SICE: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-core Platforms*

Ahmed M. Azab
North Carolina State University
amazab@ncsu.edu

Peng Ning
North Carolina State University
pning@ncsu.edu

Xiaolan Zhang
IBM T.J. Watson Research Center
cxzhang@us.ibm.com

ABSTRACT

SICE is a novel framework to provide hardware-level isolation and protection for sensitive workloads running on x86 platforms in compute clouds. Unlike existing isolation techniques, SICE does not rely on any software component in the host environment (i.e., an OS or a hypervisor). Instead, the security of the isolated environments is guaranteed by a trusted computing base that only includes the hardware, the BIOS, and the System Management Mode (SMM). SICE provides fast context switching to and from an isolated environment, allowing isolated workloads to time-share the physical platform with untrusted workloads. Moreover, SICE supports a large range (up to 4GB) of isolated memory. Finally, the most unique feature of SICE is the use of multi-core processors to allow the isolated environments to run concurrently and yet securely beside the untrusted host.

We have implemented a SICE prototype using an AMD x86 hardware platform. Our experiments show that SICE performs fast context switching (67 μs) to and from the isolated environment and that it imposes a reasonable overhead (3% on all but one benchmark) on the operation of an isolated Linux virtual machine. Our prototype demonstrates that, subject to a careful security review of the BIOS software and the SMM hardware implementation, current hardware architecture already provides abstractions that can support building strong isolation mechanisms using a very small SMM software foundation of about 300 lines of code.

Categories and Subject Descriptors

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

*We would like to thank our shepherds Úlfar Erlingsson and Leendert van Doorn for helping us improve the quality of the paper, and Adrian Perrig and Jonathan McCune for their helpful discussion. This work is supported by U.S. National Science Foundation (NSF) under grant 0910767, the U.S. Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI)), and an IBM Open Collaboration Faculty Award.

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

General Terms

Design, Security

Keywords

Isolation, Trusted Computing, Virtualization Security

1. INTRODUCTION

In the last few years, a significant portion of the IT industry has moved toward cloud computing. However, cloud computing services, particularly those relying on hardware sharing, have been the source of major security concerns.

Firstly, the owner of a workload that runs inside the cloud needs to trust the cloud service provider. This trust is imposed by the commodity x86 hardware architecture that gives full memory access to the highest privileged software, which is typically an OS or a hypervisor. Secondly, recent attacks [16, 32] and vulnerability reports [23, 24] show that hypervisors are subject to security exploits and can be compromised. These attacks form a major threat to users who intend to run their workloads beside other potentially malicious ones inside the cloud.

Thus, a need emerges for a solution that provides strong isolation to workloads running in the cloud, yet still allowing hardware sharing to reduce the operating costs. To provide strong isolation, we need to minimize the code base that is granted full access to the memory of running workloads. Thus, we can minimize the exposure to security vulnerabilities that can evade the isolation provided to these workloads. Moreover, this code base should be provided with enhanced security protection and the ability to attest to its integrity.

To achieve this objective, we introduce a prototype system that provides a strongly isolated execution environment, which relies on a trusted computing base (TCB) composed of the hardware, the BIOS, and the System Management Mode (SMM).

Our prototype represents a test system that aims to explore the capability of current hardware platforms in providing more secure isolated environments. We demonstrate that current hardware architecture already provides abstractions that can support strong isolation. Moreover, we show that building strong isolation mechanisms on top of those abstractions requires a very small software foundation of about 300 lines of code (LOC), which tremendously reduces the TCB size compared with previous techniques.

Since the SMM was neither designed nor implemented with high-assurance security mechanisms in mind, detailed security reviews by both CPU and platform vendors would be necessary to verify that current SMM implementations

are properly done to support such strong isolation guarantees provided by our prototype in practice.

1.1 Previous Attempts

There have been a few recent attempts to tackle the problem of isolating sensitive workloads while eliminating the host OS or hypervisor from the TCB of these workloads. These attempts can be divided into two main categories: (1) microhypervisor-based approaches, and (2) hardware-based approaches.

Microhypervisor-based Approaches: These approaches rely on a thin, privileged software layer (i.e., a thin hypervisor) to provide the required isolation for sensitive workloads. Among the notable research efforts in this direction are NOVA [27] and Trustvisor [19].

NOVA proposes to replace current hypervisors with a microhypervisor that is around 9 KLOC in size. Despite having a small TCB compared to commodity hypervisors, NOVA is still responsible for several management tasks (e.g., address space management, interrupt and exception handling, and communication between the running workloads). Thus, its TCB is still relatively complicated.

Trustvisor minimizes the code base of the microhypervisor even further (about 2 KLOC for its core functions), so that it can be used for isolation purposes only. However, Trustvisor is only designed to handle workloads with a small code base and only supports systems with a single processor core. This makes it unsuitable for typical compute clouds.

An effort closely related to these microhypervisor-based approaches is seL4 [15]. seL4 proposes a technique to formally verify a microkernel, which is around 8.7 KLOC, to avoid security vulnerabilities. Although this microkernel can be used for isolation purposes, the formal verification process imposes several restrictions on the microkernel functionality. Thus, it cannot be extended to fully support all the functionalities required from micorhypervisors yet.

Hardware-based Approaches: These approaches rely on hardware security extensions to provide isolation for sensitive workloads. The main advantage of these approaches is the enhanced protection for the isolated workloads (compared with software based techniques).

One notable research effort in this direction is Flicker [20], which is a system that uses the late launch capability to run a secure verifiable workload. However, the late launch capability, provided by both Intel [11] and AMD [1], incurs significant overhead (in the magnitude of hundreds of milliseconds) on every context switch to the isolated environment. Hence, it cannot provide a practical solution for cloud computing environments.

Another effort, NoHype [14], also relies on hardware isolation through assigning dedicated processor cores to each running workload. However, NoHype still relies on a thin software layer to achieve the required protection and manage the hardware resources (e.g., page tables). Moreover, it requires architectural changes to processors and hardware peripherals, which are slow to realize.

It should be noted that all of the above techniques are research prototypes that make several simplifying assumptions about their target platforms. These assumptions may hinder the applicability or even weaken the security guarantees provided by these techniques. For instance, the formal verification introduced by seL4 does not include the firmware or the SMM code, which could negate all seL4 guarantees. Thus, a practical deployment of any of these research prototypes, including the prototype we present in this paper, requires a comprehensive consideration of the target platform that includes the full hardware and firmware specifications.

1.2 Introducing SICE

In this paper, we present SICE, which stands for Strongly Isolated Computing Environment, a framework that provides a hardware-level isolated execution environment for x86 hardware platforms. SICE's main objective is to minimize the TCB required to create an isolated execution environment on commodity x86 platforms. This isolated environment can be used to host a security sensitive workload.

SICE achieves this objective by relying on a TCB that is only composed of the hardware, the BIOS, and the SMM. The TCB, which is fundamentally different from previous research, gives SICE principal advantages over both microhypervisors and hardware-based isolation techniques. We summarize these advantages below.

Smaller Attack Surface: SICE utilizes the hardware protection provided by commodity x86 processors for the SMM and the memory that hosts its code, which is called System Management RAM (SMRAM). There are two fundamental differences between the SMM and microhypervisors.

First, the SMM can only be triggered by a single interface, which is to invoke a System Management Interrupt (SMI). In SICE, the SMI handler is required to execute one of only four functions upon receiving an SMI, which are to create, enter, exit and terminate an isolated environment. Implementing these functions requires the system to run *briefly* in the SMM. Moreover, the SMI handler is not required to handle any other interrupts because all interrupts are disabled upon entering the SMM mode. The SMI handler is also *not* responsible for managing the communication between running workloads. On the other hand, microhypervisors reside at the system's highest privileged level. Thus, they have to handle all system events (e.g., hypercalls, interrupts and exceptions). They are also required to manage the communication channels (e.g, shared memory pages) between different workloads.

Second, SICE uses the SMRAM to provide the needed memory isolation. After the SMRAM is initialized by the BIOS, it can be locked so that no software can access its contents except for the SMM code. The SMM code can manage the SMRAM through modifying only two registers, which has a huge impact on decreasing the size of the TCB. In contrast, microhypervisors that rely on hardware virtualization have to manage a different set of page tables for every isolated environment to provide memory protection.

To sum up, SICE's SMM code base is better protected and less complicated than any microhypervisor. Moreover, this code does not provide any functionality other than the required isolation, which results in a very small code base. For instance, our prototype SMI handler consists of around 300 LOC (excluding cryptographic libraries). This is around an order of magnitude less than current microhypervisors (e.g., Trustvisor and NOVA). As discussed in [15], this is a significant difference when it comes to verification cost. For instance, using the industry rules-of-thumb of $10K per LOC for common criteria certification as a guideline for the cost of code verification, the cost for verifying 500LOC is $5M versus $100M for 10 KLOC.

