SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes [*]

Kun Sun, Jiang Wang[†], Fengwei Zhang, and Angelos Stavrou
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
{ksun3, jwanga, fzhang4, astavrou}@gmu.edu

Abstract

Protecting commodity systems running commercial Operating Systems (OSes) without significantly degrading performance or usability still remains an open problem. To make matters worse, the overall security depends on complex applications that perform multiple inter-dependent tasks with Internet-borne code. Recent research has shown the need for context-dependent trustworthy environments that can segregate different user activities to lower risk of cross-contamination and safeguard private information.

In this paper, we introduce a novel BIOS-assisted mechanism for secure instantiation and management of trusted execution environments. A key design characteristic of our system is usability: the ability to quickly and securely switch between operating environments without requiring any specialized hardware or code modifications. Our aim is to eliminate any mutable, non-BIOS code sharing while securely reusing existing hardware: even when an untrusted environment is compromised, there is no potential for exfiltration or inference attacks. To safeguard against spoofing attacks, we can force the user to physically set a hardware switch, an action that cannot be reproduced by software. In addition, we provide visible indication to the user about the current running environment leveraging one of the front panel Light Emitting Diodes (LEDs). In our prototype, the entire switching process takes approximately six seconds on average. This empowers users to frequently and seamlessly alternate between trusted and untrusted environments.

[*]This work was supported in part by the U.S. Army Research Office (ARO) through contract WF911NF-11-C-0048, the U.S. Air Force Research Laboratory (AFRL) through contract FA8650-10-C-7024, NSF grant CNS-TC 0915291, and a research gift from Google Inc. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Government and sponsoring agencies.

[†]Kun Sun and Jiang Wang contributed equally to this work.

1. Introduction

Nowadays, desktop computers are being employed for multiple tasks ranging from personal communication and entertainment to business and government operations: web browsing, online gaming, and social web portals are examples in the former category; online banking, shopping, and business emails belong in the latter. Unfortunately, modern software has a large and complex code base that typically contains a number of vulnerabilities [6]. To make matters worse, modern desktop applications usually operate on foreign content that is received over the network. Current operating system (OS) environments offer user- and process-level isolation for different activities; however, these levels of isolation can be easily bypassed by malware through privilege escalation or by other attacking techniques. Researchers have pointed out the need for trustworthy environments where, based on context and requirements, the user can segregate different activities in an effort to lower risk by reducing the attack space and data exposure.

There are ongoing research and commercial efforts to employ virtual machine monitors (VMMs, also referred to as hypervisors) to isolate different activities and applications [43, 20, 21, 29, 46, 28, 45, 22]. As long as the virtual machine monitor is not compromised and there is no exposed path or covert channel between the different environments, applications in different VMs remain isolated. However, their widespread adoption has attracted the attention of attackers towards VMM vulnerabilities [48, 38, 31]. According to IBM X-Force 2010 mid-year trend and risk report [4], from 1999 through the end of 2009, 373 vulnerabilities affecting virtualization solutions were disclosed, and 35.0% virtualization vulnerabilities on the server products allowed an attacker to escape from a guest VM to affect other VMs, or the VMM itself. Researchers have noticed this problem and have begun to improve hypervisor security [45, 13, 44].

An alternative or sometimes complementary approach to software isolation is hardware isolation: in many military and civilian installations users have to use multiple physically-isolated computers, merely switching controls and displays. Although attractive in terms of isolation, hardware increases the operational and maintenance cost because it requires more space, cooling, and energy. It is inflexible and cannot support the current need for a range of trusted environments. Moreover, it is inconvenient for users to switch between two computers to finish their tasks. Multi-boot supports the installation of multiple OSes on the same machine and uses a boot loader to choose between the OSes. Unfortunately, it is time consuming to shutdown one OS and boot up another. For instance, Lockdown [41] combines a hypervisor with ACPI S4 Sleep (also known as hibernation or Suspend to Disk) to provide a secure environment for sensitive applications. However, the switching latency in many cases is more than 40 seconds, rendering the system difficult to use in practice.

In this paper, we attempt to tackle the secure OS isolation problem without using a hypervisor or any mutable shared code. We design a firmware-assisted system called *SecureSwitch*, which allows users to switch between a trusted and an untrusted operating system on the *same* physical machine with a short switching time. The basic input/output system (BIOS) is the only trusted computing base that ensures the resource isolation between the two OSes and enforces a trusted path for switching between the two OSes. The attack surface in our system is significantly smaller than hypervisor- or software-based systems; we can protect the integrity of the BIOS code by using a hardware lock [5] to set the BIOS code as read-only, or by using TPM to verify the integrity of the BIOS code. Furthermore, our system guarantees a strong resource isolation between the trusted and untrusted OSes. If the untrusted OS has been compromised, it still cannot read, write, or execute any of the data and applications in the trusted OS.

Overall, our system can ensure isolation on the following computer components:

- **Memory Isolation:** All OS environments run in separate Dual In-line Memory Modules (DIMM). A physical-level memory isolation is ensured by the BIOS because only the BIOS can initialize and enable the DIMMs. No software can initialize or enable DIMMs after the system boots up.

- **CPU Isolation:** The different operating systems never run concurrently. When one OS is switched off, all CPU states are saved and flushed. We use ACPI S3 sleep mode to help achieve CPU suspend/restore.

- **Hard Disk Isolation:** Each OS can have its own dedicated encrypted hard disk. We use RAM disk to save the temporary sensitive data in the trusted OS. The untrusted OS cannot access the RAM disk in the trusted OS due to the memory isolation.

- **Other I/O Isolation:** When one OS is switched off, all contents maintained by the device drivers (e.g, graphic card, network card) are saved and the devices are then powered off. This guarantees that the untrusted OS cannot steal any sensitive data from the I/O devices.

A *trusted path* ensures users that they really are working with the operation system they intend to use. We must ensure a trusted path to prevent *Spoofing Trusted OS* attacks that deceive users into a fake trusted OS environment when the users switch from the untrusted OS to the trusted OS. For instance, a sophisticated adversary may fake an S3 sleep in the untrusted OS by manipulating the hardware (e.g., shutting down the monitor) and then deceiving the user with a fake trusted OS environment, which is controlled by the untrusted OS. Because the BIOS is the only component that we can trust to enforce the trusted path, we use the power button and power LED to ensure and indicate the user that the system enters the BIOS after one OS has been truly suspended. Then, the BIOS will wake up one OS according to a system variable that indicates which OS should be woken. The system variable can only be manually changed by the user; it cannot be changed by any software.

We harness the Advanced Configuration and Power Interface (ACPI) [24] S3 sleep mode to help achieve a short OS switching latency. Because two OSes are maintained in RAM memory at the same time, the switching latency is only about six seconds, which is much faster than switching between two OSes on a multi-boot computer or switching using ACPI S4 mode [41]. It is slower than the hypervisor-based solutions; however, we don't need to worry about the potential vulnerabilities in the hypervisor. Moreover, our system can be used as a complementary approach to existing hypervisor- and OS-protection solutions.

In summary, we make the following contributions:

- *Secure OS switching without using any mutable software layer.* Our system depends on the BIOS and existing hardware properties to enforce a discernible trusted path when switching between the two OSes. This trusted path can prevent a wide-range of attacks including the Spoofing Trusted OS attacks. Our approach requires no modification of the commodity OS.

- *No data leak between different environments.* The resource isolation enforced by the BIOS prevents any data leak from the trusted OS to the untrusted OS.

- *Fast Switching Time.* We implemented a prototype of the secure switching system using commodity hardware and both commercial and open source OSes (Microsoft Windows and Linux). Our system can switch between OSes in approximately six seconds.

