A Novel Approach for Untrusted Code Execution

Yan Wen[1], Jinjing Zhao[2], and Huaimin Wang[1]

[1] School of Computer, National University of Defense Technology,
Changsha, China, 410073
wenyan@nudt.edu.cn, whm_w@163.com
[2] Beijing Institute of System Engineering,
Beijing, China, 100101
misszhaojinjing@sina.com.cn

Abstract. In this paper, we present a new approach called Secure Virtual Execution Environment (SVEE) which enables users to "try out" untrusted software without the fear of damaging the system in any manner. A key feature of SVEE is that it implements the OS isolation by executing untrusted code in a hosted virtual machine. Another key feature is that SVEE faithfully reproduces the behavior of applications, as if they were running natively on the underlying host OS. SVEE also provides a convenient way to compare the changes within SVEE and host OS. Referring to these comparison results, users can make a decision to commit these changes or not. With these powerful characteristics, SVEE supports a wide range of tasks, including the study of malicious code, controlled execution of untrusted software and so on. This paper focuses on the execution model of SVEE and the security evaluation for this model.

Keywords: Virtual execution environment, isolated execution, execution model, virtual machine.

1 Introduction

On PC platforms, users often download and execute freeware/shareware. To benefit from the rich software resource on the Internet, most of the PC users seem to be willing to take the risk of being compromised by untrusted code.

To enhance the host security, some host-based security mechanisms have been deployed, such as access control, virus detection and so on. But the access control mechanism will be easily bypassed by authorized but malicious code. The virus detection technology has been introduced to prevent the computer system from the widely prevalent malware, yet such technology does not work well for the unknown malware. A more promising approach for defending against unknown malicious code is based on sandboxing. However, the policies which the commodity sandboxing tools incorporate trend to be too restrictive to execute most useful applications. Consequently, the PC users, often not a computer expert, will prefer functionality to security. Thus, isolation execution, an intrusion-tolerant mechanism, has been applied to allow untrusted programs to run while shields the rest of the system from their effects. But on PC platforms, existing isolation solutions fail to achieve both the OS isolation and the execution environment reproduction (reproducing the execution

S. Qing, H. Imai, and G. Wang (Eds.): ICICS 2007, LNCS 4861, pp. 398–411, 2007.

environment of the trusted environment in the untrusted environment), i.e., they cannot provide security against potential privileged malware without negating the functionality benefits of benign programs.

In this paper, we propose a new execution model called *Secure Virtual Execution Environment* (SVEE) for executing untrusted code. In this execution model, all the untrusted code should be executed within a hosted virtual machine (SVEE VM) while other programs run in host OS. This feature guarantees the OS isolation and provides security against the privileged malicious code. The most desirable feature of SVEE VM is that it boots not from a newly installed OS image but just from the underlying host OS, so the execution environment reproduction is achieved. This is significantly different from the existing VM-based security approaches. In this local-booted OS, no privileged operations will be restricted. Thus, the behavior of untrusted code is reproduced accurately. To retain the acceptable execution results within SVEE VM, SVEE also provides an approach for users to track and compare the changes within SVEE VM and host OS. Using these comparison results for reference, users can make a choice between committing these execution effects and discarding them.

The rest of the paper is organized as follows. Section 2 covers the execution model details of SVEE and discusses its implementation architecture. Section 3 proposes a qualitative security evaluation for SVEE. Section 4 shows the current implementation status and provides an evaluation of the functionality as well as the performance of our approach, then presents our plans for future work. In Section 5, we review previous works on isolated execution technology. Section 6 concludes this paper.

2 Execution Model of SVEE

As discussed in the previous section, the goal of SVEE is to accomplish three capabilities: *OS isolation*, *execution environment reproduction* and *execution effects committing*. The capability of OS isolation is a prerequisite to make the trusted environment be resistant to the attacks from kernel-mode malicious code. Execution environment reproduction is necessary to reproduce the behavior of untrusted code because the behavior of an application is usually determined by the execution environment, especially the contents of the file system. Besides, the execution environment reproduction should not be implemented via duplicating the complete resource of trusted environment, viz. reinstalling the OS and software in the untrusted environment. This is because few PC users can afford such deployment overhead from the usability's standpoint. From the security pointer of view, the resource to be reproduced must be *configurable* for users to avoid uncovering the security-sensitive or privacy-sensitive files. In addition, for many of the applications running within untrusted environment, a user would like to retain the results of activities that are acceptable. So the execution mode of SVEE should provide an approach to track and commit the execution results of the isolated programs.

To achieve OS isolation, the execution model of SVEE must introduce the virtual machine monitor as the software layer to close off the trusted environment and the untrusted ones. According to the definition of Goldberg [1], a virtual machine monitor (VMM) is software for a computer system that creates efficient, isolated programming environments that are "duplicates", which provide users with the

appearance of direct access to the real machine environment. These duplicates are referred to as virtual machines. There are two different types of VMMs that can serve as a virtualization environment: Type I VMM and Type II VMM. A Type I VMM just runs above a bare computer hardware platform. It tends to be implemented as a lightweight OS with the virtualization capabilities. A Type II VMM is executed as an application. The OS that manages the real computer hardware is called the "host OS". Every OS that runs in the Type II virtual machine is called a "guest OS ". In a Type II VMM, the host OS provides resource allocation and a standard execution environment to each guest OS.

