

# Expanding an Operating System's Working Space with a New Mode to Support Trust Measurement

Chenglong Wei, Wenchang Shi, Bo Qin, and Bin Liang

School of Information, Renmin University of China, Beijing, China  
chenglwei@163.com, {wenchang,bo.qin,liangb}@ruc.edu.cn

**Abstract.** Integrity measurement for Operating Systems (OS) is of practical significance. To make a measurement trustworthy, it is essential to protect the Integrity Measurement Mechanisms (IMM). However, much is still to be done to this end. This paper tries to take a step forward to shoot the target. Firstly, it puts forward the concept of trust mode, which expands the working space of an OS from two-mode, consisting of user mode and kernel mode, to tri-mode, consisting of user mode, kernel mode and trust mode. The trust mode is of the highest privilege level, in which the Core Measurement Mechanism (CMM) of an OS is executed. The CMM is in charge of measuring the IMM, which is running in kernel mode. Even if the OS kernel is compromised, the CMM would work normally without interference. Then, the paper proposes an approach to building the trust mode. It also develops a prototype to implement the trust mode by fully utilizing potentialities of modern hardware.

**Keywords:** Tri-mode Operating System, Trust Mode; Integrity Measurement Mechanism Protection, Hardware Virtualization, Code Measurement.

## 1 Introduction

Integrity measurement for Operating Systems (OS) is a hot research topic and there have been many achievements. Modern computer systems are constructed hierarchically, which means that a lower layer has control over a higher layer. An OS is always sitting at the lowest layer and has the highest privilege level. The whole system is controlled by the OS. To ensure the security of a computer system, the trust of an OS is very important. From different perspectives, much work [1-3] has been done in discussing the support of an OS in building trusted computer systems. It can be shown that an OS kernel has an indispensable role in this aspect [4-5].

The protection of an Integrity Measurement Mechanism (IMM) is involved in all topics related to integrity measurement. The measurement issue of a system can be stated briefly as that an object is to be measured by a measurement mechanism. The essential idea is that the mechanism determines whether the object meets security or trust requirements of the system. Obviously, the conclusion about the security or trustworthiness of the object is made by the mechanism. To protect the mechanism is paramount. Otherwise, the conclusion would be unconvincing.

Integrity measurement for an OS is a similar case, where the IMM corresponds to the measurement mechanism and the OS corresponds to object. To make the

measurement conclusion credible, it is essential to protect the IMM. The difficulty is that in current system architecture, an OS kernel has the highest privilege level and can gain access to all software layers. How to protect an IMM from interference of an OS kernel under the control of an attacker is a key problem.

There are many research results about protection of an IMM, but satisfactory methods are still hard to find. In work [6-7], an IMM is implanted into the OS kernel or applications. This kind of IMM will be compromised if the OS kernel is corrupted. Special hardware is utilized to isolate an IMM in some work [8-9]. Although security of the IMM is improved, compatibility and flexibility of the systems are lost. Some other efforts [10-11] implement an IMM based on existing Virtual Machine Monitors (VMM) [12]. Although risks from an OS kernel can be avoided and no special hardware is needed, complexity of these existing VMMs may induce security problems [13-14].

To protect an IMM from interference of an OS kernel, this paper puts forward the concept of trust mode. Compatible with current OS design, the working space of an OS will be expanded from two-mode, consisting of user and kernel mode, to tri-mode, with trust mode being the third mode. The user mode is the same as the original OS design. The kernel mode is almost the same except that when some specific instructions are executed, it will be trapped into the trust mode. This kind of traps is triggered by hardware and completely transparent to kernel mode code. The trust mode is dedicated to run a new mechanism, called Core Measurement Mechanism (CMM). The CMM is used to ensure that the kernel mode IMM is trustworthy. The trust mode is at the highest privilege level and catches traps from the kernel mode. After a trap being handled, the execution flow is returned to the kernel mode.

The development of software is always lagging far behind hardware in computer systems. This paper fully utilizes potentialities of modern hardware and proposes an approach to building the trust mode for an OS. The CMM is designed to run in the trust mode, which has the highest privilege level. Even if the OS kernel is compromised, the CMM in trust mode would work normally without interference.

The main contributions of the paper are as follows.

- It puts forward the concept of trust mode to expand the working space of an OS from two-mode to tri-mode. It designs a CMM to run in trust mode to ensure the trustworthiness of the IMM running in kernel mode.
- It proposes an approach to building the trust mode and develops a prototype to implement the new mode by utilizing potentialities of modern hardware.
- It introduces the private page table technology to protect the trust mode CMM by memory remapping. The CMM runs in a private memory region that cannot be accessed by the OS kernel. However, memory belonging to the IMM can be mapped for the CMM to measure the integrity of the IMM.