Compatibility with Existing Software Systems: Un-

like isolation techniques that monopolize the highest privileged execution level of the target platform (e.g., Trustvisor), SICE does not exclude running legacy workloads (e.g., a hypervisor with multiple VMs) on the same physical platform. In other words, a platform using SICE can offer isolated environments, and at the same time accommodate legacy virtualization software.

Feasible Hardware-based Isolation: SICE uses existing hardware features to provide the required isolation. We have successfully implemented a SICE prototype using a commodity AMD processor. Thus, it does not require fundamental changes to current hardware architecture.

Moreover, our performance evaluation shows that SICE performs a secure context switching with an isolated environment that is four orders of magnitude faster than systems that rely on the late launch capability (e.g., Flicker [20]). It also uses multi-core processors to allow isolated workloads to run in parallel to a legacy hypervisor or OS. Thus, SICE avoids the two main drawbacks of using late launch, which are the high performance overhead, and the dedication of all system resources to only one isolated workload.

1.3 SICE Overview

SICE introduces novel techniques that allow the isolated workloads to run in parallel with a host OS or hypervisor. For convenience, we refer to the OS or hypervisor along with all the software running on it as the *legacy host*, a strongly isolated computing environment, which supports an isolated workload, as an *isolated environment*, and the code that manages the isolation between these environments as the *SMI handler*, or simply SICE.

The SMI handler, which represents the TCB of the isolated environments, resides inside the SMRAM. It is the only part of our framework that executes in the SMM. The SMI handler is responsible for two main tasks: 1) maintaining the memory isolation of the isolated environments, 2) securely initializing the isolated environments and attesting to their integrity. In SICE, these tasks require the SMI handler to run for a very short time.

An isolated environment is composed of two components: an isolated workload and a security manager. The isolated workload is a user-provided system that runs in the isolated environment. It can be any software, ranging from a single program (e.g., a program that manages secret keys) to a complete VM (e.g., a VM that runs a web server).

The security manager is a thin software layer that has limited functionalities such as handling exceptions and managing page tables. It is mainly responsible for confining the isolated workload. Due to the commodity hardware limitation on SMRAM size, SICE's unique hardware-level isolation is not used to protect the legacy host from the isolated environments. Thus, SICE uses the security manager to prevent the isolated workload from accessing the memory of the legacy host. A separate copy of the security manager is generated by SICE for every isolated workload running on the system. Both the security manager and the isolated workload run after the system returns from the SMM.

Though SICE requires the legacy host to trust the security managers, it does not weaken the hardware-level isolation provided to the isolated workloads. SICE uses SMM to protect isolated environments from the legacy host. Even if a malicious workload (in one isolated environment) compromises its own security manager and consequently the legacy host, it will not be able to compromise any other isolated environment running on the same platform.

SICE's protection of the legacy host may appear to be equivalent to those provided by microhypervisor-based approaches such as NOVA. Indeed, the security manager, which is a thin privileged software layer, is similar to a microhypervisor in terms of its required tasks and code size. However, SICE also provides hardware-level, stronger protection for the isolated environments, which are not available in microhypervisor-based approaches.

SICE provides two operating modes: *time-sharing mode* and *multi-core mode*. In both modes, the SICE philosophy is based on using the isolated environments to run security sensitive workloads, while the legacy host is used for running less sensitive workloads and managing hardware peripherals. In a typical execution scenario, a communication channel is used so that the legacy host provides hardware services (e.g., networking) to the isolated environments. The communication channel can be established using a shared memory outside of the memory range protected by SICE. Communication channels are not controlled or managed by the SMI handler. Thus, the code responsible for managing the communication is not a part of the TCB.

In the time-sharing mode, time multiplexing is used to share the hardware platform between the isolated environments and the legacy host. During the environment switching, SICE guarantees a fresh start of the processor, complete memory isolation between these environments, and a timely switching that does not largely impact the performance.

In the multi-core mode, SICE assigns one or more processor cores to each isolated environment, while the other cores are used to run the legacy host. SICE guarantees isolation of both the processor cores and the memory dedicated to each of the concurrently running environments.

In both modes, SICE attests to the integrity of each isolated environment to remote users, while avoids revealing sensitive information about the workload to the legacy host.

The current SICE design relies on hardware features provided by AMD processors. Some of these features are not currently supported by Intel processors, particularly the ability to resize the SMRAM at runtime and defining a separate SMRAM for each processor core. Proposing alternative techniques to implement SICE on Intel platforms will require further research. We discuss these issues in Appendix A.

We implement a prototype of SICE on an IBM LS22 blade server that uses AMD processors. We use our SICE prototype to run a complete Linux VM in the isolated environment. Our experimental evaluation shows that the time required to enter and exit an isolated environment using SICE is around 67 μs. We conduct experiments to evaluate the performance of the isolated VM. In the multi-core mode, SICE incurs a low (under 3%) overhead on VM operations that do not require frequent communication with the legacy host. The time-sharing mode shows a higher overhead (around 10%) due to the time for the context switching. A test dedicated to measure the performance of network emulation shows a higher overhead. However, this overhead is expected to decrease using an optimized implementation of network emulation for isolated VMs.

1.4 Summary of Contributions

We make several technical contributions in this paper:

- We provide a complete and feasible solution to share

hardware resources with an isolated execution environment that does not rely on any host software. Instead, it relies on a TCB that is around an order of magnitude smaller than the state-of-the-art systems.

- We provide a novel technique that allows concurrent execution of the isolated environments with the untrusted host environment.
- We provide attestation to the integrity of the isolated workloads without revealing sensitive information about the workloads to untrusted host software.
- We implement a prototype of SICE on an AMD platform. We use this prototype to evaluate the performance overhead introduced by SICE.

The rest of this paper is organized as follows. Section 2 provides background information on SMM. Section 3 discusses our assumptions and threat model. Section 4 presents SICE in detail. Section 5 presents our prototype implementation. Section 6 presents our experimental evaluation. Section 7 presents related work, and Section 8 concludes this paper with some future research directions. Finally, The Appendix discusses SICE's portability to Intel platforms.

2. BACKGROUND

In this section, we briefly give some background information on the System Management Mode (SMM).

Commodity x86 architecture supports the SMM as one of its operating modes. The processor enters the SMM when receiving an SMI. Upon an SMI, the processor saves its state to a dedicated state save map and switches to the SMM. To return from the SMM, the special instruction `RSM` restores the saved processor state and resumes normal execution.

SMM code is stored in a designated memory called SMRAM. To provide protection of the SMM code and data, both AMD and Intel provide the capability of locking the SMRAM. When the SMRAM is locked, all accesses to it, except from within the SMM, are prevented. All interrupts, including non-maskable ones, are disabled upon entering the SMM. Thus, no other code running on the system can interfere with the SMI handler.

Current hardware can support up to 4 GB of SMRAM. There are two types of SMRAM: the `ASeg`, which is located at a fixed low address, and the `TSeg` which is located at a variable base and has a variable size. SICE uses `TSeg` due to the flexibility it offers. Defining the SMRAM location and range differs slightly between AMD and Intel architectures. In the following, we take a closer look at AMD architecture, on which this paper focuses.

The AMD architecture defines the `TSeg` memory range using two Model Specific Registers (MSRs) that are local to each processor core. The first is `SMM_Addr`, which specifies the SMRAM base address, while the second is `SMM_Mask`, which specifies its range. Since MSRs are local to each processor core, all cores have to set their MSRs to provide complete protection of the SMRAM.

The `SMMLOCK` bit in the `HWCR` MSR can be set to prevent changing the `TSeg` range. Moreover, the AMD architecture uses a password mechanism to protect the modification of the `SMMLOCK` bit. When the SMRAM is unlocked, writes to the `SMM_KEY` MSR set the 64-bit password. When the SMRAM is locked, writing the correct password to the same MSR clears the `SMMLOCK` bit.

3. THREAT MODEL AND ASSUMPTIONS

Threat Model: SICE aims at defending against all malicious activities by software running in the legacy host that are targeted at compromising the isolation offered to the isolated environment (e.g., malicious activity that result from exploiting a vulnerability in the hypervisor). Specifically, SICE protects the isolated environment from all types of unauthorized memory accesses or any modification to its execution environment. The protection starts from the moment the isolated environment is initialized by SICE. Upon initialization, the initial image of the isolated workload, which is loaded by the legacy host, is measured so that SICE can further attest to its integrity.

