Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

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

Kun Sun ksun3@gmu.edu
Jiang Wang jwanga@gmu.edu
Fengwei Zhang fzhang4@gmu.edu
Angelos Stavrou astavrou@gmu.edu

Technical Report GMU-CS-TR-2011-7

Abstract

Protecting commodity desktop systems that run commercial operating systems (OS) without adversely impacting performance or usability remains an open problem. To make matters worse, the overall system security depends on desktop applications with complex code-bases that perform multiple and inter-dependent tasks often dictated by Internet-borne code. Recent research has indicated the need for context-dependent trustworthy environments where the user can segregate different activities in an effort to lower risk and safeguard private information.

In this paper, we introduce a new BIOS-assisted mechanism for the secure generation and management of trusted execution environments, tailored to separate security-sensitive activities from untrusted ones. A key characteristic of our system is usability: the capability to quickly and securely switch between operating environments in a single physical machine without the need for any specialized hardware or extensive code modifications. Our goal was to eliminate any mutable, non-BIOS codesharing while reusing existing processor capabilities. We demonstrate that even if the untrusted OS becomes compromised, it cannot perform an exfiltration or inference attack on data or applications in the trusted OS. To avoid sophisticated attacks that fake a trusted environment, we provide visible indication to the user about the current environment. Moreover, to alternate between environments, we require the user to physically press a button, an action that cannot be reproduced by software. Using our prototype, we measured the switching process to be approximately six seconds. This short switching time empowers the user to frequently and seamlessly switch between trusted and untrusted environments.

1 Introduction

Nowadays, desktop computers are being employed for multiple tasks from Desktop computers are being employed nowadays for multiple tasks ranging from pleasure to business: 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 [5]. 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.

To this end, there is an ongoing effort to employ virtual machine monitors (VMMs, also referred to as hypervisors) to isolate different activities and applications [33, 14, 15, 22, 36, 21, 35, 16]. 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 [24] and the number and nature of attacks [38, 28] against the hypervisors are poised to grow. Researchers have noticed this problem and have begun to improve hypervisor security [35, 11, 34].

An alternative 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 inconvenient and time consuming to shutdown one OS and boot up another. For instance, Lockdown [31] 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 is still too long, in many cases 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 designed 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 (e.g., BIOS_CNTL register [4] in Intel ICHs) 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 state is 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.

To prevent fake OS attacks, we must enforce a trusted path when the system switches from the untrusted OS to the trusted OS. This guarantees that the system really suspends the untrusted OS and wakes up the trusted OS. Otherwise, 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 faked trusted OS environment, which is controlled by the compromised indicates which OS should be waken up. The value of the flag can only be manually changed by the user, and it cannot be changed by any untrusted OS. We refer to such events as "Fake OS attacks." To prevent such an attack, we use the power button and power LED to indicate to the user when the system enters the BIOS after one OS is suspended. The BIOS will then wake up one OS according to a system OS flag that indicates which OS should be woken. The value of the flag can only be manually changed by the user; it cannot be changed by any software.

We harness the Advanced Configuration and Power Interface (ACPI) [18] 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 [31]. 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 within this paper:

- *Secure OS switching without using any mutable software layer.* Our system depends on the BIOS and some hardware properties to enforce a trusted path when switching between the two OSes. Our solution requires no modification of the commodity OS.

- *Discernible trusted path and no data leakage between two environments.* The resource isolation enforced by the BIOS prevents data leakage from the trusted OS to the untrusted OS. The trusted path can prevent the dangerous fake OS attacks.

- *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 the two 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 [18]. 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 nor 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, northbridge, southbridge, hard disk, and some other necessary IO devices, such as keyboard. BIOS code is normally stored on a non-volatile ROM chip built into the system on the mother board.

The BIOS is traditionally written by assembly language and works in real-address mode. In recent years, a new generation of BIOS, referred to as Unified Extensible Firmware Interface (UEFI) [8], 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. In some sense, coreboot is 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 finding out all of the data sheets for our motherboard.

2.3 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 [10]. We use DDR2 memory in our system.

A motherboard usually has more than one DIMM slot. BIOS normally uses a variable named "DIMM_MASK" to enable the DIMMs. 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.

3 Threat Model and Assumptions

Our system operates under the assumption that an adversary can subvert the untrusted OS using any known or zero-day attacks on software applications, device drivers, user-installed code, or operating system. We assume that the attacker cannot access the physical machine or launch local physical attacks, such as opening the case or removing a hard disk or a memory DIMM.

An adversary may launch various attacks against the trusted OS after compromising the untrusted OS. A data exfiltration attack aims at stealing sensitive data from the trusted OS. Furthermore, the adversary may modify the code of trusted OS and compromise its integrity. In a fake 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 crashing the untrusted OS; however, such attacks can be easily detected and resolved. Therefore, we do not prevent DoS attacks against our system.

We assume 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; however, they lie beyond the scope of this paper.

We assume that the BIOS code, including the option ROMs on devices (e.g., video cards), does not contain vulnerabilities and can be trusted. The operating system must support ACPI S3 sleeping mode, which has been widely supported by modern OSes, such as Linux and Windows. Our system does not require hardware virtualization support (e.g, Intel VT-x or AMD-V).

