Secure-Turtles: Building a Secure Execution Environment for Guest VMs on Turtles System

Fei Liu, Lanfang Ren, Hongtao Bai
Institute of Security China Mobile, China

***Abstract*—We propose Secure-Turtle, a secure nested virtual system based on Turtles system, which provides a secure execution environment for the L2 guest VM. In particular, Secure-Turtles system builds a trust chain from L0 host hypervisor, L1 guest hypervisor, qemu-kvm daemon to L2 guest VM. Through this security chain, Secure-Turtles can protect L2 guest VM against attacks form the L1 user mode, even when the attacker has the root privilege of the L1 guest hypervisor.**

Our goal is to make Secure-Turtles possible to rule out known class of vulnerabilities from the L1 user. We proposed four general requirements for Secure-Turtles to satisfy to achieve our goal and list sixteen basic properties for the Secure-Turtles system to achieve. With these properties, the proposed four requirements can be guaranteed. We rely on the memory virtualization to build Secure-Turtles and implement a prototype based on Turtles. We evaluate its prototype using two metrics: security and performance. The security evaluation result shows that Secure-Turtles can protect L2 guest VM from attacks from the L1 user mode. The performance result shows that Secure-Turtles introduces little performance overhead to the L2 guest VM compared with the Turtles system.

***Index Terms*—Security, Nested virtualization**

I. Introduction

System virtualization has been a standard technique in many commercial usage scenarios such as server consolidation, multi-tenant cloud and virtual appliances. Commodity hypervisors or operating systems increasingly make use of the virtualization capabilities to run virtual machines (VM). Window 7 supports a backward compatible Windows XP mode by running the XP operating system as a VM. Linux runs VMs using its build-in KVM [1] hypervisor. Xen hypervisor [2] runs its VMs with the help of its privileged VM.

However, these systems are still vulnerable to security attacks. The trust base of these commodity systems (e.g., the host operating system of Window 7 and KVM or the privileged VM in Xen) are large and complex, and consequently, frequently prone to compromise. When it is compromised, guest VMs running on it are under great danger. As the host hypervisor has the full control of the underlying hardware resources and provide virtual resources for the guest VM, it can easily obtain the secure sensitive data from the guest VMs lunched by it. For example, it can get the disk I/O data from a guest VM through its virtual disk device emulator. Furthermore, guest VMs are unaware of the existing of the underlying hypervisor. This makes them more vulnerable to the attacks on their hypervisor as they may still processing sensitive data when the hypervisor has been compromised.

In this paper, we make use of nested virtualization to enhance the security of such virtual environments based on the Turtles project [3], which proposes a nested virtualization architecture based on KVM through multi-dimensional paging for MMU virtualization and multi-level device assignment for I/O virtualization. It provides a flexible virtual functionality to host virtualization capable operating system or hypervisor. In this architecture, the bare-metal hypervisor running on the hardware, which is called L0, emulates a virtual hardware environment. The guest hypervisor running on L1 will execute on such virtual environment. The guest VMs of the guest hypervisor will run on L2.

We proposed Secure-Turtles, a secure nested virtual system based on Turtles system, which can run L2 guest VMs securely. In this system, we build a trust chain from the L0 host hypervisor, the L1 guest hypervisor kernel, qemu-kvm daemon to the L2 guest VM, we figure out four requirements to satisfy inside the Secure-Turtles system: 1) lifetime kernel code integrity of the L1 guest hypervisor should be provided; 2) the code and data integrity of qemu-kvm daemons should be provided; 3) the data integrity of the L2 guest VM should be provided; and 4) the guest VM should be aware of any violation of the above three requirements. We prove these four requirements are necessary and sufficient to build a secure environment for an L2 guest VM. In order to realize these four requirements, we list sixteen properties for the Secure-Turtles system to achieve and prove that with these properties the above requirements can be guaranteed.

Here, we assume that the L0 host hypervisor is insulated from the outside world to avoid being attacked. Thus it can be trusted in the Secure-Turtles system. Based on the trusted L0 host hypervisor, we build a trusted L1 hypervisor's kernel by ensuring the kernel code integrity of the L1 guest hypervisor, who hosts the L2 guest VMs, and let L1 hypervisor to ensure its own kernel data integrity. Then based on the trusted L1 hypervisor kernel, we build a secure execution environment for the L2 guest VMs by ensuring the code and data integrity of the qemu-kvm daemon who lunches L2 guest VMs as well as the memory and disk data integrity of L2 guest VM. When an attack hijacks the user mode of the L1 hypervisor through some vulnerability of some user processes such as /bin utilities and acquires the root privilege, Secure-Turtles can still prevent it from attacking the L2 guest VMs.

In this paper, we use Trusted Platform Module (TPM) [4] to approve the booting process of L1 guest hypervisor, the lunching process of qemu-kvm VM and the booting process of L2 guest VM. (The TPM enables remote attestation by digitally signing cryptographic hashes of specific software components. The attestation will affirm that the software is genuine and correct.) We use $W \oplus X$ protection to provide code integrity and monitor the L1 kernel entries and qemu-kvm entries to enhance code integrity. We also disable all memory access interfaces that provide user processes the ability to access arbitrary memory in L1 to provide memory data integrity and use disk access protection to provide disk data integrity for L2 guest VMs. And we let L0 and L1 to inform L2 guest VMs the potential threats.