The rest of the paper is organized as follows. Section 2 outlines the problem we face. Section 3 introduces the design of the tri-mode architecture. Section 4 presents the way to build the trust mode for an OS. Section 5 shows how to implement the trust mode prototype. Section 6 evaluates the prototype. Section 7 discusses related work. The conclusion and future work will be given in section 8.

## 2 Problem Definition

We describe the threats we face at first, and then state the goals of our design to confront those threats.

### 2.1 Threat Model

We consider an attacker who controls everything in a computer system but the CPU, the memory controller, the main memory, a Trusted Platform Module (TPM) [15] and the buses that interconnect them.

We assume that the attacker can execute arbitrary code in kernel mode or user mode. The attacker can also access the system's DMA-capable devices, such as Fire-wire interface. Thus, the attacker may be able to read or write secrets in memory without modifying the legacy OS.

The traditional privilege structure of an OS leaves the attacker a chance to tamper with executing code of the OS kernel. Common manifestations of the attacker's abilities are rootkits and Trojans.

### 2.2 Design Goals

We have four design goals for the trust mode: (1) small code size, (2) no kernel code change, (3) no special hardware, and (4) the highest privilege level.

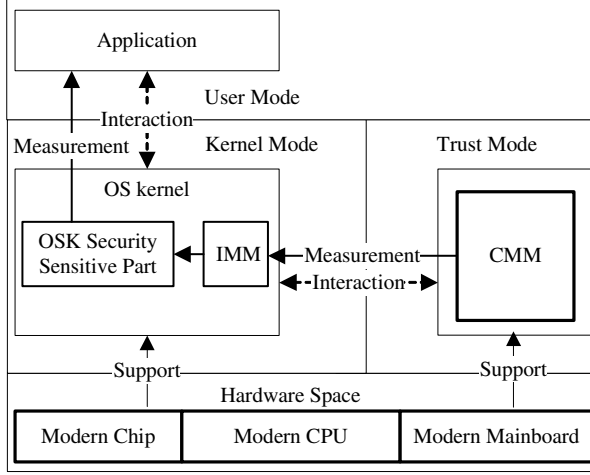
Small code size implies that we may ensure the trustworthiness of the trust mode by formal verification and manual audit. No kernel change means that commodity kernels can work directly above the trust mode without any modification. The third goal indicates that we are to fully utilize potentialities of modern commodity hardware to implement the trust mode. The last goal means that the trust mode has the highest privilege level and can work normally even if the OS kernel is compromised.

## 3 Design of the Tri-mode Architecture

Presented in this section is the conceptual design of the trust mode that is independent of any CPU or OS kernel. Expanding the working space of an OS from two-mode to tri-mode is to protect the kernel mode IMM. The CMM, a new mechanism, is added to the OS, which is to run in trust mode to ensure that the kernel mode IMM is trusted. Even if the OS kernel is compromised, the trust mode CMM can work normally.

Our architecture of a tri-mode OS is given as Fig. 1. Applications are executed in user mode and the OS kernel is executed in kernel mode. As with current design, interaction between user mode and kernel mode is implemented by system calls and interrupts.

The CMM runs in trust mode and possesses the highest privilege level. Interaction between kernel mode and trust mode is enforced by specific instructions. Execution of specific instructions in kernel mode will be trapped into trust mode. This kind of traps is triggered by hardware and completely transparent to the kernel mode. Under the



**Fig. 1.** Tri-mode OS Architecture

support of hardware, specific instructions executed in kernel mode are caught by the trust mode.

After a trap is handled, the execution flow will be returned to the kernel mode. Security sensitive parts of an OS kernel are measured by the IMM. The IMM is timely measured by the trust mode CMM. And problems are dealt with as soon as they are recognized. Even if an OS kernel is compromised, the CMM running at the highest privilege level can work normally without interference.

Our mission in this paper is to build the trust mode for an OS to protect the kernel mode IMM. To this end, the following objectives are to be reached.

- Fully utilize potentialities of modern hardware, hiding details of specific products and abstracting hardware features to support the trust mode.
- Based on hardware features, figure out a way to build the trust mode for the CMM to run separately from the OS kernel. The CMM has the ability to measure the IMM and avoid interference from the OS kernel and other software components.
- Enforce interaction between kernel- and trust-mode. Specific instructions cause kernel mode executions trapped into trust mode, in which the CMM measures the IMM. After that, the execution flow is sent back to kernel mode. Both the traps and the launch of the CMM are triggered by hardware.
- Carry out transfer of execution flows. Traps from user- into kernel-mode happen in way of system calls or interrupts. Transfer from kernel- to user mode occurs after relevant events. This is common transfer of execution flows. We focus on transfer between kernel- and trust-mode.

## 4 The Way to Build the Trust Mode

By definition, the trust mode must be of the highest privilege level. However, in the existing paradigm, the OS kernel is of that level. What we need to do is to create a new mode beyond the current situation. Modern hardware can help us to do that.