2. Background

2.1. ACPI Sleeping States

The Advanced Configuration and Power Interface (ACPI) establishes industry-standard interfaces that enable OS-directed configuration, power management, and thermal management of computer platforms [24]. ACPI defines four global states: *G0, G1, G2, and G3*. G0 is the working state wherein a machine is fully running. G1 is the sleeping state that achieves different levels of power saving. G2 is called “Soft Off,” wherein the computer consumes only a minimal amount of power. In G3, the computer is completely shutdown; aside for the real-time clock, the power consumption is zero.

G1 is subdivided into four sleeping states: *S1, S2, S3, and S4*. From S1 to S4, the power saving increases, but the wakeup time also increases. In S3, all system context (i.e., CPU, chipset, cache) aside from the RAM is lost. S3 is also referred to as *Standby* or *Suspend to RAM*. In S4, all main memory content is saved to non-volatile memory, such as a hard drive, and the machine (including RAM) is powered off. S4 is also referred to as *Hibernation* or *Suspend to Disk*. In both S3 and S4, all of the devices may be powered off.

Not every machine or operating system supports all of the ACPI states. For instance, neither S1 or S2 is used by Windows. S3 and S4, however, are supported by all Linux 2.4 and 2.6 series kernels and recent Windows distributions (XP, Vista, 7). Our SecureSwitch uses S3 operations provided by the operating system to help save the system context and later restore it. This dramatically saves our developing efforts.

2.2. BIOS, UEFI and Coreboot

The BIOS is an indispensable component for all computers. The main function of the BIOS is to initialize the hardware, including processor, main memory, chipset, hard disk, and other necessary IO devices. BIOS code is normally stored on a non-volatile ROM chip on the motherboard. In recent years, a new generation of BIOS, referred to as Unified Extensible Firmware Interface (UEFI) [9], has become increasingly popular in the market. UEFI is a specification that defines the new software interface between OS and firmware. One purpose of UEFI is to ease the development by switching to the protected mode in a very early stage and writing most of the code in C language. A portion of the Intel UEFI framework (named Tiano Core) is open source; however, the main function of the UEFI (to initialize the hardware) is still closed source.

Coreboot [2] (formerly known as LinuxBIOS) is an open-source project aimed at replacing the proprietary BIOS (firmware) in most of today’s computers. It performs a small amount of hardware initialization and then executes a so-called payload. Similar to the UEFI-based BIOS, Coreboot also switches to protected mode in a very early stage and is written mostly in C language. Our prototype implementation is based on Coreboot V4. We chose to use Coreboot rather than UEFI since Coreboot has done all of the work of hardware initialization, whereas we would need to implement UEFI firmware from scratch, including obtaining all of the data sheets for our motherboard.

2.3. SMM

System Management Mode (SMM) is a separate CPU mode from the protected mode and real address mode. It provides a transparent mechanism for implementing platform-specific system-control functions, such as power management and system security. SMM is primarily targeted for use by the basic input-output system (BIOS) and specialized low-level device drivers.

SMM is entered via the system management interrupt(SMI), when the SMM interrupt pin (SMI#) is asserted by motherboard hardware, chipset, or system software. At the next instruction boundary, the microprocessor saves its entire state in a separate address space known as system management RAM (SMRAM) and enters SMM to execute a special SMM handler. The SMRAM can be made inaccessible from other CPU operating modes; therefore, it can act as trusted storage, sealed from access by any device or even the CPU (while not in SMM mode). The program executes the RSM instruction to exit SMM. Our system includes an SMM handler to monitor the hard disk isolation between two OSes.

2.4. DQS Settings and DIMM MASK

There are many different types of RAM, and one of the most popular ones is the Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM). One feature of these DDR memories is that they include a special electrical signal referred to as “data strobes” (DQS). For proper memory reads to occur, the DQS must be properly timed to align with the data valid window of the data (DQ) lines. The data valid window refers to the specific period of time when the DRAM chip drives (i.e., makes active) the DQ lines for the memory controller to read its data. If the DQS signal is not properly aligned, the memory access will *fail*. For DDR1, the parameters of DQS can be automatically set by the hardware. For DDR2 and DDR3, the DQS settings should be programmed by the BIOS [11]. We use DDR2 memory in our system.

A motherboard usually has more than one DIMM slot. Our system assigns one DIMM to one OS. When one OS is running, the BIOS will only enable the DIMM assigned to that OS with the corresponding DQS settings. BIOS uses a variable named "DIMM_MASK" to control which DIMMs should be enabled.

3. Threat Model and Assumptions

Our system operates under the assumption that an adversary can potentially subvert the untrusted OS using known or zero-day attacks on software applications, device drivers, user-installed code, or operating system. In addition, we assume that the attacker does not have physical access to the protected machine and thus cannot mount local physical attacks, such as removing a hard disk or a memory DIMM.

However, an adversary may launch any class of attacks against the trusted OS after compromising the untrusted OS. For instance, a data exfiltration attack aims at stealing sensitive data from the trusted OS. Furthermore, the adversary may attempt to modify the code of trusted OS and compromise its integrity. In a *Spoofing Trusted OS attack*, a sophisticated attacker can create a fake trusted OS environment, which is fully controlled by the attacker, and deceive the user into performing sensitive transactions there. An attacker can perform a denial-of-service (DoS) attack against the trusted OS by forcing the untrusted OS to terminate or crash; however, since such attacks can be easily detected and resolved; we do not prevent DoS attacks against our system. Another class of attacks is the direct or indirect "inference" attack: an attacker is able to identify instances of secure OSes running at the same physical machine as insecure instances with the purpose of inferring run-time behavior. Due to its design to not run in parallel instances of OSes, our system is impervious to this class of attacks.

There is an implicit assumption that the trusted OS can be trusted when the BIOS boots it up, but this does not mean that the trusted OS is bug-free. In other words, the trusted OS may be compromised from network attacks using vulnerabilities within the OS or the applications. There are several mechanisms to alleviate these network attacks [19, 16, 15]; however, they lie beyond the scope of this paper.

Furthermore, we assume that BIOS and device firmware code, including the option ROMs on devices (*e.g., video cards*) and peripherals (e.g., mouse and keyboard), are not susceptible to a direct or indirect compromise that can subvert the control flow during or after the sleep/wake-up cycle. In addition, the operating system must support ACPI S3 sleeping mode. This mode has been widely supported by modern OSes, such as Linux and Windows. Our system does not require any hardware virtualization support including Intel VT-x and AMD-V.

4. SecureSwitch Framework

The overall architecture of the SecureSwitch system is depicted in Figure 1: two OSes are loaded into the RAM at the same time. Commercial OSes that support ACPI S3 can be installed and executed without any changes. Instead of relying on a hypervisor, we modify the BIOS to control the loading, switching, and isolation between the two OSes.

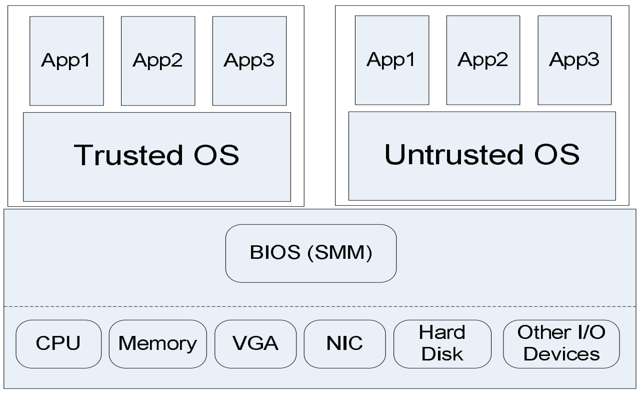

Figure 1. Architecture of SecureSwitch System

The Secure Switching operation consists of two stages: *OS loading stage* and *OS switching stage*. In the OS loading stage, the BIOS loads two OSes into separated physical memory space. The trusted OS should be loaded first and the untrusted OS second. In the OS switching stage, the system can suspend one OS and then wake up another. In particular, it must guarantee a trusted path against the spoofing trusted OS attack when the system switches from the untrusted OS to the trusted OS.