We consider the following attacks out of the scope of this paper: Attacks aimed at the availability of SICE (e.g., denying its network access), and those that directly exploit vulnerabilities of the isolated environment through legitimate communication channels. Such attacks are not specific to the isolation mechanism, and should be addressed by other techniques such as keep-alive messages and patching the vulnerabilities. Moreover, SICE is not responsible for securing the hardware services provided by the host. Thus, the isolated workload should use other techniques to achieve this objective (e.g., encrypting its network traffic).

Side-channel attacks are also out of the scope of this paper. Adopting SICE prevents cache side-channels because the hardware automatically clears the cache upon entering and exiting from the SMM. However, these attacks in general are not unique to our approach and require further research.

Assumptions: We assume that our platform is physically secure (e.g., locked in a server room) so that an adversary cannot launch any hardware attack. Moreover, we assume that the target platform is equipped with trusted computing hardware, including the Core Root of Trust Measurement (CRTM) and Trusted Platform Module (TPM) [29]. This allows the attestation to the integrity of key software components (e.g., the SMI handler).

We assume that the SMM is properly isolated from other software running on the system and the hardware provides the SMRAM with proper isolation from all unauthorized memory accesses (e.g., cache poisoning, Direct Memory Access (DMA)). Recent incidents showed that attackers were able to subvert the SMRAM using cache poisoning [6, 33]. Fortunately, such attacks cannot be mounted on AMD platforms due to its SMRAM cache protection and can be easily defeated on Intel platforms using proper setting of the System Management Range Register (SMRR) [13].

4. SICE DESIGN

In this section, we present the design of SICE. The objective is to enable hardware-level strongly isolated computing environments that run in parallel with the legacy host on the same hardware platform.

Implementation requirements: To implement SICE, the SMI handler needs to be modified to include SICE's code. Recently, most hardware vendors use the BIOS to lock the SMRAM to prevent potential SMM misuse. Thus, SICE requires hardware vendors to allow adding its code to the SMI handler before locking the SMRAM.

The legacy host (e.g., the hypervisor) is required to add an interface that invokes an SMI to trigger SICE. Hardware management functions provided by the legacy host (e.g.,

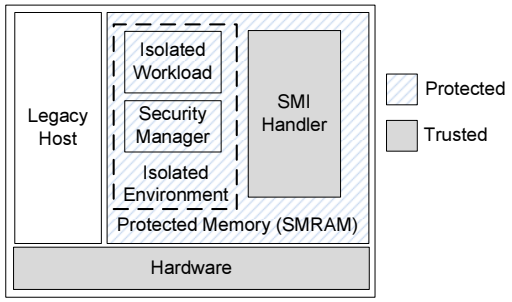

Figure 1: SICE Architecture

hardware device drivers) should be modified to use this interface to provide its services to the isolated environment. Note that the legacy host may refuse to invoke an isolated environment or deny it some services, thus threatening its availability. However, these attacks are easily detectable by SICE and its remote users (e.g., from the lack of response of the isolated workload), and can then be thwarted by replacing the faulty legacy host.

SICE Architecture: Figure 1 shows the architecture of SICE. It consists of three components: The SMI handler, the security manager, and the isolated workload. The security manager and the isolated workload constitute the isolated environment. SICE enables the isolated environment and the legacy host to share the physical platform's resources.

The security manager is a thin software layer that confines the isolated workload. It has limited functionalities such as handling exceptions and managing page tables. The initial image of the security manager should be loaded to the SMRAM, along with the SMI handler, upon the system initialization. Thus, we assume trust in both the SMI handler and the security manager based on the system's trusted boot, which can be attested to later using the TPM.

The isolated workload is a user-provided system that runs in the isolated environment. It can be any software, ranging from a single program to a complete VM. The only restrictions on this software are that it does not use more than 4GB of memory (due to SMRAM constraints) and that it does not manipulate hardware peripherals directly.

When the isolated environment is not actively running, its code and data are securely stored in the SMRAM (as shown in Figure 1). However, the SMM is a limited execution mode that is not suitable for running the isolated environment. For instance, code execution inside the SMM is slower than the protected mode [2]. Moreover, some hardware functionalities (e.g., starting a hardware assisted VM) are not supported in the SMM. Thus, SICE only uses the SMM to prepare and enter the isolated environment. SICE uses novel techniques to move the security manager and the isolated workload out of the SMRAM after the isolated environment is initialized.

In the following, we discuss SICE in a *time-sharing mode*, where the legacy host and the isolated environment time share the physical platform. We then present the *multi-core mode*, where the legacy host and the isolated environment run in parallel using multi-core processors.

4.1 Time-sharing Mode

In the time-sharing mode, SICE provides two important features: (1) fast context switch between the legacy host and the isolated environment (in the magnitude of few tens of microseconds), and (2) large protected memory range for the isolated workload (up to 4GB). The fast context switch allows the isolated workload to receive its input data and send its output to and from the legacy host without posing significant overhead on the system performance. The large protected memory range enables the isolated environment to securely keep its state across the context switches.

4.1.1 Initializing the Isolated Environment

Both the security manager and the SMI handler are initialized when the physical platform is booted. When an isolated workload is ready to be started, the legacy host first loads the initial image of this workload to a specific memory range. Then, it triggers SICE using an SMI. The SMI handler then measures the initial image of the isolated workload and copies it to the protected SMRAM. Details on measuring and attesting to the isolated environment are discussed in Section 4.3.

4.1.2 Entering the Isolated Environment

Figure 2 shows the process of entering the isolated environment. Whenever the legacy host requires the isolated workload to run, it triggers another SMI. The SMI then switches the processor's execution environment to the SMM and the execution jumps to the SMI handler. The SMI handler then prepares the isolated environment by changing the saved processor state so that the security manager, instead of the legacy host, runs after the processor returns from the SMM. The SMI handler also stores the processor state of the legacy host so that it can resume execution after the isolated environment finishes its execution.

In the time-sharing mode, SICE gives the isolated environment full control of the physical platform as soon as it runs. Thus, it changes the processor state (e.g., interrupt descriptors, system call handlers) so that only the security manager will have control after the SMI handler returns.

Unfortunately, the SMM cannot change most of the critical processor states (e.g., the interrupt descriptor table (IDT) register and the CR3 register). To overcome this challenge, the SMI handler relocates the code pointed to by the current processor state through modifying the page tables. Since the CR3 register is not writable, the SMI handler modifies the first level page table pointed to by the CR3 so that it directly points to the security manager. It also flushes the Translation Lookaside Buffer (TLB) to avoid potential race conditions with the cached page tables.

Before the SMI handler returns, we need to ensure that both the security manager and the isolated workload are accessible to the processor after it returns from the SMM. As shown in Figure 1, both the security manager and and the isolated workload are stored in the SMRAM before the isolated environment is entered. These two components need to be moved out of the SMRAM so that they can run after the processor returns from the SMM.

A straightforward solution is to copy the security manager and the isolated workload out of the SMRAM to an unprotected memory. However, this will introduce an unacceptable performance overhead.

To address this problem, we use the ability of AMD processors to resize the SMRAM during the system runtime. This feature, discussed in Section 2, relies on a password-protected mechanism to clear the SMM lock so that the `SMM_Addr` and the `SMM_Mask` registers can be updated.

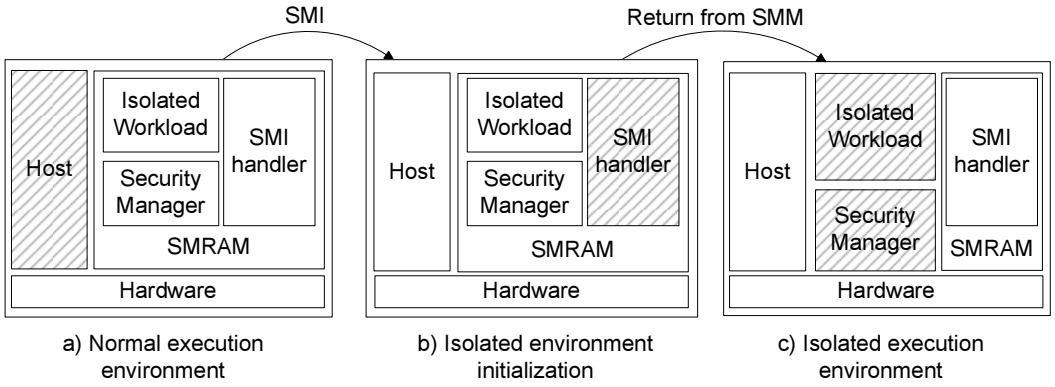

Figure 2: Entering SICE from the legacy host (actively running components are shaded): a) An SMI triggers SICE operations. b) The SMI handler prepares the isolated environment and the new SMRAM layout. c) After entering the isolated environment, the security manager prevents the isolated workload from accessing the legacy host memory.