Considering the performance of the trusted environment, Type II VMM wins an advantage over Type I VMM [1]. For Type I VMM, all OSes run above the virtual machine. So every OS, including the one serving as the trusted environment, cannot but suffer the performance penalties due to virtualization [2]. But for Type II VMM, the trusted environment, viz. the host OS, suffers no performance overhead. In addition, unlike mainframes that are configured and managed by experienced system administrators, desktop and workstation PC's are often preinstalled with a standard OS and managed by the end-user. Ignoring the difficulty of proposing a practical and seamless migration approach for PC platforms, it will maybe take several years to migrate all of them to the Type I VMM. It also might be unacceptable for a PC user to completely replace an existing OS with a Type I VMM. In contrast, Type II VMM allows co-existing with the preinstalled host OS and programs.

Thus, taking into account that PC platform is the prime concern for SVEE, as well as the significant predominance of Type II VMM on PC platforms, we select Type II VMM over Type I VMM.

The execution model is illustrated in Fig. 1. If users wish to execute any untrusted program, they should firstly configure which resource will be reproduced and then boot the local-booted virtual machine (SVEE VM) created by SVEE VMM. From then on, these two OSes, the host OS above the bare computer hardware and the local-booted OS above SVEE VM, will run concurrently. SVEE VMM catches the sensitive instruction traps and emulates their semantics to implement a Type II VMM. In this execution model, The SVEE VM serves as the untrusted execution environment wherein all untrusted programs are bounded. The local-booted OS above this virtual machine just is the virtualized instance of the underling host OS. In the local-booted OS, the behavior of untrusted programs is reproduced accurately while isolating their effects from the host OS which is the execution environment of the trusted applications.

After SVEE ending, the user may make a choice among discarding the modification effects within SVEE, reserving them and committing them. In the first case, the contents of SVEE VM will be destroyed, which means that we simply delete all the reproduced resource and leave the contents of the file system in host OS "as is". In the second case, we reserve all the reproduced resource, so we can start SVEE VM using them next time or access them at any time. And in the third case, the contents of the reproduced resource need to be merged into the host OS.

When merging the reproduced resource and the native file system of host OS, conflicting changes may have taken place within and outside the SVEE VM. For example, the same file may have been modified both in host OS and in SVEE VM. In such cases, it is unclear what the desired merging result should be. Thus, firstly we

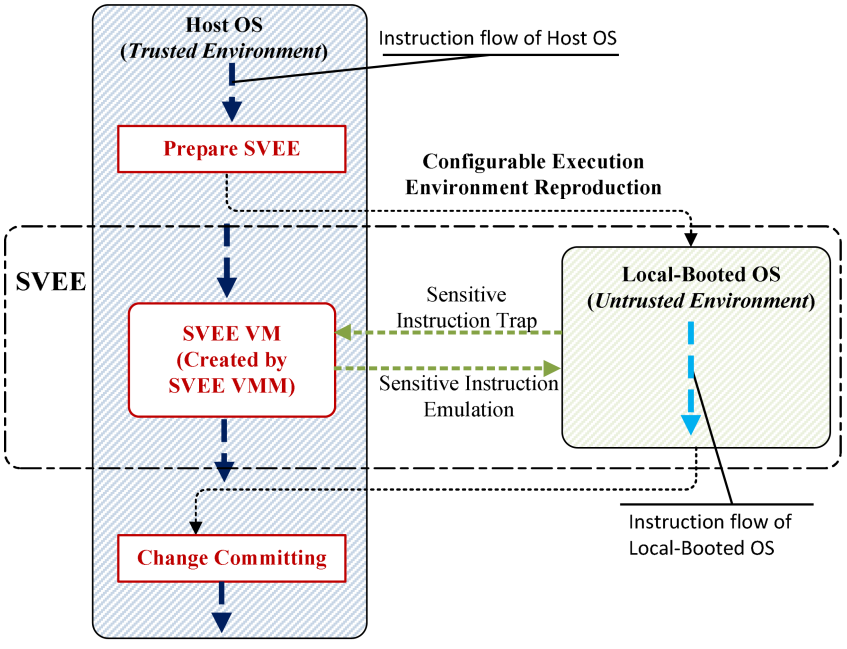

Fig. 1. Execution Model of SVEE

must identify *commit criteria* that ensure the consistency of the file systems in host OS when implementing the commit operation. We use the commit criteria described in [3]. If the commit criteria are not satisfied, then manual reconciliation of conflicting actions that took place inside the SVEE VM and outside will be needed. On this condition, SVEE will provide the user the details about such conflict. Referring to this information, the user can make a choice among optional operation:

Abort, just discards the results of SVEE VM execution.

Retry, that means discarding the results of SVEE VM execution, restarting a new SVEE VM, redoing the actions that were just performed, and then trying to commit again. Usually it often has a high probability to solve the conflicts.

Resolve conflicts, in this case, it is the user's duty to commit the contents manually.