The modern CPU hardware virtualization feature is a prospective choice. The essence of hardware virtualization is to expand CPU privilege levels. Beyond the original ring 0-3, a higher privilege level is introduced. With that, the current OS paradigm with applications in ring 3 and the kernel in ring 0 may be kept. The new privilege level is in fact the highest one, which is sound for the trust mode.

Among others, both AMD and Intel have hardware virtualization supports, which are called SVM (Secure Virtual Machine) [16] and VT (Virtualization Technology) [17], respectively. We make use of Intel VT as prototype to build the trust mode.

### 4.1 Hardware Features Analysis

Intel VT includes VT-x, VT-D and VT-C that support virtualization of processor, chipset and network. We take VT-x as the fundamental to build the trust mode.

The basic idea of VT-x is to provide two kinds of processor operations, or VMX operations. One is called VMX root operation for running a VMM. The other is called VMX non-root operation for running guest software. Specific instructions executed in non-root operation state will be trapped into root operation state. After a trap is handled by root operations, code of non-root operations continues to run. VMCS (Virtual Machine Control Structure) is the core control unit of VT-x. It will be updated automatically by the CPU when transitions between non-root and root operations occur. When VMCS is initialized, configuration can be set for instructions to cause traps or not, so that guest software may be controlled by the VMM.

A set of instructions are introduced to manage VMX operations as shown in Table 1. Another 5 instructions are also introduced to manage and configure VMCS, which are VMREAD, VMWRITE, VMCLEAR, VMPTRLD and VMRTRST.

At a first glance, VT-x is designed to support processor virtualization architecture. It simplifies the design of a VMM and improves the performance of software-based virtualization. However, the essence of hardware virtualization is to expand privilege levels of CPU beyond the original ring 0-3 privilege levels.

In order for the CMM to ensure the trustworthiness of the IMM running in kernel mode, it is critical to run the CMM at a higher privilege level than the OS kernel.

**Table 1.** VMX management instructions

Instruction	Function
VMXON	Open VMX, put CPU in root operation
VMXOFF	Close VMX, CPU leaves VMX operation
VMCALL	Non-root operation guest calls VMM for service, transit to root operation
VMLAUNCH	Launch a VMM, transit to non-root operation
VMRESUME	Resume a VM, transit to non-root operation

A higher privilege level is introduced by Intel VT, which is a fundamental hardware support. With that privilege level is built the trust mode, in which the CMM is executed to avoid interference from an OS kernel. The CMM is in charge of checking the integrity of the IMM running in kernel mode.

Gaining insight into Intel VT, we found that it can be used to enhance the security of an OS in addition to supporting hardware virtualization. Creating a trust mode for an OS with the potential capabilities of modern hardware, we can execute the CMM in the highest privilege level to monitor the activities of security sensitive components running in kernel mode.

## 4.2 Building the Trust Mode

Based on the new operation state introduced by Intel VT, we can create a trust mode and give it the highest privilege level. It is a necessary mode to execute the CMM, whose responsibility is to check the kernel mode IMM. As being shown in Fig. 2, the working process of the prototype system includes building the trust mode and monitoring the kernel mode.

To build the trust mode, a trust mode module is loaded at a proper time during the launching of an OS. The first step is to enable hardware features (enter VMX operation). Then a specific structure (VMCS) is configured. There are two parameters in the structure that directly influence an execution flow. One is the return address (RIP\_G) of the kernel mode and the other is the entrance address (RIP\_H) of the trust mode. RIP\_G is set to the next instruction right after loading the trust mode module. When hardware instruction VMLAUNCH is executed, the execution flow can be transferred to kernel mode to resume the execution of the OS from RIP\_G.

When the kernel mode is being monitored, once specific instructions are executed, a trap into the trust mode will be triggered by the CPU. Then the execution flow will be transferred to the trust mode CMM at RIP\_H. As a result, specific kernel mode instructions may be monitored by the CMM. After the trap caused by specific instructions is handled, the execution flow is returned to the kernel mode.

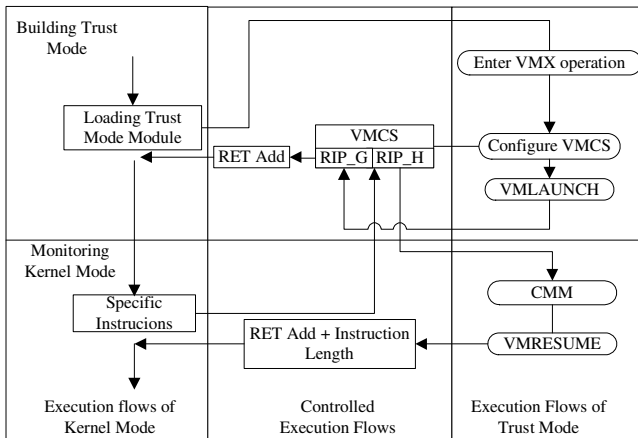


Fig. 2. Working Process of the Prototype System

## 5 Trust Mode Implementation

We developed a prototype to implement the trust mode concept in accordance with our approach. The prototype may carry out real-time monitoring of security sensitive parts of an OS kernel, including the IMM.

