *Proceedings of 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 International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[5] ADVANCED MICRO DEVICES. AMD64 architecture programmer's manual: Volume 2: System Programming. AMD Publication no. 24593 rev. 3.14, Sept. 2007.

[6] AZAB, A. M., NING, P., AND ZHANG, X. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of ACM Conference on Computer and Communications Security* (2011).

[7] 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 *Proceedings of Symposium on Operating Systems Principles* (2003).

[8] BAUMANN, A., PEINADO, M., HUNT, G., ZMUDZINSKI, K., ROZAS, C. V., AND HOEKSTRA, M. Secure execution of unmodified applications on an untrusted host. `http://research.microsoft.com/apps/pubs/default.aspx?id=204758`, 2013.

[9] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call API is a bad untrusted rpc interface. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2013).

[10] CHEN, B., AND MORRIS, R. Certifying program execution with secure processors. In *Proceedings of HotOS* (2003).

[11] 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.

[12] 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 *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[13] 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).

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

[15] DEWAN, P., DURHAM, D., KHOSRAVI, H., LONG, M., AND NAGABHUSHAN, G. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of Spring Simulation Multiconference* (2008).

[16] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation* (2008).

[17] 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 International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[18] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of ACM Symposium on Operating System Principles* (2003).

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

[20] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[21] HOFMANN, O., DUNN, A., KIM, S., LEE, M., AND WITCHEL, E. InkTag: Secure applications on an untrusted operating system. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[22] 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.

[23] JANA, S., PORTER, D. E., AND SHMATIKOV, V. TxBox: Building secure, efficient sandboxes with system transactions. In *Proceedings of IEEE Symposium on Security and Privacy* (2011),

[24] JIANG, S., SMITH, S., AND MINAMI, K. Securing web servers against insider attack. In *Proceedings of Computer Security Applications Conference* (2001).

[25] KIM, T., AND ZELDOVICH, N. Practical and effective sandboxing for non-root users. In *Proceedings of USENIX Annual Technical Conference* (2013).

[26] LI, Y., PERRIG, A., MCCUNE, J. M., NEWSOME, J., BAKER, B., AND DREWRY, W. MiniBox: A Two-Way Sandbox for x86 Native Code. Tech. Rep. CMU-CyLab-14-001, Carnegie Mellon University, 2014.

[27] LOUP GAILLY, J., AND ADLER, M. zlib open source library. `http://www.zlib.net`.

[28] MAHESHWARI, U., VINGRALEK, R., AND SHAPIRO, W. How to build a trusted database system on untrusted storage. In *Proceedings of USENIX Symposium on Operating System Design & Implementation* (2000).

[29] MAI, H., PEK, E., XUE, H., KING, S. T., AND MADHUSUDAN, P. Verifying security invariants in expressOS. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[30] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *Proceedings of USENIX Security Symposium* (2006).

[31] 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 IEEE Symposium on Security and Privacy* (2010).

[32] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of European Conference on Computer Systems* (2008).

[33] MITCHELL, M., STERLING, A., AND MILLER, A. Cbitcoin open source project. `http://code.google.com/p/naclports/`.

[34] ONARLIOGLU, K., MULLINER, C., ROBERTSON, W., AND KIRDA, E. PrivExec: Private execution as an operating system service. In *Proceedings of IEEE Symposium on Security and Privacy* (2013).

[35] OPENSSL PROJECT TEAM. OpenSSL. `http://www.openssl.org/`, May 2005.

[36] 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.

[37] PROVOS, N. Improving host security with system call policies. In *Proceedings of USENIX Security Symposium* (2003).

[38] SINGARAVELU, L., PU, C., HÄRTIG, H., AND HELMUTH, C. Reducing TCB complexity for security-sensitive applications. In *Proceedings of European Conference on Computer Systems* (2006).

[39] SMITH, S. W., AND WEINGART, S. Building a high-performance, programmable secure coprocessor. *Computer Networks 31*, 8 (Apr. 1999).

[40] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of ACM Symposium on Operating Systems Principles* (2006).

[41] 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 IEEE Symposium on Security and Privacy* (2013).

[42] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of ACM Symposium on Operating Systems Principles* (1993).

[43] WATSON, R. N. M. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of USENIX Workshop on Offensive Technologies* (2007).

[44] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical capabilities for unix. In *Proceedings of USENIX Security Symposium* (2010).

[45] WEINHOLD, C., AND HÄRTIG, H. VPFS: building a virtual private file system with a small trusted computing base. In *Proceedings of European Conference on Computer Systems* (2008).

[46] WEINHOLD, C., AND HÄRTIG, H. jVPFS: adding robustness to a secure stacked file system with untrusted local storage components. In *Proceedings of USENIX Annual Technical Conference* (2011).

[47] YANG, J., AND SHIN, K. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of ACM Conference on Virtual Execution Environments* (2008).

[48] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY T., OKASAKA, S., NARULA, N., FULLAGAR, N., AND GOOGLE INC. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of IEEE Symposium on Security and Privacy* (2009).

[49] 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.

[50] 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 ACM Symposium on Operating Systems Principles* (2011).

[51] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of ACM Conference on Computer and Communications Security* (2012).