To achieve the capabilities discussed previously, we introduce the local-booted technology implement SVEE. As shown in Fig. 2, SVEE is composed of three key components: *SVEE Virtual Machine Monitor* (SVEE VM), *Virtual Simple Disk* and *Tracking Manager*.

***SVEE Virtual Machine Monitor (SVEE VMM)*:** it's a novel local-booted virtual machine monitor which creates the local-booted virtual machine (SVEE VM). The local-booted OS, wherein untrusted programs run, just boots within this virtual machine. With the strong isolation capability of this system virtual machine, we achieve the features of **OS Isolation** and **OS & Application Transparency** effectively. With the local-booting technology, SVEE implements partial one-way isolation [4]. One-way isolation makes the host environment visible within the SVEE

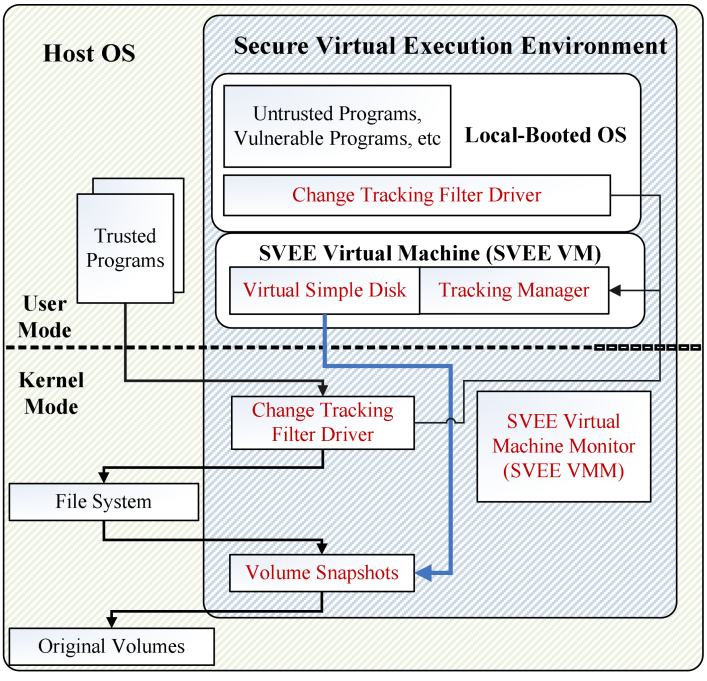

Fig. 2. SVEE Architecture

VM. Our partial one-way isolation means the environment visible within SVEE VM is a branch of host OS, and this branch was created just at the time SVEE VM started. In this sense, execution environment *reproduction* is achieved.

Virtual Simple Disk Based on Volume Snapshot. The key challenge to implement the local-booting technology is how to reuse the system volume, wherein OS is installed. While SVEE VM is running, the host OS is also modifying the same system volume. However, the local-booted OS cannot be aware of these modifications and vice versa. So they will crash because of the content inconsistency between the memory and the disks. SVEE resolves these conflicts by introducing the *virtual simple disk* based on *volume snapshot*. Volume snapshot introduces Copy-on-Write mechanism to shield the modification effects of host O from SVEE VM and vice versa. Virtual simple disk acts as the virtual storage device to export the volume snapshots to SVEE VM. Before exporting volume snapshots, the user can remove the files or folders he does not want to make visible inside SVEE VM. This characteristic makes our *execution environment reproduction* more *configurable*, i.e., the processes in SVEE VM are given access to only the volumes and files exported to SVEE VM, but not the whole file system.

From the perspective of implementation, the snapshot of an entire disk device is more intuitive than a volume snapshot. However for SVEE, such a virtual simple disk has more benefits listed as follows:

Can configure the volumes to export. If SVEE uses the disk snapshots directly, all the volumes in this disk will be visible inside SVEE VM (this is usually not the users'

desire). While in our approach, only the volumes the users want to expose will be accessible within SVEE VM.

Volume format transparent. There are several types of volumes in host OS, including single partition volume and multi-partition volumes, e.g., mirrors, stripes and RAID-5. So if the disk to export contains any multi-partition volume, we must export all other disk which this volume depends on. But our approach avoids this trouble.
Convenient to manipulate the data in snapshots. A volume is the basic unit to mount for the file system. Via mounting volume snapshots, we can expediently access their files in the host OS.

Tracking Manager. To support monitoring and committing changes, *change tracking filter drivers* are deployed within both local-booted OS and host OS. The *tracking manger* is responsible for collecting the results and comparing them to generate committing references for users.

As a summary, the key component of SVEE is the SVEE VM, a system virtual machine, whose effects are to be shielded from the host OS. Any untrusted code or the programs that trend to be attacked will be bounded inside SVEE, and share the same consistent OS. One or more such SVEE VMs can be active on the same host OS. Moreover, SVEE also provides a convenient way for users to compare the changes within local-booted OS and host OS. Using these comparison results for reference, users can make a decision to commit these changes or not.