We have built a prototype of Secure-Turtles based on the Turtles system and evaluate its prototype with two metrics: security and performance on an Intel machine with 16GByte memory. The security evaluation result shows that Secure-Turtles can protect L2 guest VM against attacks from the L1 user mode, which try to either modify its execution environment such as L1 hypervisor or the qemu-kvm daemon or modify its data directly. The performance evaluation on an L2 guest VM with 1 virtual CPU (VCPU) and 2 GByte memory using the SPECINT benchmark suite shows that Secure-Turtles introduces nearly no performance overhead to the L2 guest VM compared with the Turtles system.

The rest of this paper is organized as follows: The next section describes the background of Turtles system and TPM devices. Section III discusses our assumptions and the treat mode. Section IV lists the four requirements to build the trust chain inside Secure-Turtles and proves that with these requirements, it can provide a secure environment for L2 guest VMs. Section V provides the design of Secure-Turtles system and Section VI shows the evaluation result of the Secure-Turtles prototype. Section VII concludes our work.

II. BACKGROUND

A. Nested Virtualization

Turtles project [3] investigates the design and implementation of nested virtualization to support multi-level virtualization in KVM [1] on x86 hardware with virtualization extensions, VMX [5] and SVM [6]. Figure 1 gives an overall architecture of nested virtualization. The host hypervisor runs on bare-metal hardware emulates VMX/SVM for L1 and L2 with the highest privilege level. Since L0 provides a full emulation of the VMX/SVM hardware any time there is a trap on VMX/SVM instruction, the guest hypervisor on L1 will not know it is not running on the hardware and can run guest VMs on L2 just as it has the VMX/SVM support.

As the x86 hardware has only a single hypervisor mode, a trap at any level is handled by the most

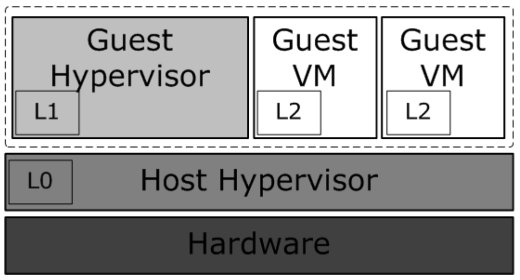

Figure1: Architecture of Nested Virtualization.

privileged hypervisor on L0 first. Figure 2 shows how traps are handled in Turtles system. When a trap occurs on L2, it will first be trapped by L0. Here L0 will decide whether the trap can be handled by L0 directly or should be sent to L1 to handle it. If it should be sent to L1, the host hypervisor will inject a virtual trap event on L1. Otherwise it will return the execution back to L2 directly. Further when a trap occurs on L1, the execution will also return to L0 and then back.

When guest hypervisor on L1 wishes to run a VM, it will launch it via VMX/SVM commands. This will cause a trap to L0, as it does not have the privilege to issue these commands. At this time, the host hypervisor on L0 can verify the genuine of the execution environment of the guest VM including the guest hypervisor on L1 as well as its booting process.

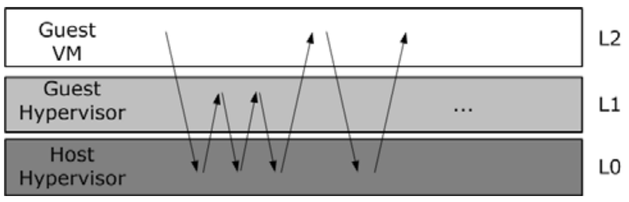

Figure2: Architecture of Nested Virtualization.

B. The Trusted Platform Module

The TPM is a security specification defined by the trusted computing group [4]. It implementation is usually available as a chip attached to a platform’s motherboard. S. Berger et.al has proposed a software based TPM implementation [7] to make TPM functions available to VMs. The TPM provides cryptographic operations including asymmetric key generation, encryption/decryption digital signing and migration of keys and random number generation and hashing. It also provides a small amount of storage to save secure sensitive information such as cryptographic keys. It enables remote attestation of specific software components to affirm that the target software is genuine and correct.

One important operation of TPM is Platform Configuration Register (PCR) extension operation. The PCRs are initialized at power up and extended cryptographically during the execution using the following equation:

$$PCR_{N+1} = Extend(PCR_N, value) = SHA1(PCR_N \ || \ value) \quad (1)$$

The N+1th extend value of PCR is calculated through Extend (PCR_N, value). SHA1 refers to the secure hash algorithm and the || operation represents a concatenation of two byte arrays, PCR_N and value.

The PCR extensions can be used during the platform boot to verify the whole boot process. The hash values of byte arrays representing code or configuration data are calculated or measured. And PCR values are extended with these values. The final PCR value represents the accumulation of a unique sequence of measurement. Such sequential list of measurements can be used to decide whether the system can be trusted.

In Secure-Turtles, we depend on the TPM to verity the genuine of the execution environment of the guest VM.

III. Assumptions And Threat Model

In this section, we state our assumptions of the software and hardware environment and describe out threat model.

A. Assumptions

We assume the CPU on which Secure-Turtles runs provides support for hardware-assisted virtualization support like Intel's VMX and AMD's SVM. We also assume the L1 kernel protected by Secure-Turtles does not use self-modifying code. We also assume that the L0 host hypervisor of Secure-Turtles are insulated from the outside world to avoid being attacked.

B. Threat Model

We consider an attacker who attacks the L1 guest hypervisor and compromises its kernel that the attacker can take control of the L1 guest hypervisor. Example attacks include modification of memory contents, injection of malicious code, and malicious DMA writes to memory using peripherals. After hijacking the L1 kernel, the attacker might easily take control of all L2 guest VMs lunched by the L1 through KVM module inside the kernel.