### 5.1 Interaction Between Kernel- and Trust-Mode

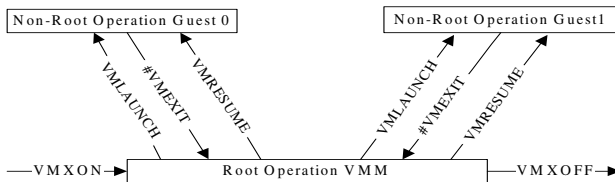
The higher privilege level introduced by Intel VT is the hardware fundamental for the creation of a trust mode. The first issue in utilizing the advanced hardware feature is to solve the problems of interaction between kernel mode and trust mode.

The advanced hardware feature to be utilized was originally designed to provide support for running a VMM. The life circle of a VMM and its guest software as well as interaction between them is illustrated in Fig. 3, which may be stated as follows.

- A program enters the VMX operation state by executing the VMXON instruction. It then runs in the root operation state, or the trust mode.
- When VMLAUNCH/VMRESUME is executed, the execution flow is transferred to a guest in non-root operation state, or the kernel mode.
- Execution of specific instructions in a guest leads to trapping into the VMM in root operation state, or trust mode.
- The VMM shuts down and leaves the VMX operation state by executing the VMXOFF instruction, which results in termination of the trust mode.

Interaction between root and non-root operation state is enforced through specific instructions. In the prototype, the first step the trust mode module takes is to enter the VMX root operation state, or trust mode, by executing VMXON. Then after VMCS having been configured, the execution flow is transferred to the non-root operation state, or kernel mode, as a result of VMLAUNCH being executed. The execution of specific kernel mode instructions will cause a trap into the trust mode. The integrity of security sensitive parts of an OS kernel, including the IMM, is measured by the trust mode CMM. After measurement, the execution flow can be turned back to the kernel mode as a result of VMRESUME.

In summary, it is by using VMX management instructions that the prototype enforces interaction between kernel mode and trust mode.



**Fig. 3.** Interaction between VMM and Guest

## 5.2 Handling Traps of Kernel Mode

Intel VT identifies two kinds of instructions that can cause traps into root operation state, or our trust mode. Some instructions cause traps unconditionally and others conditionally depending on the settings of VMCS. The way by which these instructions are handled is as follows.

- a) Handling instructions that cause traps unconditionally. Instruction CPUID and INVD are executed in trust mode. Other instructions, such as VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMRESUME, VMXOFF, VMXON, VMCALL, etc, are ignored and skipped directly.
- b) Handling instructions that cause traps conditionally. Instruction RDMSR, WRMSR and reading or writing of CR (Control Register) are executed in trust mode. Other instructions are ignored and skipped directly.

Taking instruction `mov %eax, %cr3` as an example, machine code 0F 22 D8 is handled as follows.

- a) Get instruction address, i.e. RIP, from VMCS.
- b) Get instruction length, i.e. LEN, from VMCS, which is 3.
- c) Get Exit reason from VMCS, i.e. 0x0000001C, which means that traps are caused by reading or writing CR (Control Register).
- d) Get Exit Qualification from VMCS, where information such as writing CR is available. The number of CRs is 3. The source operand is stored in EAX.
- e) Execute `mov %eax, %cr3` in trust mode.
- f) The return address in kernel mode is set to RIP + LEN, and then VMRESUME is executed to transfer control back to kernel mode.

Generally speaking, if a trapped instruction needs to be executed, it will be executed in trust mode. Otherwise, that instruction will be skipped. Finally, the original execution flow in kernel mode is restored.

## 5.3 Transfer of System Execution Flows

After the trust mode mechanism starts working, it is necessary to transfer back and forth between trust- and kernel-mode. The state of an OS, which is mainly related to general-purpose registers and flags register, must be saved before entering trust mode. The code for saving and restoring this state is shown as Fig. 4.