3 Security Evaluation of SVEE

Section 2 has covered the execution model details of SVEE and its advantages under PC platforms. In this section, we evaluate the security of SVEE qualitatively. Thus, the correlative definitions are listed as follows:

$\mathbf{S} = \{ p \mid p \text{ is a program}\}$

$\mathbf{S^U} = \{ p \mid p \in \mathbf{S} \text{ is an untrusted program}\}$, $\mathbf{S^T} = \{ p \mid p \in \mathbf{S} \text{ is a trusted program}\} = S - \mathbf{S^T}$

$\mathbf{S^M} = \{ p \mid p \in \mathbf{S^U} \text{ and contains malicious code}\}$, $\mathbf{S^I} = \mathbf{S^U} - \mathbf{S^M}$

$\mathbf{S^V} = \{ p \mid p \in \mathbf{S^T} \text{ and contains vulnerable code}\}$, $\mathbf{S^S} = \mathbf{S^T} - \mathbf{S^V}$

VMM and OSes are two types of special programs, for they are programs as well as execution environments.

$\mathbf{S}_{env} = \{ p \mid p \in \mathbf{S} \text{ and runs within } env\}$, $env \in \mathbf{ENV} = \{$*OS*, *local-booted OS, host OS*$\}$, *OS* refers to a conventional multiprogramming OS, *local-booted OS* and *host OS* are illustrated in Fig. 2.

$\mathbf{P}(p)$, $p \in \mathbf{S}_{env:}$ **:** The probability that p will cause a security violation within *env* to occur.

$\mathbf{P_M}(p)$, $p \in \mathbf{S}_{env:}$ **:** The probability that program p within *env* contains malicious code.

$\mathbf{P_V}(p)$, $p \in \mathbf{S}_{env:}$ **:** The probability that a given program p contains vulnerable code which will cause a security violation to occur.

Size (p)**:** the number of lines of a program p in source code, this is a measurement for a software scale.

Based on these definitions, we would get the following conclusions:

$$\mathbf{S} = \mathbf{S^T} \cup \mathbf{S^U} = (\mathbf{S^V} \cup \mathbf{S^S}) \cup (\mathbf{S^M} \cup \mathbf{S^I}) \tag{1}$$

$$\mathbf{P}(p) = \mathbf{P_M}(p) + \mathbf{P_V}(p), \text{ and } \sum_{p \in \mathbf{S}_{env}} \mathbf{P}(p) = \sum_{p \in \mathbf{S}_{env}} \mathbf{P_M}(p) + \sum_{p \in \mathbf{S}_{env}} \mathbf{P_V}(p) \le 1 \tag{2}$$

$$\mathbf{P}(\mathbf{S}'_{env}) = \sum_{p \in \mathbf{S}'_{env}} \mathbf{P}(p) < \mathbf{P}(\mathbf{S}''_{env}) = \sum_{p \in \mathbf{S}''_{env}} \mathbf{P}(p) \qquad \textbf{for } \mathbf{S}'_{env} \subset \mathbf{S}''_{env} \tag{3}$$

As showed in formula (3), the probability of system failure tends to increase with the load on the ***env*** (i.e., the number of different requests issued, the variety of functions provided, the frequency of requests, etc.).

Noted "secure coder" Wietse Venema estimates that there is roughly one security bug per 1000 lines in software source code. This conclusion assumed the complexities of all the programs to be analyzed are approximately same. So we can deduce that the vulnerability of a program p is proportion to **Size**(p). Thus, $\mathbf{P_V}(p)$ can be calculated as:

$$\mathbf{P_v}(p) = \alpha \times \frac{\mathbf{Size}(p)}{\sum_{p_i \in \mathbf{S}_{env}} \mathbf{Size}(p_i)},\ p \in \mathbf{S}_{env},\ \alpha \text{ is a constant.} \tag{4}$$

For a conventional multiprogramming *OS*, we can calculate $\mathbf{P}(\mathbf{S_{OS}})$ by:

$$\begin{aligned} \mathbf{P}(\mathbf{S_{OS}}) &= \mathbf{P}(\mathbf{S^T_{OS}} \cup \mathbf{S^U_{OS}}) = \mathbf{P}(\mathbf{S^T_{OS}}) + \mathbf{P}(\mathbf{S^U_{OS}}) = \mathbf{P}(\mathbf{S^V_{OS}}) + \mathbf{P}(\mathbf{S^S_{OS}}) + \mathbf{P}(\mathbf{S^M_{OS}}) + \mathbf{P}(\mathbf{S^I_{OS}}) \\ &= \mathbf{P}(\mathbf{S^V_{OS}}) + \mathbf{P}(\mathbf{S^M_{OS}}) + \mathbf{P}(\mathbf{S^I_{OS}}) \end{aligned} \tag{5}$$

In formula (5), $\mathbf{P}(\mathbf{S^S_{OS}})$ is ignored because the programs in $\mathbf{S^S_{OS}}$ are trusted and without any vulnerability. Then, with these basic conclusions, we can evaluate the security of the isolation mechanism for hosted SVEE architecture as follows.