Upon the initialization of the system, SICE generates a 64-bit random number, sets it as the password for locking/unlocking the SMRAM, and stores it inside the SMRAM. Before entering the isolated environment, the SMI handler uses this password to unlock the SMRAM and modify its protection range (by modifying the `SMM_Addr` and the `SMM_Mask` registers) to exclude both the security manager and the isolated workload. Thus, they can be accessed by the processor after the SMI handler returns.

As shown in Figure 2, the SMRAM protection remains for the SMI handler while the isolated workload runs. This guarantees that the SMI handler (including the SMRAM password) can never be accessed by any other software running on the system, including the isolated workload.

A subtle issue needs to be clarified. To guarantee full control over the system, the isolated environment should run in the highest privileged level after the SMI returns. However, this requires the legacy host to trigger the SMI from within the highest privileged level. In other words, the SMI cannot be triggered by a guest VM. Otherwise, the isolated environment will return to the context of a guest VM controlled by the legacy host.

To verify that the SMI is triggered by the highest privileged level, SICE requires the legacy host to disable virtualization by clearing the `SVME` bit in the `EFER` register before triggering the SMI. The SMI handler verifies that this bit is clear. Otherwise, it will not enter the isolated environment and keep the SMRAM protection for its memory.

4.1.3 Managing the Isolated Workload

Since the isolated workload is provided by the user, it is not trusted by SICE. Hence, it should not be allowed to tamper with the hardware configuration or access memory regions that belong to the security manager or legacy host.

Unfortunately, commodity hardware architecture cannot assign more than 4GB of memory to the SMRAM. Thus, using the SMRAM to protect the legacy host memory from the isolated workload will not be a feasible solution due to its restriction on the memory capacity of the legacy host.

To address this challenge, the security manager plays the role of a hypervisor and runs the isolated workload in the context of a guest VM. Therefore, the isolated workload's execution environment is restricted so that it cannot execute privileged instructions. Moreover, the security manager crafts the page tables of the isolated workload so that it can access a limited specific range of physical memory. Thus, the isolated workload cannot access any memory range that belongs to either the security manager or the legacy host.

4.1.4 Exiting the Isolated Environment

To allow the legacy host and the isolated environment to efficiently share the hardware resources, SICE provides a technique to securely and properly exit the isolated environment and return the execution back to the legacy host.

To return back to the legacy host, the isolated environment triggers an SMI. The SMI handler then: (1) Clears all general purpose registers, (2) flushes all cache levels, (3) uses the SMM password to change the SMRAM range so that it covers both the security manager and the isolated workload, and (4) restores all changes done to the legacy host page tables before the isolated environment was initialized.

The first two steps ensures that no sensitive data is leaked to the legacy host. The third step ensures the protection of the isolated environment's memory. Finally, the last step ensures that the legacy host resumes its operations correctly.

4.1.5 Terminating the Isolated Workload

When the isolated workload finishes execution, it needs to be securely terminated so that no confidential information are leaked to the legacy host. Thus, isolated workload termination requests must be forwarded to the SMI handler, which securely erases the memory belonging to the isolated workload and then removes it from the protected SMRAM.

4.2 Multi-core Mode

The SICE multi-core mode enables the legacy host and the isolated environments to run in parallel on different processor cores to better utilize the hardware resources.

Before presenting our solution, we first give some background information on multi-core processors. As shown in Figure 3, multi-core processors are equipped with one or more processing nodes. Each node has its own memory control hub (north bridge) and one or more processor cores. North bridge configuration registers can be accessed by any core on any node. On the other hand, each processor core has its own general-purpose registers, MSRs, APIC, and two

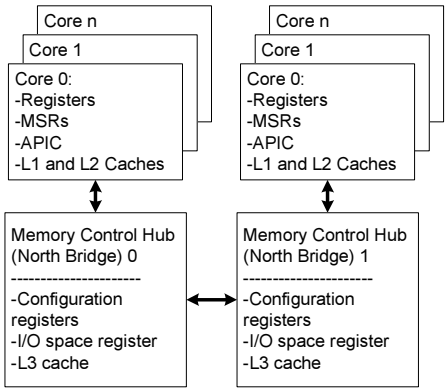

Figure 3: General multi-core processors architecture

levels of cache. These components can only be accessed by their designated core. Among these core-dependent components are the `SMM_Addr` and `SMM_Mask` MSRs that define the SMRAM memory range. Thus, each processor core can have its own protected SMRAM memory range that is independent from the SMRAM memory range defined on other processor cores that share the same node. SICE utilizes this architecture to provide the necessary protection for the isolated environment in the multi-core mode.

For simplicity, we assume a platform with two cores in our discussion: the *host core* that runs the legacy host, and the *isolated core* that runs the isolated environment. However, the same techniques can be used to assign more than one core to either the legacy host or the isolated environment. In general, two processor cores that belong to different execution environments can be either co-located on the same processing node or located among different nodes.

In the multi-core mode, SICE initializes the isolated environment in the same way as the time-sharing mode (i.e., by triggering an SMI in the legacy host). The SMI switches the execution environment of all processor cores that exist on the target platform to the SMM. The SMI handler performs a different task on each processor core. On the isolated core, the SMI handler prepares the isolated environment and returns to the security manager using the techniques described in Section 4.1.2. Meanwhile, on the host core, the SMI handler returns directly to the legacy host so that it can resume its operations at the same time.

Since both the legacy host and the isolated environment run in parallel on different processor cores, no context switching between these environments is needed.

To ensure the isolation between the legacy host and the isolated environments, SICE prevents the legacy host from affecting the execution of the isolated environment, and the isolated workload (excluding the security manager, which needs to be trusted by the legacy host) from affecting the execution of the legacy host. In particular, we need to prevent the interference through explicit inter-core communication as well as those through modifying each other's memory.

4.2.1 Handling Inter-core Communication

Communication between processor cores, which is done through Inter-Processor Interrupts (IPIs), can modify the execution environment on the recipient code by changing the execution path. This can pose a risk on the integrity of the isolated environment. Therefore, all IPIs should either be blocked or securely handled by the recipient core.

There are two types of IPIs: (1) Maskable IPIs that can be blocked by the recipient core's Advanced Programmable Interrupt Controller (APIC), and (2) non-maskable IPIs (e.g., NMI, SMI, startup, and INIT) that cannot be blocked.

Upon initializing the isolated environment, the SMI handler disables all maskable interrupts on the isolated core so that the host core cannot interfere with the isolated core. The security manager, which runs as a thin hypervisor on the isolated core, keeps these interrupts disabled.

On the other hand, non-maskable interrupts cannot be blocked. Moreover, a specific non-maskable interrupt, the startup interrupt, resets the processor core to start execution at a low physical memory address. Intuitively, this address can be modified by the host core to alter the course of execution on the isolated core.

To overcome this problem, SICE relies on the Global Interrupt Flag (GIF) introduced by AMD. When the GIF is clear, all interrupts, including non-maskable ones, are ignored or held pending. The security manager thus clears the GIF of the isolated core when the isolated environment is entered, and sets it only after the isolated environment completes its execution and all memory protection measures are taken.

The security manager then runs the isolated workload in the guest VM mode. Upon entering the VM mode, both global interrupts and maskable interrupts are re-enabled. Hence, the isolated workload can receive all interrupts including both IPIs and local processor interrupts. However, the received interrupts will cause a VM exit and jump to the security manager rather than modifying the execution environment.

Global interrupts are automatically disabled again before exiting the VM mode and jumping to the security manager. The security manager identifies the reason that causes the VM exit. If it is an interrupt that aims to modify the execution environment (e.g., INIT interrupt), then it indicates a malicious activity from the host core and all memory protection measures are taken. However, if the VM exit is caused by a local interrupt or an IPI, the interrupt is forwarded back to the isolated workload. This architecture allows the isolated workload to use IPIs as method of signaling, which is required to build a communication channel with the legacy host. This channel can be used to provide hardware services offered by the legacy host to the isolated workload.

On the other hand, SICE relies on hardware virtualization to prevent the isolated workload from perturbing the legacy host. The isolated workload, which runs in a guest VM, is not allowed to directly access the APIC to send any interrupts that can perturb the execution of the host core.

4.2.2 Memory Isolation

To prevent the legacy host from accessing the isolated environment's memory, SICE relies on a novel protection called the *memory double-view technique*, shown in Figure 4. This technique relies on the fact that AMD processor's SMRAM is defined based on the *core-dependent* MSRs. Thus, different cores can view the SMRAM differently depending on the values of their own `SMM_Addr` and `SMM_Mask` registers.