<pre> /* Store the state of OS */ asm volatile("pusha\n"); asm volatile("pop GuestEDI\n"); asm volatile("pop GuestESI\n"); asm volatile("pop GuestEBP\n"); asm volatile("pop GuestESP\n"); asm volatile("pop GuestEBX\n"); asm volatile("pop GuestEDX\n"); asm volatile("pop GuestECX\n"); asm volatile("pop GuestEAX\n"); asm volatile("pushf\n"); asm volatile("pop GuestEFlags\n"); </pre>	<pre> /* Restore the state of OS */ asm volatile("push GuestEFlags\n"); asm volatile("popf\n"); asm volatile("push GuestEAX\n"); asm volatile("push GuestECX\n"); asm volatile("push GuestEDX\n"); asm volatile("push GuestEBX\n"); asm volatile("push GuestESP\n"); asm volatile("push GuestEBP\n"); asm volatile("push GuestESI\n"); asm volatile("push GuestEDI\n"); asm volatile("popa\n"); </pre>
---	--

**Fig. 4.** Save and Restore the State of an OS



<pre> /* Code that sets stack frame */ push %ebp mov %esp %ebp </pre>
<pre> /* Get return address for function with stack frame */ /* and return address is stored in ret_EIP */ push 0x4(%ebp); pop ret_EIP; </pre>
<pre> /* Get return address for function without stack frame */ /* and return address is stored in ret_EIP */ pop ret_EIP; push ret_EIP; </pre>

**Fig. 5.** Store return address in ret\_EIP

The address of the instruction right after the one that invokes the trust mode module should be saved, so that the execution flow of the original OS can be restored after work in trust mode is finished. In our experiments, there exist two different cases. In one case, there is a stack frame before a function begins. In the other case, there is no stack frame. The code that gets return address is shown in Fig. 5.

After the trust mode is set up, the address for return to kernel mode, i.e. a field in VMCS, is set to ret\_EIP. Therefore, the original execution flow will turn back to executing from the instruction right after the one that invokes the trust mode module when VMLAUNCH is executed.

In the process of monitoring kernel mode, when specific instructions are executed in kernel mode, a trap into trust mode, in which the CMM is executed, occurs. The CMM gets the address (RIP) and length (LEN) of the instruction that causes the trap. After the trap is handled, the address for return to kernel mode is set to RIP + LEN. So when VMRESUME is used to restore the execution in kernel mode, the execution flow goes on from the instruction next to the one that causes the trap.

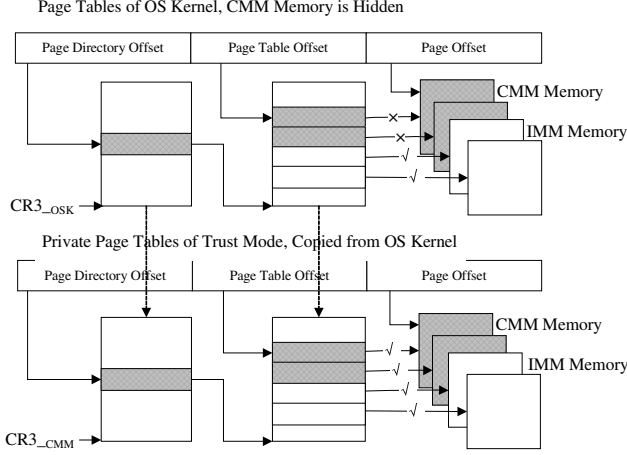
#### 5.4 Technology of Private Page Table

Memory is managed by the kernel in a legacy OS. The CMM is vulnerable if it can be accessed by the kernel. To protect CMM's memory from being tampered with by kernel mode code, a memory hiding mechanism is applied to conceal the trust mode CMM completely. On the contrary, the CMM must be able to measure the integrity of the kernel mode IMM. Therefore, memory belonging to the IMM has to be accessed by the CMM. The technology of private page table is used to fulfill these two goals.

The private page table technology, as shown in Fig. 6, patches the page table entries belonging to the CMM running in the OS kernel. It copies page tables of the CMM for private use in trust mode and then changes the page tables in kernel mode to refer to a spare physical memory. As a result, the OS kernel cannot access the memory area belonging to the CMM.

As to memory area allocated to the IMM, it is mapped for the CMM, because the integrity of the IMM must be measured by the CMM. As a result, the CMM is capable of accessing memory area of the IMM for the purpose of measurement.

Both trust mode and kernel mode have their own CR3 register, which points to the base address of their own page directory table. CR3 is stored in VMCS, it can



**Fig. 6.** Private Page Table Technology

be loaded automatically by the CPU when interaction between trust mode and kernel mode takes place.

## 6 Evaluation

The platform used for our prototype is a Thinkpad T400 with Intel Core2 Duo P8600 processor, 2GB RAM and a Trust Platform Mode (TPM). The operating system is Ubuntu 11.04. We evaluated the prototype with two metrics, i.e. effectiveness and performance, where the former stands for the ability of the trust mode CMM to measure the integrity of the kernel mode IMM.

To evaluate the effectiveness, we develop a kernel integrity measurement mechanism called OSKIM, which is an implementation of the IMM.

Here, we describe how to implement the OSKIM, introduce the protected function of the CMM, and demonstrate the performance cost of the prototype.

### 6.1 Design and Implementation of OSKIM

The OSKIM is responsible for measuring the integrity of security sensitive components of the OS kernel. In order for the OSKIM to get clear semantics of the OS kernel, it is implemented in kernel mode. The trustworthiness of the OSKIM, i.e. its integrity, is checked by the trust mode CMM at proper time.