For the local-booted OS within SVEE and underlying host OS:

$$\mathbf{P}(\mathbf{S_{Local\text{-}Booted\ OS}}) = \mathbf{P}(\mathbf{S^U_{Local\text{-}Booted\ OS}}) = \mathbf{P}(\mathbf{S^M_{Local\text{-}Booted\ OS}}) + \mathbf{P}(\mathbf{S^I_{Local\text{-}Booted\ OS}}) \tag{6}$$

$$\mathbf{P}(\mathbf{S_{Host\ OS}}) = \mathbf{P}(\mathbf{S^T_{Host\ OS}}) = \mathbf{P}(\mathbf{S^V_{Host\ OS}}) + \mathbf{P}(\mathbf{S^S_{Host\ OS}}) = \mathbf{P}(\mathbf{S^V_{Host\ OS}}) \tag{7}$$

Considering that within host OS, only the *SVEE VMM*, network adapter driver and network protocol components of OS will exchange data with other environments, we can deduce the following formula:

$$\begin{aligned} &\mathbf{S^V_{Host\ OS}} \cong \{\textit{SVEE VMM}, \textit{Network Components}\},\ \mathbf{S^T_{Host\ OS}} \subset \mathbf{S_{OS}} \textbf{ and } |\mathbf{S^T_{Host\ OS}}| \ll |\mathbf{S_{OS}}| \\ &\mathbf{Size}(\textit{SVEE VMM}) + \mathbf{Size}(\textit{Network Components}) \ll \textit{Size}(\textit{OS}) \end{aligned} \tag{8}$$

Based on formula (3), (4), (7) and (8), inequality (9) is reached:

$$\mathbf{P}(\mathbf{S_{Host\ OS}}) \cong \mathbf{P}(\textit{SVEE VMM}) + P(\textit{Network Components}) \ll \mathbf{P}(\mathbf{S_{OS}}) \tag{9}$$

Since *SVEE VMM*, a Type II VMM, tends to be shorter, simpler, and easier to debug than conventional multiprogramming OSes, even when $\mathbf{S_{SVEE\ VMM} = S_{OS}}$, the VMM is less error-prone. For example, since the VMM is defined by the hardware specifications of the real machine, the field engineer's hardware diagnostic software can be used to checkout the correctness of the VMM.

For all untrusted programs run within SVEE VM, and SVEE is particularly concerned about the host OS security, we can define the probability of a program *p* on one SVEE VM violating the security of another concurrent program on host OS as:

$$\mathbf{P}(\mathbf{S}_{\text{Local-Booted OS}} \mid VMM \mid Host\ OS) + \mathbf{P}(\mathbf{S}_{\text{Host OS}}) = \mathbf{P}(\mathbf{S}_{\text{Local-Booted OS}}) \times \mathbf{P}(VMM) \times \mathbf{P}(Host\ OS) + \mathbf{P}(\mathbf{S}_{\text{Host OS}}) \tag{10}$$

$\mathbf{P}$ ($\mathbf{S_{local\text{-}booted\ OS}}$ |*VMM* | *host OS*) is the probability of the simultaneous security failure of local-booted OS, VMM and host OS. If a single OS's security fails, the VMM isolates this failure from the other virtual machines. If the VMM'S security fails, the malicious code will have to break the protection of host OS. But, if functioning correctly, malicious code within local-booted OS will not take advantage of the security breach. This assumes that the designers of the individual OSes are not in collusion with malicious users. This seems to be a reasonable hypothesis.

Based on the formulas of (3), (9) and |{*VMM*}|=|{*host OS*}|=1 << |$\mathbf{S_{OS}}$|, we arrive at the following conclusion:

$$\mathbf{P}(\mathbf{S}_{\text{Local-Booted OS}} \mid VMM \mid Host\ OS) + \mathbf{P}(\mathbf{S}_{\text{Host OS}}) = \mathbf{P}(\mathbf{S}_{\text{Local-Booted OS}}) \times \mathbf{P}(VMM) \times \mathbf{P}(Host\ OS) + \mathbf{P}(\mathbf{S}_{\text{Host OS}}) << \mathbf{P}(\mathbf{S}_{\text{OS}}) \tag{11}$$

As a summary, based on the inequality (11), the conclusion that the isolation architecture of SVEE improves the security of host OS observably can be reached.

4 Status and Future Work

SVEE has been firstly implemented on Windows with Intel x86 processors because of the prevalence of Windows and Intel processors under PC platforms. A detailed description of SVEE implementation is beyond the scope of this paper. Instead, the framework of the three key components in SVEE is outlined in this section.

It's well-known that Intel x86 processor is not virtualizable [5]. Ignoring the legacy "real" and "virtual 8086" modes of x86, even the more recently architected 32- and 64-bit protected modes are not classically virtualizable for its visibility of privileged state and lack of traps when privileged instructions run at user-level. To address this problem, we have come up with a set of unique techniques that we call *ISDT* (*Instruction Scan and Dynamic Translation*) technology, which is composed of two components: *Code Scanner* (*CS*) and *Code Patcher* (*CP*). Before executing any ring 0 code, *CS* scans it recursively to discover problematic instructions. *CP* then performs in-situ patching, i.e. replace the instruction with a jump to hypervisor memory where an integrated code generator has placed a more suitable implementation. In reality, this is a very complex task as there are lots of odd situations to be discovered and handled correctly.