The system must guarantee a thorough isolation between the two OSes. Usually one OS is not aware of the other co-existing OS in the memory. Even if the untrusted OS has been compromised and can detect the coexisting trusted OS on the same computer, it still cannot access any data or execute any code on the trusted OS.

4.1. Secure Switching

Figure 2 shows the state machine for loading and switching between two operating systems in the system. Two variables are maintained to denote the system states. In each parenthesis, the first number records which OS is running in the system (1 for the trusted OS and 0 for the untrusted OS); the second number records if the untrusted OS has been loaded into the system.

When the machine is powered off, the system state is (0,0). After the system is powered on, the BIOS always

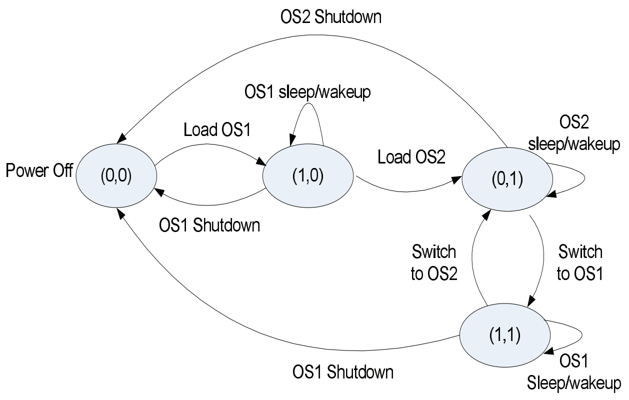

Figure 2. State Machine for OS Switching.

boots up the trusted OS (OS1) first. The BIOS constrains the trusted OS to use only part of the physical RAM. The trusted OS can be shut down or can perform sleep/wakeup. (1,0) means the trusted OS has been loaded and is running now, but the untrusted OS has not been loaded. When loading the untrusted OS(OS2), the BIOS first suspends the trusted OS, and then boots up the untrusted OS into the non-allocated physical RAM, which has no overlap with the memory used by the trusted OS. (0,1) means both OSes have been loaded into the memory while the untrusted OS is running and the trusted OS is suspended. If a user wants to switch from the untrusted to the trusted OS, the untrusted OS will be suspended first and then the system will wake up the trusted OS. At this time, the system state is (1,1). Similarly, the user can switch back from the trusted OS to the untrusted OS. To save power energy, the system still supports the normal OS sleep/wakeup.

4.1.1 Stateful vs. Stateless Trusted OS

When the system switches into the trusted OS, there are two options for restoring the OS context: *Stateless mode* and *Stateful mode*. In the stateless mode, each time when the system switches into the trusted OS, it starts from a pristine state. A copy of the trusted OS in its pristine state is maintained either on the hard disk or in a reserved memory area. In the stateful mode, when the trusted OS is switched in, it resumes from the system context wherein it was last switched out. All states of the trusted OS should be saved in the memory or on the hard disk.

The stateless mode does not save any system state when the trusted OS is switched out. It can mitigate the impacts of network attacks since the trusted OS will start from a pristine state that has not been compromised. The drawback is that the user loses the system context, so it cannot resume previous sessions or tasks within the trusted OS. Moreover, an adversary may easily fake a trusted OS environment if it knows the pristine state of the OS. In a stateful mode, since all of the system states are saved and can be restored, a user may resume sessions and tasks within the trusted OS. However, when the trusted OS has been continuously used for a long time, the risk of being compromised from network attacks increases.

4.1.2 Trusted Path

In our system, a *trusted path* assures users that they really are working with the operation system they intend to use. Our system must ensure a trusted path to prevent Spoof Trusted OS attacks, which deceive users into a fake trusted OS environment when the users switch from the untrusted OS to the trusted OS.

The secure switching consists of two sequential steps: *OS Suspend* and *OS Wakeup*. In x86 architecture, the suspend step is performed entirely by the operating system without involving the BIOS; the wakeup step is initiated by the BIOS and then handed over to the OS. Since the BIOS is the only component that we can trust to enforce the trusted path, we must guarantee the OS has been truly suspended so that the BIOS will be triggered. Otherwise, an attacker may launch a spoofing trusted OS attack by faking a suspend of the untrusted OS (e.g., power off the monitor) that totally circumvents the BIOS and then deceiving the user into a fake trusted OS. The untrusted OS can create such a fake trusted OS environment by installing a virtual machine similar to the trusted OS [26].

It is critical to protect the integrity of the system variable that is used by the BIOS to decide which OS should be woken up. Otherwise, when a user wants to switch from the untrusted OS to the trusted OS, an attacker may launch another spoof trusted OS attack by manipulating the system variable to make the BIOS wake up the untrusted OS and then deceiving the user into a fake trusted OS environment. In our system, the system variable is controlled by a physical jumper, which can only be manually set by the local user. The design details are described in Section 5.2.1.

4.2. Secure Isolation

The system must guarantee a strong isolation between the two OSes to protect the confidentiality and integrity of the information on the trusted OS. According to the von Neumann architecture, we must enforce the resource isolation on major computer components, including CPU, memory, and I/O devices.

CPU Isolation: When one OS is running directly on a physical machine, it has full control of the CPU. Therefore, the CPU contexts of the trusted OS should be completely isolated from that of the untrusted OS. In particular, no data information should be left in CPU caches or registers after one OS has been switched out.

CPU isolation can be enforced in three steps: saving the current CPU context, clearing the CPU context, and loading the new CPU context. For example, when one OS is switching off, the cache is flushed back to the main memory. When one OS is switching in, the cache is empty. The content of CPU registers should also be saved separately for each OS and isolated from the other OS.

Memory Isolation: It is critical to completely separate the RAM between the two OSes so that the untrusted OS cannot access the memory allocated to the trusted OS. A hypervisor can control and restrict the RAM access requests from the OSes. Without a hypervisor, our system includes a hardware solution to achieve memory isolation. The BIOS allocates non-overlapping physical memory spaces for two OSes and enforces constrained memory access for each OS with a specific hardware configuration (DQS and DIMM Mask) that can only be set by the BIOS. The OS cannot change the hardware settings to enable access to the other OS's physical memory. Details regarding this are included in Section 5.3.

I/O Device Isolation: Typical I/O devices include hard disk, keyboard, mouse, network card (NIC), graphics card (VGA), etc. The running OS has full control of these I/O devices. For devices with their own *volatile memory* (e.g., NIC, VGA), we must guarantee that the untrusted OS cannot obtain any information remaining within the volatile memory (e.g., pixel data in the VGA buffer) after the trusted OS has been suspended. When a stateful trusted OS is switched out, the device buffer should be saved in the RAM or hard disk and then flushed. when a stateless trusted OS is switched out, the device buffer is simply flushed.

For I/O devices with *non-volatile memory* (e.g., USB, hard disk), the system must guarantee that the untrusted OS cannot obtain any sensitive data information from the I/O devices used by the trusted OS. One possible solution is to encrypt/decrypt the hard disk when the trusted OS is suspended/woken. However, this method will increase the OS switching time due to costly encryption/decryption operations. Another solution is to use two hard disks for two OSes separately, and use BIOS(SMM) to ensure the isolation. When we are targeting at browser-based applications that don't need to keep local state, as opposed to a local copy of TurboTax, it is secure to save the temporary sensitive data in a RAM disk, which can maintain its content during OS sleep, but gets cleaned when the system reboots. Details can be found in Section 5.3.

5. Overall System Design

We combine the BIOS and the standard ACPI S3 mode to enforce resource isolation between the two OSes. BIOS is the control center and the only trusted computing base to enforce a trusted path during the OS switching process. Our system uses ACPI S3 to support both secure switching and the normal OS sleep/wakeup. The BIOS uses two system variables to control the OS loading and switching process. An *OS flag* indicates which OS (and corresponding resources) should be started; a *Boot flag* indicates whether the untrusted OS has been loaded into the memory.