4 SecureSwitch Framework

Figure 1 illustrates the overall architecture of the SecureSwitch system. Two OSes are loaded into the RAM at the same time. Instead of relying on a hypervisor, we modify the BIOS to control the loading, switching, and isolation between the two OSes. Commercial OSes that support ACPI

S3 can be installed and executed without any changes. We require that the computer have at least two DIMMs and two hard disks and that the two OSes be installed on each hard disk.

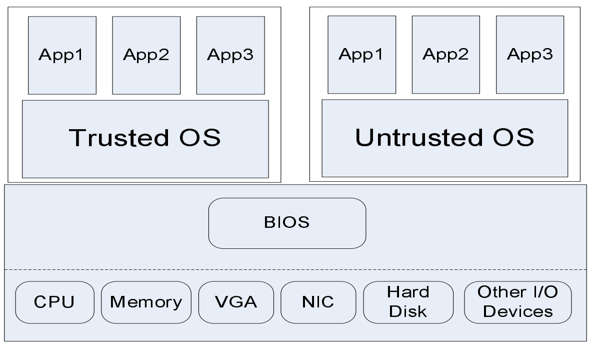

Figure 1: Architecture of SecureSwitch System

Secure Switching 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 when the system switches from the untrusted to the trusted OS.

The system must guarantee a thorough isolation between the two OSes. Usually one OS is not aware of the other coexisting 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 OS Loading

Figure 2 shows the state machine for loading and switching between two operating systems in the system. Two variables are maintained to denote the respective states of two OSes. In each parenthesis, the first and second number records the state of OS1 and OS2 respectively. 0 means the OS has not been loaded into the system; 1 means the OS has been loaded and is currently running; -1 means the OS has been loaded but switched out (i.e. is in sleep mode). For instance, (-1, 1) means that OS1 has been loaded but switched out and that OS2 is running. Since the system has only four valid states, we could use two bits to record the current state of the machine.

When the machine is powered off, the system state is (0,0). After the system is powered on, the BIOS always boots up the trusted OS (OS1) first. When loading the trusted OS from non-volatile memory (e.g., hard disk, USB), BIOS constrains the trusted OS to use only half of the physical RAM. The trusted OS is not aware of the existence of the second half of the physical memory. The trusted OS can be shut down or can perform sleep/wakeup. Before loading the untrusted OS (OS2), the trusted OS is first suspended. The BIOS then boots up the untrusted OS into the second half of the physical

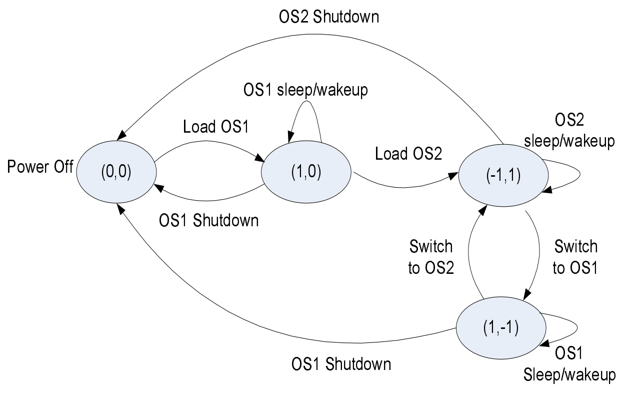

Figure 2: State Machine for OS Switching.

RAM, which has no overlap with the memory used by the trusted OS. At this point, both OSes have been loaded into the memory while the untrusted OS is running and the trusted OS is suspended. When either OS is shut down, the whole system will be shut down.

4.2 Secure OS Switching

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. Similarly, the user can switch back from the trusted to the untrusted OS. To save power energy, the user can still sleep and wake up the same OS.

4.2.1 Stateful vs. Stateless Trusted OS

When the system switches into the trusted OS, there are two options for restoring OS context.

- *Stateless mode*: Each time 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.
- *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 can be saved in the memory or on the hard disk.

The stateless mode does not save any system state when the OS is switched out. It can mitigate the impacts of network attacks to the trusted OS since it 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 a previous session or task within the trusted OS. Moreover, an adversary can 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.

4.2.2 Protecting Control Variables

In Figure 2, two variables record the system's current states. The system also uses these as control variables to control the system actions on OS loading and switching. It is critical to protect these control variables from being manipulated by an attacker.

Malicious code may manipulate the two variables to launch a fake OS attack. For instance, when the untrusted OS is running with system state (-1,1), the user wants to switch to the trusted OS. The control variables should first be updated from (-1,1) to (1,-1). To launch the attack, the untrusted OS can keep the control variables unchanged and then suspend itself. When the system reads the control variables, it will simply wake up the untrusted OS, which may create a faked trusted OS environment by installing a virtual machine similar to the trusted OS on the untrusted OS [19]. Since all of these actions may be transparent to the user, the attacker can deceive the user into a fake trusted OS to perform sensitive transactions.

In our prototype, we define two system flags to serve as the control variables. We could prevent the fake OS attacks by using some common hardware components to indicate the value of a system flag. Thus, the system flags can only be manually set by the user and cannot be configured by any software. The design details are described in Section 5.2.1.

4.2.3 Trusted Path

The secure switching consists of two sequential steps: *OS1 Suspend* and *OS2 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, which then hands over control to the operating system. Since a compromised untrusted OS can gain full control of the suspend step, it may fake an OS suspend (e.g., power off the monitor) and deceive the user into a fake trusted OS.