Further, even when the attacker can't hijack the L1 kernel, but hijacking the L1 user space, it can still take control of the L2 guest VM by lunching the VM with a modified qemu-kvm daemon with injected malicious code or uses ptrace interface to control the qemu-kvm daemon.

Here, we do not consider an attacker that attacks the L2 guest VM directly by compromising it or using denial of service attacks. Several works [8, 9] have been proposed to handle such threats in the cloud environments.

IV. Design Requirements of Secure-Turtles

In this section, we discuss the design requirements of building the Secure-Turtles system. We start by describing the design goal of Secure-Turtles followed by a serial of properties that need to be achieved in order to guarantee the goal.

A. Design Goal of Secure-Turtles

The general goal of Secure-Turtles is to provide a secure execution environment for L2 guest VMs even when the L1 guest hypervisor is compromised by an attacker. To achieve this, following requirements should be achieved by Secure-Turtles:

- **Requirement1:** The Secure-Turtles system should provide lifetime kernel code integrity for the L1 guest hypervisor.
- **Requirement2:** The Secure-Turtles system should provide the code and data integrity of the corresponding qemu-kvm daemon of a guest VM during its lifetime.
- **Requirement3:** The data integrity of the L2 guest VM should be guaranteed.
- **Requirement4:** The guest VM should be aware of any violation of the above three requirements.

Theorem1: With Requirement1 to Requirement4, Secure-Turtles can provide a secure execution environment for L2 guest VM.

Proof: With Requirement1 and Requirement2, the code integrity of the L1 kernel and qemu-kvm daemon is guaranteed so that no attacker can alter the virtual environment emulation functionality. Further the data integrity of qemu-kvm daemon is guaranteed so that no attacker can alter the execution flow of the qemu-kvm daemon by changing the stack or heap of qemu-kvm daemon. Thus, the virtual environment emulated by the L1 kernel and the qemu-kvm daemon cannot be altered by the attacker. With Requirement3, as the data integrity of the guest VM is also guaranteed by Secure-Turtles, no attacker can eavesdrop or modify the guest VM's data. Finally, with requirement4, as Secure-Turtles will notify the L2 guest VM the danger of any violation of the above three requirements which will threaten the security of L2.

B. Required Properties for Secure L2 Guest Virtual Machine Execution

We start designing Secure-Turtles by converting the requirements into properties. Our first requirement is to provide kernel code integrity for the L1 guest hypervisor. Thus, Secure-Turtles must provide the following six properties for L1 guest hypervisor:

- **Property1**: The booting process of L1 guest hypervisor should be approved.
- **Property2**: The whole kernel code of L1 must be approved to be unmodified.
- **Property3**: Memory containing the kernel code should not be modified or extend by any time.
- **Property4**: Every entry into kernel mode should set the CPU's Instruction Pointer (IP) to an instruction within the kernel code approved by Secure-Turtles in Property1.
- **Property5**: After entering kernel mode, the IP should continue to point to the approved kernel code until the CPU exits kernel mode.
- **Property6**: Every exit from kernel mode should set the execution privilege into user mode.

Theorem2: With Property1 to Property6, Requirement1 can be achieved.

Proof: Property1 guarantees the L1 guest hypervisor boots correctly and the kernel code loaded during the boot process are secure. Property2 together with Property3 guarantee that the loaded kernel code will neither be extended nor be modified. Property4 to Property6 guarantee that when the control flow enters the kernel mode, the CPU can only execute the kernel code that is approved according to Property2 and Property3 as the CPU's IP can only point to the approved kernel code when the L1 is running in the kernel mode. Thus, with Property1 to Property6, Secure-Turtles can provide lifetime kernel code integrity for L1 guest hypervisor.

The second requirement is to provide code integrity of the qemu-kvm daemon of a guest VM. Thus, Secure-Turtles must provide the following six properties:

- **Property7**: The lunching process of qemu-kvm should be approved.
- **Property8**: The qemu-kvm lunching the L2 guest VM must be approved to be unmodified.
- **Property9**: Memory containing the qemu-kvm code should not be modified by any time.
- **Property10**: Every entry into the qemu-kvm daemon should set the CPU's Instruction Pointer (IP) to an instruction within the qemu-kvm code.
- **Property11**: During its execution, the IP should continue to point at approved qemu-kvm code.
- **Property12**: Memory containing the qemu-kvm data should not be accessed by other processes.

Theorem3: With Property7 to Property12, Requirement2 can be achieved.

Proof: Property7 guarantees the qemu-kvm daemon is created correctly and its code is safe. Property8 together with Property9 make sure that the loaded code of the qemu-kvm daemon is not changed at its setup time (Property8) and will not be changed during the daemon's lifetime (Property9). Propertl0 guarantees that whenever qemu-kvm daemon is scheduled, it will start at a valid instruction address within its code base. Further, Property11 guarantees that the daemon can only execute the code approved according to Property8 and Property9. Thus, Property8 to Property11 provide code integrity of the qemu-kvm daemon. While Property12 forbids any access to the memory used by the qemu-kvm daemon from other processes, which provides data integrity of the daemon. Thus, Requirement2 can be achieved with Property7 to Property12.

The third requirement is to guarantee the data integrity of the guest VM. Thus, Secure-Turtles must provide the following two properties.