5.1. Bootstrapping Two OSes

During the OS bootstrapping stage, the system loads both OSes in the RAM from the hard drive. To enforce RAM isolation and hard disk isolation, our system requires the motherboard to support at least two DIMMs and two hard disks, and it assigns one DIMM and one hard disk to each OS. When the computer boots up from a power-off state, the BIOS first loads the trusted OS using only one DIMM. Because BIOS is responsible for detecting and initializing the memory controller, it can enable and report only half of the RAM to each OS. Similarly, the BIOS only enables and reports one hard disk for each OS.

After the system is powered on, the BIOS always boots up the trusted OS first. To load the untrusted OS, the trusted OS should be suspended in S3 sleep. Then, the BIOS tries to wake up the untrusted OS when the OS flag is set to 0. However, the untrusted OS has not been loaded into the RAM at this time. To solve this problem, we use a Boot flag to indicate whether the untrusted OS has been loaded. When the system is powered on, the Boot flag is reset to 0 to reflect that the untrusted OS has not been loaded. When the BIOS detects that it is trying to wake up an untrusted OS that has not been loaded, it will load the untrusted OS and then set the Boot flag to 1.

One major drawback of this method is that the granularity for memory allocation is the size of DIMM. When one OS is running, only a portion of the RAM in the system can be used. We consider this the price of enhancing system security and plan to improve it in the future work.

5.2. Switching Between Two OSes

OS switching is conducted by both the operating system and the BIOS. After both OSes have been loaded into the memory, the switching is done by putting the currently-running OS into ACPI S3 sleep mode and then waking up the other OS from ACPI S3 sleep mode. We use ACPI S3 sleep/wakeup because it has defined functionality to save the CPU context and hardware devices' states. In ACPI S3 sleep mode, the CPU stops executing any instruction, and the CPU context is not maintained. The operating system will flush all dirty cache to RAM. The RAM context is maintained by placing the memory into a low-power self-refresh state. Only those devices that reference power re-

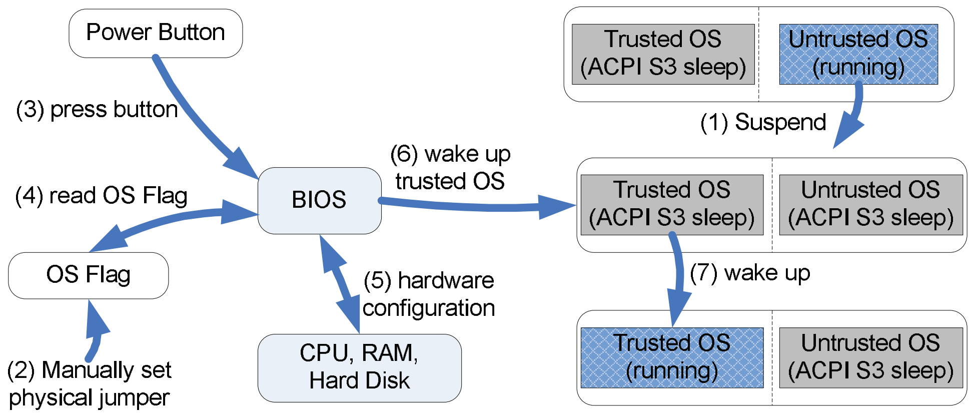
Power Button
(3) press button
(4) read OS Flag
BIOS
(6) wake up trusted OS
OS Flag
(5) hardware configuration
(2) Manually set physical jumper
CPU, RAM, Hard Disk
Trusted OS (ACPI S3 sleep)
Untrusted OS (running)
(1) Suspend
Trusted OS (ACPI S3 sleep)
Untrusted OS (ACPI S3 sleep)
(7) wake up
Trusted OS (running)
Untrusted OS (ACPI S3 sleep)

Figure 3. Switching Flow from Untrusted OS to Trusted OS.

sources are in the ON state. All the other devices (e.g., VGA, NIC) are in the D3 (OFF) state while their states are saved by the OS or the device drivers.

Figure 3 shows the control flow when the system is switching from the untrusted OS to the trusted OS. The user first suspends the untrusted OS by executing a userspace program or clicking a pre-defined *Standby* button, which is responsible for saving the CPU context and hardware devices' states. Afterwards, both OSes stay in the ACPI sleep mode. The user manually sets a physical jumper to indicate the BIOS that the trusted OS should be woken up next. In other words, the physical jumper controls the value of the *OS* flag. The user then presses the power button to wake up the system. This step is critical to make the system enter the BIOS first to enforce a trusted path.

The BIOS can distinguish OS S3 wakeup from OS booting using some register in the southbridge. In the south bridge VT8237R [42], the three bits of "Sleep Type" in the Power Management Control register is set to 001 for S3 sleep. After the BIOS reads the OS flag and decides to wake up the trusted OS, it programs the initial boot configuration of the CPU (e.g., the MSR and MTRR registers), initializes the cache controller, enables the memory controller, and jumps to the waking vector. Then, the BIOS forwards the system control to the trusted OS, which OS continues to perform the ACPI S3 wakeup and recover its CPU context and device states.

5.2.1 OS Flag Integrity

We must ensure the integrity of the OS flag to prevent Spoofing Trusted OS attacks. One challenge is to find a safe place to save the flag. First, we cannot simply save it in the RAM because the BIOS who needs the OS flag to enable the memory DIMM(s) cannot read the flag from RAM before the RAM has been enabled. Second, we cannot save the OS flag in the CMOS either, since the untrusted OS can manipulate the OS flag stored in the CMOS as the BIOS does. However, we can save the Boot flag in the CMOS. Because the Boot flag records if the untrusted OS has been loaded into the memory, the adversary can gain nothing aside from rebooting the untrusted OS by modifying the Boot flag.

Our system uses a physical jumper to control the value of the OS flag. This jumper can only be physically set by the local user, while the BIOS and the OS can only read it. In our system prototype, we use the standard parallel port to control the OS flag. In the D-Type 25-Pin Parallel Port Connector, the Pin Number 15 is used to signal an Error to the computer. The Status Port (base address +1) is a read-only port where Bit 3 reports the Error events. When the user connects Pin 15 (Error) and Pin 25 (the ground pin) with a jumper, the bit 3 of the Status port equals 0 and the BIOS will always wake up the trusted OS. When the user disconnects the two pins, the bit 3 of Status port equals to 1 and the BIOS will always wake up the untrusted OS. Our system uses parallel port connector due to its simplicity and availability on the prototype motherboard; many other hardware bits or devices can serve the same purpose in a computer.

5.2.2 Trusted Path Enforcement

Our system can enforce a trusted path when switching from the untrusted OS to the trusted OS. Because the BIOS is the only component that we can trust to enforce the trusted path, our system must ensure the BIOS is entered in the OS wakeup stage. Otherwise, an adversary could fake both the untrusted OS sleep and the trusted OS wakeup that totally bypass the BIOS. Our system uses system power Light Emitting Diodes (LED) to make sure the untrusted OS has been suspended. The power LED shows current system mode: it lights up when the OS is running, and it blinks (or changes to another color) when the system is in sleep mode. Since the power LED is hardware-controlled, the user can trust it to reveal if the untrusted OS has been suspended or not. Moreover, the system uses the power button to ensure the system enters the BIOS first. When the user presses the power button, the system will enter the BIOS first, no matter whether it boots up from scratch or wakes up from S3 sleep mode. Next, the BIOS is responsible for ensuring the trusted path to wake up the trusted OS.

The physical jumper indicates which OS is running and it is read-only, the user can use it to detect the spoofing trusted OS attacks when the system seems running a trusted OS environment but the jumper indicates an untrusted OS environment.