We implement the volume snapshot using the Windows volume filter driver. This driver creates two types of device objects, one is a *volume filter device object* just located above the original volume to filter all I/O Request Package (IRP) sent to it and execute the *COW* operations, and the other is a *volume snapshot device object* which exports all general volume interfaces to provide a way to access volume snapshots.

To support change committing, we must track all the modification made within SVEE VM and host OS. For Windows, besides file changes, the registry changes are also pivotal. Our approach accomplishes file change tracking as a file system filter driver, and adopts a Native API interceptor to monitor the registry modification. On termination of SVEE, tracking manager collects the change results generated by *change tracking filter driver* and *registry monitor*, and compares them to provider committing reference for uses.

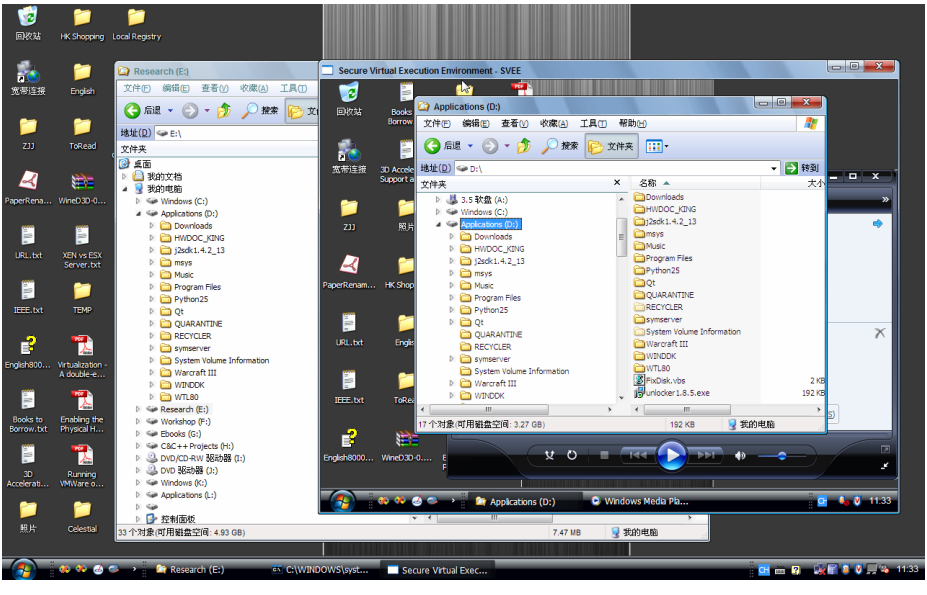

Fig. 3. Screenshot of a Running SVEE

Fig. 3 is a screenshot of a running SVEE VM showed in the window with a title of *Secure Virtual Execution Environment*. The resolution of the local-booted Windows within SVEE VM is 1024x768 while the resolution of host Windows is 800x600, so the icon arrangement within its desktop differs from that of host Windows. As showed in Fig. 3, the programs of *Explorer* and *MediaPlayer* are running in this local-booted Windows. Compared the file system volumes showed in the Explorer programs running within local-booted Windows and outside, we can find that only the volumes C: and D: are exported to it. This just brings forth the SVEE's capability of *configurable execution environment reproduction*: the resource to be reproduced to SVEE VM can be configurable for users. In the Explorer of host Windows, volume K: and L: are the relevant snapshots of C: and D:. When local-booted Windows is running, no programs except SVEE VMM on host Windows can access these

snapshots which compose the virtual simple disk of SVEE VM. After local-booted Windows ends, SVEE will help users to access its file system contents for *execution effects committing*.

We have tested the basic functions of SVEE VMM, including instruction set and hardware virtualization. Instruction set virtualization is verified by the QEMU's test-i386 tool, which we have ported to Windows. This tool tests all the x86 user-mode instructions, including SSE, MMX and VM86 instructions. The results show that the execution results of all the instructions are equivalent with those in Host Windows. In addition, we ran PassMark on local-booted Windows. All the virtual hardware devices works perfectly well, including IDE disk, CD-ROM, network card, display adapter and so forth.

For a desktop-oriented workload, we ran Everest Ultimate 2006 both natively and in a local-booted Windows. Everest Ultimate is a synthetic suite of microbenchmarks intended to isolate various aspects of workstation performance. Since user-level computation is almost not taxing for VMMs, we expect local-booted Windows runs to score close to native. Fig. 4 confirms this expectation, showing a slowdown over native of 0.41-4.18%, with a 1.75% average slowdown for SVEE VMM.