- **Property13**: Memory containing the guest VM's memory should not be accessed by other processes except the qemu-kvm daemon in L1.
- **Property14**: The disk image of the guest VM should not be accessed by other processes except the qemu-kvm daemon in L1.

Theorem4: With Property13 to Property 14, Requirement3 can be achieved.

Proof: Property13 guarantees the guest VM's memory is not accessible to any processes except the qemu-kvm daemon in L1 so that even the attacker can control the L1, it cannot access the L2 guest VM's memory to alter its state or steal secure sensitive data. Property14 guarantees that the disk data is only accessible to the qemu-kvm daemon that the attacker cannot access it through the disk file (e.g. mounting the disk image). With Property13 and Property14, we can provide both memory data integrity and disk data integrity for the L2 guest VM.

The forth requirement is let the guest VM be aware of any violation of the first two requirements. Thus, Secure-Turtles must provide the following two properties:

- **Property15**: The booting process of L2 guest VM should be approved.
- **Property16**: Any violation of any properties listed above should be informed to L2.

Theorem5: With Property13 to 14, Requirement4 can be achieved.

Proof: Property15 guarantees the L2 guest VM boots correctly according to the user's requirements. If it is not boot correctly, the booting process may be hijacked that the guest VM may not be secure. Property16 ensures that the L2 guest VM will know whether it is running on a secure environment. If any property from Property1 to Property14 is violated, the running environment of the L2 guest VM is not secure ever. The guest VM must stop processing any secure sensitive data. With Property15 and Property16, the guest VM will be securely running and be aware of any kinds of secure threats from L1 guest hypervisor.

V. Design of Secure-Turtles

In this section, we show how each property is achieved based on Turtles system in Secure-Turtles.

A. Approving BootingProcess through TPM

Every time an L1 guest hypervisor is booted, a qemu-kvm daemon is created or an L2 guest VM is lunched, the booting/setup process should be approved. Secure-Turtles use the TPM to verify the correctness of these processes.

Here L0 host hypervisor will approve the booting process of L1 guest hypervisor using the hardware TPM device [4] through a chain of measurements using the "equation (1)". In order to satisfy Property2, the kernel code loading process is also added into the measurement chain of the boot process so that only predefined code is loaded into memory. Here, the predefined code is decided at the time the L1 hypervisor is installed, when it is disconnected with the outside world. Thus, it is insulated against the attackers from the Internet and is safe. After adding the predefined code into the measurement chain, the whole kernel code of L1 after the boot is approved to be secure. Here the L0 will make an attestation of the approved booting process using the equation:

$$Attes_{L1} = (PCR_{final-L1} == Key_{L1} \ ?\ yes : no) . \quad (2)$$

$PCR_{final-L1}$ refers to the accumulation of the sequence of measurements of the L1 booting process. The Key_{L1} is a predefined hash value representing that the L1 is booted securely, the result of the equation shows whether L1 boots securely or not.

With the approved booting process of L1 guest hypervisor, both Property1 and Property2 are guaranteed.

When starting a L2 guest VM, the L1 guest hypervisor will approve the lunching process of the qemu-kvm daemon as well as the booting process of the L2 guest VM. The L1 guest hypervisor uses a virtual TPM device, proposed by S. Berger et.al. [7], to measure the lunching process of qemu-kvm daemon and the booting process of the L2 guest VM. In order to satisfy Property8, the code loading process is also added into the measurement chain of the lunching process of qemu-kvm daemon. Here, we use "equation 3" to generate the attestation of the approved lunching process of the qemu-kvm daemon and "equation 4" to generate the attestation of the approved booting process of L2 guest VM.

$$Attes_{qemu-kvm} = (PCR_{final-qemu} == Key_{qemu} \ ? \ yes : no). \quad (3)$$

$$Attes_{L2} = (PCR_{final-L2} == Key_{L2} \ ? \ yes : no). \quad (4)$$

$PCR_{final-qemu}$ refers to the accumulation of the sequence of measurements of the qemu-kvm lunching process. The Key_{qemu} is a predefined hash value representing that the qemu-kvm daemon is lunched securely. $PCR_{final-L2}$ refers to the accumulation of the sequence of measurements of the L2 booting process. The Key_{L2} is a predefined hash value representing that the L2 guest VM is booted securely.

With the approved lunching process of the qemu-kvm daemon as well as the L2 guest VM, Property7, Property8 and Property15 are guaranteed.

B. W ⊕X Protection

On each entry L1 falls into kernel mode, Secure-Turtles sets execution permissions in the protection page table so that only approved L1 kernel code can execute. For the kernel code, this can be simply achieved by setting the execution permission of pages containing kernel code and clean such permission for any other kernel pages. However, as the user code is also executable in kernel, Secure-Turtles also need to clean the execution permission of all user pages. Figure 3 shows the memory protections in the protection page tables for user and kernel mode. It can be seen that, in kernel mode, only the pages of kernel code will be executable but not writable. This type of memory protection is referred to as W ⊕ X protection.