To prevent such fake OS attacks, our system must ensure a trusted path that guarantees that the system enter the real trusted OS when it switches to the trusted OS. In our system, the BIOS and some hardware are the only components that we can trust to enforce the trusted path. We must guarantee that one OS will be truly suspended in order to trigger the BIOS to enforce the trusted path.

4.3 Secure OS 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, chipset, and I/O devices.

4.3.1 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.

4.3.2 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 novel 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 5.3.2.

4.3.3 I/O Device Isolation

Typical I/O devices include a 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 its 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 the OS is switched in, the device buffer can be reloaded from the RAM or hard disk. 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 switching time dramatically due to costly encryption/decryption operations. Another solution is to use the RAM disk to save temporary sensitive data for the trusted OS because the untrusted OS cannot access the trusted OS's memory. Details regarding this can be found in Section 5.3.3.

5 System Design

We combine the BIOS and the standard ACPI S3 sleep 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. The integrity of the BIOS can be protected by the Trusted Platform Module (TPM) or by a signed signature from vendors.

The BIOS relies on two flags in the OS loading and switching process. The *OS_FLAG* indicates which OS (and corresponding resources) should be started; the *BOOT_FLAG* indicates whether the untrusted OS is being woken up or loaded.

5.1 Loading Two OSes

In the OS loading stage, the system loads both OSes in the RAM. To enforce RAM isolation without a hypervisor, our system requires that the motherboard have at least two DIMMs and two hard disks, and it assigns one DIMM and one 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 and hard disk to each OS. The other loading steps are the same as for a normal OS booting.

Our system uses ACPI S3 sleep for both secure switching and the normal OS sleep/wakeup. The BIOS uses the *OS_FLAG* to distinguish the two cases. When the trusted OS is suspended in S3 sleep, the BIOS can either wake up the trusted OS when *OS_FLAG* is set to 0 or wake up the untrusted OS when *OS_FLAG* is set to 1. However, a problem occurs when the BIOS tries to wake up the untrusted OS, which has not been loaded into the RAM at this time. To solve this problem, we use the *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 yet 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 *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.

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 to put the currently-running OS into ACPI S3 sleep mode and then to wake up the other OS from ACPI S3 sleep mode. We use ACPI S3 sleep/wakeup because it has defined functionalities 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 and save the CPU context. The Dynamic RAM context is maintained by placing the memory into a low-power self-refresh state. Only those devices that reference power resources 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.

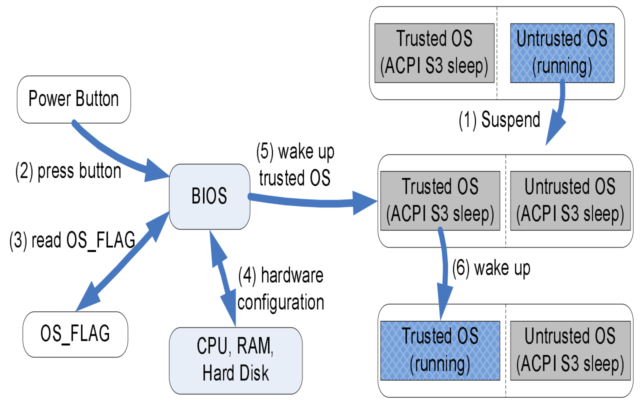

Figure 3: Switching Flow from Untrusted OS to Trusted OS.

Figure 3 shows the control flow when the system is switching from the untrusted to the trusted OS. The user first suspends the untrusted OS, which is responsible for saving the CPU context and hardware devices' states. Afterwards, both OSes stay in the ACPI sleep mode. The user then presses the power button to wake up the system. This step is critical to enforce a trusted path by making the system enter the BIOS first. We will discuss this further at a later point. The BIOS can distinguish OS S3 wakeup from OS booting using some register in the southbridge. For instance, in the south bridge VT8237R [32], the three bits of "Sleep Type" in the Power Management Control register is set to 001 for S3 sleep.

Next, the BIOS reads the flag *OS_FLAG* to decide which OS should be woken. According to *OS_FLAG*, the BIOS 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. After the BIOS forwards the system control to the OS, the trusted OS continues to perform the ACPI S3 wakeup and recover its CPU context and device states.

5.2.1 Integrity of System Flags

We must ensure the integrity of the system flags, *OS_FLAG* and *BOOT_FLAG*, to protect the system from fake OS attacks. However, the first challenge is to find a location to save these flags. We cannot simply save the two flags in the RAM because the BIOS uses the flags to enable the memory DIMM(s) and cannot read the flags before the RAM has been enabled. One solution is to save the flags in some unused bytes in CMOS. For example, we could save the *OS_FLAG* at offset 125 bytes of CMOS, and the *BOOT_FLAG* at offset 126 bytes of CMOS.

The second challenge is to ensure the integrity of the flags. Since the OS can also access CMOS, a compromised OS can change the flags in CMOS. For the *BOOT_FLAG* flag, the default value is 1, and it changes to 0 after booting the untrusted OS. If the untrusted OS changes the *BOOT_FLAG* flag to 1, the BIOS will load the untrusted OS again. Thus, the adversary can gain nothing aside from rebooting the untrusted OS, so we save the *BOOT_FLAG* flag in CMOS.

However, CMOS is not secure for the *OS_FLAG* flag. By manipulating the *OS_FLAG* flag from the OS, the adversary can launch the fake OS attack by waking the untrusted OS instead of the trusted OS and then faking the trusted OS GUI under the control of the untrusted OS.