As shown in Figure 4, each processor core has its own view of the physical memory. From the host core's perspective, the isolated environment uses a physical memory that lies in the SMRAM memory range. Hence, the legacy host cannot

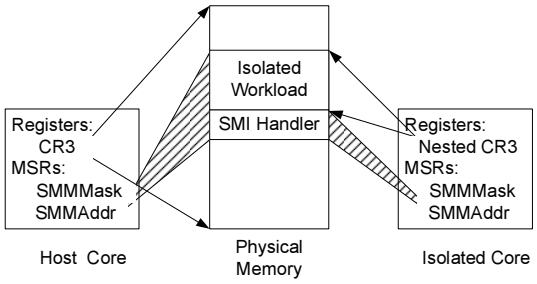

Figure 4: Double view of memory from different processor cores. From the host core's view, the isolated environment is part of the SMRAM. From the isolated workload's view, the SMI handler is protected by the SMRAM and the legacy host is protected by the security manager via hardware assisted paging.

access such memory due to the SMRAM protection. From the isolated core's perspective, the isolated environment lies within a normal memory region that is not part of the SMRAM. Hence, the isolated environment can run normally on the isolated core.

SICE also prevents the isolated workload from accessing the legacy host memory. The security manager, which plays the role of a hypervisor, restricts the isolated workload to its assigned physical memory. Nevertheless, a shared memory, which does not fall in the host core's SMRAM, is mapped to allow communication between the two environments.

4.3 Attestation and Secure Communication

Cloud computing users are required to trust the environment that runs their workloads. In SICE, attesting the integrity of the isolated environment is complicated by the fact that the legacy host has full access to all hardware peripherals and communication channels. In this section we provide a three-step solution to address this challenge.

Attesting to Integrity of SICE: SICE requires the platform to use standard trusted boot [28]. After a typical trusted boot, the TPM securely stores the measurement of the boot process, which includes the SMI handler, the code image of the security manager and the code that loads them and locks the SMRAM. The measurement is stored in TPM special registers that cannot be erased by malicious software.

The TPM can further use its private Attestation Identity Key (AIK) to attest to the integrity of this measurement to remote users. Unlike other system software that is continuously interacting with potential attackers, the SMI handler cannot be modified by any code running on the system and the trust in it can be maintained.

Attesting to Integrity of Isolated Workload: In turn, the SMI handler attests to the integrity of the isolated workload. It measures the isolated workload before it is first invoked and stores the measurement in the secure SMRAM.

To enable the attestation, we adopt an approach we previously used is HyperSentry [2]. Specifically, SICE generates a public/private key pair during system boot. The private key is securely stored in the SMRAM, and the public key is securely stored inside the TPM.

To attest to the integrity of the isolated workload, the SMI handler signs the measurement of the initial workload image using its private key. The remote user accepts the measurement output only if the private key matches the public key stored in, and attested to by, the TPM.

Secure Communication with the Isolated Environment: SICE allows the establishment of a secure communication channel between the isolated workload and its remote owner using standard cryptographic techniques. To enable the remote owner to authenticate the isolated workload, SICE provides a public/private key pair for the isolated workload. The public key is sent to the user as a part of the workload attestation evidence signed by the SMI handler. The private key is directly provided to the isolated workload. Thus, it is never exposed to potentially malicious code running inside the legacy host. On the other hand, the isolated workload should include the public key of its remote owner as a part of its initial image.

With both the security manager and the remote owner being able to validate each other's public keys, we can easily modify, for example, SSL for secure communication without leaking information to the (untrusted) legacy host.

4.4 Security Analysis

Now we discuss the security of the isolated environment, including its confidentiality, integrity, availability, and TCB. We also discuss the security implication of using the SMM.

4.4.1 The Security of the Isolated Environment

Confidentiality: To protect the confidentiality of the isolated environment, SICE prevents potential attackers from accessing its memory. SICE uses the SMRAM to achieve this objective. The hardware protects the SMRAM from access requests made by both the CPU and direct memory access (DMA) capable devices.

In the time-sharing mode, SICE modifies the SMRAM memory range according to the running environment. Whenever the legacy host is running, SICE extends the SMRAM to include the memory of the isolated environment. Thus, this memory is protected from all memory requests.

When the isolated environment is triggered, SICE takes two security measures to guarantee its confidentiality. First, SICE assures that the isolated environment fully controls physical platform. Hence, the legacy host is not active to threaten the isolated environment, given that modern processors do not use cached memory upon entering the SMM to avoid cache poisoning attacks [1, 13]. Second, SICE uses hardware DMA exclusion (e.g., AMD's DMA exclusion vector (DEV) [1] and Intel's VT-d [12]) to prevent DMA capable devices, which could be maliciously programmed, from accessing the protected memory.

In the multi-core mode, SICE's memory double-view technique provides the needed protection. According to architecture manuals, memory access requests made from a processor core are checked against its own SMRAM before being routed. Hence, the host core will not be able to access the protected memory that falls within its own SMRAM range. The same is true for requests to update or retrieve cache entries. Hence, cache poisoning is not possible.

In the multi-core mode, SICE cannot rely on hardware DMA exclusion because its control registers exist in the memory control hub, which is accessible by all processor cores. Thus, SICE relies on the SMRAM protection to prevent DMA access to the isolated environment's memory. Typically, the memory control hub does not allow DMA access to the SMRAM. However, the AMD hardware ar-

chitecture manual does not precisely define which processor core's SMRAM range is used to prevent DMA access, particularly when different SMRAM ranges are defined on different cores. Thus, implementing SICE on a specific platform requires verifying which SMRAM range is protected from DMA. The specific processor core(s) that defines this range is the same one that should be used as the untrusted host core. As shown in figure 4, the host core's SMRAM memory range includes the whole memory range used by the running, one or more, isolated environments.

A final threat to the confidentiality of the isolated environment is brute force attacks against the SMM password. For each processor core, SICE generates a 64-bit random value to be used as the SMM password. Thus, it is computationally infeasible to break this password. The random password generation can be done using the TPM.

Integrity: In the time-sharing mode, the legacy host cannot threaten the integrity of the isolated environment because they do not run concurrently. In the multi-core mode, the host core cannot access the registers, MSRs and APIC of the isolated core. However, the host core can still modify system-wide configurations that rely on shared resources (e.g., the memory control hub and the IO control hub). Next, we prove that such configurations can only perturb the isolated environment, for example, by rebooting the system, without threatening its confidentiality or integrity.

According to the Advanced Configuration and Power Interface (ACPI) specification [10], processor cores keep their state as long as the processor runs in the S0 or S1 power modes. Thus, the isolated core will keep running the isolated environment as long as either S0 or S1 state is maintained. On the other hand, all other ACPI states will cause the processor core to lose its state and resume execution from the non-volatile memory (i.e., the BIOS). Since the BIOS belongs to the trusted computing base, changing the ACPI state will be detected by SICE. In all cases, the ACPI implementation of the target platform needs to be carefully reviewed to securely implement SICE.

As presented in Section 4.2, attacks that use IPI between processor cores are thwarted using APIC setting and the GIF of the processor core that runs the isolated environment.

Availability: SICE does not provide protection against attacks that target the availability of the isolated environment. Example attacks include perturbing the isolated environment through system reboots or denying it network access. However, this type of attacks is easily detectable by SICE and its remote users (e.g., from the lack of response of the isolated environment), and can be easily thwarted by removing the malicious code from the legacy host.

The TCB of SICE: SICE aims to minimize and enhance the protection of the TCB of the isolated environment so as to maximize its security.

The TCB of the isolated environment consists of the hardware, the BIOS, and the SMM. Using the SMM gives SICE two main advantages over microhypervisor-based isolation. First, SICE TCB enjoys the hardware protection provided for the SMRAM. Second, the SMM's attack surface is much smaller than that of microhypervisors. Thus, The TCB of the isolated environment is better protected and less complicated than that of microhypervisors. Section 1.2 compares the TCB of SICE with that of microhypervisors in detail.

Trusting the BIOS is required to start the trust chain based on the static root of trust management (SRTM) technique. However, recent incidents [17] show that the SRTM can be compromised. Nevertheless, SICE can adopt the Dynamic root of trust management (DRTM) to start the trust chain, which consequently eliminates the BIOS from the TCB. SICE can use the DRTM to invoke a trusted code that securely initializes the SMRAM, given that it will not be locked by the BIOS. A similar technique was used by Trustvisor [19] to initialize its TCB.