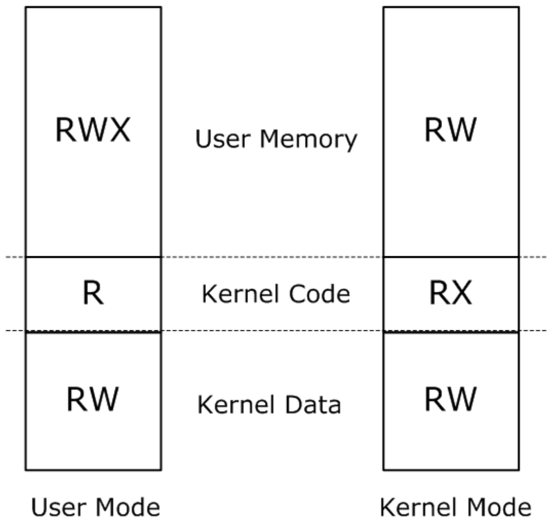

Figure3: Memory protections in the protection page table for user mode and kernel mode

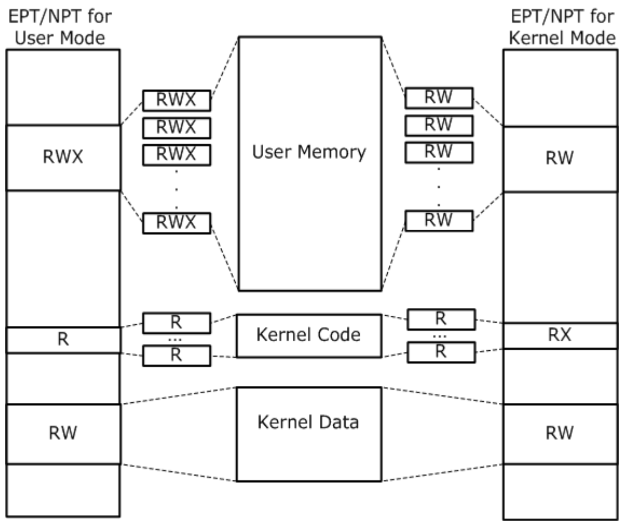

Figure4: How EPT/NPT table is built for L1 guest hypervisor in guest model and kernel mode.

In order to provide the separate page tables for user mode and kernel mode, Secure-Turtles rely on the hardware virtualization extensions as Intel VMX [10] and AMD SVM [11], which provide nested page table support. The extend page table (the EPT) for Intel or the nested page table (the NPT) for AMD is used to translate guest physical address to system physical address. In Turtles, L0 use EPT/NPT to translate L1's guest physical address to the system physical address. Thus, to realize W ⊕X protection for L1 kernel code in Secure-Turtles, L0 host hypervisor will maintains two EPT/NPT tables for the L1 guest hypervisor, one for user mode and one for kernel mode. Figure 4 shows how these two tables are built. Both EPT/NPT tables for user mode and kernel mode use the same mapping for the kernel data (RW). The EPT/NPT table takes R page table entries to map the kernel code for the user mode and it takes RX page table entries for the kernel mode. Finally, the EPT/NPT table takes RWX page table entries to map the user memory for user mode and takes RW page table entries for kernel mode. With these two tables, when L1 falls into kernel mode, only the kernel code part is executable. When the IP point to an address outside the kernel code range, an exception will be caught by L0 indicating the property violation of Property5.

To achieve Property2, the mapping of the kernel code must be established following the kernel code loading process of the L1 guest hypervisor. In Secure-Turtles, L0 will forbid further modification of this mapping area inside the EPT/NPT table after the code loading process.

For example, any kernel module loaded after the establishment of this mapping will not have the execution permission.

With W ⊕ X protection for the kernel code as well as the restricted kernel code mapping mechanism Property5 is guaranteed. Other than that Property3 is partially guaranteed, as it is impossible to modify the kernel code in kernel mode.

The same memory protection methodology is also used to protect the qemu-kvm code from modification. Figure5 shows the memory protection layout in the protection page table for qemu-kvm daemon. It can be seen that only the qemu-kvm code is set to executable. Any data area such as stack and heap are none-executable. With this protection page table, when qemu-kvm daemon is running, only the code part is executable. When the IP point to an address outside the code range, an exception will be caught by L1 indicating the property violation of Property11.

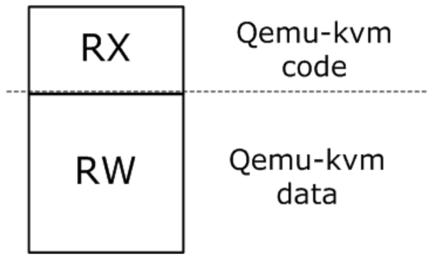

Figure5: Memory protections in the protection page table for qemu-kvm daemon

To achieve Property9, the mapping of the qemu-kvm daemon code must be established following the qemu-kvm lunching process. In Secure-Turtles, L1 will forbid further modification of this mapping area after the code is loaded.

With W ⊕ X protection for the qemu-kvm code as well as the restricted code mapping mechanism and Property11 are guaranteed. Other than that, Property9 is partially guaranteed, as it is impossible to modify the qemu-kvm code in its daemon's context.

C. Disabing Physical Memory Mapping from Root

Security sensitive data should not be accessible to user processes even when the process has the root privilege so that none process in L1 can read or modify the kernel code and only the qemu-kvm daemon process can access and modify its own code and data as well as the L2 guest VM's memory (guest VM takes the same context with the qemu-kvm daemon in KVM). To do so, L1 hypervisor disables the /dev/mem and /dev/kmem device. The /dev/mem and /dev/kmem are special files that provide access to pseudo device drivers which allows read and write accesses to system memory. High privilege users such as root can use these two devices to read and write any memory of the L1 hypervisor. By disabling these two devices, L1 hypervisor closes all interfaces to access arbitrary memory of L1 in the user mode. Thus, together with W ⊕ X protection Property3 is guaranteed.