To protect the *OS_FLAG* flag and support the normal S3 sleep and wakeup, we can use a hardware bit to indicate the value (0 or 1) of *OS_FLAG* flag. This hardware bit can be manually and physically set only by the user, while the BIOS and OS can only read it. For example, we could use the standard parallel port to control the bit. 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 wire or switch, 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. Many such hardware bits or devices can serve the same purpose in a computer.

Since the user controls which OS should be woken, our system can prevent the fake OS attacks while supporting the normal S3 sleep/wakeup. When users want to do a normal sleep and wakeup, they do not change the parallel port connection; if they want to switch to another OS, they change the connection. The connection correctly shows the current OS that the user is running, and we can build a small switch with a long wire (so that the user can reach it easily) and put the 'TRUSTED' and 'UNTRUSTED' text on the two ends of the switch.

5.2.2 Trusted Path Enforcement

Since the BIOS is the only component that we can trust to enforce the trusted path from the untrusted to the trusted OS, we use the power button to ensure that the BIOS is entered during the OS wakeup stage, as shown in the second step in Figure 3.

Since the *OS_FLAG* alone is not enough to enforce the trusted path, the above-mentioned step is critical to prevent fake OS attacks. Supposing that the untrusted OS could fake the untrusted OS S3 sleep and trusted OS wakeup process on the monitor, the BIOS and *OS_FLAG* could be totally bypassed. For such an attack, the attacker would also need to disable the power button events so that when the user presses the power button, the OS does not do what it is supposed to. To prevent such an attack, the user can use the system power LED to know the current system mode. The power LED lights up when the user turns on the power and blinks (or changes to another light, depending on the implementation) when the system is in sleep mode. Since the LED is hardware-controlled, the user can trust it to reveal if the untrusted OS has been suspended or not. If not, then this most likely is an attack.

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 trusted and the untrusted OSes. Modern OSes (e.g., Linux and Windows XP) support ACPI S3 suspend/wakeup mechanisms, which can be used to enforce the isolation on CPU and I/O devices (e.g., VGA and NIC), without any modification of the OS. 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 new mechanism using BIOS and OS to enforce the isolation of RAM and the hard drive.

5.3.1 Isolation based on Trusted OS

CPU Isolation: According to ACPI standards, 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. 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 and does not save the CPU context, the untrusted OS cannot be resumed and this becomes a DoS attack. The trusted OS always follows the standard and saves the CPU context.

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 functions to wake up the computer when the card stays in D0 or D3 power state. SecureSwitch 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 [**?**].

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 BIOS does not know how to wake up the video card from an unprogrammed state. One easy way around this is to execute code from the video option BIOS in the same way as the system BIOS does. vbetool [9] is one such small application that executes code from the video option BIOS. It can run in the user space but may introduce some time delay in S3 sleep and wakeup.

5.3.2 Memory Isolation

Memory isolation is physically enforced by the BIOS. According to the OS_FLAG 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 un-configured (though it may still maintain its data content). Prior to having the memory controller bring system memory out of the self-refresh mode, the BIOS is responsible for restoring the state of the processor's memory controller upon waking up from the S3 sleep state.

DDR2 and DDR3 memories need the BIOS to set the DQS settings for read and write. When the BIOS boots up the system, it searches the best settings for DQS. In addition to setting the memory controller in the north bridge, the BIOS also saves a copy of the setting in non-volatile RAM (NVRAM) of the south bridge. In normal S3 sleep mode, system power is removed from the memory controller (the north bridge); however, a copy of DQS setting is maintained in NVRAM in the south bridge. During an S3 wakeup, the BIOS copies the DQS settings from the south bridge to the memory controller (the north bridge).

A normal system keeps only one set of the DQS configurations, while our secure-switching system must keep two sets of different DQS configurations to initialize/enable different DIMMs for two OSes. To wake up one OS, the BIOS should reset the DQS setting in the memory controller using the corresponding set of DQS settings.

Since we cannot save two sets of memory controller configurations in the same set of chipset registers, we must store them somewhere during the S3 sleep mode. For custom-designed computers with specific hardware devices, we may save the two settings in the BIOS. However, in many scenarios, the memory control should be dynamically adjusted to achieve optimal performance under different voltages and temperatures. We cannot keep them in the RAM, either, because the RAM is not accessible at that time. Our solution is to save the other set of 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).

In our system, the memory controller can only be reset by the BIOS, while the untrusted OS cannot initialize/enable the memory controller to access the DIMM for the trusted OS. The DQS settings contain more than one hardware register. For instance, there are 16 registers on AMD K8 and 4 registers on AMD family 10h processors. This means that 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, the program cannot obtain the next instruction from the main memory and the system will hang. BIOS can modify the DQS settings because it reads the instructions from the ROM that is not controlled by the DQS settings.

The untrusted OS can modify both DQS settings saved in the south bridge and in the CMOS. However, besides the DQS setting, the BIOS uses a "DIMM Mask" byte to control which DIMM should be enabled. In our system, the BIOS sets "DIMM Mask" according to the OS_FLAG flag. We set the DIMM Mask to 0x01 only to enable the first DIMM, and to 0x10 only to enable the second DIMM. If the Mask conflicts with the DQS setting, the system will hang.