Two design principles are set for the OSKIM, i.e. small code size and the least dependency on the OS kernel. As being implied by software engineering, small code size means fewer bugs. Less dependency on the OS kernel means that it is easier for the CMM to perform measurement. A TPM is fully used to help obtain the goals.

To implement the OSKIM, the following tasks should be undertaken.

#### 1) Extract security sensitive components from the OS kernel

Threat vectors should be taken into account. Kernel level rootkit is one of the most severe threats to an OS kernel. System call table is often tampered with by this kind of rootkits [18]. Consequently, it belongs to kernel security sensitive components.

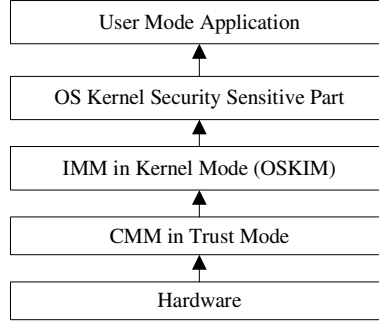
**Table 2.** Security Sensitive Components of OS Kernel

Type	Data or functions
Kernel Rootkit Target	system call table
PDIMS	do_page_fault(); do_mmap_pgoff(); arch_setup_additional_pages();
Scheduler Program	load_balance(); scheduler_tick(); try_to_wake_up(); recalc_task_prio(); schedule();

A mechanism called PDIMS [19], which we previously developed to dynamically measure the integrity of a process, is another component needs to be considered. It is necessary to build a trust chain from trust mode to user mode as shown in Fig. 7.

The scheduler program in the OS kernel is of great importance to the OS, it is another component needs to be considered.

Listed in Table 2 are all presently considered security sensitive components of the OS kernel, which should be measured by the OSKIM. An interface is provided for users to dynamically add other security sensitive components as required.

**Fig. 7.** Chain of trust from hardware to application

## 2) Determine how to measure security sensitive components

The SHA-1 cryptographic hash function is used for measurement. To reduce code size and make hash function trustworthy, a hardware TPM is employed. Three TPM commands, i.e. TPM\_SHA1Start, TPM\_SHA1Update and TPM\_SHA1Complete, are used to perform SHA-1 calculation. The first command starts an SHA-1 session. The second transfers data to the TPM. The third gets the result of the SHA-1 calculation.

## 3) Get the start address and length of a security sensitive component

In a Linux-based prototype, kernel file System.map-xxx may be used to get the start address, or start\_add, of a kernel component with the following commands.

```
cat /boot/System.map-xxx | grep security_sensitive_part
```

In the case of system call table, the length is a constant. Considering a kernel sensitive function, we can get the address next to its last instruction, or `end_add`, and take  $(\text{end\_add} - \text{start\_add})$  as its length.

#### 4) Determine the time to run the OSKIM

The OSKIM is implemented as a kernel thread and runs at a proper time. For simplicity, the OSKIM may measure the security sensitive part of the OS kernel at a regular interval. Other policies may be used as well.

To enforce the measurement time policy, we need to get trustworthy time. As we presume that the kernel may be compromised, we have to get trustworthy time from other place. We use the hardware TPM to get a trusted time with the `TPM_GetTicks` command. A trusted time interval may be produced with the following code.

```
/* a kernel thread */
while(1) {
    measure integrity of kernel security sensitive part;
    startTicks = TPM_GetTicks();
    do {
        schedule();
        endTicks = TPM_GetTicks();
    } while(endTicks - startTicks < TIMEOUT);
}
```

## 6.2 Protected Functions of the CMM

Supported by the CPU, specific instructions may be caught by the trust mode CMM. Instructions caught by our prototype CMM include `VMCLEAR`, `VMLAUNCH`, `VMPTRLD`, `VMPTRST`, `VMREAD`, `VMWRITE`, `VMRESUME`, `VMXOFF`, `VMXON`, `VMCALL`, `CPUID`, `INVD`, `RDMSR`, `WRMSR`, reading or writing `CR1`, `CR2`, `CR3`, `CR4`, etc.

Experimental test shows that the trust mode mechanism we proposed can effectively monitor sensitive kernel mode operations and measure the integrity of the kernel mode IMM. Running in the highest privilege level and supported by memory protection mechanism, the CMM is of high security and reliability.

## 6.3 Performance Test Results

We used `UnixBench` version 5.1.3 to evaluate the performance cost of our prototype. As is shown in Fig. 8, the IMM has little influence on system performance. The overall performance degrades by 1.4% after the IMM is loaded. When the IMM and the CMM are both loaded, the Pipe-based Context Switching test is most heavily influenced by the prototype mechanism, with 51.5% of performance penalty, because the test contains lots of sensitive operations such as reading and writing `CR3`. Fortunately, with the IMM and the CMM in action, the overall system performance cost is about 8.9%, as shown by the last column in Fig. 8, which is acceptable.