Although, user processes can only access memory belonging to its context. Attackers can use another way to take control of qemu-kvm daemon through the ptrace interface. The ptrace system call provides a way for a process (tracer) to observe or take control of another process (tracee) and examine and change the tracee's memory and registers. High privilege users such as root can use this interface to take control of the qemu-kvm daemon and alter its memory and control flow. After disabling this interface in the L1 hypervisor, qemu-kvm daemons are free from being observed or controlled. Thus, together with W ⊕ X protection, Property9, Property12 and Property13 are guaranteed.

D. Managing L1 kernel Entries and Exits and Qemu-kvm Entries

Secure-Turtles ensures that kernel mode entries and exits satisfy Property4 and Property6. Also it will ensure that the qemu-kvm daemon entries satisfy Property10.

Kernel mode entries: Secure-Turtles needs to ensure that all control transfers from user mode into kernel mode in L1 guest hypervisor will set the IP to an address within the kernel code approved at the L1 booting process.

In the x86 architecture, control transfer to kernel mode is only allowed through the entry points designated by kernel. The kernel informs the CPU of the permitted entry points by writing the address of these points in CPU registers and data structures. All entry points exist in the global description table (GDT), local description table (LDT), interrupt description table (IDT) and some model specific registers (MSRs). Thus, Secure-Turtles should protect the entry points inside this tables or registers. Fortunately, all these entry points can only be modified by L1 within the kernel mode so that they will point to the approved kernel code after L1 hypervisor has booted according to Property1 and Property2. Further, according to Property3, the attacker cannot change these entry points even with the root privilege. Thus with Property1, Property2 and Property3, Property4 will be guaranteed.

Further, in order to enforce the W ⊕ X protection, L0 need to switch the EPT/NPT table for the L1 from user mode to kernel mode. As shown in Figure3, when control transfers from user mode to kernel mode, the EPT/NPT table used is the one for the user mode, thus when the kernel code is executed, an EPT/NPT violation will be caught by L0 as the kernel code pages are none-executable. At this point L0 will switch EPT/NPT table into the one for the kernel mode.

Kernel mode exits: Secure-Turtles also needs to ensure that every exit from kernel mode will set the execution privilege into user mode. As shown in Figure3, the user memory mapping in the kernel mode EPT/NPT table is none executable, when there is an exit from kernel mode to user mode, the execution of the first user instruction will cause an EPT/NTP violation which will be caught by L0. At this point, L0 will switch EPT/NPT table into the one for the user mode. Also L0 will set the CPL field of the VMCB to 3, thus, ensuring that when user process resumes execution has switch to user mode. Here Property6 will be guaranteed.

Qemu-kvm entries: Secure-Turtles needs to ensure that when control transfers to qemu-kvm daemon in L1, it will set the IP to an address within the qemu-kvm code approved at qemu-kvm daemon lunch process.

At every time, there is a context switch to the qemu-kvm daemon, the page table shown in Figure 5 is loaded and the CPL will be set to 3 according to Property6. Thus, if the entry points to the IP outside the qemu-kvm code, a page fault will be caught by L1, which means any violation of Property10 will be detected directly by Secure-Turtles.

After managing L1 kernel entries and exits as well as qemu-kvm entries, Property4, Property6 and Property10 are guaranteed.

E. *Protecting Disk Accesses*

L2 disk images exist as local files in L1 hypervisor. Processes with root privilege can access these files directly. To protect L2 images from malicious accesses from arbitrary processes, we propose two solutions: 1) Store encrypted L2 disk images in L1. L1 needs to decrypt the disk blocks when they are loaded into the L2 guest VM's memory and encrypt the data when it is written into the disk; and 2) Take access control over L2 disk images. Only the approved qemu-kvm daemon has the right to access it. As disk encryption and decryption waste lots of computing resources, we choose the second solution in Secure-Turtles.

The opening an L2 disk image is only allowed within the context of an approved qemu-kvm daemon. Before opening the image, L1 will first check the $Attes_{qemu-kvm}$ of the requesting qemu-kvm daemon to ensure that it is lunched correctly. Only the approved qemu-kvm daemon has the right to access the L2 disk images. However, as no process other than the qemu-kvm daemons can access the images, who create these images? The answer is L0. The disk images are copy into L1 guest hypervisor by the L0 host hypervisor who will also mark these files as L2 images that L1 should apply special access control over them.

With the disk access control mechanism, Property14 is guaranteed.

F. *Informing Property Violation*

In Secure-Turtles, L2 guest VM will be aware of potential security threats. After booting the L2 guest VM, the user of guest VM can verify its execution environment by verifying the value of $Attes_{L1}$, $Attes_{qemu-kvm}$ and $Attes_{L2}$. Only when all these three values are true the execution environment is secure.

During L2 guest VM's lifetime, any violation of properties from Property1 to Property15 will be propagated to the L2 guest VM and trigger a signal to all user processes. Any user process requiring a secure execution environment should register a signal handler to handle this signal. Here, Property16 is guaranteed.

VI. Evaluation

In this section, we evaluate our Secure-Turtles prototype using two metrics: security and performance.