Note that another requirement for our system is that one memory controller should control more than one DIMM. In that case, even if there is more than one memory controller, the BIOS can initialize all of them to use part of the DIMMs connected with the controllers. Then the attacker still cannot modify the memory settings. Otherwise, if memory controllers are mapped one-to-one to the DIMMs, then some memory controllers have to be in an uninitialized (i.e., unconfigured) state that must be used by the trusted OS, and the attacker may try to initialize it by not affecting other memory controllers and DIMMs.

5.3.3 Hard Disk Isolation

The nonvolatile storage, such as hard disks used by the trusted OS, should be completely isolated from the untrusted OS in order to prevent information leakage. One 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) has more than one SATA Channels to support more than one hard disk. When each OS can have its own hard disk, the BIOS can 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, no one can 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 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 the attacker knows the southbridge data sheet, it may reset the SATA Channel enable register and initialize both hard disks.

With the observation that most applications in the trusted OS only require that a small amount of data (e.g., browser cookies) be saved on the hard disk, our system uses RAM disk to store the dynamic sensitive data in the RAM, which can be better protected by the system. In the beginning, we set up a special directory in the RAM disk and ask the user to save sensitive data into this directory. With Linux kernel version 2.6.18, we set the *ramdisk_size* parameter in memu.list to initialize about 256MB RAM disk. After booting into the trusted OS, we create a directory called */ramdisk* and mount RAM disk */dev/ram0* to the directory. However, the basic solution is not very user friendly, so we improve upon it by using a stackable file system (e.g., aufs [25, 33]) to mount a read-write layer of RAM disk on top of regular directories, which are mounted as read-only. We create a home directory under the */ramdisk* directory and mount */home* to */ramdisk/home*.

The */home* directory is mounted as read-only, and 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, so an attacker cannot access the sensitive data saved on the RAM disk after rebooting. For other binary program files, the system only needs to protect their integrity. For example, the user can run them from a CD or use other integrity-checking mechanisms to check those files.

5.4 Security Analysis

Our system can enforce a trusted path during secure switching and ensure a firmware-assisted resource isolation between two OSes. The untrusted OS cannot steal data from the trusted OS or compromise the integrity of the data in the trusted OS. Our system can prevent the Data Exfiltration attacks and the fake 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 and hard disks. Since the untrusted OS cannot change memory DQS settings without crashing the system, an attacker cannot access the memory DIMM of the trusted OS. To protect the small amount of data saved in the hard disk, we use RAM disk to save those sensitive data in the memory.

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.

Suppose that the user has started a sensitive Web transaction in the trusted OS, and he/she switches to the untrusted OS before the end of the transaction. If the remote server keeps sending sensitive data without any encryption, the attacker may receive this data. This problem can be solved by (1) protecting all Web transactions with encryption, or (2) closing all Web transactions by the OS before the switching occurs.

Fake OS attacks. Our system can prevent fake OS attacks by enforcing a trusted path during OS switching and protecting the system flag OS_FLAG. Another promising solution is to employ TPM for remote attestation, wherein a trusted remote server can verify the identify of the running OS

Network Attacks on Trusted OS. We assume that the trusted OS is secure and can be trusted. However, because 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 [17, 34] to detect any OS tampering attempts by comparing the newly-generated OS states with a clean state. However, that is beyond the scope of this paper.

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 with the prototype system through a serial port to debug and collect the experimental results. We also use a POST card to display the debugging codes from the BIOS.

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 [7] as the BIOS.

6.1 Trusted Computing Base (TCB)

The BIOS provides the Trusted Computing Base (TCB) of our system. We measure the size of our prototype using sloccount[1], and the total lines of code(LOC) we added is just 120. This LOC is significant smaller than the 8471 LOC in Lockdown [31], and it is also smaller than other hypervisor- or microkernel-based methods that rely on an extra software layer in addition to the BIOS.

6.2 OS Loading and Switching Latency

We measure two latencies during SecureSwitch: *system loading time* and *switching latency*. System loading time is the time duration for loading two OSes into the memory. We use the real-time clock (RTC) to measure it. To record the beginning time, we print out the RTC time through the serial port console at the beginning of the BIOS code. For the ending time, we record the time when the "rc.local" file is executed in CentOS or when a startup application is called in Windows XP. The loading times for both OSes are very close within our system: 74 seconds for loading CentOS and 79 seconds for loading Windows XP. The total loading time is 153 seconds. Though the loading time is relatively long, it only occurs once when the user boots up the system. Moreover, the loading time may be reduced by using solid-state drive.

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

OS Switching latency measures the time duration when switching from one OS to another. It consists of two parts: the time to suspend the current OS and the time to wake up another OS. We use the system's Time Stamp Counter (TSC) to measure the OS wakeup time. TSC is a 64-bit register that is present on all x86 processors since the Pentium, and it counts the number of ticks since reset. After pressing the power button, the TSC is reset to 0. We write a user-level program to obtain the current TSC value continuously. We then calculate the wakeup time as TSC*(1/CPU frequency). TSC can be used to measure the wakeup time for both CentOS and Windows XP. However, it is difficult to use TSC to measure Windows XP's suspension time without the Windows source code. Since the OS suspend does not involve the BIOS, we cannot use the BIOS to read the TSC value either. Instead, we use an Oscilloscope, Tektronix TDS 220, to measure the suspension time. We connect the oscilloscope to the serial port on the motherboard. When we initiate the ACPI S3 sleep, a customized program sends an electrical signal to the serial port to indicate the start of S3 sleep. When the system finishes S3 sleep, the oscilloscope receives a power-off electrical signal from the serial port. We use this method to measure the suspension delay for both CentOS and Windows XP.