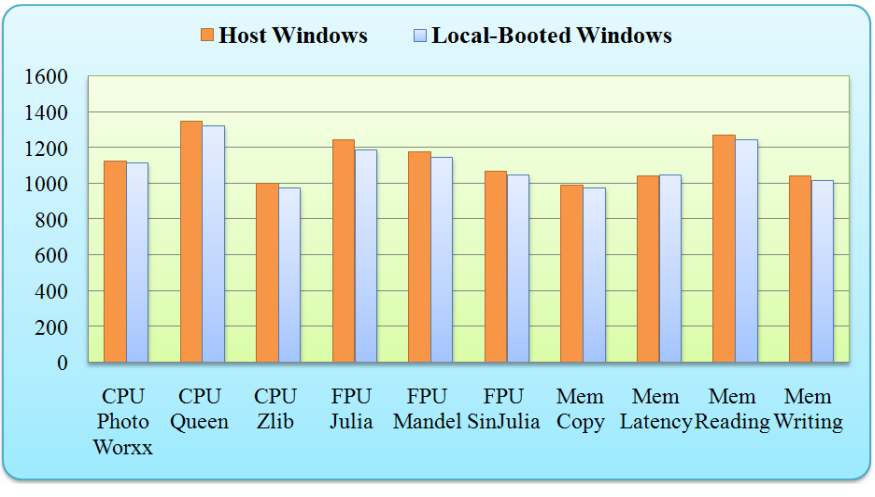

Fig. 4. Performance Comparison between host OS and SVEE VM

To improve the usability and performance of SVEE, we are currently improving the memory management mechanism of SVEE VMM to share the memory pages between SVEE VM and host OS. Multiprocessor virtualization capability is also to be added to SVEE VMM to support Multiprocessor-Specialized host OS version. In addition, we are integrating some intrusion detection mechanisms into SVEE at the virtual hardware layer. To make SVEE VM support the multimedia programs such as 3D games, we plan to reimplement the graphic virtualization mechanism referring to the approach proposed by H. Andres Lagar-Cavilla, et al [6]. Finally, we will investigate the automated change committing technology for SVEE.

5 Related Work

Sandbox. A sandbox is an environment in which the actions of a process are restricted according to a security policy. Sandboxing based approaches involve observing a program's behavior and blocking actions that may compromise the system's security. Janus [7] and Chakravyuha [8] implement sandboxing using kernel interposition mechanism. MAPbox [9] introduces a sandboxing mechanism with the aim at making the sandbox more configurable and usable via providing a template for sandbox policies based on a classification of application behaviors. Safe Virtual Execution (SVE) [10] implements sandboxing using *software dynamic translation*, a technology for modifying binaries at runtime. Systrace [11] proposes a sandboxing system that notifies the user about all the system calls that an program tries to invoke and then generates a policy for the program according to the response from the user.

However, use of sandboxing approaches in practice has been hampered by the difficulty of policy selection: determining resource access rights that would allow the code to execute successfully without compromising system security. Sandboxing tools often adopt highly restrictive policies that preclude execution of most useful applications. So users usually choose functionality over security, i.e., executing untrusted code outside such sandboxing tools, exposing themselves to unbounded damage if this code turned out to be malicious.

Isolation Technology within Mono-OS. Isolated execution has previously been studied by researchers in the context of Java applets [12, 13]. Compared with general applications, such applets do not make much access to system resources. So the approach used by applets often relied on executing these untrusted applets on a "remote playground", i.e., an isolated computer. However, most of the desktop applications will usually require access to more resources such as the file system on the user's computer. To run such applications on a remote playground, the complete execution environment on the user's computer, especially the entire file system contents, should be duplicated to the remote playground.

Literature [4] is the first approach to present a systematic development of the concept of *one-way isolation* as an effective means to isolate the effects of running processes from the point they are compromised. They developed protocols for realizing one-way isolation in the context of databases and file systems. However, they did not present an implementation of their approach. As a result, they do not consider the research challenges that arise due to the nature of COTS applications and commodity OSes.

Alcatraz [14] and its improved version [3], *Security Execution Environment (SEE)*, proposes its improved version with the name of. A key property of SEE is that it reproduces the behavior of applications, as if they were running natively on the underlying host OS. But this approach does not achieve OS isolation, so such protection mechanism can be bypassed by kernel-mode malicious code. And in SEE, a number of privileged operations, such as mounting file systems, and loading/unloading modules are not permitted.

All these approaches suffer from the same problem: they can be turned off if intruders compromise the operating system and gain system privileges [15]. The file protection they provide is thus less effective in a compromised environment.

Isolation Based on Virtual Machine. Covirt [16] proposes that most of applications may be run inside virtual machine instead of host machines. User-mode VMs have been proposed for the Linux OS [17]. All the above approaches suffer from the difficulty of environment reproduction.

Denali [18, 19] is another virtual machine based approach that runs untrusted distributed server applications. Denali focuses on supporting lightweight VMs, relying on modifications to the virtual instruction set exposed to the guest OS and thus requiring modifications to the guest OS. In contrast, we are focusing on heavier weight VMs and make no OS modifications.

VMWare ESX Server provides an isolation approach for server platform with a similar objective to ours. XEN [20] and L4-based virtual machine [21] also implement isolated virtual execution environments. But all of these three environments are just located above computer hardware in form of Type I VMM. So as discussed in section 2, they are not fit for PC platforms because of their drawbacks caused by Type I VMM architecture.