SICE requires the legacy host to trust the security manager(s). However, this does not make the security guarantee provided to the legacy host weaker than microhypervisor-based approaches such as NOVA, which assume trust in a thin hypervisor with typical duties. Indeed, SICE and the microhypervisor-based approaches provide similar security guarantees for the legacy host, while SICE additionally also provides a stronger protection for the isolated environments, as discussed in Section 1.3.

4.4.2 SMM Security

Despite being an integral part of the TCB of SICE, the SMM was not designed to provide security services. It was originally designed to create a shadow environment to run hardware management tasks. In the following, we discuss some of the important issues that need to be considered before deploying SICE, or any other system that relies on the SMM, in a production environment.

Legacy SMM Tasks: Intuitively, the TCB of SICE includes any code that exists in the SMRAM. To keep the TCB minimal, legacy system management tasks need to be eliminated from the SMM. This does not mean that SICE needs to completely eliminate them from the system. Instead, SICE can simply forward specific SMIs to (possibly relocated) legacy SMM code after taking the required memory protection measures.

Previous SMM Attacks: As mentioned in Section 3, there have been attempts to subvert the SMRAM using cache poisoning [6,33]. Although these attacks can be easily prevented by using proper hardware configuration, it shows that hardware vendors should undergo a rigorous review of the SMM security to avoid any further security problems.

It is worth mentioning that the implication of SMM attacks is beyond the systems that rely on the SMM for security. It is shown in [7] that a subverted SMM can be used to host a stealthy rootkit that undermines the security of any system. In general, trusting the underlying hardware, including the SMM, is imposed on all software systems.

Hardware Vendors Cooperation: As mentioned in Section 4, implementing SICE requires hardware vendors to allow adding its code to the SMI handler before locking the SMRAM. In fact, we advocate that SICE should be entirely implemented by hardware vendors. In this case, the system BIOS will be responsible for initializing SICE upon booting the system. This guarantees that SICE will always be compatible with the specification of the underlying hardware.

5. SICE PROTOTYPE: AN ISOLATED VM

In this section, we present a SICE prototype implemented on an IBM LS22 blade server, which is equipped with two 2.7GHz AMD Opteron 10h family quad-core processors. We use an Ubuntu 9.10 Linux as the OS. In our prototype, we replace the original SMI handler with SICE's SMI handler. Implementing the core functions of SICE's SMI handler,

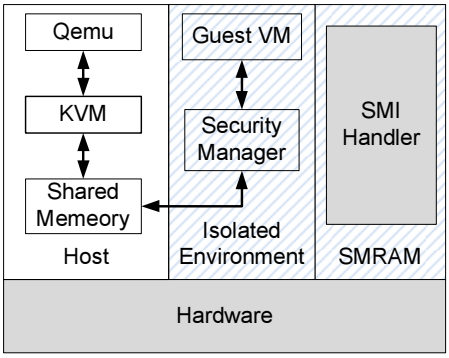

Figure 5: SICE Prototype

which are to prepare and enter the isolated environment, requires around 300 LOC (excluding cryptographic libraries)

As mentioned in Section 1.3, the isolated environment can support running any software, ranging from a single program to a complete VM. To run a single program inside SICE, it needs to be instrumented so that it directly runs on top of the security manager. The required instrumentation is similar to that required by previous research on running isolated security sensitive code (e.g., Flicker [20], Trustvisor [19]).

In our prototype, we use SICE to run a Linux VM, rather than running an instrumented single program. This is to demonstrate the flexibility provided to the isolated environment. Running a whole VM requires the ability to support a diverse set of applications, frequent context switching, and a large range of protected memory.

Figure 5 shows our SICE prototype. The security manager plays the role of the hypervisor in the isolated environment. It uses hardware assisted virtualization to manage and run the VM as the isolated workload. All hardware management functions are delegated to the legacy host, specifically, a modified version of Qemu/KVM [18] running in the legacy host. Qemu is a program that emulates hardware peripherals. KVM is a kernel module that manages other VM operations like scheduling and memory management.

We modified KVM so that it uses a shared memory to send the required VM configurations to the security manager. The same shared memory is used by the security manager to pass the information required to request hardware services from the legacy host. This architecture is mainly chosen to simplify the implementation efforts. In our prototype, the security manager is composed of around 2.1 KLOC, which is comparable to the size of current microhypervisors.

Our implementation provides both network interface and serial port emulation using Qemu. Networking is necessary to allow the VM to communicate with remote users, while the serial port is used as a console for debugging.

Our prototype does not support graphic display emulation, which is not necessary for cloud computing applications. Moreover, we do not support disk drive emulation because our experiments do not need a permanent storage.

Supporting a disk drive emulation is straightforward based on the technique used to implement the network interface emulation. However, the main question is how to secure the disk access from a compromised legacy host. The answer is dependent on the isolated workload rather than the isolation mechanism. For instance, security sensitive applications can either use full disk encryption (similar to Bitlocker [8]), or selectively encrypt secret files only (e.g., a file that contains user passwords). Other applications only need to keep secret data in memory without a permanent storage (e.g., web servers that process online purchases without storing customer credit card numbers).

In the following, we present more details about our prototype implementation.

5.1 Preparing and Initializing the VM

As discussed in Section 4.1.1, the legacy host provides SICE with the initial VM image, composed of the VM kernel and the initial ram disk. The SMI handler copies this image to the SMRAM. KVM also places the required VM configuration parameters (e.g., RAM size) in the shared memory then triggers an SMI to initialize the isolated workload. As mentioned in Section 4.3, the SMI handler measures the initial VM image to attest to its integrity to the remote owner of this VM. The integrity evidence should be extended to include the VM configuration parameters passed by KVM. Some of these parameters (e.g., the VM execution entry point, initial register values) are critical to its integrity.

The security manager prepares the VM page tables and its virtual machine control block (VMCB), based on the provided configuration parameters. The page tables are crafted using AMD's nested page tables (NPT) [1], which adds another layer to the virtual-to-physical memory translation.

5.2 Handling VM Exits

Certain events force the VM to exit its operations (e.g., external interrupts, page faults). These events need to be handled by the hypervisor.

We forward most of the VM exit events to KVM in the legacy host. Nevertheless, the security manager handles some specific VM exits that do not require much computation to avoid unnecessary context switching. Among these VM exit events are control register accesses and requests to execute privileged instructions like `CPUID` or `INVD`.

Other VM exists (e.g., writing to an IO port, accessing an IO memory) are directly forwarded to the legacy host. To preserve the VM confidentiality, the security manager should only send information that is necessary for handling the VM exits (e.g., VM exit reason and error code). This information is sent to the KVM in the legacy host through an established communication channel between the isolated environment and the legacy host.

5.3 Communication with the Legacy Host

Establishing a communication channel with the legacy host is not managed by the SMI handler. Instead, it is directly managed by the security manager and the legacy host. As discussed in Section 4, a shared memory that is outside the SMRAM protection is used to establish the communication between the legacy host and the isolated workload.

Signaling is required between the two environments to send notifications that data is placed into the shared memory. In the time-sharing mode, we use the context switching as the method of signaling. In the multi-core mode, IPIs between the host and the isolated cores are used for signaling, as discussed in Section 4.2.1.

5.4 Using Hardware Peripherals

In our prototype, hardware peripherals in the isolated workload are emulated using Qemu in the legacy host. There are three main methods to control a hardware peripheral:

IO ports, IO memory, and Direct Memory Access (DMA). When the VM accesses an IO port or an IO memory, a VM exit occurs and the control is transferred to the host to emulate the hardware access.

DMA requests work differently because DMA is supposed to directly read or write physical memory. In legacy VMs, Qemu is granted full access to the VM physical memory to emulate DMA accesses. However, since Qemu is located in the legacy host, SICE prevents it from directly accessing the memory of the isolated environment. Thus, our prototype modifies Qemu to send a request to the security manager with the address, size, and type (read or write) of the emulated DMA. The security manager in turn copies the required memory between the VM memory and the shared memory. To preserve the confidentiality of the isolated workload, the security manager only allows Qemu to retrieve or modify VM memory areas assigned to DMA operations.

In our prototype, the security manager handles DMA access to simplify our implementation. Nevertheless, this operations should be directly handled by the isolated VM by allowing it to directly access a part of the shared memory, which will reduce the tasks required from the security manager, and consequently reduce its code size.

5.5 Attestation

As mentioned in Section 4.3, the SMI handler should be a part of a trusted boot process. However, the LS22 servers, used for our implementation prototype, are not equipped with a TPM. Due to the lack of a TPM, the attestation process is not included in our prototype. Nevertheless, it is worth mentioning that static attestation using the TPM, signature key generation and signing are all known techniques that have been implemented previously.