Table 1: 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

The latency when the system switches from the trusted OS to the untrusted one is different from the latency when the system switches back, as shown in Table 1. We can see that switching from Windows XP to CentOS requires 5.03 seconds, which is a little faster than switching from CentOS to the Windows XP. 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, these measurements will change when we use an external VGA card or execute a large number of processes in the OS. The ASUS motherboard has one integrated VGA card with VIA chip and 256 MB video memory. When we insert an external VGA card with S3 chip and 64 MB memory, the external VGA card needs less suspension time than the integrated one since it has a smaller video memory size. To our surprise, the external VGA card requires a wakeup time that is three times longer than the integrated one due to the fact that coreboot needs to call the option ROM of the external video card, but it encounters a computability problem and dramatically delays the wakeup.

In addition, we run multiple while(1) programs on the Linux to see how the CPU intensive processes affect the switching time. When we run five while(1) programs at the same time, the switching time is about three times longer. We deduced that most of the increasing is due to the user space suspend and wakeup, while the delay in kernel space does not change much. This leads us to breakdown the operations in BIOS, user space, and kernel space to understand the major contributors for the time delay. Due to the closed-source nature of Windows XP, we only break down the operations on the CentOS 5.5 with Coreboot V4.

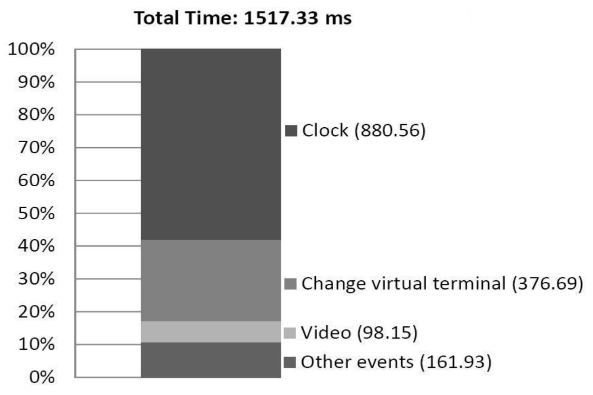

Figure 4: User Space Suspend Breakdown

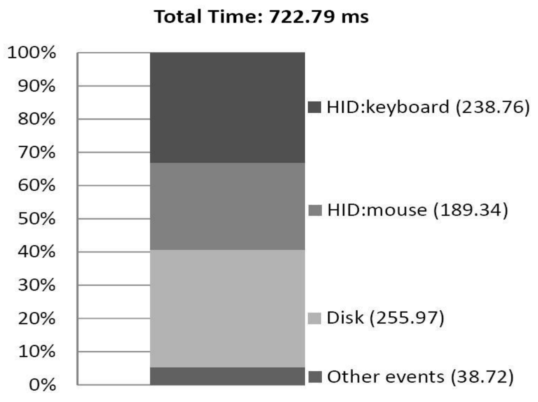

Figure 5: Kernel Space Suspend Breakdown

6.2.1 Linux Suspend Breakdown

We use Ftrace [3] to trace the suspension function calls in Linux S3 sleep. According to the function call graph generated by Ftrace, we divide the suspend operations into two phases: *user space suspend* and *kernel space suspend*. We use the *pm-suspend* script provided by the OS to trigger the suspend. The script basically notifies the Network Manager to shut down networking and uses vbetool [9] to call functions at video option ROM to save VGA states. It then echoes string "mem" to /sys/power/state. This jumps to the kernel space and stops the user space. 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, 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.

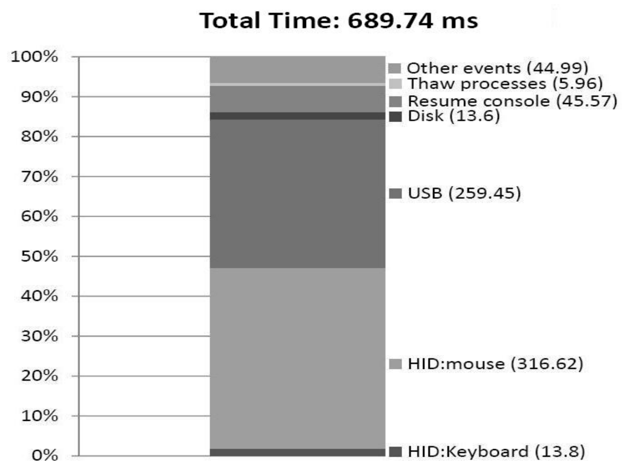

Figure 6: Kernel Space Wakeup Breakdown

Figure 4 shows the time breakdown for user space suspend, and Figure 5 shows the time breakdown for kernel space suspend. The total suspend times for user space and kernel space are 1517.33 *ms* and 722.79 *ms* respectively. In the user space, by running command *chvt 63*, the monitor changes the GUI terminal to */dev/tty63* as the foreground virtual terminal. In the clock operation, the OS stops the Network Time Protocol Daemon and writes the current system time to RTC time in CMOS. For the video operation, the OS uses vbetool [9] to save current video state to the */var/run* directory in memory. Other events include stopping network manager and saving the state of CPU frequency governors, etc. In the kernel space, the most time is consumed by stopping the keyboard, mouse, and hard disks. We use a PS/2 mouse and keyboard in our system. The suspending functions of the mouse and keyboard drivers reset the devices, which causes the delay. For the hard disk, delay comes from synchronizing the cache. The two hard disks each have 16 MB caches, and cache write is enabled by default for the SATA disk [1]. The OS also needs to stop other devices, such as the USB and serial ports, which takes relatively less time.