A. *Security Measurements*

We evaluate Secure-Turtles' security on an Intel machine with 4 cores and 16 GByte memory. The L0 host hypervisor is Debian 6.0 with kernel version 3.10.2. The L1 host hypervisor is booted with 4 VCPU and 8 GByte memory. The L1 host hypervisor is Debian 6.0 with kernel version 3.8.4. The L2 guest VM is booted with 1 VCPU and 2 GByte memory. In this evaluation, we assume the attacker has taken control over the root privilege of L1 guest hypervisor, and we takes following three tests:

- The L1 guest hypervisor's kernel image is changed by the attacker.
- The attacker tries to control qemu-kvm daemon through ptrace.
- The attacker tries to modify the L2 disk image.

Test One: After modifying the L1 guest hypervisor's kernel image, the booting process of L1 guest hypervisor generates a wrong attestation as the hash of the loaded code altered with the predefined one. Here, the $Attes_{L1}$ is set to false. Under this condition, L2 does not execute after booting, as it finds out that its execution environment is insecure as shown in section V.F.

Test Two: The attacker tries to attack the L2 guest VM through the qemu-kvm daemon. When it tries to use ptrace to take control of the daemon, it finds that this interface is closed by the L1 hypervisor. Thus, the attacker cannot take control of the daemon.

Test Three: The attacker tries to access the L2 disk image by mounting it into a directory (e.g. /mnt) and access it. However, it fails as Secure-Turtles forbids any process other than the qemu-kvm daemon to access the L2 disk images.

It can be seen that Secure-Turtles can successfully defense attacks from the L1 user mode even when the attacker has the root privilege.

B. *Performance Measurements*

We evaluate Secure-Turtles' performance on the same Intel machine as we evaluate its security. We use the same software environment and L1 guest hypervisor and L2 guest VM configurations.

We use five applications (Bzip2, GCC, MCF, Hummer and H264ref) from SPECINT benchmark test suite. Bzip2, GCC and MCF have a modest memory footprint while Hummer and H264ref have a small memory footprint. Each application is run with three times and we list the result of each run.

Figure 6 shows the performance of Bzip2, GCC and MCF on Turtles and Secure-Turtles. It can be seen that for each run, Secure-Turtles incurs small performance overhead over Turtles system. The average performance overhead for these three applications is 2.67%. Figure 7 shows the performance of Hummer and H264ref on Turtles and Secure-Turtles. It can be seen that for each run, Secure-Turtles incurs little performance overhead over Turtles system. The average performance overhead for these two applications is 0.85%. The performance overhead here is mainly due to the additional cost in the page fault handling routine that L1 has to build the EPT for the L2 guest VM during which the W ⊕ X protection mechanism is activated to protect the L1 hypervisor and qemu-kvm daemon. As Hummer and H264ref have much less memory footprint, the performance overhead of them is less than Bzip2, GCC and MCF.

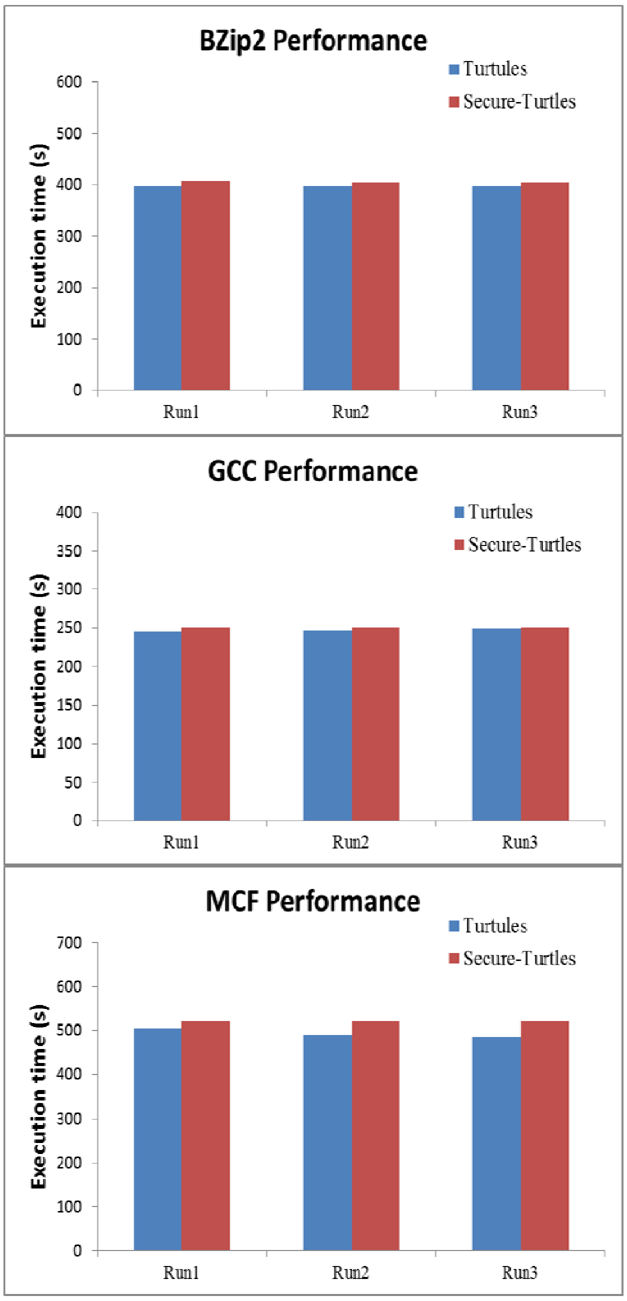

Figure6: Performance of Bzip2, GCC and MCF on Turtles and Secure-Turtles

VII. Conclusion

The trust bases of most hypervisors (KVM or Xen) are large and complex, and consequently, frequently prone to compromised. When the hypervisor is compromised, guest VMs running on it are under great danger. In this paper, we proposed Secure-Turtles, a secure nested virtual system based on Turtles system, which can run L2 guest VMs securely. In Secure-Turtles, even when the user space of the L1 guest hypervisor is compromised by an attacker, L2 guest VMs can still execute securely. We have built a prototype of Secure-Turtles based on the Turtles system and evaluate its prototype with two metrics: security and performance on an Intel machine.