It is worth mentioning here that implementing these cryptographic operations may increase the size of the code base of SICE. For instance, a typical SHA1 library is around 120 LOC. Other cryptographic functions (e.g., generating an RSA signature) can be done using the TPM.

6. EXPERIMENTAL EVALUATION

We perform a set of experiments to evaluate the performance of SICE. There are two anticipated sources of performance overhead associated with SICE. The first is the direct overhead resulting from entering/exiting the isolated environment. The second is the indirect overhead that results from the cache and TLB flushing required for SICE operations. On the other hand, running the isolated environment outside the SMM avoids any execution slow-down and is not anticipated to cause any performance overhead, compared to running the same workload without SICE.

In the rest of this section, we present a measurement of the anticipated overhead. First, we measure the execution time needed for a full context switching to and from the isolated environment. Second, we use the SICE prototype (See Section 5) to compare the performance of the isolated guest VM with the same VM running without SICE isolation.

6.1 SICE Execution Time

We measure the execution time needed to perform each of the four major steps of entering and exiting the isolated environment. The measured steps include: (1) triggering the SMI, (2) preparing the isolated environment by the SMI handler, (3) entering the isolated environment, and (4) returning to the legacy host. These measures are obtained from the average of 100 runs.

To precisely measure the end-to-end execution time of each step, we use the `RDTSC` instruction to read the processor's Time Stamp Counter (TSC). We then convert cycles to microseconds based on the TSC speed (2.7GHz in our experimental platform).

Table 1: SICE execution time

Operation	Time (in μs)	Std. Dev.
Triggering an SMI	6.8	0.074
Preparing the isolated env.	20.7	0.155
Entering the isolated env.	9.3	0.233
Exiting the isolated env.	30.1	0.644
Total ($\approx$)	67	

Table 1 shows the experimental results for the time-sharing mode. Triggering an SMI and switching the processor context from the legacy host to the SMI handler needs an average of 6.8 μs. The SMI is invoked by the local APIC of the processor core by sending an IPI to all cores on the system.

The next step, which is to prepare the isolated environment, needs an average of 20.7 μs. The latency of this step is relatively high given that this step only requires changing a few entries in the page tables, modifying the interrupt vector table descriptor, and verifying the values of several registers and MSRs. We anticipate that this relatively high latency is caused by a slower processing speed in the SMM. A similar observation was made in [2].

The next step is entering the isolated environment. In this step, we execute the "return from SMM" instruction (`RSM`) to jump to the security manager. The time needed for this step is 9.3 μs, which is relatively high. We anticipate this latency is due to the change of the processor execution environment, particularly the page tables, which leads to invalidating all cache and TLB entries.

Finally, the time needed to return from the isolated environment to the legacy host is 30.1 μs, which is similar to executing all the previous steps in the reverse order (i.e., from the isolated environment to the legacy host). In total, the end-to-end time required to enter and exit the isolated environment is 67 μs.

In the time-sharing mode, the end-to-end execution time overhead obtained in this experiment occurs on every context switch between the isolated environment and the legacy host. A context switch is required every time the isolated environment needs a service from the legacy host. In contrast, it is a one-time overhead in the multi-core mode.

6.2 Isolated Environment Performance

Our next experiment is to evaluate the overall performance overhead on the isolated workload. We run this experiment using the same system configuration of our prototype system, discussed in Section 5. In our experiments, we assign a 256 MB to the VM. Our VM runs Linux kernel v2.6.28. and a ram disk that is equipped with BusyBox v1.10.2 [4]. (BusyBox is a program that combines common utilities into a single executable.)

To evaluate the performance of the guest VM, we assemble a set of tests and run each test on the same guest VM in three different environments: (1) unmodified Qemu/KVM using

one processor core, (2) SICE time-sharing environment using a single processor core, and (3) SICE multi-core environment using two processor cores, used as the host core and the isolated core. In the multi-core test, we do not run any program other than Qemu/KVM on the host core, so that we can get a precise measurement of the performance overhead.

To measure the performance overhead, we use Qemu to emulate a serial port for the guest VM. We then configure the guest VM to use this serial console to accept shell command. Our performance measurement is the execution time needed for the guest VM to complete each of these tests. The execution time is calculated by using the `RDTSC` instruction in the legacy host. The time measurement is taken just before sending the last letter of the shell command (the carriage return) and right after the first response is received (echoing the carriage return).

To calculate the performance overhead of each test, we first measure T_K, the time needed to run the test using unmodified Qemu/KVM. Afterward, we measure T_S, the time needed to run the same experiment using SICE. The performance overhead is then calculated using the equation $(T_S - T_K)/T_K \times 100\%$.

We selected our test cases in three categories: The first is user level programs to test the user level operations of guest VMs running with SICE. The second is a group of tests that test the throughput of the guest VM kernel. The third tests the performance of the emulated network interface.

In the first category, we run three tests. The first measures the latency of copying a 2.1 MB file. Our prototype system does not support hard drive emulation. Thus the file is copied inside the virtual file system that represents the initial ram disk. Our main objective of this test is to estimate the impact of SICE on the guest VM memory operations, because it is basically a copy between two locations in memory rather than an actual disk access. The next two tests in the same category use the programs `gzip` and `gunzip` to compress and decompress the same file. These tests aim to test the impact of SICE on the guest VM's user level computations. The final results of these tests are obtained as an average of 100 runs.

In the second category, we measure the guest VM kernel responses to three main operations: `fork`, `getpid` and `insmod`. `fork` and `getpid` are tested using a custom program that directly calls these system calls 10,000 times and measures the average response time. For the `insmod` test case, we use the `insmod` program to insert a 16.4 KB loadable kernel module in the guest kernel. This test is repeated 10 times, and all test runs show very little variance.

In the last category, we run a single test that uses `wget` to retrieve a 156.6 KB file from an Apache server running in the legacy host. The test is repeated 10 times.

Figure 6 shows the results of our experiments. In the multi-core mode, all tests, except for `wget`, showed a slight performance overhead under 3%. We expect that the slight performance overhead is due to the time required to copy information to and from the shared memory between the legacy host and the security manager.

The `wget` test shows a 17% performance overhead. However, a significant portion of this overhead is due to the *non-optimized* implementation of the Qemu network emulator for SICE. When a packet is ready to be sent through the network, the VM sends an IO command to the network adapter to inform it that a packet is ready. In normal operations,

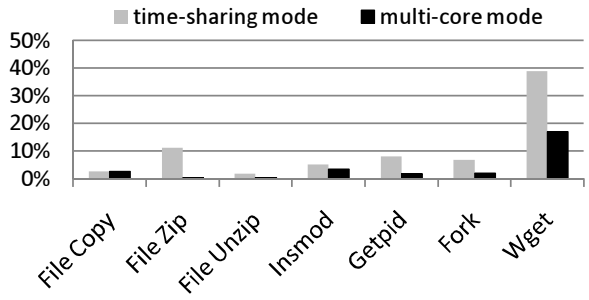

Figure 6: Performance overhead of an isolated VM compared with original Qemu/KVM

Qemu first receives this IO command, and then directly accesses the VM physical memory to emulate the DMA read of the packet. In our prototype, Qemu does not have direct access to the physical memory of the guest VM. Thus, we modified Qemu to send DMA requests to the security manager, which copies the packet from the protected guest VM memory to the shared memory so that Qemu can process it correctly. Similarly, an extra step is necessary when the VM receives a packet.

Hence, our prototype requires an extra communication step between Qemu in the legacy host and the security manager for every network packet. Intuitively, this extra communication step requires an extra context switch from the legacy host to the isolated environment in the time-sharing mode or an extra inter-process communication between the host core and the isolated core in the multi-core mode.

This extra communication step is the cause of the high overhead in the `wget` test. To support this claim, we measured the number of exits to and from the isolated environment during this test. In normal operations, there was an average of 1,249 guest VM exits that required hardware emulation by KVM and/or Qemu. In SICE's time-sharing mode, there was an average of 1,877 context switches between the isolated environment and the legacy host. There was a similar number of messages passed between the host core and the isolated core in the multi-core mode. This 50% increase in the communication time between the two environments explains the high overhead in this test.

To avoid this overhead, the VM device driver can be modified so that it uses the shared memory for the network adapter's DMA operations. Thus, the Qemu emulated driver would be able to directly read the passed packets without sending an extra request to the security manager.

The time-sharing mode showed a higher performance overhead, which is expectable due to the 67 μs needed to switch between the host and the isolated guest VM. The tests that show higher overhead, such as `gzip` and `wget`, are those that require a higher number of switching between the host and the isolated environment.