6.2.2 Linux Wakeup Breakdown

Unlike S3 suspend, S3 wakeup operations are handled by both the BIOS and the OS. The wakeup process starts from a hardware reset. The system enters the BIOS first, then jumps to the OS wakeup vector. The total latency time in BIOS is almost constant and equal to 1259.25 *ms*. Again, the OS wakeup operations can be divided into two parts: kernel space wakeup and user space wakeup. The wakeup latency in kernel space and user space are 698.74 *ms* and 612.04 *ms* respectively. Figure 6 shows the time breakdown for the major components in the kernel space. The major delay contributors in

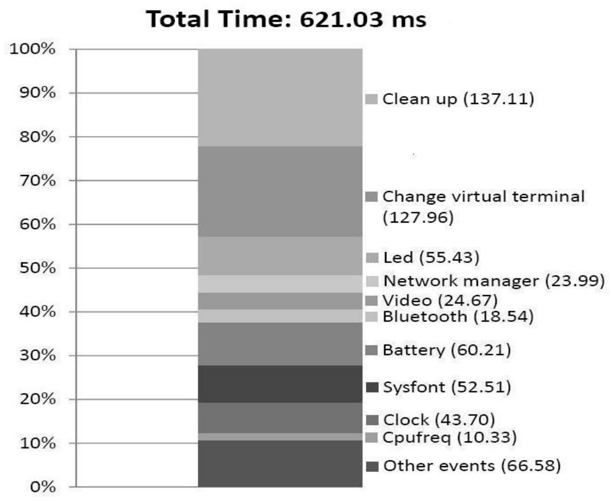

Figure 7: User Space Wakeup Breakdown

kernel space are the USB and the mouse. There are four USB ports on the motherboard. Since coreboot doesn't provide an optimized support for the USB, OS needs to initialize all four of the USB ports. The BIOS must initialize the keyboard, but not necessarily the mouse. We discovered that the mouse takes more time than the keyboard in kernel-space wakeup due to the OS initialization of the mouse. Figure 7 shows the time breakdown for wakeup in the user space. We can see that cleaning up the files and changing the foreground's virtual terminal (*chvt 1*) take up most of the time.

6.3 Comparison with Other Methods

There is recent research on protecting the execution of security-sensitive code on legacy systems [22, 21, 31]. We contrasted these with SecureSwitch using the following metrics: trusted computing base, switching time, software compatibility, hardware dependency, and performance impact on the system. Table 2 presents the comparison results.

The BIOS code is the trusted computing base (TCB) for both SecureSwitch and Flicker [21], while Lockdown [31] and TrustVisor [22] must also ensure the security and integrity of a hypervisor when loading it from the hard disk. Both Flicker and TrustVisor have a very small switching time since they can ensure a hardware-assisted, trusted execution environment without OS sleep/wakeup. The switching delay in SecureSwitch is small and acceptable, while Lockdown requires a relatively long switching time.

In Flicker and TrustVisor, the security code must be custom-compiled or ported to run in the secure environment, while the legacy programs can run directly on both SecureSwitch and Lockdown without any changes. SecureSwitch requires the hardware devices to support ACPI, which has already been widely supported by hardware manufactures for efficient power management. All of the other solutions depend on TPM to protect the integrity of the hypervisor or to provide the Dynamic Root of Trust Measurement (DRTM) [17] feature. The memory overhead in SecureSwitch

Table 2: Comparing SecureSwitch with Other systems

	SecureSwitch	Lockdown [31]	TrustVisor [22]	Flicker [21]
Trusted Computing Base	BIOS	BIOS+Hypervisor	BIOS+Hypervisor	BIOS
Switching Time (second)	≈ 6	40	¡1	1
Software Compatibility	High	High	Low	Low
Hardware Dependency	ACPI	ACPI + TPM	TPM (DRTM)	TPM (DRTM)
Memory Overhead	High	Low	Low	Low
Computation Overhead	Low	Median	Low	High

is high due to the coarse physical isolation on the DIMMs. The memory overheads in other methods are fairly low.

In SecureSwitch, Lockdown, and Flicker, when a security code is running in the trusted environment, the applications in the untrusted environment are fully stopped. Lockdown requires 15-55% more computation overhead in the trusted environment due to the NPT pages. Flicker incurs significant computation overhead due to its frequent use of hardware support for DRTM. SecureSwitch adds no computation overhead in the trusted environment. TrustVisor can execute the applications in the untrusted environments with little overhead when the security code is running in the trusted environment; however,this requires code modification. Although possible, it would seem to be an engineering challenge to port all existing code to support this, especially for an entire commercial OS.