5.3. Enforcing System Isolation

Our system depends on the BIOS and the ACPI S3 mode of the trusted OS to enforce resource isolation between the two OSes. Most modern OSes (e.g., Linux and Windows XP) support ACPI S3 suspend/wakeup mechanisms, which is used to enforce the isolation on CPU and I/O devices (e.g., VGA and NIC). This dramatically lessens our need to save/recover the CPU context and devices' states. The BIOS must be customized to enforce isolation on RAM and hard disk, which cannot be thoroughly isolated by the OS alone. In the following, we first introduce the isolation capability of the ACPI S3 on CPU, NIC, and video devices. We then present the mechanisms using the BIOS and the OS to enforce the isolation of RAM and the hard drive.

CPU Isolation: According to ACPI standards [24], the CPU context will be lost during the S3 sleep, and the untrusted OS cannot get any CPU context information of the trusted OS. The OS is responsible for saving and restoring the CPU context. The trusted OS always follows the standard and saves the CPU context. In the untrusted OS, an attacker has only two options: either saving the CPU context or not saving it. If the attacker modifies the OS to avoid saving the CPU context, the untrusted OS cannot be resumed and this becomes a DoS attack.

NIC Isolation: In S3 sleep, most of the devices are put into D3 (a no-power state for devices) state, during which the contexts for these devices are lost. Thus, there is no information leakage during the switching from the trusted OS to the untrusted OS. According to ACPI specifications, a network card may provide Wake-on-LAN (WOL) functions to wake up the computer when the card stays in D0 or D3 power state. Our system only supports the network card in D3 state to wake up the computer, since the device in D0 state keeps its context that may be misused by the attacker. Fortunately, most of the current network cards support WOL at the D3 state [25].

Video Device Isolation: In S3 sleep, the content in the video buffer is lost. The ACPI specification does not require the BIOS to reprogram the video hardware or to save the video buffer, so the BIOS does not know how to wake up the video card from an non-programmed state. One easy way around this is to execute code from the video option ROM in the same way as the system BIOS does. vbetool [10] is one such small application that executes code from the video option ROM. It can run in the user space but may introduce some time delay in S3 sleep and wakeup.

Memory Isolation: Memory isolation is physically enforced by the BIOS. According to the OS flag, the BIOS knows which OS is going to be booted or woken up, and it then initializes or wakes up the corresponding DIMM for that OS. The other DIMM remains uninitialized or unconfigured (though it may still maintain its data content). Our system uses DDR2 memory.

DDR2 memory requires the BIOS to set the DQS settings in the memory controller (the north bridge) for memory read and write. In normal S3 sleep mode, system power is removed from the the memory controller; however, a copy of the DQS settings is still maintained in non-volatile RAM (NVRAM) of the south bridge. During an S3 wakeup, the BIOS copies the DQS settings from the south bridge to the memory controller.

Normally, a machine maintains only one set of the DQS settings. Our Secure Switch system must store two sets of different DQS settings to initialize/enable different DIMMs for two OSes. To wake up one OS, the BIOS should reset the DQS settings in the memory controller using the corresponding set of DQS settings. Since the NVRAM of the south bridge can only save one set of DQS settings, we must store the other set of DQS settings in some other non-volatile memory. Our solution is to save the other set of DQS settings in the CMOS. We save 64 bytes of Data Strobe Signal(DQS) settings, starting from the offset 56 of CMOS, which by default are not used according to the CMOS layout of the motherboard (ASUS M2V-MX SE).

The untrusted OS cannot initialize/enable the memory controller to access the DIMM for the trusted OS. First, the DQS settings contain more than one hardware register (*i.e., 16 registers on AMD K8 and 4 registers on AMD family 10h processors*), which means there is a transient state wherein the system cannot access any DIMM before all the DQS settings are complete. When an attacker exploits a short program to modify the DQS settings in the memory controller, the program cannot obtain the next instruction from the main memory and the system will hang. The BIOS can modify the DQS settings because it reads the instructions from the non-volatile ROM that is not controlled by the DQS settings. Second, even if the attacker can pre-fetch all the instructions and save them in the CPU cache or some non-volatile memory, it still cannot enable or have access to the other DIMM for the trusted OS. This is because besides the DQS settings, the BIOS uses a "DIMM Mask" byte to control which DIMM should be enabled, and the DIMM mask is set by the OS flag. When the DIMM mask conflicts with the DQS settings, the system will hang. Moreover, it is a very challenging task for the untrusted OS to (1) load the DIMM initialization and DQS setting instructions from RAM into the transparent CPU Cache; (2) control the instruction flow from RAM to Cache and then back from Cache to RAM; and (3) map the memory space of the trusted OS into its memory space and then fill the context gaps to read meaningful information from the RAM belonging to the trusted OS. Third, the untrusted OS can modify both DQS settings saved in the south bridge and in the CMOS. However, the conflicts between DQS settings and the DIMM Mask will hang the system.

Hard Disk Isolation: The non-volatile storage, such as hard disks used by the trusted OS, should be completely isolated from the untrusted OS to prevent information leakage. One direct solution is to encrypt a portion or the entirety of the hard disk before sleeping the trusted OS and to decrypt it after waking it up. However, the encryption/decryption operations will increase the switching time, along with the size of the hard disk.

Most motherboards (e.g., ASUS M2V-MX SE, in our implementation) have more than one SATA Channels to support more than one hard disk. When each OS can have its own hard disk, there are two methods to constrain access to the hard disk of the trusted OS. First, some hard disks support disk lock, an optional security feature defined by AT Attachment (ATA) specification [1]. This lock allows the user to set a password to lock a hard disk. Without knowing the password, an adversary cannot access the hard disk. The limitation of this method is that not all hard disks are provided with this feature. Second, according to the OS flag, the payload of BIOS (e.g., SeaBIOS), which is responsible for hard disk initialization, can initialize only one of the two hard disks by setting the SATA Channel enable register (e.g., Bus0, Device15, Function0, offset0x40 on southbridge VT8237r). However, if an attacker knows the southbridge data sheet, the untrusted OS may reset the SATA Channel enable register and initialize both hard disks. To prevent the attacker from re-enabling the hard disk, we use the SMM-based monitoring mechanism to check the settings of the hard disk configuration. If SMM detects that the hard disk used by the trusted OS is enabled when the untrusted OS is running, it will trigger an alarm to notify the user. The details of an SMM-based monitoring mechanism can be found in [44, 13].

With the observation that most browser-based applications in the trusted OS only require a small amount of data (e.g., browser cookies) be saved temporarily, our system uses RAM disk to store the dynamic sensitive data in the RAM. With Linux kernel version 2.6.18, we set the kernel parameter *ramdisk_size* to initialize 256MB RAM disk. After booting into the trusted Linux OS, we create a directory called */ramdisk* and mount RAM disk */dev/ram0* to the directory. However, it is not very user friendly. We improve upon it by using a stackable file system *aufs* [32, 43] to mount a read-write layer of RAM disk on top of regular directories, which are mounted as read-only. We mount a read-only *home* directory to */ramdisk/home*, so all the files created under the */home* directory will be written into */ramdisk/home*, which is in RAM. Since the RAM is isolated between the trusted and untrusted OSes, the files in the RAM disk cannot be accessed by the attacker. Moreover, the files in RAM disk are lost after a reboot. We have considered using Live CD as the trusted OS and tried Live CDs for Chrome OS, Centos, Linux Mint, Ubuntu, and Fedora; however, none of them provide enough support for ACPI S3 sleep/wakeup.

5.4. Security Analysis

Our system can ensure a firmware-assisted resource isolation between two OSes to prevent data exfiltration attacks. The untrusted OS cannot steal data from the trusted OS or compromise the integrity of the data in the trusted OS. Our system can also enforce a trusted path during secure switching to prevent the spoofing trusted OS attacks. We do not prevent DoS attacks because the user can easily notice this attack and boot the machine to recover.