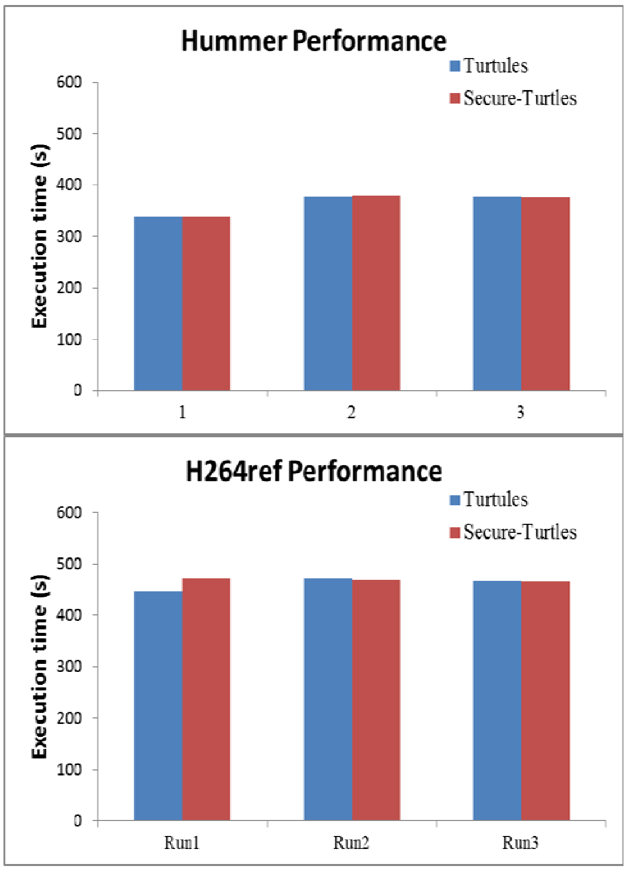

Figure7: Performance of Hummer and H264ref on Turtles and Secure-Turtles

The evaluation result shows that Secure-Turtles can protect L2 guest VM against attacks from the L1 user mode and it introduces nearly no performance overhead to the L2 guest VM compared with the Turtles system.

Acknowledgment

This work was supported in part by a grant from National Science Special Program "New wireless mobile communication network" named "Study of security of mobile network under cloud computing". (No.2012ZX03002002)

References

[1] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in Proceedings of the Linux Symposium, vol. 1,2007, pp. 225–230.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in Proc. SOSP. ACM, 2003, pp. 164–177.

[3] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. HarEl, A. Gordon,A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation

of nested virtualization," in Proceedings of the 9th USENIX conference on Operating systems design and implementation. USENIX Association, 2010, pp. 1–6

[4] Trusted Computing Group. http://www.trustedcomputinggroup.org.

[5] Uhlig, Rich and Neiger, Gil and Rodgers, Dion and Santoni, Amy L and Martins, Fernando CM and Anderson, Andrew V and Bennett, Steven M and Kagi, Alain and Leung, Felix H and Smith, Larry. Intel virtualization technology. Computer 38,5 (2005), 49-56

[6] AMD. Secure virtual machine architecture reference manual.

[7] S. Berger, R. C´aceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In Proceeding of USENIX-Security'06, Berkeley, CA, USA, 2006.

[8] Lin, Guoyuan and Bie, Yuyu and Lei, Min. Trust Based Access Control Policy in Multi-domain of Cloud Computing. Journal of Computers 8,5 (2013), 1357--1365

[9] Meng, Bo and Huang, Wei and Li, Zimao. Automated Proof of Resistance of Denial of Service Attacks Using Event with Theorem Prover. Journal of Computers 8,7 (2013), 1728--1741

[10] Uhlig, Rich and Neiger, Gil and Rodgers, Dion and Santoni, Amy L and Martins, Fernando CM and Anderson, Andrew V and Bennett, Steven M and Kagi, Alain and Leung, Felix H and Smith, Larry. Intel virtualization technology. Computer 38,5 (2005), 48-56

[11] AMD. Secure virtual machine architecture reference manual.

Fei Liu Beijing China, October, 1972.

She is the deputy head of the security research institute of China Mobile Research Institute. She works on security infrastructure, terminals and smart card/business/network security.

Ms. Liu is the deputy head of the security group of China Communication Standards Association (CCSA-TC5). Ms. Liu was awarded with the second place of National Institute of Communications Science and Technology Progress Award once, the first place of China Moble and Technological Progress Award once, and the second place of China Moble and Technological Progress Award multiple times.

Langfang Ren Beijing China, July, 1982. Master of Engineering, Beijing Jiaotong University, March, 2007.

She is an imtermediate engineer of the security research institute of China Mobile Research Institute. She works on cloud security and network security area

Hongtao Bai Beijing China, December, 1982. Master of Engineering. Peking University, January, 2008

He is the technical manager, intermediate engineer of the security research institute of China Mobile Research Institute. He works on security protocol, cloud security, mobile security and web security area.