7 Related Work

SecureSwitch was inspired by Lampson's Red/Green separation idea [20]. The closest in terms of concept is the Lockdown [31] system that places two OSes on one machine and isolates them with help of a small hypervisor. To switch, it hibernates one OS and then wakes up another one. If implemented carefully, Lockdown can provide isolation between two OSes. 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 switching times of approximately 6 seconds. In addition, Lockdown relys on mutable shared code using a light-weight hypervisor, while SecureSwitch does not.

There is a line of research that uses hypervisors to add an extra layer of control between the OSes and the underlying hardware, including HyperSpace [6], Terra [15], Safefox [33], Tahoma system [14], Overshadow [12], and Nettop [23]. Others attempt to protect the integrity of the hypervisor [29, 13, 11, 34, 35], or to protect the kernel [30, 27, 26]. 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 [24, 38, 37]. 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.

Flicker [21] and TrustVisor [22] employ TPM to provide a small TCB and then run security-sensitive code in a trusted environment. Flicker is a pure hardware, TPM-based method, while TrustVisor adds a small hypervisor to accelerate the TPM operation. Both Flicker and TrustVisor require Dynamic Root of Trust Measurement (DRTM), while the SecureSwitch system does not. In addition, applications must be ported to support TPM-based methods. For Flicker, the code running in TPM-provided, trusted environments may not take long because the normal OS is frozen when the trusted environment is running. The SecureSwitch system is capable of running the legacy applications in the trusted OS for a long time.

8 Conclusions

The increasing number, size, and complexity of the applications running on desktop computers, coupled with their capability to operate on content and code generated by different sources, brought forward the need for context-dependent, trustworthy environments. Having such environments will enable the user to segregate different activities and lower the attack surface while maintaining system usability.

To that end, we propose a novel firmware-assisted mechanism to foster the secure management of execution environments, tailored to segregate security-sensitive applications from untrusted ones. A design tenet of our system was the ability to quickly and securely switch between operating environments without extensive code modifications or a need for specialized hardware. At the same time, we wanted to minimize the code attack surface and prevent mutable, non-BIOS code from controlling the switching process. Finally, 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. We believe that the proposed framework achieves all of these goals. 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.

References

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

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

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

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

[5] Mitre cve vulnerability database.

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

[7] Seabios, http://www.coreboot.org/seabios.

[8] Unified Extensible Firmware Interface, http://www.uefi.org/home/.

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

[10] ADVANCED MICRO DEVICES, INC. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, April 22, 2010.

[11] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), CCS '10, pp. 38–49.

[12] CHEN, X., GARFINKEL, T., LEWIS, E., SUBRAHMANYAM, P., WALDSPURGER, C., BONEH, D., DWOSKIN, J., AND PORTS, D. 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* (2008), ACM, pp. 2–13.

[13] COKER, G. Xen security modules (xsm). *Xen Summit* (2006).

[14] COX, R., HANSEN, J., GRIBBLE, S., AND LEVY, H. A safety-oriented platform for web applications. In *Security and Privacy, 2006 IEEE Symposium on* (2006), IEEE, pp. 15–364.

[15] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 193–206.

[16] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium* (2003), pp. 191–206.

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

[18] HEWLETT-PACKARD, INTEL, MICROSOFT, PHOENIX, AND TOSHIBA. ACPI, http://www.acpi.info/.

[19] KING, S., AND CHEN, P. SubVirt: implementing malware with virtual machines. In *Security and Privacy, 2006 IEEE Symposium on*, IEEE, pp. 14–pp.

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

[21] MCCUNE, J., PARNO, B., PERRIG, A., REITER, M., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (2008), ACM, pp. 315–328.

[22] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[23] MEUSHAW, R., AND SIMARD, D. Nettop-commercial technology in high assurance applications. *VMware Tech Trend Notes* (2000).

[24] NATIONAL INSTITUTE OF STANDARDS, NIST. National vulnerability database, http://nvd.nist.gov.

[25] OKAJIMA, J. R. Aufs, http://aufs.sourceforge.net/.

[26] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An architecture for secure active monitoring using virtualization. *IEEE Symposium on Security and Privacy* (2008), 233–247.

[27] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection* (2008), Springer, pp. 1–20.

[28] RUTKOWSKA, J. Beyond the CPU: Defeating hardware based RAM acquisition. *Proceedings of BlackHat DC 2007* (2007).

[29] SAILER, R., JAEGER, T., VALDEZ, E., CACERES, R., PEREZ, R., BERGER, S., GRIFFIN, J. L., AND VAN DOORN, L. Building a mac-based security architecture for the xen open-source hypervisor. *Computer Security Applications Conference, Annual 0* (2005), 276–285.

[30] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. 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* (2007), ACM, p. 350.

[31] VASUDEVAN, A., PARNO, B., QU, N., GLIGOR, V., AND PERRIG, A. Lockdown: A Safe and Practical Environment for Security Applications (CMU-CyLab-09-011). Tech. rep., 2009.

[32] VIA TECHNOLOGIES, I. VT8237R South Bridge, Revision 2.06, December 2005.

[33] WANG, J., HUANG, Y., AND GHOSH, A. SafeFox: A Safe Lightweight Virtual Browsing Environment. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on* (2010), IEEE, pp. 1–10.

[34] WANG, J., STAVROU, A., AND GHOSH, A. HyperCheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection* (2010), Springer, pp. 158–177.

[35] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010), SP '10, pp. 380–395.

[36] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 545–554.

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

[38] WOJTCZUK, R. Subverting the Xen hypervisor, 2008.