Data Exfiltration Attacks. The untrusted OS cannot steal any data information from the trusted OS using either shared or separated devices. The two OSes have separated RAM DIMMs and hard disks. Since the untrusted OS cannot change memory DQS settings without crashing the system, an attacker cannot access the DIMM of the trusted OS. To protect the dynamic sensitive data in the trusted OS, we could either save the data in RAM disk or save them in the hard disk, which is protected by a SMM-based monitoring mechanism.

The two OSes share all other hardware devices aside from RAM and the hard disk. The ACPI S3 sleep guarantees that the trusted OS won't leave any sensitive data on those devices to be accessed by the untrusted OS. First, the CPU context, including registers and caches, will be flushed during S3 sleep. In AMD K8, the north bridge is integrated in CPU and its content is flushed, too. The NVRAM in south bridge only records some system configuration data. Second, for hardware devices with their own buffers, such as VGA and NIC, all of the content in their buffers will be lost because those devices lose power in S3 sleep.

Spoofing Trusted OS Attacks. Our system can prevent spoofing trusted OS attacks by enforcing a trusted path during OS switching and protecting the integrity of the OS flag. We use the system power LED to ensure that the untrusted

OS has been suspended, and use the power button to enforce that the system enters the BIOS first when it is powered on. We use a physical jumper to protect the integrity of the OS flag. Note the power LED, the power button, and the physical jumper are all hardware-controlled, so the untrusted OS cannot change them.

Network Attacks on Trusted OS. We assume that the trusted OS is secure and can be trusted when it boots up. However, since an OS contains tens of thousands of lines of code, vulnerabilities exist that can be misused by attackers from the network. Our system can guarantee that the trusted OS won't be compromised from the untrusted OS. However, if normal users use the trusted OS for a long time, we cannot guarantee that the trusted OS won't be compromised from network attacks. The stateless OS mode can only alleviate this attack by restoring the trusted OS to a pristine state every time it is woken, but it cannot prevent this attack. One promising solution is to employ some TPM- or SMM-based integrity checking mechanisms [23, 44] to detect any OS tampering attempts by comparing the newly-generated OS states with a clean state. This, however, is beyond the scope of this paper.

Side Channel Attacks. VMM-based solutions (e.g., Xen [14]) provide virtual resource isolation and may be susceptible to side channel attacks [33, 35]. For instance, co-located VMs on the same machine may have implicit resource sharing (e.g., Cache) that may be manipuated by attackers to extract sensitive information such as workload information [37]. Instead, SecureSwitch provides physical resource isolation and can prevent Cache-based side channel attacks by flushing the Cache during OS switching. SecureSwitch cannot protect against side channels that take advantage of malicious device firmware. The problem of malicious device firmware is an orthogonal problem and but is an active research area that SecureSwitch is not trying to solve. For instance, the Interactive Link system [12] can help prevent side channels in keyboard, mouse, and monitor.

6. Implementation & Experimental Results

We implement a prototype of the SecureSwitch system using an ASUS M2V-MX_SE motherboard with VIA K8M890 as the northbridge and VIA VT8237R as the southbridge. The CPU is AMD Sempron 64 LE-1300. Two Kingston HyperX 1GB DDR2 memory modules and two Seagate Barracuda 7200 RPM 500GB hard disks are installed. We connect a laptop to the motherboard through a serial port for debugging and data collection.

We install CentOS 5.5 on one hard disk as the trusted OS, and Windows XP SP3 on another hard disk as the untrusted OS. Our implementation also supports two CentOS 5.5 (or Windows XP) OSes. We use the open-source Coreboot V4 [2] and SeaBIOS [8] as the BIOS. The total new lines of code(LOC) we added in the BIOS is 120. It is significantly smaller than hypervisor- or microkernel-based methods [41], which rely on an extra software layer in addition to the BIOS.

Table 1. OS Switching Time

Switching Operation	Secure Switch(s)
Windows XP Suspend	4.41
CentOS Wakeup	1.96
Total	6.37
CentOS Suspend	2.24
Windows XP Wakeup	2.79
Total	5.03

6.1. OS Loading and Switching Time

OS loading time is the time duration for loading two OSes into the memory, and the OS switching time measures the time duration when the system switches from one OS to another OS. We use the real-time clock (RTC) to measure the OS loading time. At the beginning of the BIOS code, we print out the RTC time to the laptop through the serial port. This time records the beginning time of OS loading. We record the ending time when the "rc.local" file is executed in CentOS or when a startup application is called in Windows XP. The total OS loading time is 153 seconds, 74 seconds for loading Centos and 79 seconds for loading Windows XP. OS loading time only occurs once when the user boots up the system, and it may be reduced by using solid-state drive.

OS Switching time consists of two parts: the time to suspend current OS and the time to wake up the other OS. We use the 64-bit Time Stamp Counter (TSC) to measure the OS wakeup time for both CentOS and Windows XP. TSC counts the number of ticks since reset. We write a user-level program to obtain the current TSC value once it is being executed, and then calculate the wakeup time as TSC*(1/CPU frequency). However, it is difficult to use TSC to measure Windows XP's suspend time. First, we cannot change the source code of Windows to record the time when the OS suspend ends. Second, since the BIOS is not involved in the OS suspend process, it does not know when the OS suspend ends either. Instead, we use an Oscilloscope, Tektronix TDS 220, to measure the suspend time. Before a customized program initiates the ACPI S3 sleep, it sends an electrical signal to the Oscilloscope to indicate the start of S3 sleep. When the OS finishes S3 sleep, the oscilloscope will receive a power-off electrical signal. We use Oscilloscope to measure the suspend times for both CentOS and Windows XP.

In table 1 we show that the OS switching time from CentOS to Windows XP is 5.03 seconds, which is a little faster than switching from Windows XP to CentOS. For both OSes, the suspend time is longer than the wakeup time. Windows XP's suspend and wakeup times are longer than those of CentOS. Table 1 only provides a rough latency measurement that is constrained to the specific hardware and software used in our prototype system. For instance, when we replace the integrated VGA card (VIA chip, 256 MB memory) with an external VGA card (S3 chip, 64 MB memory), the OS suspend time is reduced due to a smaller video memory size. Moreover, when we run multiple while(1) programs on CentOS, the switching time is three times longer. This leads us to breakdown the operations in BIOS, user space, and kernel space to understand the major contributors for the OS switching delay. Due to the closed-source nature of Windows XP, we only break down the operations on the CentOS.

6.1.1 Linux Suspend Breakdown

We use Ftrace [3] to trace the suspend function calls in Linux S3 sleep. According to the function call graph generated by Ftrace, the suspend operations can be divided into two phases: *user space suspend* and *kernel space suspend*. We use the *pm-suspend* script in CentOS to trigger the OS suspend. This script first notifies the Network Manager to shut down networking, and then uses vbetool [10] to call functions at video option ROM to save VGA states. Next, it jumps to the kernel space by echoing string "mem" to /sys/power/state. In the kernel space, the suspend code goes through the device tree and calls the device suspend function in each driver. The kernel then powers off these devices. To measure the user space suspend time, we record the TSC time stamp in file /var/log/pm/suspend.log. For kernel time measurement, we add $printk$ statements between various components of the kernel.

Figure 4 shows the time breakdown for user space suspend. Each bar is an average of 10 measurements, and the Y axis error bars show confidence interval at 95% confidence. The total suspend times for user space is 1517.33 ms. The OS spends time on calling vbetool [10] to save video states to the */var/run* directory, changing the GUI terminal to */dev/tty63* as the foreground virtual terminal, and stopping the Network Time Protocol Daemon and writing the current system time to RTC time in CMOS. Other operations include stopping network manager and saving the states of CPU frequency governors, etc. Figure 5 shows the time breakdown for kernel space suspend, where the total suspend times is 1590.14 ms. The most time-consuming operations are to stop the keyboard, mouse, and hard disks.