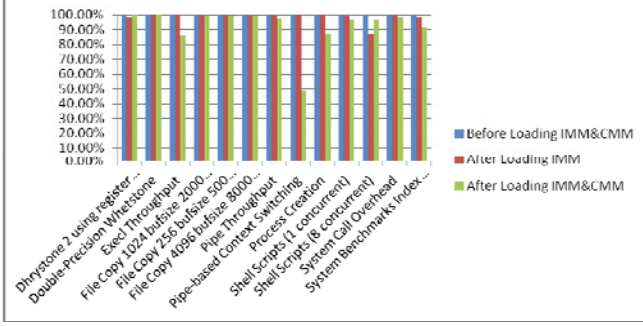


Fig. 8. UnixBench results

## 7 Related Work

To measure system integrity with hardware support is not a new idea. Dyad [20], BITS [21], AEGIS [22], IBM 4758 [8] and TPod [23] are representatives of earlier efforts that use hardware to support system integrity measurement. They focus on boot-time integrity measurement, not taking runtime integrity into consideration.

IMA [2] makes load-time integrity measurement for executable objects, which pushes a step forward from boot-time measurement. PRIMA [24] is an improvement on IMA. By introducing the CW-Lite access control model, it tries to dynamically measure system integrity with information flow being taken into account. Neither IMA nor PRIMA can provide effective way to protect measurement mechanisms.

Intel TXT [25], AMD SVM [16] and AMD TrustZone [26] are technologies that provide hardware support to system integrity measurement. Intel TXT is similar to AMD SVM. TrustZone is dedicated to embedded system platforms. These technologies may provide hardware features to be utilized by our work, which is on software mechanisms. We make use of Intel VT [17], which is a part of Intel TXT.

SecVisor [27] is a tiny hypervisor that ensures code integrity for OS kernels. It runs in the VMM privilege level that is higher than the kernel. It uses CPU-Based memory virtualization to implement isolation between monitor and OS kernel. To this point, our work is similar to SecVisor in some sense. However, SecVisor treats kernel code as a whole and does not identify security sensitive parts.

Flicker [28] is an architecture for isolating sensitive code execution with a minimal Trusted Computing Base. It utilizes hardware support of AMD SVM to create a completely isolated environment for security-sensitive code to run. It can provide attestation of the executed code to a remote party. It shows no concern on OS integrity, while OS kernel integrity monitoring is a key point of our work.

TrustVisor [29] is a special hypervisor that protects security sensitive code based on AMD SVM hardware features. It has three basic operating modes called Host mode, secure guest mode and legacy guest mode. Although our work is also involved with three modes, they are different from those of TrustVisor.

SIM [10] is a framework that monitor OS kernel based on Intel VT. It copes with a situation where a monitor and the OS kernel to be monitored coexist in the same guest VM. It implements isolation between a monitor and the OS kernel. The biggest problem of SIM is that it depends on VMM. Our work does not bother with VMM.

## 8 Conclusion and Future Work

Based on popular modern hardware features, this paper puts forward the concept of trust mode to expand the working space of an OS from two-mode, consisting of user mode and kernel mode, to tri-mode, with trust mode as the third newly added mode. It then proposes an approach to building the trust mode for an OS to monitor and protect the integrity measurement mechanism running in kernel mode. A prototype has been developed to implement the trust mode by fully utilizing potentialities of modern hardware, i.e. hardware-virtualization-enabled CPU and TPM. The effectiveness and performance of the trust mode are demonstrated with experiments.

To improve the performance of our prototype is one part of our future work. We are using multi-core technologies to enable measuring and measured entities to work in parallel so as to reduce performance cost. Since the code size of a trust mode mechanism (3500 lines of code) is small, we also want to use formal verification to prove the trustworthiness of the mechanism.

**Acknowledgments.** We would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. The work was supported in part by the National Natural Science Foundation of China under grant No. (61472429, 61070192, 91018008, 61303074, 61170240), Beijing Natural Science Foundation under grant No. 4122041, and National High-Tech Research Development Program of China under grant No. 2007AA01Z414.