It is also noticeable that the time-sharing mode showed a higher overhead (around 40%) in the `wget` test. Receiving the target file in the normal VM operations needs 0.532 seconds compared to 0.739 seconds in the time-sharing mode. As mentioned above, this test requires 1,877 context switches from the isolated environment to the legacy host. Given that each context switch requires an average of 67 μs, the context switch overhead in this test was 0.126 seconds. This test clearly shows that the time-sharing mode is not suitable for operations that are IO intensive. It will be rather

useful to run programs that require an enhanced isolation and a smaller IO footprint (e.g., a program that processes secret key operations).

7. RELATED WORK

Several researchers (e.g, [3, 5, 9, 22, 34]) attempted to use hypervisors to enable strong isolation between workloads running in a cloud computing environment. Nevertheless, recent attacks [16,32] and vulnerability reports [23,24] show that hypervisors are subject to security exploitation.

There have been several attempts (e.g., [2,30,31]) to verify the runtime integrity of hypervisors. These techniques still require cloud computing users to trust the host environment, which has a relatively large TCB and continuously interacts with mutually distrusted workloads.

As discussed in Section 1, the recent attempts that aim at eliminating the hypervisor from the TCB can be divided into microhypervisor-based and hardware-based approaches.

Hardware-based approaches: The introduction discussed how systems that rely on the late launch capability (e.g., Flicker [20], BIND [25]) fall short from our objectives. Moreover, Intel proposed a hardware service called Processor Measured Application Protection Service (P-MAPS) [21] to offer runtime memory isolation. However, there is insufficient detail about its ability to provide multi-core isolation. IBM also introduced interesting security features in its Cell Broadband Engine (Cell BE) architecture [26], which provides multi-core isolation. However, these features are unique to this specific architecture. SICE achieves the same level of isolation on the x86 architecture, which is used in the vast majority of general purpose computing platforms.

Microhypervisor-based approaches: We discussed two notable microhypervisors, NOVA [27] and Trustvisor [19], as well as seL4 [15] in the introduction. In particular, seL4 is a promising approach that uses formal verification to introduce a vulnerability-free microkernel, though it cannot be extended to commodity hypervisors due to its restrictions (e.g., it should run with interrupts mostly disabled). It would be interesting to study how we can apply formal verification to guarantee the integrity of SICE's security manager.

8. CONCLUSION

In this paper, we presented SICE, a research prototype that aims to explore the capability of current hardware platforms in providing more secure isolated environments. We demonstrated that current hardware architecture already provides abstractions such as SMM that can support strong isolation. We showed that building strong isolation mechanisms on top of the SMM abstraction requires a very small TCB of about 300 LOC, which tremendously reduces the TCB size compared with previous techniques. Nevertheless, practical deployment of SICE would require CPU and hardware platform vendors to undergo detailed security reviews of the BIOS software and the SMM implementation.

SICE provides a set of unique capabilities to the isolated environment, including (1) fast context switch to and from the legacy host, (2) protected concurrent execution with the legacy host, and (3) attestation to its integrity. Moreover, its ability to run isolated workloads concurrently with a legacy host provides a cost-effective solution to security sensitive workloads without using dedicated hardware platforms.

Our future research will focus on extending SICE to work on Intel platforms. In addition, we will explore how to apply formal verification methods on SICE's security manager.

9. REFERENCES

[1] Advanced Micro Devices. Amd64 architecture programmer's manual: Volume 2: System programming, September 2007.

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

[3] S. Berger, R. Cáceres, D. Pendarakis, R. Sailer, E. Valdez, R. Perez, W. Schildhauer, and D. Srinivasan. TVDc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev.*, 42(1):40–47, 2008.

[4] BusyBox. `http://www.busybox.net/`.

[5] X. Chen, T. Garfinkel, E. C. Lewis, P Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D.R.K Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS'13)*, pages 2–13, 2008.

[6] L. Duflot. Getting into the SMRAM: SMM reloaded. In *Proceedings of the 10th CanSecWest conference*, 2009.

[7] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: a new breed of OS independent malware. In *Proceedings of the 4th international conference on Security and privacy in communication networks*, pages 1–12, August 2008.

[8] Niels Ferguson. Aes-cbc + elephant diffuser: A disk encryption algorithm for windows vista. Microsoft Corporation Technical Report, 2006.

[9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP '03)*, pages 193–206, 2003.

[10] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification. revision 3.0b, October 2006.

[11] Intel Corporation. Trusted eXecution Technology preliminary architecture specification and enabling considerations. Document number 31516803, 2006.

[12] Intel Corporation. Intel virtualization technology for directed I/O. Document number 31516803, 2007.

[13] Intel Corporation. Software developer's manual vol. 3: System programming guide, June 2009.

[14] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10)*, pages 350–361, 2010.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an OS kernel.

In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, pages 207–220, 2009.

[16] K. Kortchinsky. Hacking 3D (and breaking out of VMWare). In *Black Hat conference*, 2009.

[17] Dartmouth PKI Lab. TPM reset attack. `http://www.cs.dartmouth.edu/~pkilab/sparks/`. Accessed in August 2011.

[18] Kernel Based Virtual Machine. `http://www.linux-kvm.org/`.

[19] J. McCune, Y. Li, N. Qu, A. Datta, V. Gligor, and A. Perrig. Efficient TCB reduction and attestation. In *the 31st IEEE Symposium on Security and Privacy*, May 2010.

[20] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, March/April 2008.

[21] R. Sahita, U. Warrier, and P. Dewan. Dynamic software application protection. Intel Corporation, April 2009.

[22] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based security architecture for the xen opensource hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pages 276–285, 2005.

[23] Secunia. Vulnerability report: Vmware esx server 3.x. `http://secunia.com/advisories/product/10757/`. Accessed in August 2011.

[24] Secunia. Xen multiple vulnerability report. `http://secunia.com/advisories/44502/`. Accessed in August 2011.

[25] E. Shi, A. Perrig, and L. van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[26] K. Shimizu, H. P. Hofstee, and J. S. Liberty. Cell broadband engine processor vault security architecture. *IBM J. Res. Dev.*, pages 521–528, September 2007.

[27] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems (EuroSys'10)*, pages 209–222. ACM, 2010.

[28] Trusted Computing Group. `https://www.trustedcomputinggroup.org/`.

[29] Trusted Computing Group. TPM specifications version 1.2. `https://www.trustedcomputinggroup.org/downloads/specifications/tpm/tpm`, July 2005.

[30] J. Wang, A. Stavrou, and A. K. Ghosh. HyperCheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID'10)*, September 2010.

[31] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, May 2010.

[32] R. Wojtczuk and J. Rutkowska. Xen 0wning trilogy. In *Black Hat conference*, 2008.

[33] R. Wojtczuk and J. Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. Invisible Things Lab, 2009.

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

APPENDIX

A. PORTABILITY TO INTEL PLATFORMS

In this section, we discuss the portability of SICE to Intel platforms. Although SMM is supported by both AMD and Intel, SICE uses some hardware features that are only provided by AMD. In the following, we discuss these features and propose alternative techniques for Intel platforms. Implementing SICE on Intel platforms using these techniques needs further research, particularly to achieve both the security and performance required from this system.

SMRAM Size and Location Modification: After the SMRAM is locked, Intel platforms do not allow runtime change of its size and location. This restriction has a direct impact on our techniques, which rely on changing the SMRAM size at run time to enable fast context switching to the isolated environment. However, some other Intel features can be used for SICE. For instance, new Intel hardware is equipped with special instructions to do a high throughput cryptographic operations using the Advanced Encryption Standard (AES). In the time-sharing mode, this feature can be used to encrypt and authenticate the memory of the isolated workload while the legacy host is running, and decrypt and verify the same memory in the isolated environment. Thus, the SMRAM can only be used as a permanent storage of encryption keys. Further research is needed to evaluate the feasibility and performance of this approach.

Memory Double-view: Intel platforms define the `TSeg` SMRAM in its memory controller hub, which is shared among multiple cores. Unless the implementation is modified to move the SMRAM definition to a core specific location, our memory double-view technique will not be able to isolate individual Intel processor cores.

Nevertheless, current Intel implementation can apply this technique on processor nodes as they do not share the same memory control hub, allowing a more coarse grained isolation of the resources on physical platforms.

Multi-core Protection: Intel does not provide a capability that can disable all system interrupts. As mentioned in Section 4.2, we rely on this capability to prevent the host core from using the startup IPI to modify the execution environment of the isolated core. However, the startup IPI can only set the instruction pointer to an address in the lowest 1MB of the physical memory. This address range can be easily modified to point to a non-writable memory chip.