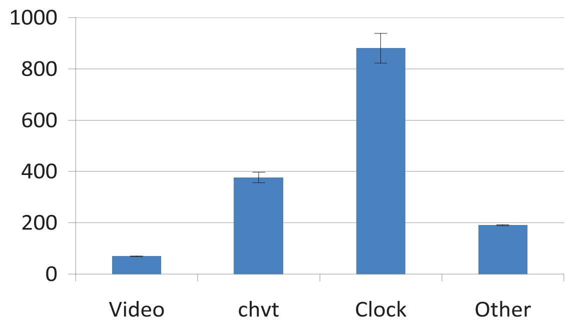

Figure 4. User Space Suspend Breakdown (ms).

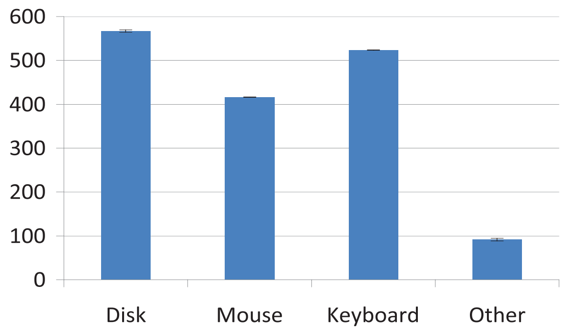

Figure 5. Kernel Space Suspend Breakdown (ms).

It takes a while to reset the PS/2 mouse and keyboard devices in our system. The hard disk delay comes from synchronizing the 16 MB cache on each SATA disk [1]. The kernel stops other devices (e.g., USB, serial ports) with relatively less time.

6.1.2 Linux Wakeup Breakdown

S3 wakeup operations are provided by both the BIOS and the OS. The wakeup process starts from a hardware reset. The system enters the BIOS first, and then jumps to the OS wakeup vector. The latency time in BIOS is constant and equals to 1259.25 ms. The OS wakeup operations can be divided into two parts: kernel space wakeup and user space wakeup. Figure 7 shows the time breakdown for the major components in the kernel space, where the total latency is 1537.22 ms. The major delay contributors are the USB and the mouse. Since Coreboot doesn't provide an optimized support for the USB, the OS must initialize the four USB ports on the motherboard. Moreover, the mouse initialization takes more time than the keyboard due to lack of support in the Coreboot. Figure 6 shows the wakeup time breakdown in the user space, where the total latency

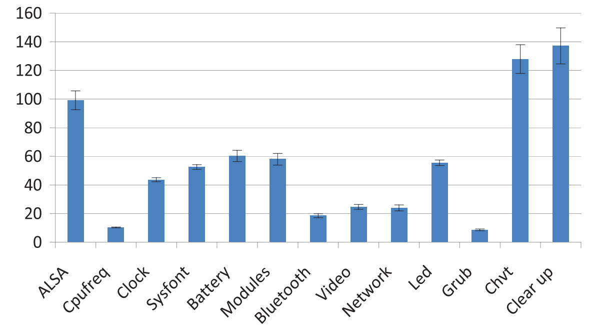

Figure 6. User Space Wakeup Breakdown (ms).

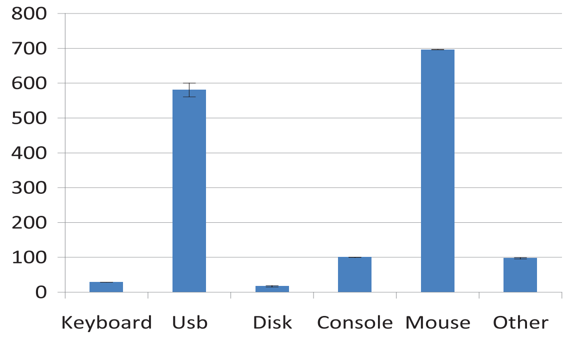

Figure 7. Kernel Space Wakeup Breakdown (ms).

is 621.04 ms. Initializing the advanced Linux sound architecture (ALSA) for sound card, changing the foreground's virtual terminal, and cleaning up the files take most of the time.

6.2. Comparison with Other Methods

We compare the SecureSwitch system with other solutions that target at protecting the execution of security-sensitive code on legacy systems [29, 28, 41, 14]. Table 2 presents the comparison results. In SecureSwitch, the trusted computing base (TCB) is the BIOS code, which has 248,421 lines of code (LOC) including Coreboot and its payload SeaBIOS. We only add around 120 new LOC in the BIOS. The TCB in Lockdown [41] includes not only the BIOS, but also a light-weight hypervisor with 8,471 LOC. Both Flicker [28] and TrustVisor [29] don't rely on BIOS's security. TrustVisor has a TCB of 6,351 LOC, and Flicker has only 250 LOC in its TCB. Xen 4.1 release [14] provides a hypervisor with 263,173 LOC.

Since the system will always enter the BIOS first after the system either boots up or wakes up, the BIOS plays an important role on enforcing the trusted path. SecureSwitch needs to modify the source code of the BIOS in the system. For the motherboards and processors that are not supported by the open source Coreboot [2], we could either work with BIOS vendors to obtain the BIOS source code or perform reverse engineering tasks on the BIOS binary code.

Both Xen and TrustVisor use hypercalls and page faults to swap context, and their switching times are less than 1 millisecond. Flicker has a 1 second switching time due to its frequent use of the Dynamic Root of Trust Measurement (DRTM) [23] hardware support for every switching operation, while TrustVisor is only initialized via DRTM. SecureSwitch can achieve a 6-second switching time by using the ACPI standard that has been widely supported by hardware manufactures for efficient power management. Lockdown requires around 40 seconds to switch from one OS to another one. TPM is required by Lockdown, TrustVisor, and Flicker to provide more secure functions such as remote attestation and secure storage that are not supported in SecureSwitch; however, it may not be available on legacy systems.

In Flicker and TrustVisor, the security code must be custom-compiled or ported to run in the secure environment. Although possible, it would seem to be an engineering challenge to port all existing code to support this, especially for an entire commercial OS. The legacy applications and OSes can run directly on SecureSwitch, Lockdown, and Xen without any changes. The memory overhead in SecureSwitch is high due to the coarse physical isolation on the DIMMs. The memory overheads in Lockdown, TrustVisor, and Flicker are fairly low; while since Xen can provide more functions, it needs more memory space. In SecureSwitch, Lockdown, and Flicker, when a security code is running in the trusted environment, the untrusted OS and the applications in the untrusted environment are fully stopped. TrustVisor and Xen support more than one OS concurrent execution.

We compare the computation overhead among the above systems as the performance of applications when they are running in the trusted environments. SecureSwitch and Flicker have a low computation overhead, since they run applications on the bare metal. Lockdown, TrustVisor, and Xen require 5-10% more computation overhead in the trusted environment due to usage of the Extended/Nested page tables. TrustVisor and Xen might experience slightly more slowdown due to the CPU contention among multiple OSes.

7. Related Work

SecureSwitch was inspired by Lampson's Red/Green system separation idea [27]. Instead of trying to solve all the Red/Green system's challenges, such as how the users decide what applications go into each OS and how to give the user certain control over data exchanges between the two OSes, we focus on providing a strong resource isolation between the two OSes.

Table 2. Comparing SecureSwitch with Other systems

	SecureSwitch	Lockdown [41]	TrustVisor [29]	Flicker [28]	Xen [14]
Trusted Computing Base	BIOS	Hypervisor+BIOS	Hypervisor	250 LOC	Hypervisor
Switching Time (second)	≈ 6	40	<0.001	1	<0.001
Hardware Dependency	ACPI	ACPI+TPM+ VT-x/SVM	TPM+ VT-x/SVM	TPM+ VT-x/SVM	VT-x/SVM*
Software Compatibility	High	High	Low	Low	High
Memory Overhead	High	Low	Low	Low	Medium
OS Concurrency	No	No	Yes	No	Yes
Computation Overhead	Low	Medium	Medium	Low	Medium

* Xen requires VT-x/SVM to support full virtualization.