## References

1. Loscocco, P.A., Wilson, P.W., Pendergrass, J.A., et al.: Linux Kernel Integrity Measurement Using Contextual Inspection. In: 2007 ACM workshop on Scalable Trusted Computing, pp. 21–29. ACM Press, New York (2007)
2. Sailer, R., Zhang, X., Jaeger, T., et al.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: 13th USENIX Security Symposium, pp. 223–238 (2004)
3. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-Reduced Integrity Measurement Architecture. In: 11th ACM Symposium on Access Control Models and Technologies, pp. 19–28. ACM Press, New York (2006)
4. Shi, W.: On Design of a Trusted Software Base with Support of TPCM. In: Chen, L., Yung, M. (eds.) INTRUST 2009. LNCS, vol. 6163, pp. 1–15. Springer, Heidelberg (2010)
5. Loscocco, P.A., Smalley, S.D., Muckelbauer, P.A., et al.: The Flawed Assumption of Security in Modern Computing Environments. In: 21st National Information Systems Security Conference, pp. 303–314 (1998)
6. Swift, M.M., Bershad, B.N., Levy, H.M.: Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems* 23(1), 77–110 (2005)
7. Venema, W.: Isolation Mechanisms for Commodity Applications and Platforms. IBM Technical Report, RC24725(W0901-048) (2009)
8. Dyer, J.G., Lindemann, M., Perez, R., et al.: Building the IBM 4758 Secure Coprocessor. *IEEE Computer* 34(10), 57–66 (2001)
9. Suh, G.E., Clarke, D., Gassend, B., et al.: AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In: 17th Annual International Conference on Supercomputing (ICS 2003), pp. 160–171. ACM Press, New York (2003)

10. Sharif, M., Lee, W., Cui, W., et al.: Secure In-VM Monitoring Using Hardware Virtualization. In: 16th ACM Conference on Computer and Communications Security (CCS 2009), pp. 477–487. ACM Press, New York (2009)
11. Azab, A.M., Ning, P., Sezer, E.C., et al.: HIMA: A Hypervisor Based Integrity Measurement Agent. In: 25th Annual Computer Security Applications Conference (ACSAC 2009), pp. 461–470. IEEE Press (2009)
12. Rosenblum, M., Garfinkel, T.: Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer* 38(5), 39–47 (2005)
13. Garfinkel, T., Rosenblum, M.: When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In: 10th USENIX Workshop on Hot Topics in Operating Systems. USENIX Press, Berkeley (2005)
14. Drepper, U.: The Cost of Virtualization. *ACM QUEUE*, 30–35 (January/February 2008)
15. TPM Main - Part 1 Design Principles - Specification Version 1.2. Trusted Computing Group (July 2007)
16. Advanced Micro Devices: AMD64 Virtualization: Secure Virtual Machine Architecture Reference Manual. AMD Publication, no.33047, rev. 3.01. (2005)
17. Neiger, G., Santoni, A., Leung, F.: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal* 10(03), 167–177 (2006)
18. Levine, J.F., Grizzard, J.B., Owen, H.L.: Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection. *IEEE Security & Privacy* 4(1), 24–32 (2006)
19. Wei, C., Song, S., Hua, W.: Operating Systems Support for Process Dynamic Integrity Measurement. In: IEEE Youth Conference on Information, Computing and Telecommunication (YC-ICT 2009), pp. 339–342. IEEE Press (2009)
20. Tygar, J.D., Yee, B.: Dyad: A System for Using Physically Secure Coprocessors. Technical Report, CMU-CS-91-140R, Carnegie Mellon University (1991)
21. Clark, P.C., Hoffman, L.J.: BITS: A Smartcard Protected Operating System. *Communications of the ACM* 37(11), 66–70, 94 (1994)
22. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A Secure and Reliable Bootstrap Architecture. In: 1997 IEEE Symposium on Security and Privacy (S&P 1997), pp. 65–71 (1997)
23. Maruyama, H., Seliger, F., Nagaratnam, N., et al.: Trusted Platform on Demand. Technical Report, RT0564, IBM (2004)
24. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-Reduced Integrity Measurement Architecture. In: 11th ACM Symposium on Access Control Models and Technologies, pp. 19–28. ACM Press, New York (2006)
25. Intel Trusted Execution Technology - Software Development Guide - Measured Launched Environment Developer's Guide. Document Number: 315168-005, Intel (2008)
26. Alves, T., Felton, D.: TrustZone: Integrated Hardware and Software Security - Enabling Trusted Computing in Embedded Systems. *Information Quarterly* 3(4), 18–24 (2004)
27. Seshadri, A., Luk, M., Qu, N., et al.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In: 21st ACM Symposium on Operating Systems Principles (SOSP 2007), pp. 335–350. ACM Press, New York (2007)
28. McCune, J.M., Parno, B., Perrig, A.: Flicker: An Execution Infrastructure for TCB Minimization. In: ACM European Conference on Computer Systems, EuroSys 2008 (2008)
29. McCune, J.M., Li, Y., Qu, N., et al.: TrustVisor: Efficient TCB Reduction and Attestation. In: 2010 IEEE Symposium on Security and Privacy (SP 2010), pp. 143–158 (2010)

Information Security Practice and Experience  
11th International Conference, ISPEC 2015, Beijing,  
China, May 5-8, 2015, Proceedings  
Lopez, J.; Wu, Y. (Eds.)  
2015, XIV, 576 p. 76 illus., Softcover  
ISBN: 978-3-319-17532-4