The COW/COW2 mechanism of QEMU [22], an open source emulator, can only isolate the Guest OS's modifications to file system from host OS. But modifications made by host OS will cause the conflicts between the disk and file system content in Guest OS and crash it. Thus QEMU failed to achieve the environment reproduction. Besides, its poor performance prevents it from server as an effective virtual environment. KVM [23], a Kernel-based Virtual Machine based on QEMU, significantly improves the performance. But it also cannot provide the capability of environment reproduction. Besides, it must modify the host OS and rely on the hardware virtualization technology, such as Intel VT and AMD-V.

6 Conclusions

In this paper, we proposed a new execution model called SVEE for executing untrusted code safely and shown the security evaluation for this model. SVEE is versatile enough to coexist with the existing OS and programs. The most considerable benefit of SVEE is that it provides the capability of OS isolation while accomplishing the configurable execution environment reproduction. SVEE also provides a convenient way for users to track the changes made within the SVEE VM, viz. the untrusted execution environment. These changes can be discarded if the user does not accept them. Otherwise, the changes can be committed so as to become visible within host OS.

SVEE accomplishes all the capabilities discussed in section 2: *OS isolation*, *configurable execution environment reproduction* and *execution effects committing*.

Consequently, SVEE provides security against potential malicious code without negating the functionality benefits provided by benign programs. With these capabilities, SVEE supports a wide range of tasks, including the study of malicious code, controlled execution of untrusted software, experimentation with software configuration changes, testing of software patches and so on.

Acknowledgements

This research is supported by National Basic Research Program of China (Grant No. 2005CB321801), National Natural Science Foundation of China (Grant No. 60673169), and National Science Fund for Outstanding Youths under Grant No. 60625203.

References

1. Goldberg, R.P.: Architectural Principles for Virtual Computer Systems, Ph.D. Thesis. Harvard University, Cambridge, MA (1972)
2. Adams, K., Agesen, O.: A Comparison of Software and Hardware Techniques for x86 Virtualization. In: Proceedings of The 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006), pp. 2–13 (2006)
3. Sun, W., Liang, Z., Sekar, R., Venkatakrishnany, V.N.: One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. ISOC Network and Distributed System Security (NDSS 2005) (2005)
4. Liu, P., Jajodia, S., McCollum, C.D.: Intrusion confinement by isolation in information systems. Journal of Computer Security 8, 243–279 (2000)
5. ScottRobin, J.: Analyzing the Intel Pentium's Capability to Support a Secure Virtual Machine Monitor, Master's Thesis. Naval Postgraduate School, Monterey, CA, 133 (1999)
6. Lagar-Cavilla, H.A.e., Tolia, N., Satyanarayanan, M., Lara, E.d.: VMM-Independent Graphics Acceleration. In: Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2007), San Diego, CA (2007)
7. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.: A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In: Proceedings of the Sixth USENIX UNIX Security Symposium, San Jose, California (1996)
8. Dan, A., Mohindra, A., Ramaswami, R., Sitaram, D.: ChakraVyuha(CV): A Sandbox Operating System Environment for Controlled Execution of Alien Code. IBM T.J. Watson research center (1997)
9. Acharya, A., Raje, M.: Mapbox: Using Parameterized Behavior Classes to Confine Applications. In: Proceedings of the 9th USENIX Security Symposium (2000)
10. Scott, K., Davidson, J.: Safe Virtual Execution using Software Dynamic Translation. In: Computer Security Applications Conference, pp. 209–218 (2002)
11. Provos, N.: Improving Host Security with System Call Policies. In: Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA (2003)
12. Chiueh, T.-c., Sankaran, H., Neogi, A.: Spout: A Transparent Distributed Execution Engine for Java Applets. In: Proceedings of the 20th International Conference on Distributed Computing Systems, vol. 394 (2000)
13. Malkhi, D., Reiter, M.K.: Secure Execution of Java Applets using A Remote Playground. IEEE Transactions on Software Engineering 26, 1197–1209 (2000)
14. Liang, Z., Venkatakrishnan, V.N., Sekar, R.: Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In: Omondi, A.R., Sedukhin, S. (eds.) ACSAC 2003. LNCS, vol. 2823, Springer, Heidelberg (2003)
15. Kernel brk() vulnerability, http://seclists.org/lists/bugtraq/2003/Dec/0064.html
16. Chen, P.M., Noble, B.D.: When Virtual is Better Than Real. In: 8th Workshop on Hot Topics in Operating Systems (2001)

17. Dike, J.: A User-mode Port of the Linux Kernel. In: Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, Georgia, USA (2000)
18. Whitaker, A., Shaw, M., Gribble, S.D.: Denali: A Scalable Isolation Kernel. In: Proceedings of the Tenth ACM SIGOPS European Workshop, Saint-Emilion, France (2002)
19. Whitaker, A., Shaw, M., Gribble, S.D.: Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In: Proceedings of the 2002 USENIX Technical Conference (2002)
20. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauery, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003), pp. 164–177. ACM Press, New York (2003)
21. Biemueller, S., Dannowski, U.: L4-Based Real Virtual Machines - An API Proposal. In: Proceedings of the MIKES 2007: First International Workshop on MicroKernels for Embedded Systems, Sydney, Australia, pp. 36–42 (2007)
22. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: USENIX Association Technical Conference (2005)
23. Qumranet: KVM: Kernel-based Virtualization Driver (2006)