The closest in terms of concept and implementation is the Lockdown [41] system that uses a hardware switch and LEDs to provide a trusted path to a small hypervisor, which ensures virtual resource isolation between two OSes. Lockdown relies on the light-weight hypervisor to ensure that trusted applications can only communicate with trusted sites and thus can prevent malicious sites from corrupting the applications, while SecureSwitch does not. To switch, it also uses a ACPI-based mechanism (S4) to hibernate one OS and then wake up another one. Unfortunately, it requires more than 40 seconds to switch because hibernating requires writing the whole main memory content to the hard disk and reading it back later on. In contrast, SecureSwitch can accommodate two OSes into the memory at the same time and offers a switching time of approximately 6 seconds.

Flicker [28] and TrustVisor [29] employ a hardware support called Dynamic Root of Trust Measurement (DRTM) with a small trusted computing base to create a secure environment. Flicker creates an on-demand secure environment using DRTM, while TrustVisor uses DRTM to securely initialize a light-weight hypervisor that uses hardware virtualization (VT-x/SVM) to protect the applications running in the secure environments. The two systems use the TPM to provide remote attestations and to securely store data when they are not executing. The major concern about Flicker and TrustVisor is that the security code must be custom-compiled or ported to run in the secure environment, and it is an engineering challenge to port an entire commercial OS in the secure environment.

There is a line of research that uses hypervisors (VMMs) to add an extra layer of control between the OSes and the underlying hardware, including HyperSpace [7], Terra [21], Safefox [43], Tahoma system [20], Overshadow [17], and Nettop [30]. Others attempt to protect the integrity of the hypervisor [39, 18, 13, 44, 45], or to protect the kernel [40, 36, 34]. All of these systems depend upon the integrity of the shared hypervisor code for the isolation between two environments. Nevertheless, attacks against the hypervisors are more and more frequent today [31, 48, 47]. Although the hypervisor may have a smaller attack surface compared to the traditional OSes, it is still vulnerable to attack. SecureSwitch employs immutable BIOS-protected code so that minimal code is shared between the trusted and the untrusted environments. When comparing with VMM-based solution, SecureSwitch has some usage limitations. SecureSwitch can only provide coarse-grained isolation on memory DIMMs and hard disks, so it is not as scalable as the VMM-based approaches on accomodating multiple OSes on the same machine. Moreoever, SecureSwitch does not support concurrent execution of the two OSes. It also requires local user attention and hence is not effective when the user is away from his/her computer.

8. Conclusions

The ever increasing size and complexity of desktop applications is an undeniable trend. This fact coupled with the requirement to operate on foreign, untrusted content using software that is constantly updated has given rise to the need for disposable and context-dependent, trustworthy environments. These environments will empower the user to segregate different activities thus lowering the attack surface and data exposure while maintaining system usability.

To meet this need for usable trustworthy workspaces, we propose a novel BIOS-assisted mechanism for the secure management of execution environments. A design tenet of our system was the ability to quickly and securely switch between operating environments without extensive code modifications or need for specialized hardware. At the same time, we wanted to minimize the code attack surface and prevent mutable, non-BIOS code from being able to subvert the switching process. Lastly, the system had to offer protection against attacks that aim to deceive the user's perception of the operating environment he/she is currently in. The proposed framework achieves all of these goals and can operate in conjunction with existing VMM-based isolation systems offering multi-layer isolation approach. In our prototype implementation, the switching process takes approximately six seconds. Moreover, the user can clearly discern the state of the system and seamlessly switch between untrusted and trusted OSes to perform sensitive transactions. Finally, SecureSwitch is more suitable for environments where users have both trusted and untrusted workloads that do not have to be executed in parallel but rather frequently alternate.

9. Acknowledgments

The authors would like to thank the shepherd of this paper Bryan Parno from Microsoft Research and the anonymous reviewers for their valuable comments and suggestions. The authors also want to thank Rudolf Marek from the Department of Computer Science and Engineering, FEE CTU in Prague for his helpful comments on Coreboot development within this project.

References

[1] AT Attachment specification. http://www.t13.org/.

[2] Coreboot. , http://coreboot.org/.

[3] Ftrace. http://elinux.org/Ftrace.

[4] IBM X-Force 2010 Mid-Year Trend and Risk Report. ftp://public.dhe.ibm.com/common/ssi/ecm/en/wgl03003usen/WGL03003USEN.PDF.

[5] Intel Corp. Intel I/O Controller Hub 9 (ICH9) Family Datasheet (2008) .

[6] MITRE CVE Vulnerability Database. http://cve.mitre.org/.

[7] Phoenix hyperspace. http://en.wikipedia.org/wiki/Phoenix_Technologies.

[8] Seabios. http://www.coreboot.org/SeaBIOS.

[9] Unified Extensible Firmware Interface (UEFI). http://www.uefi.org/home/.

[10] vbetool, http://linux.die.net/man/1/vbetool. http://linux.die.net/man/1/vbetool.

[11] Advanced Micro Devices, Inc. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, April 22, 2010.

[12] M. Anderson, C. North, J. Griffin, R. Milner, J. Yesberg, and K. Yiu. Starlight: Interactive link. *Computer Security Applications Conference, Annual*, 0:55, 1996.

[13] 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.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[15] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 281–289, 2003.

[16] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[17] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. 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*, pages 2–13. ACM, 2008.

[18] G. Coker. Xen security modules (xsm). *Xen Summit*, 2006.

[19] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, 2006.

[20] R. Cox, J. Hansen, S. Gribble, and H. Levy. A safety-oriented platform for web applications. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–364. IEEE, 2006.

[21] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. *ACM SIGOPS Operating Systems Review*, 37(5):193–206, 2003.

[22] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[23] T. C. Group. Trusted platform module main specification. version 1.2, revision 103, 2007.

[24] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced Configuration and Power Interface (ACPI). http://www.acpi.info/.

[25] Intel. PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual. http://download.intel.com/design/network/manuals/8254x_GBe_SDM.pdf.

[26] S. T. King, P. M. Chen, Y. min Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, pages 314–327, 2006.

[27] B. Lampson. Privacy and security: Usable security: how to get it. *Commun. ACM*, 52:25–27, November 2009.

[28] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328. ACM, 2008.

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

[30] R. Meushaw and D. Simard. Nettop-commercial technology in high assurance applications. *VMware Tech Trend Notes*, 2000.

[31] National Institute of Standards and Technology (NIST). National Vulnerability Database. http://nvd.nist.gov.

[32] J. R. Okajima. AnotherUnionFS(aufs). http://aufs.sourceforge.net/.

[33] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*, pages 1–20, 2005.

[34] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. *IEEE Symposium on Security and Privacy*, pages 233–247, 2008.

[35] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[36] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2008.

[37] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, 2009.

[38] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. *Proceedings of BlackHat DC 2007*, 2007.

[39] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. *Computer Security Applications Conference, Annual*, 0:276–285, 2005.

[40] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, page 350. ACM, 2007.

[41] A. Vasudevan, B. Parno, N. Qu, V. Gligor, and A. Perrig. Lockdown: A Safe and Practical Environment for Security Applications (CMU-CyLab-09-011). Technical report, 2009.

[42] I. VIA Technologies. VT8237R South Bridge, Revision 2.06, December 2005.

[43] J. Wang, Y. Huang, and A. Ghosh. SafeFox: A Safe Lightweight Virtual Browsing Environment. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–10. IEEE, 2010.

[44] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[45] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, 2010.

[46] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554. ACM, 2009.

[47] R. Wojtczuk. Adventures with a certain Xen vulnerability (in the PVFB backend). http://www.invisiblethingslab.com/resources/misc08/xenfb-adventures-10.pdf.

[48] R. Wojtczuk. Subverting the Xen hypervisor. http://www.invisiblethingslab.com/bh08/papers/part1-subverting_xen.pdf, 2008.